TOSHIBA

32 ビット TX System RISC
TX19A ファミリー
アーキテクチャ編

Rev 1 .0

株式会社 東芝 セミコンダクター社

- 当社は品質、信頼性の向上に努めておりますが、一般に半導体製品は誤作動したり故障することがあります。当社半導体製品をご使用いただく場合は、半導体製品の誤作動や故障により、生命･身体･財産が侵害されることのないように、購入者側の責任において、機器の安全設計を行うことをお願いします。
  なお、設計に際しては、最新の製品仕様をご確認の上、製品保証範囲内でご使用いただくと共に、考慮されるべき注意事項や条件について「東芝半導体製品の取り扱い上のご注意とお願い」、「半導体信頼性ハンドブック」などでご確認ください。 021023_A

- 本資料に掲載されている製品は、一般的電子機器 (コンピュータ、パーソナル機器、事務機器、計測機器、産業用ロボット、家電機器など) に使用されることを意図しています。特別に高い品質･信頼性が要求され、その故障や誤作動が直接人命を脅かしたり人体に危害を及ぼす恐れのある機器(原子力制御機器、航空宇宙機器、輸送機器、交通信号機器、燃焼制御、医療機器、各種安全装置など) にこれらの製品を使用すること (以下 “特定用途” という) は意図もされていませんし、また保証もされていません。本資料に掲載されている製品を当該特定用途に使用することは、お客様の責任でなされることとなります。 021023_B

- 本資料に掲載されている製品を、国内外の法令、規則及び命令により製造、使用、販売を禁止されている応用製品に使用することはできません。 060106_Q

- 本資料に掲載してある技術情報は、製品の代表的動作･応用を説明するためのもので、その使用に際して当社及び第三者の知的財産権その他の権利に対する保証または実施権の許諾を行うものではありません。 021023_C

- 本資料に掲載されている製品は、外国為替及び外国貿易法により、輸出または海外への提供が規制されているものです。 021023_E

- 本資料の掲載内容は、技術の進歩などにより予告なしに変更されることがあります。 021023_D

- MIPS16,Application Specific Extensions 及び R3000A は MIPS Group , a division of Silicon Graphics, Inc の商標です。

はじめに

このマニュアルは当社 TX19A シリーズアーキテクチャについて説明します。このマニュアルの構成を以下に示します。

1 章　TX19A の概要
TX19A の一般的な特長を説明します。

2 章　CPU アーキテクチャ
CPU レジスタとメモリにどのようにデータがとりこまれるかを説明します。TX19A のレジスタの機能の概要も説明します。

3 章　32 ビット ISA の概要
TX19A で使用される 32 ビット命令セットアーキテクチャ(ISA)の概要を説明します。

4 章　16 ビット ISA の概要
TX19A で使用される 16 ビット命令セットアーキテクチャ(ISA)の概要を説明します。

5 章　CPU パイプライン
TX19A で実行される命令パイプラインについて説明します。

6章　メモリ管理
仮想アドレス空間と物理アドレス空間、マッピングの方法について説明します。

7章　内部 I/O バスオペレーション
ハーバードアーキテクチャと内部バスオペレーションのタイミングについて、簡単に説明します。

8 章　システム制御コプロセッサ(CP0)レジスタ
例外処理に関係するレジスタのグループについて説明します。

9 章　例外処理
例外の原因となるイベントと取り扱われる順序について説明します。

10 章　低消費電力モード
動作中の消費電力を調整する方法を説明します。

付録 A　32 ビット ISA の詳細
32 ビット ISA モードで使用できる命令を詳細に説明します。

付録 B　16 ビット ISA の詳細
16 ビット ISA モードで使用できる命令を詳細に説明します。

付録 C　プログラミング制約
アセンブリ言語プログラムで守らなければならない制約について説明します。

付録 D TX19・TX19A・TX39 の相違点
3 つの RISC プロセッサファミリを比較する。

付録 E CPU 命令（32 ビット ISA）のオペコードのビットエンコード

付録 F CPU 命令（16 ビット ISA）のオペコードのビットエンコード

対象者

このマニュアルは当社TX19Aプロセッサとコントローラを使って製品を開発しようとするソフトウエア、ハードウエア設計者を対象にしています。

TX19A のような RISC プロセッサは、CISC プロセッサとは異なる特長があります。RISC プロセッサを初めて使うユーザーは、1 章を最初にお読みください。この章を読めば RISC プロセッサの特長が理解できます。RISC プロセッサには小規模な命令セットしかない事に注意してください。RISC の命令セットは小さいので、CISC プロセッサにある LDIR(ブロック転送)、CPIR(ブロックサーチ)、BS1B（ビットスキャン）などの複合命令はありません。複合命令は RISC の命令を組み合わせて、コンパイラで作成するか、またはユーザーが作成します。

C 言語のような高級言語を使用するユーザーは、2 章のアーキテクチャの概要を読めばソフトウエアの開発が可能です。

アセンブリ言語を使用するユーザーは、ハードウエア機能を理解する必要があります。ソフトウエアの性能はユーザーがどの程度ハードウエア機能を理解しているかに影響されます。このマニュアルはアセンブリ言語を使用するユーザーを考慮して TX19A アーキテクチャを詳細に説明しています。ソフトウエア設計者は一通りお読みください。

参考マニュアル

このマニュアルを使用する上で、参考となるマニュアルを以下に示します。

- 半導体信頼性ハンドブック（集積回路編）
  半導体製品の設計と、大量生産で高品質と高い信頼性を約束する当社の方法について説明します。

第1章　TX19A の概要

この章では、まず TX19A の特長について説明します。また、当社が提供している TLCS-900/L1 などの CISC プロセッサと比較して TX19A RISC アーキテクチャが具体的にどのような特長をもっているのか、ということについて説明します。

1.1　プロセッサの一般的特長

TX19A は Silicon Graphics 社 MIPS グループの 32 ビット RISC プロセッサをベースにして、高コード効率を追求した 16 ビット命令セット MIPS16e-TX 命令セットを実装した高性能 32 ビット RISC プロセッサです。TX19A の命令セットには、TX39 の 32 ビット命令がサブセットとして含まれています。また、TX19 の 16 ビット命令もサブセットとして含まれていますので、TX19A は、TX39 と TX19 に対しソフトウエアの上位互換性があります。

TX19A ファミリーは、TX19A プロセッサコア、内部バス、および特定の用途向けにさまざまな周辺回路を組み合わせて、ASSP として、またシステム ASIC のコアとして提供されています。

■　MIPS16e-TX 命令セットと MIPS32 命令セット

◆　MIPS16e-TX 命令セットは、高コード効率を追求した MIPS16 ASE を MIPS 社合意の上で東芝にて独自に拡張したものです。東芝独自の拡張を除けば、MIPS16 ASE とオブジェクトレベルで互換性があります。

> 注意:　MIPS16 ASE の 64 ビットデータ処理の命令は TX19A にはありません。

◆　MIPS32 命令セットは、演算性能の優れた TX39 とオブジェクトレベルで互換性があります。

- MIPS16e-TX 命令セットと MIPS32 命令セットは、実行時に命令により切り換えられます。これらの命令セットを実行している状態をそれぞれ 16 ビット ISA モード、32 ビット ISA モードと呼びます。
- ハードウエアインタロック機構により、ロード命令直後にロードされたレジスタの内容を参照する命令を配置できるため、NOP 命令を挿入する必要がありません。
- 分岐ライクリ命令により分岐先で実行する命令を、分岐命令の直後に配置できるため、NOP 命令を挿入する必要がありません。

■　高性能

◆　ほとんどの命令を、1 クロックサイクルで実行します。

- 3 オペランドの演算命令
- 内部 32 ビット構成
  32 ビット汎用レジスタと 32 ビットプログラムカウンタなど
- Shadow Register Set を内蔵
  7 つの割り込みレベルに応じて自動切り替え
- 5 段パイプライン
- アクセス時間が 1 クロックサイクルの高速メモリを、命令用とデータ用に独立して内蔵可能
- ライトバッファ内蔵可能
- ハーバードアーキテクチャを採用
  TX19A は､命令とデータ(オペランド)用に別々のバスを使用します｡TX19A は、プロセッサコアにデータの出し入れをするデータバス、データ用のアドレスバス、オペコードを運ぶバス、オペコード用のアドレスの 4 本のバスをもっています。別々のバスで命令とデータを同時にアクセスできるので、命令のスループットが高くなります。
- ノンブロッキングロード機能
  外部メモリからのロードで大きな遅延を発生した場合でも、ロード遅延スロット中の後続命令を実行できます。
- 積和演算器（MAC）を内蔵
  32 ビット×32 ビット＋64 ビットの演算を 1 クロックで実行します。
  64 ビット−32 ビット×32 ビットの演算を 1 クロックで実行します。
- 4G バイトの仮想アドレス空間
- コプロセッサを内蔵
  システム構成、例外処理、メモリ管理などをつかさどるシステム制御コプロセッサ（CP0）を内蔵しています。

■　**低消費電力**

- 消費電力を抑えた最適化設計
- プログラムにより設定できる低消費電力モード（HALT モード・DOZE モード）
  外部マスタからのバス制御権の要求に応答できる DOZE モードと、応答しない HALT モードがあります。

■　リアルタイム制御に向いた高速割り込み応答

◆　各割り込みサービスルーチンのエントリーアドレスを独立化

◆　各割り込みソースに対するベクタアドレスを自動生成
例外ベクタを読み出すとき優先順位を判定し、割り込み要求レベルが現在の割り込みマスクレベルより大きい場合のみ、割り込み例外を発生します。これにより、高位の割り込み要求に対する保留を最短にします。

◆　割り込みレベルに応じたレジスタバンクの自動更新

■　システム ASIC 用マイコンコア

◆　ASIC と同一の製品プロセス、開発環境

◆　コンパクトなコア

◆　TX シリーズの標準バスである G-Bus に直接接続可能

■　システム開発環境

◆　言語ツール　C コンパイラ、アセンブラなど
当社とサードパーティのツールを整備

◆　リアルタイムオペレーティングシステム
当社(μITRON)とサードパーティのリアルタイムオペレーティングシステムを整備

◆　テバック支援システム

- ソースレベルデバック環境を提供するリアルタイムエミュレータとして、当社とサードパーティのツールを整備
- デバック支援回路（DSU）を ASIC に挿入する簡易ツールをサポート

1.2　RISC とは?

1980 年代の初めまでは、すべての CPU は複合命令セットコンピュータ（CISC）という方式をとっており、既存のソフトウエアとの互換性を維持するために、新しい種類の命令や、より複雑な演算を可能にするための機能を追加しながら進化してきました。通常、CISC はあらゆる状況を想定した何百もの命令を実現した CPU を意味します。何百もの命令を解読・実行することのできる CPU は、当然のことながら回路が非常に複雑になり、設計にも時間、コストがかかります。

ところが、1980 年代の初めになると、プロセッサの資源がソフトウエアにより実際にどのように使用されているかという統計的な分析にもとづいて、新しい概念のコンピュータが提唱されるようになります。これを縮小命令セットコンピュータ(RISC）といいます。RISC の最大の特長は、プログラマやコンパイラによりあまり使用されることのない複雑な命令をなくし、命令セットを簡素化したことです。

■　特長 1　命令数が少ない

RISC プロセッサは複雑な命令をなくし、基本的な命令に絞り込んでいます。例えば、ブロック転送、ブロックサーチ、ビットスキャンなどの複雑な命令はありません。

また、RISC プロセッサはロード・ストアアーキテクチャを採用しています。CISC プロセッサでは、メモリ中のデータをオペランドとして直接指定できる命令があります。例えば、当社 16 ビット CISC プロセッサ TLCS-900/L1 に用意されている命令、“ADD A,（1000H)” はメモリの 1000H 番地のデータを CPU に取り込んで、A レジスタの内容と加算し、その結果を再び A レジスタに書き戻す命令です。RISC ではこのような命令はなく、メモリに対する操作には、メモリ中のデータを CPU レジスタにロードするか、または CPU レジスタ中のデータをメモリにストアする命令しかありません。すなわち、すべての演算は CPU レジスタ中のオペランドを対象とします。

CISC プロセッサには非常に多くの命令があり、さらに各命令でいくつものアドレッシングモードをサポートしているため、命令を実行するのにマイクロコードが使用されます。RISC に比べるとプログラミングが容易で、コードサイズが縮小されるという利点がある一方で、マイクロコードの実現にチップのかなりの面積が割かれるため、プロセッサの性能を改良する上での障害になっています。

■　特長 2　命令が固定長である

RISC プロセッサの命令は固定長です。反対に、CISC プロセッサの命令は可変長であり、1 バイト長命令、2 バイト長命令などの命令から、長いものでは 7 バイト長の命令もあります。命令長がばらばらであると、入ってくる命令の長さが分からないため、命令デコーダは極めて複雑になります。これに対して、TX19A の 32 ビット ISA の場合、命令長はすべて 32 ビットに固定されています。命令を固定長にすることにより、命令のデコードが高速になります。

■　特長 3　多段パイプライン処理

RISC には少数の単純な命令しかないので、大部分の命令は 1 クロックサイクルで実行できます。そのため、パイプライン中の各命令が異なるクロックサイクルを必要とする CISC と比べると、RISC はパイプライン化が簡単です。通常、RISC プロセッサは多段パイプラインを備えています。

1.3　TX19A の特徴

前項では、一般的な RISC プロセッサと CISC プロセッサの違いを説明しました。この項では、TX19A の命令セットアーキテクチャ（ISA)の特徴について、当社の 8 ビット CISC プロセッサ TLCS-870/C や 16 ビット CISC プロセッサ TLCS-900/L1 と比較しながら説明します。

TX19A には 16 ビットと 32 ビットの 2 つの命令セット(ISA)があります。16 ビットの命令セットを実行している状態を 16 ビット ISA モード、32 ビットの命令セットを実行している状態を 32 ビット ISA モードと呼びます。16 ビット ISA モードと 32 ビット ISA モードは、プログラムの実行中にサブルーサン単位で、命令により切り換えられます。16 ビット ISA は独立した命令セットではなく、32 ビット MIPS アーキテクチャを拡張したものです。32 ビット ISA には 103 の命令があり、16 ビット ISA には 128 の命令があります。一般的にコードサイズを縮小したい部分は 16 ビット ISA を用い、高速化したい部分は 32 ビット ISA を用います。

一方、870/C や 900/L1 には約 1000 個の命令と多くのアドレッシングモードがあります。一般的に CISC プロセッサはコード効率の点で優れています。

1.3.1　命令セットアーキテクチャ

TX19A の命令セットアーキテクチャの特徴を以下に示します。

◆　**複雑な処理を行う命令がない**

TX19A にはロード、ストア、加算、減算、乗算、除算、AND、OR、XOR、シフト、ジャンプ、分岐などの基本的な命令しかありません。900/L1 にある LDIR (ブロック転送）、CPIR（ブロック検索）などの複雑な命令はありません。これらの複雑な処理を行う命令を CISC プロセッサがハードウエアで実行するのに対して、TX19A では基本的な命令を組み合わせてソフトウエアルーチンを作成しなければなりません。これはコンパイラ(またはプログラマ)の役目になります。ただし、例外として、高速処理が要求される積和/積差演算命令（MADD/U、MSUB/U）は命令セットの中に用意されています。（これらの命令は専用の MAC ユニットで実行されます。）

◆　**他の命令で代替できる命令はない**

命令数を減らすために、TX19A は他の命令を使って実行できる命令は除いてあります。例えば、TX19A には、NOP (No Operation）命令、INC (Increment）命令、DEC (Decrement）命令がありません。例えば、TX19A では、NOP 命令と

同様の処理は、以下のようにシフト命令を代用します。

```
SLL  r0,r0,0
```

TX19A では、レジスタ r0 はハードウエア的に 0 に固定されています。前ページの命令は、実際にはレジスタ r0 の内容を 0 ビットシフトし、その結果をレジスタ r0 に格納する命令です。（アセンブラではプログラムを分かりやすくするための疑似命令として NOP を許可しています。）

また、レジスタのインクリメントは以下のように即値加算命令 ADDIU（Add Immediate Unsigned）を用いて実行します。

```
ADDIU  rt,rs,1
```

ここで *rt* と *rs* はそれぞれターゲットレジスタとソースレジスタです。同様にレジスタのデクリメントは以下のように実行します。

```
ADDIU  rt,rs,-1
```

◆ **複数の単純な命令で実現できる命令はない**

TX19A は、2 つ以上の単純な命令から実現できる命令は切り捨て、命令数を減らしています。例えば、TX19A にはスタックに対するポップ命令、プッシュ命令はありません。CISC プロセッサでは、プッシュ命令を実行すると、レジスタの内容がスタックに退避され、スタックポインタレジスタがオペランドサイズ分デクリメントされます。TX19A では、慣例として 32 個ある汎用レジスタのうちの 1 つをスタックポインタレジスタとして使い、プッシュをこのレジスタに対する加算命令とストア命令で実行します。

◆ **ロード・ストアアーキテクチャを採用**

870/C や 900/L1 のような CISC プロセッサでは、ADD A,（1000H）のように演算命令でもメモリをアクセスできます。これに対して、従来の TX19 ではメモリへのアクセスは、ロード・ストア命令によるメモリと CPU の汎用レジスタとの間でデータを動かす命令に限定していますが、TX19A ではビット操作命令やメモリに対して即値データを加算する命令を追加しています。

◆ **メモリのアドレッシングモードを限定**

900/L1 や 870/C にはメモリをアクセスためのアドレッシングモードが 7 種類以上あります。例えば、レジスタ間接モード、オートインクリメント付きレジスタ間接モード、インデックス相対モード、ベースドインデックス相対モードなどです。これらのアドレッシングモードは、アセンブリ言語のプログラマには便利で、コードサイズを小さくするのに役立ちます。

これに対して、TX19A には、32 ビット ISA モードの場合、メモリのアドレッシングモードは、ベースド相対(オフセット付きレジスタ間接)の 1 種類しかありません。また、16 ビット ISA モードでも、これに加えて PC 相対と SP 相対と FP 相対の 4 種類のアドレッシングモードしかありません(PC 相対モードと SP

相対モードを使えるのは 3 命令に限定されています)。このようにアドレッシングモードの数が少ないため、TX19A は 900/L1 や 870/C などの CISC プロセッサに比べると、回路が単純になります。

◆ **3 オペランドの演算命令を採用**

TX19A では、2 つのソースレジスタと 1 つのデスティネーションレジスタを別々に指定できる 3 オペランドの命令があります。例を以下に示します。

```
ADD  rd,rs1,rs2
```

この命令は *rs1* と *rs2* の 2 つのソースレジスタの内容を加算し、その結果をデスティネーションレジスタ *rd* に格納します。これに対して、900/L1 では、以下のように演算命令は 2 つのオペランドしかとれません。この命令は、XWA と XBC の内容を加算して、その結果を XWA に書き戻します。

```
ADD  XWA,XBC
```

◆ **フラグレジスタがない**

TX19A には、キャリー、オーバーフロー、サインなどの演算フラグがありません。例えば、900/L1 ではキャリーフラグは加算、減算でキャリー、ボローが発生したことを記録するのに使われます。キャリーフラグは、多桁の数値を加算する際によく使われます。900/L1 には、そのためにキャリーフラグすなわち、桁上がりのビットを 2 つのレジスタの内容に加算する ADC 命令が用意されています。

一方、TX19A では、32 ビットの演算が 1 つの命令で実行できるので、フラグビットを必要とすることはあまりありません。しかし、キャリーが発生する可能性のある大きな数値の加算をするとき、最初に加算の結果キャリーが発生した場合はそれを汎用レジスタに記録する必要があります。したがって、複数ワードサイズ以上の加算をするときには、キャリーが発生する場合のコードと、キャリーが発生しない場合のコードの 2 つのコードが必要になります。

また、900/L1 の比較命令 CP(Compare)命令は、減算でボローが発生したかどうかを示すためにキャリーフラグを使用します。これに対して TX19A では、SLT(Set On Less Than)などの比較命令の結果は、汎用レジスタに格納されます。

1.3.2　命令フォーマット

TX19A には、MIPS16 と MIPS32 の 2 つの ISA モードがあります。MIPS32 ISA モードの命令は、すべて 32 ビットの固定長です。MIPS16 ISA モードの命令は、一部の例外を除き 16 ビットの固定長です。

870/C には、1 バイト命令から 5 バイト命令まであり、命令長は可変長です。900/L1 は最大で 7 バイト命令があります。命令長が可変長であることにはコードサイズが小さくなるという利点がある反面、命令のサイズが分からないため、命令デコーダは複雑になり、処理が遅くなる欠点もあります。

1.3.3　パイプライン処理

TX19A は 5 段パイプラインで命令を処理します。5 段パイプラインは、各命令の実行を 5 段に分けて、5 つの命令を同時に処理します。各ステージは 1 クロックで動作します。

TX19A の大きな特長は、ほとんどの命令は同じクロック数で実行されることです。そのため、TX19A は比較的パイプライン化するのが簡単で、1 命令当たりほぼ 1 クロックで実行できます。

CISC プロセッサのように異なる命令長で命令が構成されている場合、パイプラインの管理が複雑になります。また、コンパイラによるスケジューリングでも、パイプラインのストールをなくすことが難しくなります。

第2章　CPU アーキテクチャ

この章ではデータ形式、プログラミングモデル、ISA モード、コプロセッサ、パイプライン、メモリ管理など、TX19A のアーキテクチャの概要について説明します。

2.1　データ形式

この項ではレジスタおよびメモリにおけるデータの形について、オペランドの符号拡張、ゼロ拡張について説明します。

2.1.1　バイト順序

TX19A は 8 ビット、16 ビット、32 ビット、64 ビットのデータ形式をサポートしています。1 バイトは 8 ビット、ハーフワードは 2 バイト(16 ビット)、1 ワードは 4 バイト(32 ビット)、ダブルワードは 2 ワード(64 ビット)と定義されています。

データ形式がダブルワード、ワード、ハーフワードの場合、TX19A ではバイト順序として、ビッグエンディアン方式とリトルエンディアン方式をサポートしています。バイト順序はリセット時に入力端子 ENDIAN の状態により設定できます (派生品によっては、どちらかのエンディアンに固定されています)。

図 2-1 にビッグエンディアン方式とリトルエンディアン方式のバイト順序を示します。TX19Aはバイトアドレッシング方式を用いています。ビッグエンディアン方式では、最上位(左端)のバイトから下位のアドレスを割り当てられます。リトルエンディアン方式では、最下位(右端)のバイトから下位のアドレスが割り当てられます。リトルエンディアン方式では、同じ整数値であればデータ形式がハーフワード・ワードに関係なく、各アドレスには同じ値が入るという特徴があります。

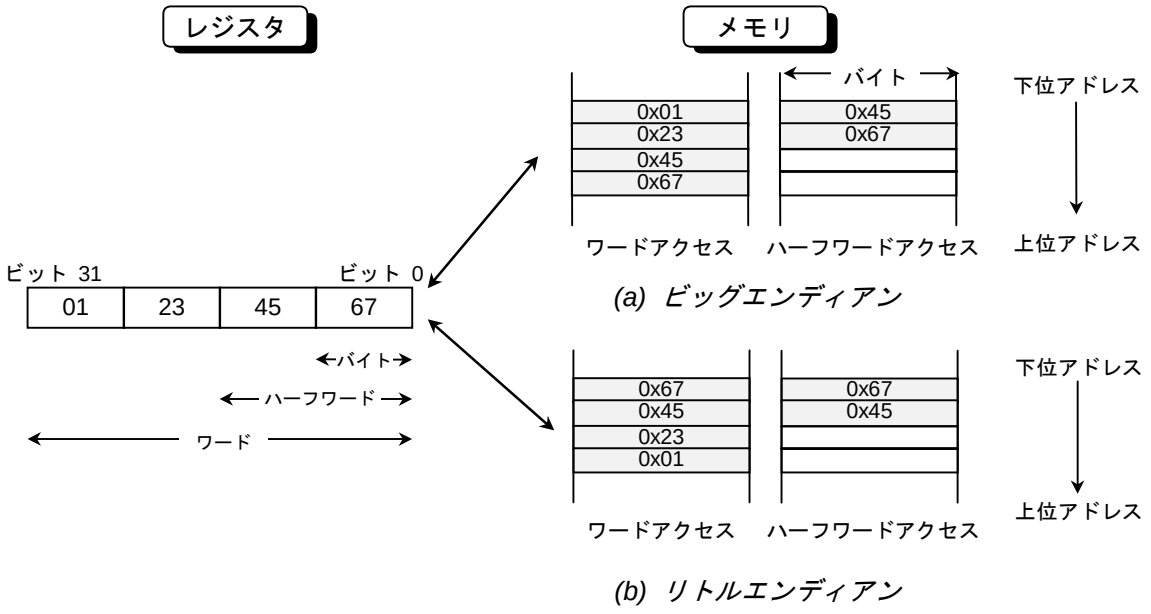
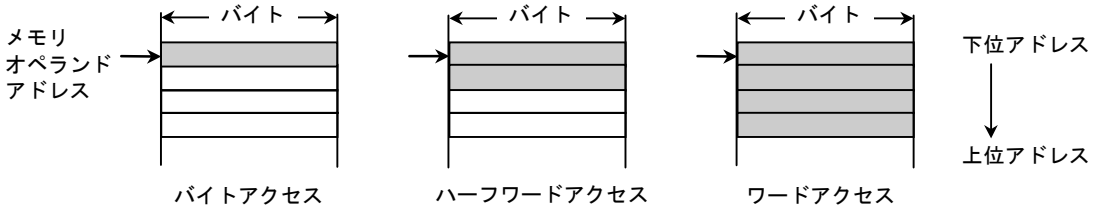


図 2-1　バイト順序

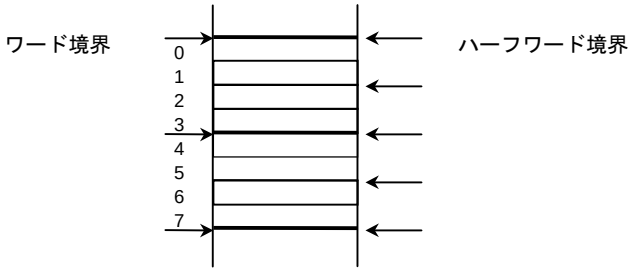
2.1.2　メモリアドレッシング

TX19A のメモリアクセスには、バイトアクセス、ハーフワードアクセス、ワードアクセスがあります。ハーフワード、ワードのアドレス指定には、データが格納されているメモリの最下位アドレスを使用します。このアドレスはビッグエンディアンの場合、データの最上位バイトのアドレス、リトルエンディアンの場合、データの最下位バイトのアドレスとなります。

メモリをアクセスする命令には、図 2-2(b)に示すように位置合わせ（アライメント)境界があります。ハーフワードアクセスの場合は、2 の倍数のバイト境界上に位置合わせしなければならず、ワードアクセスの場合は 4 の倍数のバイト境界上に位置合わせしなければなりません。



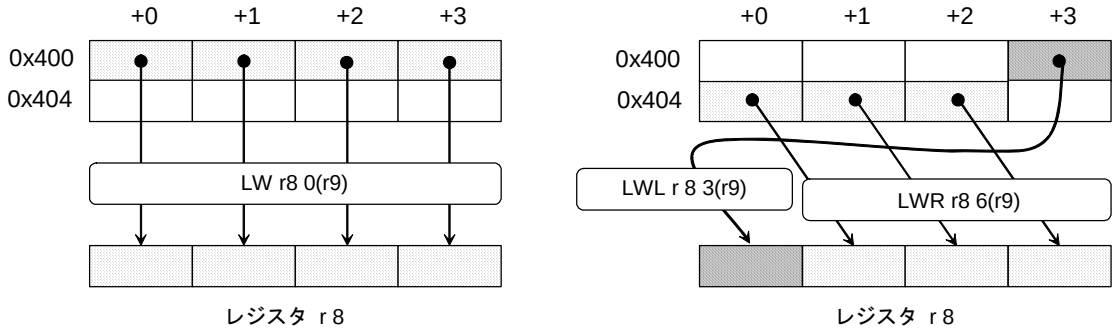
(a) メモリアクセス



(b) ワード境界・ハーフワード境界

図 2-2　位置合わせされたデータ

位置合わせは、処理速度に影響するので、ほとんどの命令で位置合わせが必要です。ワード境界に整列されていないデータのロード、ストアの場合、LWL、LWR、SWL、SWRという特殊な命令を使用する必要があります。このときLWLはLWRと、SWLはSWRとペアで使用します。図 2-3にワード境界に整列されているデータと整列されていないデータのロードの方法を示します。



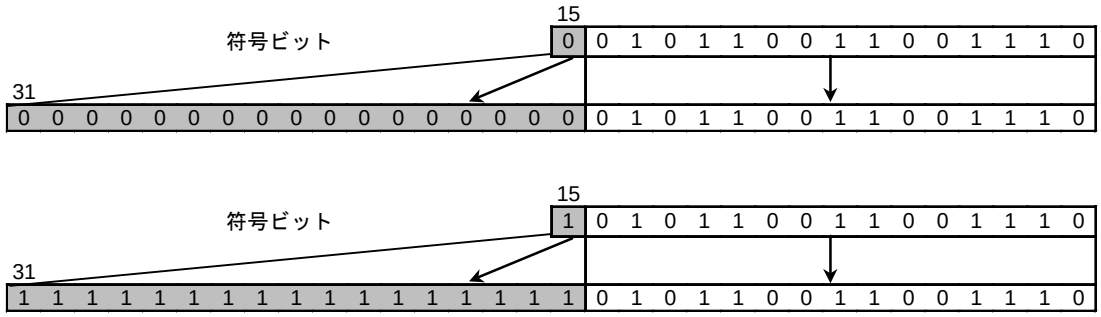
(a) 位置合わせされてるデータのロード（ビッグエンディアン方式）

(b) 位置合わせされていないデータのロード（ビッグエンディアン方式）

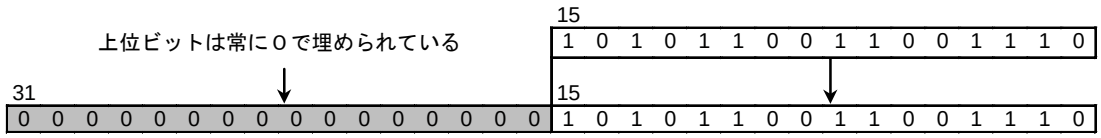
図 2-3　位置合わせされているデータとされていないデータのロード

2.1.3　データ拡張

図 2-4 に符号拡張とゼロ拡張を示します。符号付き数値では、最上位ビットは符号を表し、残りのビットは値の大きさを表します。符号拡張は、16 ビットの即値、またはロードしたバイト、またはハーフワードデータの最上位ビット(符号ビット)を上位のビットにコピーします。ゼロ拡張は、16 ビットの即値、またはロードしたバイト、またはハーフワードデータの最上位ビットの値に関係なく、上位のビットを 0 で埋めます。



(a) 16 ビットから 32 ビットへの符号拡張



(b) 16 ビットから 32 ビットへのゼロ拡張

図 2-4　符号拡張とゼロ拡張

符号拡張は、通常、算術演算で使われます。例えば、ADDI(Add Immediate Signed)命令は ADDI r3，r1，0x1234 のようにオペランドとして 16 ビットの即値をとることができます。この命令は、0x1234 を 32 ビットに符号拡張してからレジスタ r1 の内容に加算し、結果をレジスタ r3 に格納します。

また、符号拡張は LB (Load Byte)、LBU (Load Byte Unsigned)、LH (Load Halfword)、LHU (Load Halfword Unsigned)、LW (Load Word)、SB (Store Byte)、SH (Store Halfword)、SW (Store word) などのロード、ストア命令でも使われます。ロード、ストア命令でサポートされているアドレッシングモードは「ベースレジスタ＋16 ビットオフセット」のみで、LW r9，4(r8) のような書式になります。この命令は、オフセットの 4(0100)を符号拡張し、r8 レジスタに格納されているベースアドレスに加算することにより、実効アドレスを生成します。そして、実効アドレスにより指定されたワードデータを r9 レジスタにロードします。

バイトデータ、ハーフワードデータをレジスタにロードする場合、命令によって符号拡張、ゼロ拡張が選択されます。LB 命令、LH 命令はロードしたデータを符号拡張し、LBU 命令、LHU 命令はゼロ拡張してから、レジスタに格納します。

また、論理積命令と論理和命令には、AND・ANDI と OR・ORI があります。AND 命令と OR 命令はワードデータ同士の論理積・論理和をとる命令に対し、ANDI (AND Immediate)命令と ORI(OR Immediate)命令はワードデータとハーフワードデータの論理積・論理和をとる命令です。ANDI 命令と ORI 命令では 16 ビットの即値をゼロ拡張して、レジスタのビットごとに論理積、論理和をとります。

2.2　プログラミングモデル

TX19A のプログラミングモデルは、CPU レジスタとシステム制御コプロセッサ(CP0)レジスタの 2 つのレジスタグループで構成されています。

2.2.1　CPU レジスタ

図 2-5にCPUレジスタを示します。TX19Aには 256 本の汎用レジスタ (Shadow Register Set)、1 本のプログラムカウンタ(PC)レジスタ、整数の乗除算結果が格納される 2 本の特殊レジスタ(HI/LO)があります。CPUレジスタはすべて 32 ビットです。

(a) 汎用レジスタ

Shadow	0	1	2	3	4	5	6	7
Register	r0							
Set 番号	r26 (k0)							
	r27 (k1)							
	r28 (gp)							
	r29 (sp)	r29 (sp)						
	r1 (at)	r1 (at)	r1 (at)	r1 (at)	r1 (at)	r1 (at)	r1 (at)	r1 (at)
	r2 (v0)	r2 (v0)	r2 (v0)	r2 (v0)	r2 (v0)	r2 (v0)	r2 (v0)	r2 (v0)
	r3 (v1)	r3 (v1)	r3 (v1)	r3 (v1)	r3 (v1)	r3 (v1)	r3 (v1)	r3 (v1)
	r4 (a0)	r3 (a0)	r4 (a0)	r4 (a0)	r4 (a0)	r4 (a0)	r4 (a0)	r4 (a0)
	r5 (a1)	r5 (a1)	r5 (a1)	r5 (a1)	r5 (a1)	r5 (a1)	r5 (a1)	r5 (a1)
	r6 (a2)	r6 (a2)	r6 (a2)	r6 (a2)	r6 (a2)	r6 (a2)	r6 (a2)	r6 (a2)
	r7 (a3)	r7 (a3)	r7 (a3)	r7 (a3)	r7 (a3)	r7 (a3)	r7 (a3)	r7 (a3)
	r8 (t0)	r8 (t0)	r8 (t0)	r8 (t0)	r8 (t0)	r8 (t0)	r8 (t0)	r8 (t0)
	r9 (t1)	r9 (t1)	r9 (t1)	r9 (t1)	r9 (t1)	r9 (t1)	r9 (t1)	r9 (t1)
	r10 (t2)	r10 (t2)	r10 (t2)	r10 (t2)	r10 (t2)	r10 (t2)	r10 (t2)	r10 (t2)
	r11 (t3)	r11 (t3)	r11 (t3)	r11 (t3)	r11 (t3)	r11 (t3)	r11 (t3)	r11 (t3)
	r12 (t4)	r12 (t4)	r12 (t4)	r12 (t4)	r12 (t4)	r12 (t4)	r12 (t4)	r12 (t4)
	r13 (t5)	r13 (t5)	r13 (t5)	r13 (t5)	r13 (t5)	r13 (t5)	r13 (t5)	r13 (t5)
	r14 (t6)	r14 (t6)	r14 (t6)	r14 (t6)	r14 (t6)	r14 (t6)	r14 (t6)	r14 (t6)
	r15 (t7)	r15 (t7)	r15 (t7)	r15 (t7)	r15 (t7)	r15 (t7)	r15 (t7)	r15 (t7)
	r16 (s0)	r16 (s0)	r16 (s0)	r16 (s0)	r16 (s0)	r16 (s0)	r16 (s0)	r16 (s0)
	r17 (s1)	r17 (s1)	r17 (s1)	r17 (s1)	r17 (s1)	r17 (s1)	r17 (s1)	r17 (s1)
	r18 (s2)	r18 (s2)	r18 (s2)	r18 (s2)	r18 (s2)	r18 (s2)	r18 (s2)	r18 (s2)
	r19 (s3)	r19 (s3)	r19 (s3)	r19 (s3)	r19 (s3)	r19 (s3)	r19 (s3)	r19 (s3)
	r20 (s4)	r20 (s4)	r20 (s4)	r20 (s4)	r20 (s4)	r20 (s4)	r20 (s4)	r20 (s4)
	r21 (s5)	r21 (s5)	r21 (s5)	r21 (s5)	r21 (s5)	r21 (s5)	r21 (s5)	r21 (s5)
	r22 (s6)	r22 (s6)	r22 (s6)	r22 (s6)	r22 (s6)	r22 (s6)	r22 (s6)	r22 (s6)
	r23 (s7)	r23 (s7)	r23 (s7)	r23 (s7)	r23 (s7)	r23 (s7)	r23 (s7)	r23 (s7)
	r24 (t8)	r24 (t8)	r24 (t8)	r24 (t8)	r24 (t8)	r24 (t8)	r24 (t8)	r24 (t8)
	r25 (t9)	r25 (t9)	r25 (t9)	r25 (t9)	r25 (t9)	r25 (t9)	r25 (t9)	r25 (t9)
	r30 (fp)	r30 (fp)	r30 (fp)	r30 (fp)	r30 (fp)	r30 (fp)	r30 (fp)	r30 (fp)
	r31 (ra)	r31 (ra)	r31 (ra)	r31 (ra)	r31 (ra)	r31 (ra)	r31 (ra)	r31 (ra)

(b) 乗算･除算レジスタ

HI
LO

(c) プログラムカウンタ

PC

図 2-5　CPU レジスタ

■　汎用レジスタ

TX19A コアは、Shadow Register Set と呼ばれるレジスタバンク構成 (8 バンク) となっています。この Shadow Register Set は、割り込みの応答、命令 (MTC0) によって切り替えることが可能です。レジスタの r0、r26、r27、r28 はすべてのバンクに共通で、r29 は Shadow Register Set 番号 1~7 まで共通のレジスタとなります。その他のレジスタは、それぞれのバンクごとに用意されています。

32 ビットISA命令では、図 2-5に示す汎用レジスタをすべて使用できます。レジスタにはr0～r31 まで番号がつけられています。r0 以外の汎用レジスタは、v0～v1、a0～a3 などのようにアセンブラで使用できるシンボル名(ソフトウエア名)をもっています。32 ビットISA命令では、r0 とr31 を除いて、汎用レジスタはすべて同じように扱われます。r0 はハードウエア的に常に値が 0 に固定されています。そのため、r0 は演算結果を破棄したい場合のターゲットレジスタや、値として 0 が必要な場合のソースレジスタとして使用します。r31(ra: リターンアドレス)はリンク機能付きのジャンプ命令、分岐命令、分岐ライクリ命令で使われるリンクレジスタです。これらの命令は、サブルーチンからのリターンアドレスをr31 に格納します。

16 ビット命令では、基本的に、図 2-5に示す汎用レジスタのうちr2～r7、r16、r17 の 8 本のレジスタしかアクセスできません。ただし、プロセッサには 32 ビットISA モードで使用する 32 本のレジスタが存在するため、16 ビットISAでアクセスできる 8 本のレジスタと残りの 24 本のレジスタ間で値を移送するためのmove命令が 16 ビットISAに用意されています。また、特定の命令では、暗黙的にr24 (t8)、r28 (gp)、r29 (sp)、r30 (fp)、r31 (ra)を使用します。r24 は比較結果が格納されるレジスタとして、r28 はグローバルポインタとして、r29 はスタックポインタとして、r30 はフレームポインタとして、r31 はリンクレジスタとして用いられます。

■ **(注意) レジスタ r1 はアセンブラの予約レジスタになっています。プログラム中では r1 を使用しないでください。**

HI レジスタ･LO レジスタ

HI レジスタと LO レジスタは整数の乗除算、積和/積差の演算結果が格納されるレジスタです。整数の乗算、積和/積差では、ダブルワード(64 ビット)の結果の上位 32 ビットが HI レジスタに、下位 32 ビットが LO レジスタに格納されます。整数の除算では、商が LO レジスタに、剰余が HI レジスタに格納されます。HI レジスタ、LO レジスタと汎用レジスタのあいだのデータの移送には、MFHI、MFLO、MTHI、MTLO 命令を使います。

■　プログラムカウンタ(PC)

プログラムカウンタの最下位ビットは、ISA モードビットで、0 は 32 ビット ISA モードを、1 は 16 ビット ISA モードを示します。最下位ビットはアドレスの一部とは見なされず、最下位ビットを 0 にクリアしたときの 32 ビット全体の値が現在実行中の命令のアドレスを表します。

2.2.2　システム制御コプロセッサ(CP0)レジスタ

TX19Aには、システム制御コプロセッサCP0 が内蔵されています。CP0 には、図 2-6に示すユーザーがアクセスできる 17 本のレジスタがあります。

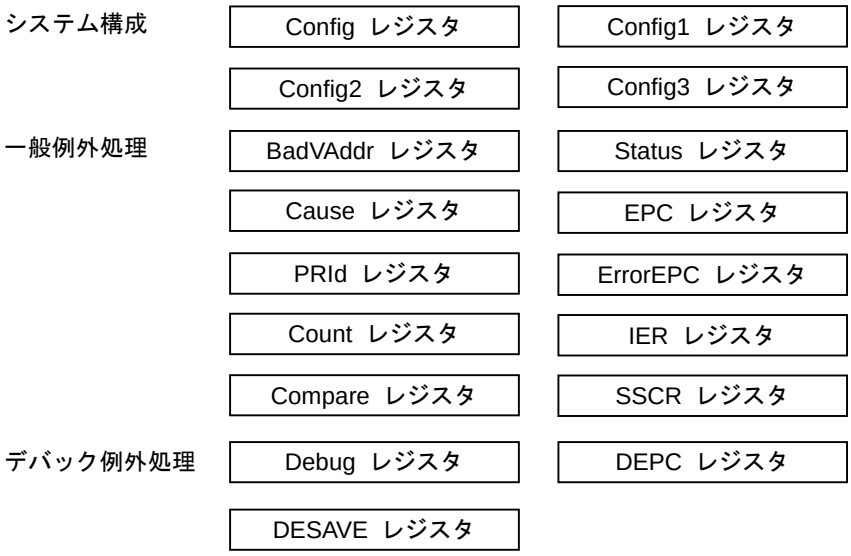


図 2-6　システム制御コプロセッサ (CP0) レジスタ

CP0 レジスタは、システム構成レジスタ、一般例外処理レジスタ、デバック例外処理レジスタの 3 種類に分類されます。プロセッサがカーネルモードのときは、システム制御、コプロセッサCP0 命令によって、CP0 レジスタに常にアクセスできます。また、プロセッサがユーザーモードのときは、StatusレジスタのCU0 ビットが 1 のときのみ、CP0 レジスタにアクセスできます。プロセッサの動作モードについては「2.7 メモリ管理」で説明します。

表 2-1　システム構成レジスタ

レジスタ名	機能説明
Config、Config1、Config2、Config3	EJTAG、16 ビット ISA モード、キャッシュ実装などのシステム構成の設定をします。

表 2-2　一般例外処理レジスタ

レジスタ名	機能説明
BadVAddr	仮想アドレスから物理アドレスへの変換で、エラーを起こした仮想アドレスを保持します。読み出し専用です。
Status	動作モード(ユーザーモード･カーネルモード)、割り込みイネーブル状態などのプロセッサの状態を保持します。
Cause	直前に発生した例外の原因を保持します。
EPC	例外プログラムカウンタ。例外を発生した命令のアドレス、ISA モードを保持します。
ErrorEPC	例外プログラムカウンタ。リセット、NMI 例外を発生した命令のアドレス、ISA モードを保持します。
Count	CPUCLK の 1/2 速度で 1 ずつインクリメントされるカウントアップレジスタです。
Compare	カウントアップレジスタとの比較用データを格納するレジスタです。
PRId	TX19A プロセッサコアのリビジョンを示します。読み出し専用です。
IER	Status レジスタの割り込み許可・禁止ビットを操作するレジスタです。
SSCR	Shadow Register Set の使用 Set の設定

表 2-3　デバック例外処理レジスタ

レジスタ名	機能説明
Debug	デバック例外の原因とデバック時の状態を保持します。
DEPC	デバック例外プログラムカウンタ。デバック例外からのリターンアドレス、ISA モードを保持します。
DESAVE	Context スイッチのときに、一時的にレジスタを退避するために使われます。

2.3　32 ビット ISA モード・16 ビット ISA モード

TX19A には、16 ビット ISA と 32 ビット ISA の 2 つの命令セット（ISA）があります。16 ビット ISA のプログラムを実行している状態を 16 ビット ISA モード、32 ビット ISA のプログラムを実行している状態を 32 ビット ISA モードと呼びます。16 ビット ISA モードと 32 ビット ISA モードは、プログラムの実行中にサブルーチン単位で、命令により切り換えられます。一般的に、コードサイズを縮小したい部分は、16 ビット ISA でプログラムを生成し、高速化したい部分は 32 ビット ISA を用いてプログラムを生成します。

プログラムカウンタ(PC)の最下位ビットが ISA モードビットで、0 のときは 32 ビット ISA モードになり、1 のときは 16 ビット ISA モードになります。32 ビット ISA モードと 16 ビット ISA モードは JALX、JR、JALR、JRC、JALRC 命令により切り換えられます。

16 ビット ISA モード中で例外が発生すると、プロセッサは自動的に 32 ビット ISA モードに切り換わります。このとき、リターンアドレスと ISA モードビットが、例外プログラムカウンタ(EPC、ErrorEPC)またはデバック例外プログラムカウンタ(DEPC)に保存されます。EPC あるいは ErrorEPC レジスタに保存されているリターンアドレスへ戻るには、ERET 命令を使います。また、デバック例外の場合は、DERET 命令により DEPC レジスタに保存されているリターンアドレスへ戻ります。

32 ビットISA、16 ビットISAにはそれぞれ 図 2-7に示す命令があります。

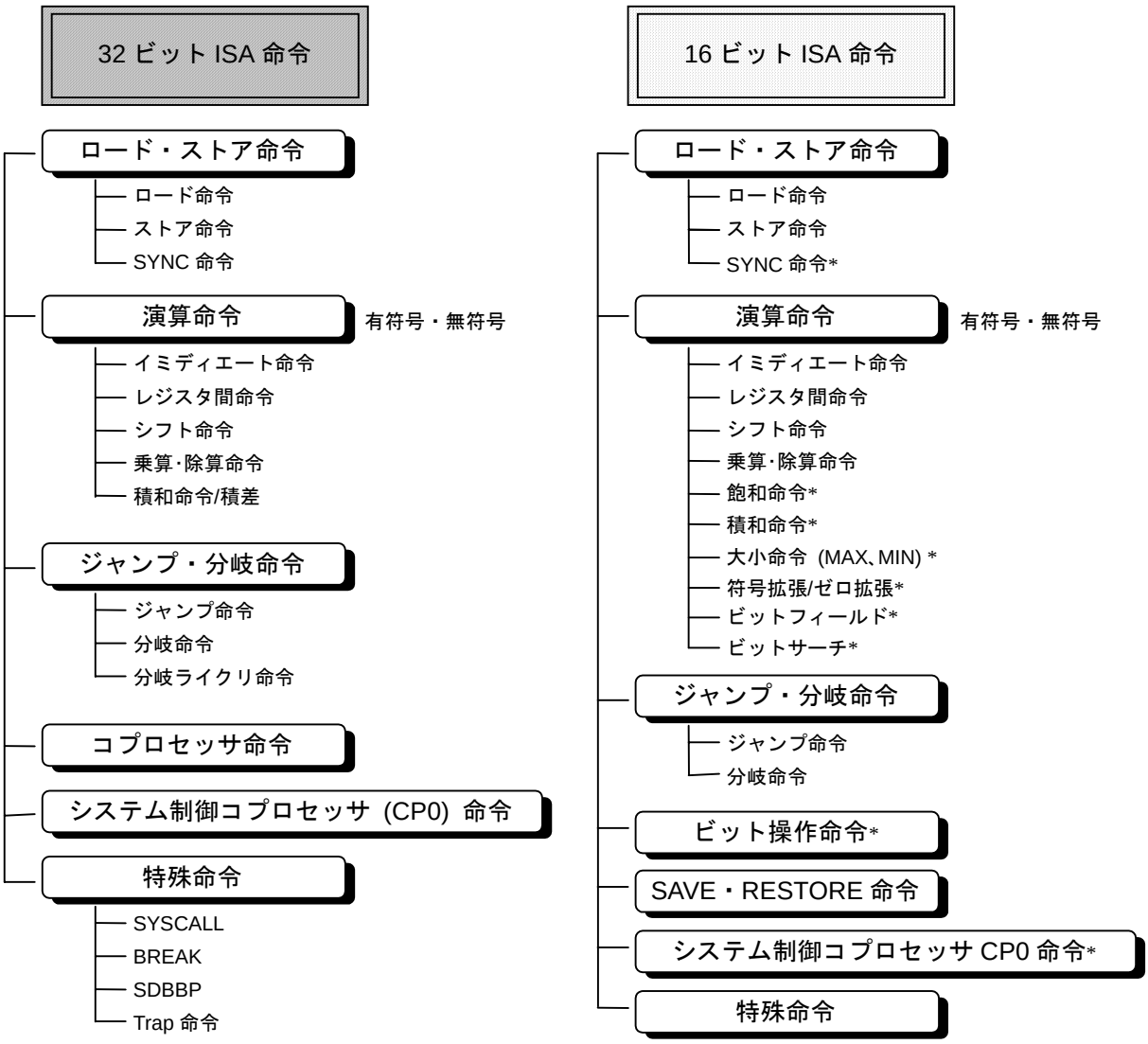


図 2-7　32 ビット ISA と 16 ビット ISA の命令

32 ビット ISA の命令長は、すべて 32 ビットです。16 ビット ISA の命令長は原則として 16 ビットですが、EXTEND 命令を連結することによって命令長を 32 ビットにすることができます。EXTEND 命令自体は 16 ビットで、5 ビットのオペコードと 11 ビットの即値のみで構成されますが、一部の命令では 11 ビットの即値の部分をオペコードとして使用する場合もあります。EXTEND 命令は、それだけでは機械

語の命令を生成しませんが、自身の即値を後続命令の即値に連結することにより、16 ビットの即値を使えるようにします。命令長が３２ビットの 16 ビット ISA 命令を拡張命令と呼びます。SYNC、ERET、DERET、WAIT、BS1F、MAX、MIN 命令は拡張命令しかありません。

2.4　コプロセッサ

コプロセッサとは、CPU の負荷を減らすことにより、処理スピードを上げるためのユニットです。

CP0 はシステム制御コプロセッサで、CP0 は TX19A に内蔵されています。システム構成、例外処理、メモリ管理などを行います。CP0 の基本機能はプロセッサコアに、拡張機能はメモリ管理ユニット(MMU)に内蔵されています。

ユーザーモードでは、CP0 命令および CP0 レジスタを使用できるかどうかは、Status レジスタの CU0 ビットにより制御します。CU0 ビットがクリアされているときに、ユーザーモードのプログラムで CP0 命令を実行しようとすると、コプロセッサ使用不可例外が発生します。カーネルモード、デバッグモードでは、CU0 ビットの設定に関係なく、すべての CP0 命令が実行できます。

Status レジスタの CU[3:1]ビットはユーザーモード、カーネルモードでの各コプロセッサの使用可能、不可能状態を制御します。対応する CU ビットがクリアされているときに、コプロセッサ命令を実行すると、コプロセッサ使用不可例外が発生します。

システム制御コプロセッサ(CP0)には、ユーザーがアクセスできる 17 本のレジスタがあります。各レジスタの機能については 8 章で詳しく説明します。

2.5　パイプライン

TX19Aは 5 段パイプラインを内蔵しています。各命令は 5 つのステージに分けて実行されます。それぞれのステージは、ほぼ 1 クロックで実行されるため、1 命令の実行は最短で 5 クロックかかります（16 ビットISAのJAL、JALX命令はそれ以上のサイクルが必要です)。パイプラインは、各命令の実行をフェッチ(F)、デコード(D)、実行(E)、メモリアクセス(M)、レジスタライトバック(W)の 5 つの部分に分割し、図 2-8に示すように最大 5 つの命令を同時に実行します。したがって、1 命令当たり、ほぼ 1 クロックで実行できます。

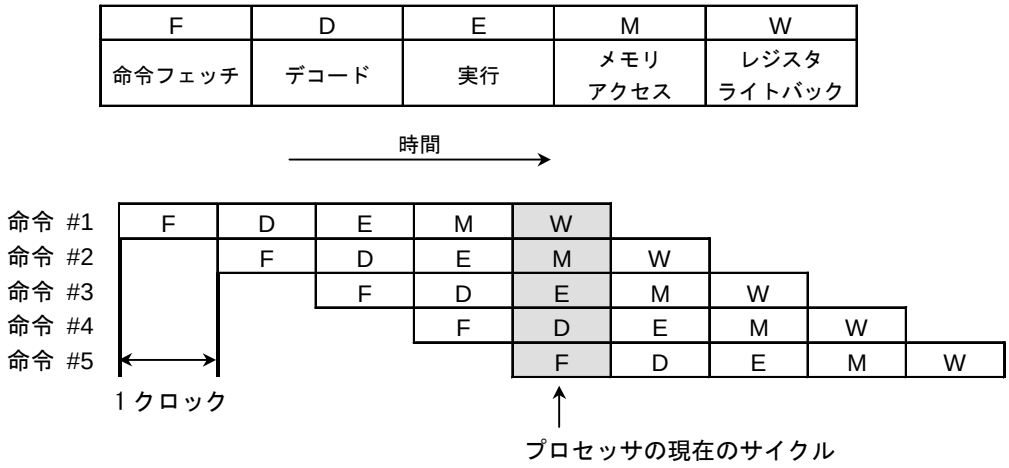


図 2-8　5 段パイプライン

2.6　ライトバッファ

ライトバッファは 4 エントリの FIFO バッファです。処理している命令が内蔵メモリ以外への書き込みを必要とする場合、前章で述べたように各パイプラインステージは原則として 1 クロックで実行されるのに対して、内蔵メモリ以外へのライトバスサイクルは必ずしも 1 クロックでは終了しません。ライトバッファはこのとき発生するスピードの差を吸収し、プログラム実行時のパフォーマンスを改善する効果があります。

2.6.1　ライトバッファを使用する命令

ライトバッファを使用する命令は、メモリへのライトバスサイクルを発生する命令です。具体的には以下の命令群になります。

- ストア命令の全て
  32ISA:　SW / SH / SB / SWL / SWR
  16ISA:　SW / SH / SB

- ビット演算命令の一部、メモリオペランド加算
  32ISA:　該当する命令はありません
  16ISA:　BCLR / BSET / BINS

- その他
  32ISA:　該当する命令はありません
  16ISA:　ADDMIU / SAVE

(注意) それぞれの命令の詳細については「付録 A　32 ビット ISA の詳細」、「付録 B 16 ビット ISA の詳細」を参照してください。

2.6.2　ライトバッファを使用する命令の実行手順

ライトバッファを使用する命令を実行すると、命令の実行のために必要なバス動作がライトバッファに登録されます。これをここでは「ライトバッファにエントリされる」といいます。ライトバッファへのエントリは命令実行順に実行されます。

ライトバッファを使用する命令が実行ステージ(E ステージ)にあるとき、ライトバッファに空きがあれば、命令のバスオペレーションがライトバッファにエントリされます。その際にオペランドバスで実行しているバスサイクルがなければ(オペランドバスが空いていれば)、ライトバッファにエントリするのと同時にバスサイクルを開始します。ライトバッファに空きがなければ、空きができるまで命令は E ステージでストールします。

ライトバッファにエントリされたオペレーションは、オペランドバスに空きがあればエントリされた順番に実行されます。ライトバッファ上で実行順序が変更されることはありません。オペランドバスに空きが無い場合は、空きができるまでライトバスサイクルの実行は待たされます。このとき後続の命令(ロード命令など)がリードバスサイクルを要求した場合、ライトバッファにエントリされたオペレーションが全て終了するまで命令は E ステージでストールします。

図 2-9にライトバッファを使用する命令の処理手順の例を示します。この図では3番目の命令がロード命令のため、ロードによるリードバスサイクルはライトバッファ中のオペレーションによるライトサイクルが終了するまで実行されません。

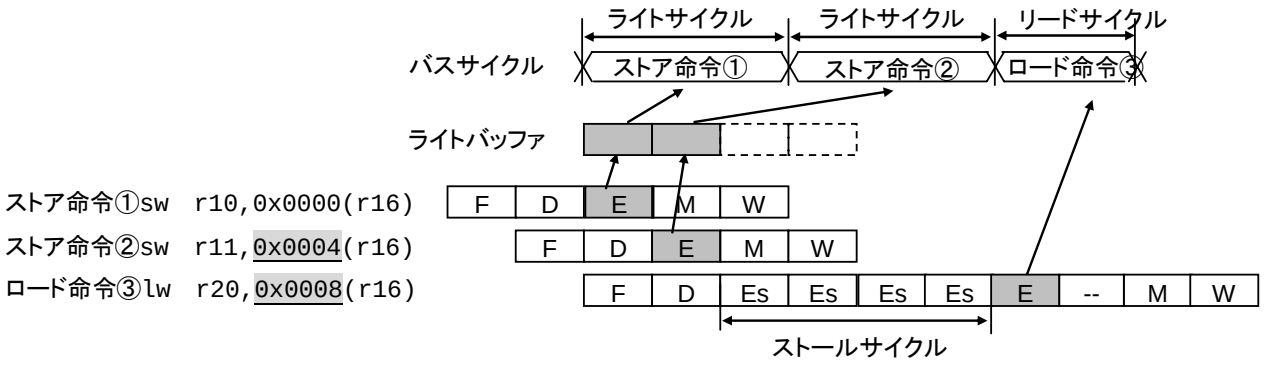


図 2-9　ライトバッファを使用する命令の処理手順の例

2.6.3　ビット演算命令・ADDMIU 命令

ビット演算命令、ADDMIU 命令のようにオペランドリードを伴うものは、オペランドリードバスサイクルを開始します。これらの命令はリードモディファイライト動作としてリードバスサイクルとライトバスサイクルは不可分である必要があるため、必ず連続して実行されます。

このとき、ビット演算のライトサイクルは後続の命令よりも必ず優先して実行されます。

図 2－10 にビット演算命令処理手順の例を示します。

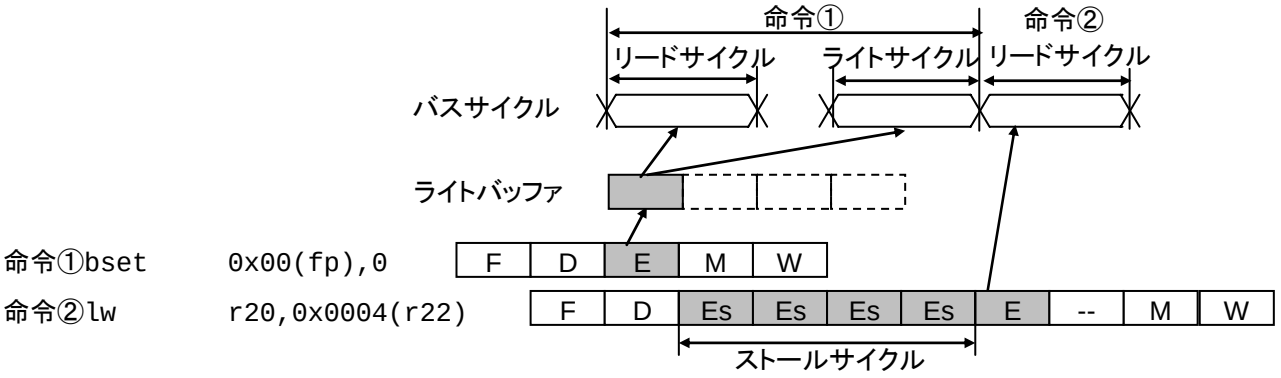


図 2-10　ビット演算命令処理手順の例

2.6.4　SAVE 命令

SAVE 命令は 1 命令で複数のストアを発生する命令です。ライトバッファには SAVE 命令が発生するストアの順にエントリされます。その間、SAVE 命令は実行ステージを占拠するので、他の命令に起因する動作がライトバッファにエントリされることはありません。

2.6.5　SYNC 命令

SYNC 命令を実行すると、メモリデータのコンシステンシを保つ為にライトバッファにエントリされたライトバスオペレーションはすべて実行されます。SYNC 命令はエントリされたオペレーションによるバスサイクルが全て終了するまでストールするので、メモリや IO の状態と命令実行の間で同期を取るために使用することができます。

なお、割り込み／例外発生時、バス権開放時に、ライトバッファの内容が自動的にフラッシュされることはありません。必要に応じて SYNC 命令などでコンシステンシを保つ必要があります。

2.7　メモリ管理

TX19A にはユーザーモードとカーネルモードの 2 種類の動作モードがあります。TX19A は例外が発生すると、自動的にカーネルモードに移行します。また、システムリセットがかかると、プロセッサはリセット例外により立ち上がるため、立ち上げ時はカーネルモードになります。図　2－11 に示すとおり、カーネルモードからユーザーモードへの移行は ERET（Exception Return）命令または、DERET（Debug Exception Return）命令により行います。

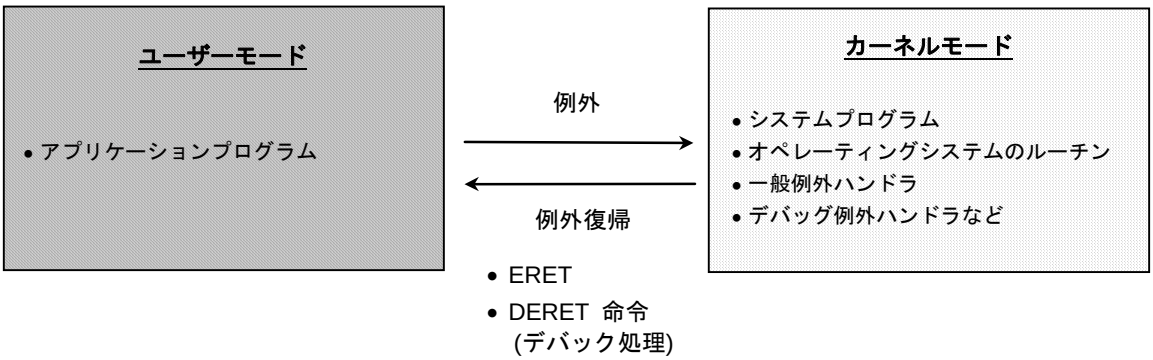


図 2-11　動作モード

動作モードによりプログラムで使用できる仮想アドレス空間、レジスタ、命令が異なります。カーネルモードはユーザーモードより高い特権レベルが与えられ、すべての仮想アドレス空間、レジスタ、命令を使用できます。ユーザーモードではこれらの使用が制限されます。オペレーティングシステムのルーチン、例外ハンドラ、デバックハンドラなどのプログラムは、カーネルモードで実行します。これによりカーネルモードでのみ使用できるアドレス空間に、ユーザーモードからアクセスできないようにシステムを保護できます。

(注意) TX19A では、カーネルモードのみ使用してください。

TX19A のメモリ管理ユニット(MMU)はダイレクトセグメントマッピング方式を使っており、TLB を内蔵していません。仮想アドレスから物理アドレスへのマッピングを図　2－12 に示します。仮想アドレス空間は、4 つの領域に分けられています。kuseg はカーネルモード、またはユーザーモードのどちらかでもアクセスできる領域です。他の 3 つの領域 kseg0、kseg1、kseg2 は、カーネルモードでのみ使用できます。メモリ管理については 6 章で詳しく説明します。

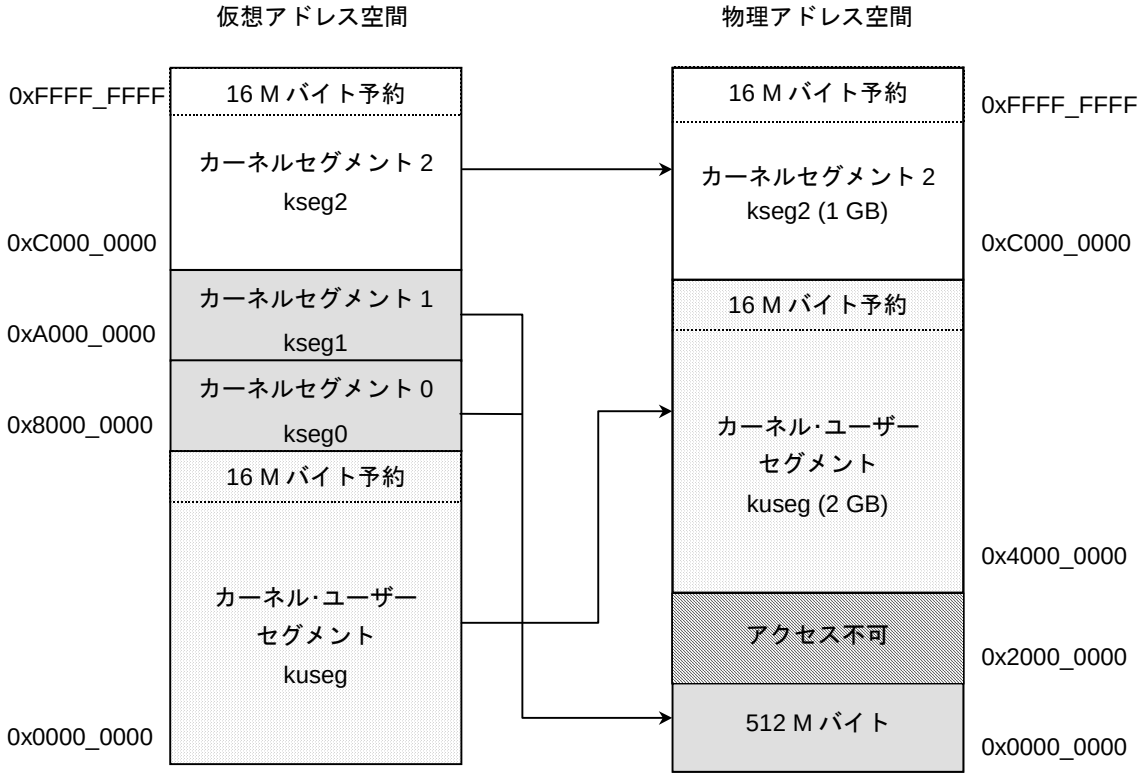


図 2-12　ダイレクトセグメントマッピング方式

第3章　32 ビット ISA の概要

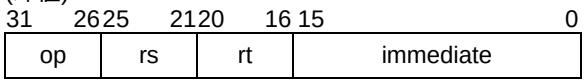
この章では、32 ビット ISA の命令とアドレッシングモードの概要を説明します。また、32 ビット命令を使った基本的なプログラミングについても説明します。
32 ビット ISA の命令は以下のように分類されます。

- ロード命令・ストア命令
- 演算命令
- ジャンプ命令・分岐命令・分岐ライクリ命令
- システム制御コプロセッサ (CP0) 命令
- 特殊命令

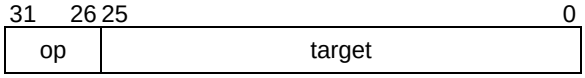
3.1　命令形式

32 ビットISAの命令は、すべて 32 ビット長で、図 3-1に示す 3 種類の命令形式があります。命令形式を 3 種類に限定することにより、命令のデコードを簡素化しています。複雑で使用頻度の低いオペレーション命令は、コンパイラにより複数の命令を組み合わせることにより実現します。32 ビット命令はすべてワード境界で位置合わせしなければなりません。

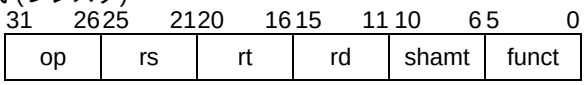
I 形式 (即値)



J 形式 (ジャンプ)



R 形式 (レジスタ)



op	6 ビットの命令コード
rs	5 ビットのソースレジスタ番号
rt	5 ビットのターゲットレジスタ番号または分岐条件
immediate	16 ビットの即値、分岐オフセット値 またはアドレスのオフセット値
target	26 ビットのジャンプターゲットアドレス
rd	5 ビットのデスティネーションレジスタ番号
shamt	5 ビットのシフト量
funct	6 ビットの機能コード

図 3-1　命令形式

3.2　ロード・ストア命令

ロード・ストア命令は、メモリと CPU 汎用レジスタ間でデータを移送するのに使います。ロード・ストア命令はメモリからレジスタへデータをロードするか、またはレジスタからメモリへデータをストアするだけです。レジスタとメモリの内容に対して、算術・論理演算を実行する命令はありません。

3.2.1　アドレッシングモード

32 ビットISAのロード・ストア命令は、すべてI形式の命令です。ロード・ストア命令は、図 3-2に示すオフセット付きレジスタ間接アドレッシングモードを使います。実効アドレスは、ベースレジスタとして指定した汎用レジスタの値に 16 ビットの即値を符号拡張した値を加算することにより、計算されます。以下に例を示します。

```
LW r9,4(r8)
```

この命令では、4 (0100) がオフセットで、r8 はベースアドレスが格納されている汎用レジスタ、r9 はロード先のターゲットレジスタを表します。

オフセット付きレジスタ間接アドレッシングモードでは、r0 をベースレジスタに指定すると、即値アドレッシングモードと同じになり、オフセット 0 を指定すると、レジスタ直接アドレッシングモードと同じになります。

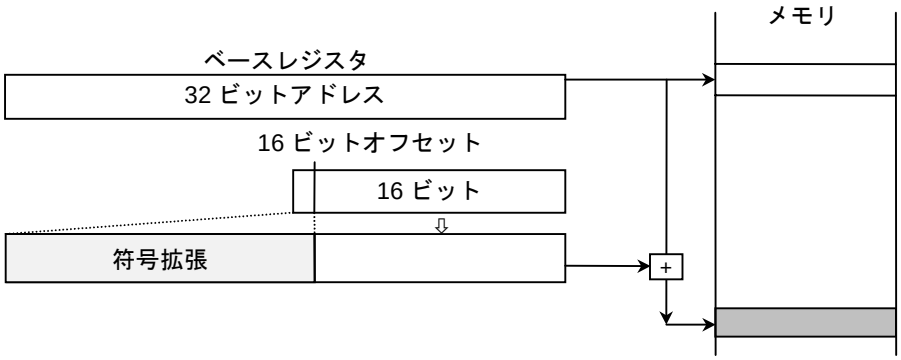


図 3-2　オフセット付きレジスタ間接アドレッシングモード

3.2.2　位置合わせされているデータのロード・ストア

表 3-1に、バイトアクセス、ハーフワードアクセス、ワードアクセス用のロード・ストア命令を示します。LB命令とLH命令では、ロードしたバイトまたはハーフワードは符号拡張されて、レジスタに格納されます。それに対して、接尾辞 (unsigned) が付いたLBU命令とLHU命令では、ロードしたバイトまたはハーフワードはゼロ拡張されて、レジスタに格納されます。

表 3-1　位置合わせされているデータのロード・ストア

データ形式	符号なしロード	符号付きロード	ストア
バイト	LBU	LB	SB
ハーフワード	LHU	LH	SH
ワード	LW	—	SW

3.2.3　位置合わせされていないデータのロード・ストア

前項に示したロード・ストア命令を使って、ハーフワード境界に位置合わせされていないハーフワードや、ワード境界に位置合わせされていないワードをロードまたはストア命令を実行すると、アドレスエラー例外が発生します。位置合わせされていないワードのロード・ストアについては、表 3-2に示す特別な命令が用意されています。LWL (Load Word Left) 命令とLWR (Load Word Right) 命令はペアで、SWL (Store Word Left) 命令とSWR (Store Word Right) 命令はペアで使います。これらの命令は、位置合わせされていないワードで、ロード・ストア命令とシフト命令を組み合わせて使うより効率的です。8 ビット、16 ビットCPU用に作成した古いプログラムを再利用するのに便利です。

表 3-2　位置合わせされていないデータのロード・ストア

データ形式	符号なしロード	ストア
左部分 (上位バイト)	LWL	SWL
右部分 (下位バイト)	LWR	SWR

3.2.4　SYNC 命令

SYNC 命令は、SYNC 命令の直前に実行したロード、ストア、命令フェッチまで、命令パイプラインをインタロックし後続のロード、ストアの実行を遅らせます。これにより、SYNC 命令の前の命令と後続の命令の実行順序を守ることができます。

3.2.5　32 ビットのアドレスの生成

32 ビット ISA のロード・ストア命令は、オフセットとして 16 ビットの符号付き即値しかとることができません。最上位ビットは符号ビットで、残りの 15 ビットでオフセットの大きさを指定します。したがって、オフセットの範囲は、–32768 から +32767 となります。オフセットがこの範囲外の場合は、オフセットをいったん汎用レジスタに格納する必要があります。3 つの例を以下に示します。

◆　例 1　ベースアドレス + 32 ビットオフセット
以下の例では、ADDU (Add Unsigned) 命令によりレジスタ r5 に格納されているオフセットをレジスタ r4 のベースアドレスに加算し、その結果を r4 に書き戻しています。次に LW 命令で、r4 をベースレジスタとして指定しています。

```
ADDU    r4,r4,r5
LW      r6,0(r4)
```

◆ 例 2　ベースアドレス + 32 ビットオフセット

以下の例では、LUI (Load Upper Immediate) 命令を使って、指定された 16 ビットの即値をレジスタ r5 の上位 16 ビットにロードしています。下位 16 ビットはゼロで埋められます。次の ADDU (Add Unsigned) 命令で r4 のベースアドレスに r5 を加算しています。つまり、ベースアドレスにオフセットの上位の 16 ビットが加算されたことになります。次に、LW 命令にて、r4 にオフセットの下位 16 ビットを加算することにより、最終的なターゲットアドレスを生成しています。

```
LUI     r5,0x12
ADDU    r4,r4,r5
LW      r6,0x3454(r4)
```

◆ 例 3　任意の 32 ビットのアドレス

以下の例で、LUI (Load upper Immediate) 命令を使って、16 ビットの即値をレジスタ r4 の上位 16 ビットにロードしています。次に ADDIU (Add Immediate Unsigned) 命令でオフセットの下位 16 ビット 0x3456 を r4 に加算しています。これで、r4 に 32 ビットのオフセットが格納されました。したがって、LW 命令では、r4 をベースレジスタとして使えば、オフセットをゼロとすることができます。

```
LUI     r4,0x12
ADDIU   r4,r4,0x3456
LW      r6,0(r4)
```

3.3　演算命令

この項では、32 ビットISAの演算命令について説明します。

3.3.1項では、演算命令の分類を示します。

3.3.2項では、32 ビットの定数を使用する演算について説明します。

3.3.3項では、64 ビットの加減算をどのように実行するか、例を使って説明します。

3.3.4項では、例外を使用せず整数演算のオーバフローを検出する方法を示します。

3.3.5項では、64 ビット × 64 ビットの乗算を実行する方法について説明します。

3.3.6項では、ローテートの実現方法について説明します。

3.3.1　演算命令の分類

32 ビットISAの演算命令は、表 3-3に示す 5 つのグループに分類されます。演算命令は、算術、比較、論理、シフト、乗算、除算、積和演算命令があります。演算命令のオペランドは、16 ビットの即値をとるI形式か、2 つまたは 3 つのレジスタを指定するR形式になります。

表 3-3　演算命令

分類	命令	オペコード
ALU 即値	加算	ADDI・ADDIU
	大小比較	SLTI・SLTIU
	論理積	ANDI
	論理和	ORI
	排他的論理和	XORI
	上位即値のロード	LUI
2/3 オペランドレジスタタイプ	加算	ADD・ADDU
	減算	SUB・SUBU
	大小比較	SLT・SLTU
	論理積	AND
	論理和	OR
	排他的論理和	XOR
	否定論理和	NOR
	カウント	CLO・CLZ
	条件付き移動	MOVN・MOVZ
シフト	論理シフト	SLL・SLLV・SRL・SRLV
	算術シフト	SRA・SRAV
乗算・除算	乗算	MULT・MULTU・MUL
	除算	DIV・DIVU
	HI・LO レジスタと汎用レジスタ間の転送	MFHI・MFLO・MTHI・MTLO
積和/積差		MADD・MADDU・MSUB・MSUBU

ALU 即値命令では、ソースオペランドは汎用レジスタと 16 ビットの符号付き即値です。例えば、ADDI (Add Immediate) 命令のシンタックスは「ADDI *rd*, *rs*, *immediate*」で、ソースレジスタ (*rs*) と符号拡張した即値 (*immediate*) を加算し、その結果をデスティネーションレジスタ (*rd*) に格納します。

2/3 オペランドレジスタタイプの命令は、汎用レジスタに格納されている 2 つの値を対象に演算を実行し、その結果を汎用レジスタに格納します。

シフト命令は、汎用レジスタの内容を指定されたビット数だけ左または右へずらします。シフト命令には、論理シフト命令と算術シフト命令の 2 種類があります。また、シフト変数命令 (SLLV、SRLV、SRAV) には、シフト量を指定するフィールド (*shamt*) がなく、代わりにシフト量が格納されている汎用レジスタを指定します。

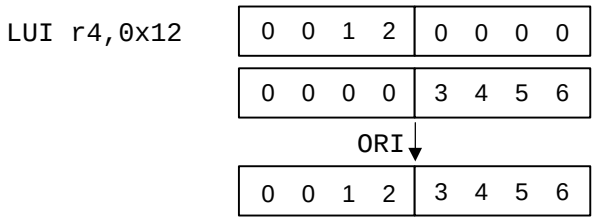
乗除算命令は、汎用レジスタに格納されている 2 つの整数値を対象に乗算または除算を行い、その結果を特殊レジスタのHIレジスタとLOレジスタに格納します。一般の命令では、HIレジスタ・LOレジスタにはアクセスできません。汎用レジスタとHIレジスタ・LOレジスタ間でデータを移送するには、MFHI、MFLO、MTHI、MTLO命令を使います。TX19Aでは、MIPSの命令を当社で拡張しており、乗算の場合、積の下位32ビットをLOレジスタと汎用レジスタの両方に同時に格納できます。「3.3.5 64 ビット × 64 ビットの乗算」で、使用例を示します。

積和命令は、32 ビットの 2 つの整数を掛け合わせ、HO・LO レジスタの 64 ビットの値に加算します。また、同時に結果の下位 32 ビットを汎用レジスタに格納することもできます。積和演算は、デジタル信号処理 (DSP) で頻繁に使用されるため、専用の積和演算器 (MAC) で高速に実行されます。

3.3.2　32 ビットの定数

I 形式の命令では、即値フィールドは 16 ビットしかありません。即値が 16 ビットより大きい場合は、32 ビットの定数を生成し、それを一時的に汎用レジスタに格納しておく必要があります。以下の例では、LUI (Load Upper Immediate) 命令にて指定された即値をレジスタ r4 の上位 16 ビットに格納し、下位 16 ビットを 0 で埋めます。そして、次の ORI (OR Immediate) 命令で、r4 の内容と ORI 命令自体の即値の論理和をとり、結果を r4 に書き戻しています。ORI 命令の即値はゼロ拡張されます。

```
LUI     r4,0x12
ORI     r4,r4,0x3456
```



次に、汎用レジスタの内容に 32 ビットの定数を加算する例を示します。この例では、LUI 命令で r5 の上位 16 ビットに 0x1234 を格納し、それに、ADDIU (Add Immediate Unsigned) 命令により 0x5678 を加算して、32 ビットの定数 0x12345678 を得ています。最後に ADDU (Add Unsigned) 命令により r4 と r5 を加算し、その結果を r6 に格納しています。

```
LUI     r5,0x1234
ADDIU   r5,r5,0x5678
ADDU    r6,r4,r5
```

注意: ADDI 命令、SLTI 命令では、即値は 32 ビットに符号拡張されます。また、ADDIU 命令と SLTIU 命令のニモニックは、それぞれ Add Immediate *Unsigned* (符号なし) と Set On Less Than Immediate *Unsigned* (符号なし) を表しますが、即値は ADDI 命令、SLTI 命令と同様、符号拡張されます。ADDI 命令と ADDIU 命令の唯一の違いは、ADDIU ではオーバフロー例外が絶対に発生しない点です。したがって、ADDIU 命令は、オーバフロー例外を発生せずに、負の数値の加算を行いたいときに使うことができます。TX19A の命令セットに符号付き即値をともなう減算命令がないので、ADDIU 命令は便利です。また、SLTI 命令と SLTIU 命令の違いは、SLTI 命令が 2 値 (*rs* と符号拡張した即値) を符号付き整数として比較するのに対して、SLTIU 命令は 2 値 (*rs* と符号拡張した即値) を符号なし整数として比較する点です。

3.3.3　64 ビットの加算・減数

加算、減算したい数値が 32 ビットより長い場合があります。その場合、汎用レジスタは 32 ビットなので、32 ビットずつに分けて演算を行う必要があります。図 3-3 に 64 ビットの定数の加算・減算を示します。この図では、64 ビットの定数の上位 32 ビットが r3 に、下位 32 ビットが r2 に格納されています。同様に r5 と r4 にも、64 ビットの定数上位 32 ビットと下位 32 ビットが格納されています。

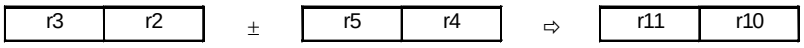


図 3-3　64 ビット加算・減算

■　キャリーが発生する加算

2 つの 64 ビットの定数を加算するためのコード例を以下に示します。

```
ADDU    r10,r2,r4  # r10 ← r2 + r4
SLTU    r11,r10,r2 # r10 (和)<r2 ならば r11=1
ADD(U)  r11,r11,r3 # r11 ← r11 (キャリー) + r3
ADD(U)  r11,r11,r5 # r11 ← r11 + r5
```

最初の ADDU 命令は、2 つの定数の下位 32 ビットを加算し、結果を r10 に格納しています。TX19A には、算術演算でキャリーが発生したことを記録するフラグがありません。そのため、加算でキャリーが発生した場合、何らかの方法でそれを記録しておく必要があります。正の数同士の加算の場合、加算結果がどちらかの加算前の値よりも小さくなったとき、キャリーが発生したと判断できます。したがって、SLTU (Set On Less Than Unsigned) 命令にて、r10 が r2 より小さいかどうかを調べ、小さい場合は、r11 に 1 の設定を行います。そして、次の 2 つの ADD (U) 命令で、キャリービット (0 または 1) と 2 つの定数の上位 32 ビットの加算を行います。

最後の 2 つの命令は ADD 命令でも ADDU 命令のどちらでもかまいません。ADD 命令と ADDU 命令の違いは、ADDU (Add Unsigned) 命令ではオーバフロー例外が発生しないという点のみです。

ただし、ADDU 命令を使用するときは、オーバフローが発生するかどうかを判断し、オーバフローが発生する場合の処理と、発生しない場合の処理を行うためのコードを別々に用意しておかなければなりません。これについて次の項で説明します。

■　ボローが発生する減算

64 ビットの減算を行う場合、上位 32 ビットのオペランドからの下位 32 ビットオペランドへのボローに注意しなければなりません。ボローが発生する減算は、キャリーが発生する加算によく似ています。以下に 64 ビットの定数から 64 ビットの定数を減算する例を示します。

```
SLTU    r8,r2,r4     # r2<r4 ならば r8=1
SUBU    r10,r2,r4    # r10 ← r2 – r4
SUB(U) r11,r3,r5     # r11 ← r3 – r5
SUB(U) r11,r11,r8    # r11 ← r11 - r8 (ボロー)
```

最初に、SLTU 命令で r2 (被減数) が r4 (減数) より小さいかどうか調べ、小さければ、r8 を 1 に設定します。これで、下位 32 ビットの減算でボローが発生したとき、ボローが r8 に記録されます。r8 の内容は最後の SUB (U) 命令で減算しています。

SUB 命令と SUBU 命令の違いは、SUBU 命令ではオーバフロー例外が発生しないという点のみです。

3.3.4　オーバフローが発生するかどうかの判断

前項で説明したように、符号付き加算命令 (ADD) 、符号付き減算命令 (SUB) は、加算・減数結果で、2 の補数のオーバフローが発生した場合、オーバフロー例外を発生します。これに対して、符号なし加算・減算命令 (ADDU・SUBU) ではオーバフロー例外は発生しません。符号付き演算で例外を発生させずにオーバフローを検出したい場合、または符号なし演算でオーバフローを検出したい場合には、オーバフロー検出用のルーチンを作成する必要があります。

加算では、オペランドの符号が同じ場合に加算結果 (和) の符号が異なったとき、オーバフローが発生したことになります。以下に、符号付き加算の結果オーバフローが発生したかどうかを調べるためのコードを示します。

```
    ADDU r2,r3,r4    # r2 ← r3 + r4
    XOR  r5,r3,r4    # r3 と r4 の符号を比較。
                     # 異なる場合、オーバフローなし(r5<0)
    BLTZ r5, No_Ov   # r4<0 ならば、No_Ov に分岐
    XOR  r5,r2,r3    # 和(r2)と加数(r3)の符号を比較。
                     # 異なれば、オーバフロー発生 (r5<0)
    BLTZ r4,Ov       # r5<0 ならば、Ov に分岐

No_Ov:
```

また、減算では、減算結果の符号が被減数の符号と異なる場合、オーバフローが発生したことになります。以下に、符号付き減算の結果オーバフローが発生したかどうかを調べるためのコードを示します。

```
    SUBU r2,r3,r4    # r2 ← r3 – r4
    XOR  r5,r3,r4    # r3 と r4 の符号を比較。
                     # 同じならば、オーバフローなし
    BGEZ r5,No_Ov    # r5=>0 ならば、No_Ov に分岐
    XOR  r5,r2,r3    # 差(r1)と被減数(r3)の符号を比較。
                     # 異なれば、オーバフロー発生
    BLTZ r5,Ov       # r5<0 ならば、Ov に分岐

No_Ov:
```

3.3.5　64 ビット × 64 ビットの乗算

TX19A で 2 つの整数を乗算する場合、整数は汎用レジスタに格納されていなければなりません。汎用レジスタは 32 ビットなので 64 ビット × 64 ビットの乗算では、被乗数、乗数のオペランドを格納するのに、レジスタを 2 つずつ使用します。

図 3-4に例を示します。この例では被乗数の上位 32 ビットがr3、下位 32 ビットがr2 に格納されているものとします。また、同様に乗数はr5 とr4 に格納されているものとします。

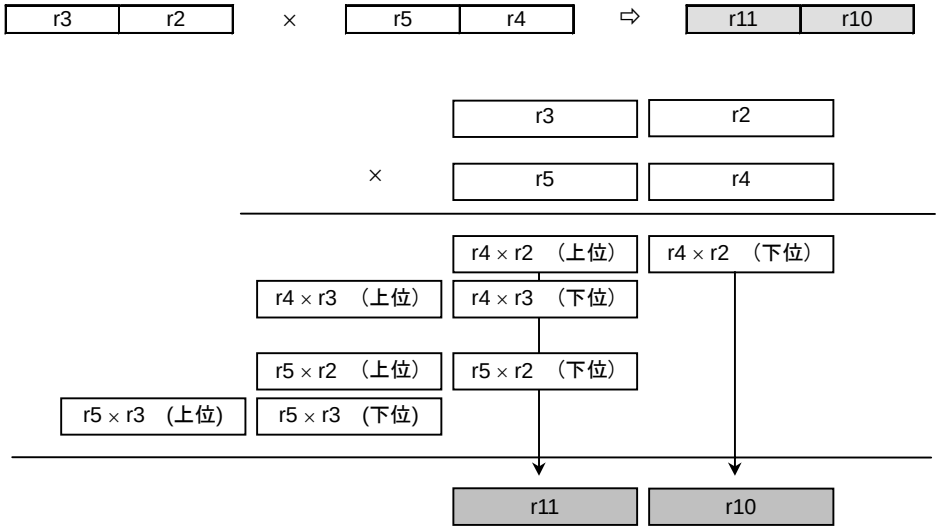


図 3-4　64 ビット × 64 ビットの乗算

以下に 64 ビット × 64 ビットの乗算を実行するためのコードの例を示します。積は最大 128 ビットになりますが、説明を簡単にするため、積の下位 64 ビットのみを考えます。

```
MULTU   r10,r2,r4   # r4 × r2, 積の下位 32 ビットを r10 に格納
MFHI    r11         # 積の上位 32 ビットを HI から r11 に転送
MULTU   r9,r3,r4    # r3 × r4, 積の下位 32 ビットを r9 に格納
ADDU    r11,r11,r9  # r11 ← r11 + r9
MULTU   r9,r2,r5    # r5 × r2, 積の下位 32 ビットを r9 に格納
ADDU    r11,r11,r9  # r11 ← r11 + r9
```

TX19A のアーキテクチャでは、MIPS の MULTU (Multiply Unsigned) 命令の機能が拡張されています。MIPS アーキテクチャでは、MULTU 命令は、被乗数と乗数が格納されているソースレジスタを 2 つしか指定できず、積は HI レジスタと LO レジスタに格納されます。それに対して、TX19A では、MULTU 命令は 3 つのオペランドを指定することができます。これにより、TX19A の MULTU 命令は積の下位 32 ビットを LO レジスタのほかに汎用レジスタにも同時に格納することができます。したがって、LO レジスタの内容を汎用レジスタに転送するのに、MFLO (Move From LO) 命令を使う必要がありません。

MFHI (Move From HI) 命令は、HI レジスタの内容、すなわち積の上位 32 ビットを汎用レジスタに転送します。

3.3.6　ローテート命令

TX19A には、機械語命令としてシフト命令は用意されていますが、ローテート命令は用意されていません。シフト命令では、左端(右シフト命令の場合は右端)からあふれたビットが廃棄され、右側(右シフト命令の場合は左側)の空きビットは 0 で埋められます。それに対しローテート命令では、各ビットの値を左方向(右ローテート命令の場合は右方向)へシフトし、あふれたビットは右端(右ローテート命令の場合は左端)に入ります。

TX19Aでは、ローテート命令はシフト命令と論理OR命令を組み合わせて実現します。図 3-5にレジスタの内容を 6 ビット左ローテートする例を示します。

```
SLL   r9,r8,6

SRL   r8,r8,(32-6)

OR    r8,r8,r9
```

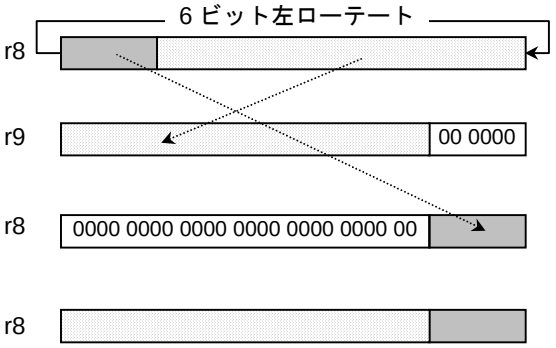


図 3-5　6 ビット左ローテート

図 3-5では、SLL (Shift Left Logical) 命令にてr8 の内容を 6 ビット左にシフトし、その結果をr9 に格納します。このとき空いた下位ビットは 0 で埋められます。次に SRL (Shift Right Logical) 命令でr8 の内容を 26 (32 – 6) ビット右にシフトします。最後にOR命令でr8 とr9 の内容の論理ORをとり、その結果をr8 に格納します。その結果、r8 を 6 ビット左ローテートしたのと同じ結果が得られます。

3.4　ジャンプ・分岐・分岐ライクリ命令

プログラムの流れを変える命令には、ジャンプ命令、分岐命令、分岐ライクリ命令があります。

3.4.1項で命令の概要を説明します。

3.4.2項では、ジャンプ命令、分岐命令、分岐ライクリ命令でサポートされているアドレッシングモードについて説明します。

3.4.3項では、32 ビットISAモードと 16 ビットISAモードの切り換え方法について説明します。

3.4.4項では、一般分岐命令と分岐ライクリ命令の違いについて説明します。

3.4.5項では、大小関係に基づき分岐する方法について説明します。

3.4.6項では、32 ビットの絶対アドレスへジャンプする方法について説明します。

3.4.7項では、サブルーチンコールとサブルーチンからの復帰について説明します。

3.4.1　ジャンプ・分岐・分岐ライクリ命令の概要

TX19Aでは、ジャンプ命令は無条件分岐のための命令です。それに対して、分岐命令と分岐ライクリ命令は､多くのマイクロプロセッサで条件付きジャンプと呼んでいる命令で、条件が成立した場合のみプログラムの流れを変えます。表 3-4と 表 3-5 に 32 ビットISAで用意されているジャンプ、分岐、分岐ライクリ命令を示します。

表 3-4　ジャンプ命令 (32 ビット ISA)

オペコード	命令	アドレッシング	命令形式
J	Jump	ページ内絶対	I 形式
JAL	Jump And Link	ページ内絶対	I 形式
JALX	Jump And Link eXchange	ページ内絶対	I 形式
JR	Jump Register	レジスタ間接	R 形式
JALR	Jump And Link Register	レジスタ間接	R 形式

表 3-5　分岐命令・分岐ライクリ命令 (32 ビット ISA)

オペコード	命令	条件	アドレッシング	命令形式
B	Unconditional Branch	*always*	PC 相対	I 形式
BAL	Branch And Link	*always*	PC 相対	I 形式
BEQ(L)	Branch On Equal (Likely)	$rs = rt$	PC 相対	I 形式
BNE(L)	Branch On Not Equal (Likely)	$rs \neq rt$	PC 相対	I 形式
BGTZ(L)	Branch On Greater Than Zero (Likely)	$rs > 0$	PC 相対	I 形式
BGEZ(L)	Branch On Greater Than or Equal To Zero (Likely)	$rs \geq 0$	PC 相対	I 形式
BLTZ(L)	Branch On Less Than Zero (Likely)	$rs < 0$	PC 相対	I 形式
BLEZ(L)	Branch On Less Than or Equal To Zero (Likely)	$rs \leq 0$	PC 相対	I 形式
BLTZAL(L)	Branch On Less Than Zero And Link (Likely)	$rs < 0$	PC 相対	I 形式
BGEZAL(L)	Branch On Greater Than or Equal To Zero And Link (Likely)	$rs \geq 0$	PC 相対	I 形式

リンク機能付きジャンプ命令とリンク機能付き分岐命令では、レジスタ r31 に戻りアドレスが格納されます。これらの命令はサブルーチンコールで使われます。

ジャンプ命令と一般分岐命令では、ターゲット命令をメモリからフェッチしているあいだに、その直後に置かれた遅延スロットの命令が先に実行されます。これは、分岐するかしないかにかかわらず、すべての一般分岐命令であてはまります。これに対して、分岐ライクリ命令では、分岐条件が成立したときのみ、遅延スロットの命令が実行され、条件が成立しなかったときは、遅延スロットの命令は廃棄されます。遅延スロットについては、「5 章　CPU パイプライン」を参照してください。

3.4.2　アドレッシングモード

表 3-4と 表 3-5に示したように、ジャンプ命令、分岐命令、分岐ライクリ命令は、以下のアドレッシングモードを使って、ターゲット命令の実効アドレスを計算します。

- ページ内絶対アドレッシング
- レジスタ間接アドレッシング
- オフセット付き PC 相対アドレッシング

■　ページ内絶対アドレッシングモード

J、JAL、JALX命令は、絶対アドレッシングモードを使って、ターゲットアドレスへ無条件にジャンプします。これらの命令では、指定された 26 ビットのオフセット値を左に 2 ビットシフトした値に、プログラムカウンタ (PC) の上位 4 ビットと連結した結果がターゲットアドレスになります。アドレスの生成方法を 図 3-6に示します。ターゲットアドレスはジャンプ命令の直後の命令、すなわちジャンプ遅延スロットのアドレスから生成されます。PCの上位 4 ビットは 16 ページアドレス空間の特定のページを示します。

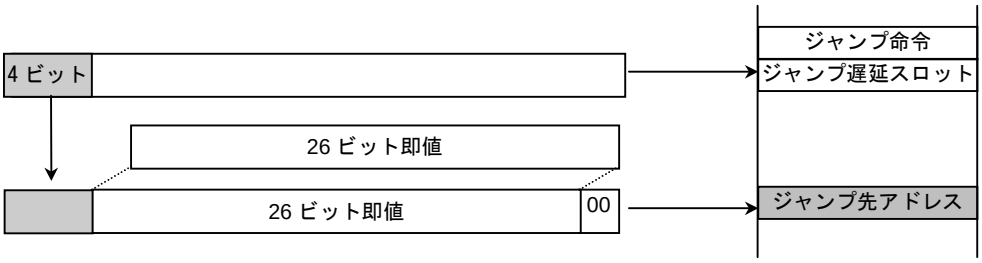


図 3-6　ページ内絶対アドレッシング (32 ビット ISA)

■　レジスタ間接アドレッシングモード

JR、JALR 命令は、汎用レジスタに格納されている 32 ビットの絶対アドレスに無条件にジャンプします。ジャンプ先のアドレスは指定されたターゲットレジスタの最下位ビットを 0 にマスクした値です。32 ビット ISA の命令はワード境界に位置合わせするため、JR 命令や JALR 命令で指定するレジスタのビット 1 を 0 にする必要があります。

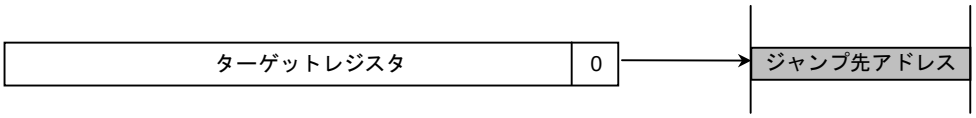


図 3-7　レジスタ間接アドレッシング (32 ビット ISA)

■　オフセット付 PC 相対アドレッシングモード

分岐命令、分岐ライクリ命令は、すべてPC相対アドレッシングモードを使います。これらの命令では、指定された 16 ビットの即値 (オフセット) を 2 ビットシフトして符号拡張した値をプログラムカウンタ (PC) の値に加算した結果がターゲットアドレスになります。アドレスの生成方法を 図 3-8に示します。分岐先のアドレスは分岐命令の直後の命令、すなわち分岐遅延スロットのアドレスから生成されます。

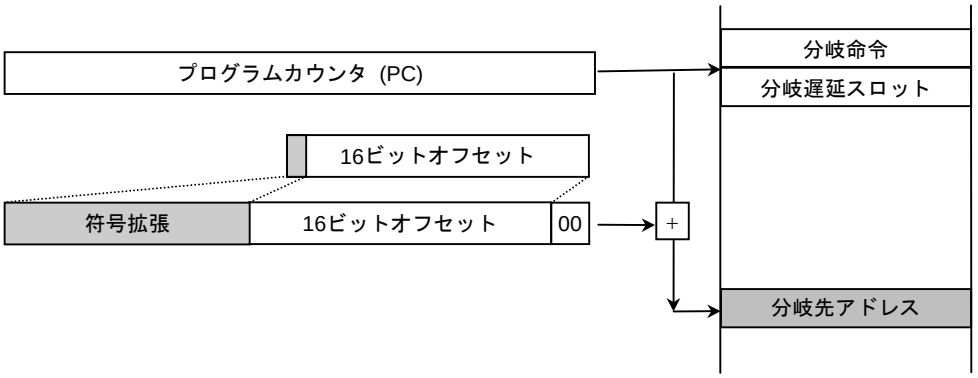


図 3-8　オフセット付き PC 相対アドレッシング (32 ビット ISA)

3.4.3　ISA モードの切り換え

TX19A には、16 ビット ISA モードと 32 ビット ISA モードの 2 つの ISA モードがあります。16 ビット ISA モードと 32 ビット ISA モードはプログラムの実行中に、JALX、JR、JALR 命令により切り換えられます。プログラムカウンタ (PC) の最下位ビットが ISA モードビットで、0 のときは 32 ビット ISA モードになり、1 のときは 16 ビット ISA モードになります。JALX 命令では、ジャンプ後の PC の ISA モードビット (最下位ビット) が他の ISA モードに無条件で切り換えられます。JR 命令と JALR 命令では、ジャンプアドレスが格納されているレジスタの最下位ビットから ISA モードビットが設定されます。ジャンプアドレスは、ISA モードビットをゼロにマスクした値になります。

32 ビット ISA モードでは、命令はワード境界上に位置合わせしなければなりません。そのため、16 ビット ISA モードから 32 ビット ISA モードへ切り換えるときは、JR 命令または JALR 命令で指定するレジスタの値は下位 2 ビットが 00 でなければなりません。下位 2 ビットが 10 の場合、ジャンプ先の命令をフェッチするときに、アドレスエラー例外が発生します。

JALX 命令、JR 命令、JALR 命令のジャンプ遅延スロットにある命令は、ジャンプ前の ISA モードで実行されます。

リンク機能付きジャンプ命令、分岐命令、分岐ライクリ命令では、レジスタ r31 (ra) または指定されたデスティネーションレジスタ (*rd*) に、サブルーチンからの戻りアドレスが自動的に格納されます。レジスタの最下位ビットには、サブルーチンが実行された後の ISA モードが格納されます。サブルーチンからの復帰後はサブルーチンに入る前の ISA モードになります。

3.4.4　分岐ライクリ命令

ジャンプ命令、分岐命令では、ジャンプまたは分岐先の実効アドレスを計算し、命令をフェッチしなければならないので、プログラムのフローを変えるのに 2 命令の遅延が発生します。この遅延をジャンプ遅延、分岐遅延といいます。TX19A では、遅延スロットの処理をソフトウエアに任せており、ターゲット命令をメモリからフェッチしているあいだに、ジャンプまたは分岐直後の命令を実行するようにコンパイラまたはアセンブラにより命令の順序が再編成されます。

ジャンプ命令は、プログラムの流れを無条件に変更するので何の問題もありません。つまり、ジャンプ直後の命令を、常に遅延スロットに置くことができます。しかし、分岐命令では、プロセッサは分岐条件が成立するかどうか、あらかじめ知ることはできません。したがって、遅延スロットの命令は、プログラムの論理に影響を与えない命令でなければなりません。そのような命令がない場合は、NOP (No Operation) 命令で命令パイプラインを埋める必要があります。(NOP 命令はアセンブラで受け付けられる疑似命令で、1 章で説明したように NOP 命令はアセンブラにより r0 レジスタへの 0 ビットシフト命令に変換されます。)

図 3-9に、r8 の値が 0 か 1 かという判断に基づいて、レジスタr2 を 0 または 1 に設定するプログラムを示します。この例では一般分岐命令を使っています。ADDI命令はプログラムの論理の上からBEQ命令より先に実行できないので、BEQ命令の直後にNOPが必要です。

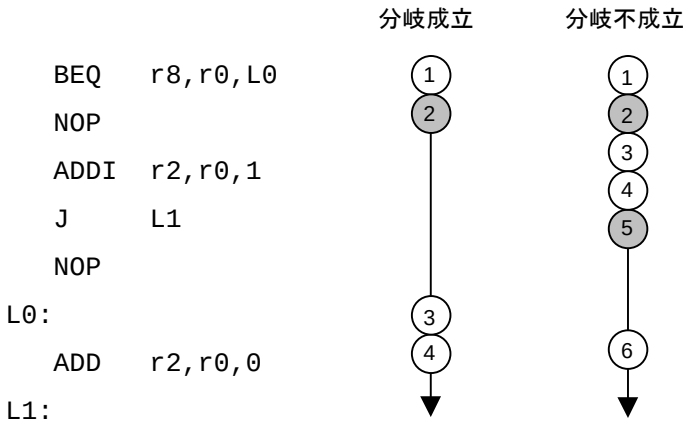


図 3-9　一般分岐命令

図 3-10では、BEQ命令の代わりに分岐ライクリ命令BEQL (Branch On Equal Likely) を使っています。分岐ライクリ命令は、分岐条件が成立したとき遅延スロットの命令が実行される命令です。分岐条件が成立しなかったときは、遅延スロットの命令は廃棄されます。このため遅延スロットにNOP命令を挿入する必要がなく、コードサイズの縮小、分岐処理の高速化に役立ちます。

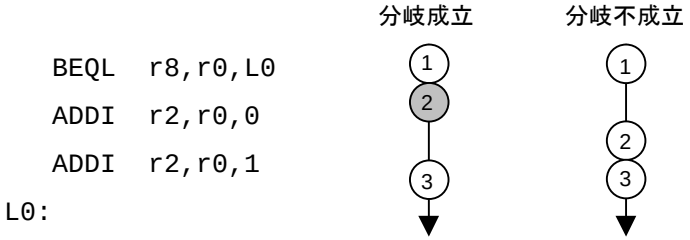


図 3-10　分岐ライクリ命令

3.4.5　大小関係に基づく分岐

2 つのレジスタの値を比較して、その結果に基づいて分岐する命令は、BEQ (Branch On Equal) 命令、BNE (Branch On Not Equal) 命令、およびそれらの分岐ライクリ命令 (BEQL・BNEL) があります。以下に例を示します。

```
BEQ r2,r3,Equal
```

上記の例では、レジスタ r2 の内容とレジスタ r3 の内容を比較し、値が等しい場合、Equal に分岐します。しかし、r2 の内容が r3 の内容より大きいかどうかという判断に基づいて分岐する命令はありません。2 つのレジスタ、またはレジスタと即値の比較を行うには、2 つの命令を組み合わせて実現する必要があります。以下にレジスタ同士の大小比較、レジスタと即値の大小比較、レジスタと即値の比較のプログラム例を示します。(アセンブラには、マクロ命令が用意されているものがあります。そのようなアセンブラを使うとマクロ命令を機械語命令に自動的に変換してくれるので、プログラミングの負荷を軽くすることができます。)

◆ 例 1　r6 ≥ r7 の場合の分岐

以下に r6 の値が r7 の値以上の場合に分岐する例を示します。r6 の値が r7 より小さい場合、SLT (Set On Less Than) 命令により r24 が 1 に設定されます。r6 が r7 以上の場合 r24 は 0 に設定されます。BEQ 命令で r24 の値を判断することにより大小関係の分岐を実現することができます。
(r0 はハードウエア的に 0 に固定されています。)

```
SLT     r24,r6,r7
BEQ     r24,r0,Label
```

◆ 例 2　r7 ≥ 0x1234 の場合の分岐

以下に、r7 の値が 0x1234 以上の場合に分岐する例を示します。この例では、SLTI (Set On Less Than Immediate) 命令により r7 の値と 0x1234 が比較され、r7 が大きい場合は r24 が 0 に設定されます。

```
SLTI    r24,r7,0x1234
BEQ     r24,r0,Label
```

◆ 例 3　r7 ≠ 0x1234 の場合の分岐

以下に、レジスタの値と即値が等しいかどうか調べて、それに基づき分岐する例を示します。この例では、ORI (OR Immediate) 命令を使って、0x1234 を r10 に一時的に格納します。次に BEQ 命令で r10 の値と r7 の値を比較します。

```
ORI     r10,r0,0x1234
BEQ     r10,r7,Label
```

3.4.6　32 ビットのアドレスへのジャンプ

3.4.2項で説明したように、ページ内絶対アドレッシングモードを使うJ命令、JAL命令、JALX命令は、最大 26 ビットです。26 ビットの即値は 2 ビット左シフトされるので、ターゲットアドレスは、256Mバイトセグメント内でなければなりません。任意の 32 ビットのアドレスへジャンプするには、LUI命令とORI命令を使って、希望するアドレスをいったんレジスタに格納し、次にJR (Jump Register) 命令を使う必要があります。以下に、アドレス 0x76543210 へジャンプするための例を示します。

```
LUI     r8,0x7654
ORI     r8,0x3210
JR      r8
```

3.4.7　サブルーチンコール

32 ビット ISA には、リンク機能付きジャンプ命令 (JAL、JALX、JALR)、リンク機能付き分岐命令 (BLTZAL、BGEZAL)、リンク機能付き分岐ライクリ命令 (BLTZALL、BGEZALL) があります。これらの命令は通常サブルーチンコールに使われ、サブルーチンの戻りアドレスがレジスタ r31 (ra) に格納されます。JALR (Jump-And-Link Register) 命令では、r31 以外の汎用レジスタ (*rd*) を使うこともできます。戻りアドレスを格納するレジスタをリンクレジスタといいます。

戻りアドレスは、遅延スロットの直後の命令のアドレスになります。また、リンク機能付きのジャンプ命令ではリンクレジスタの最下位ビットに ISA モードが保存されます。

サブルーチンから復帰するには、JR 命令を使います。ISA モードビット (PC の最下位ビット) は、リンクレジスタの最下位ビットから復帰されます。

サブルーチンをネスティングする場合は、次のサブルーチンをコールするまえにリンクレジスタの戻りアドレスをスタック領域に退避させなければなりません。

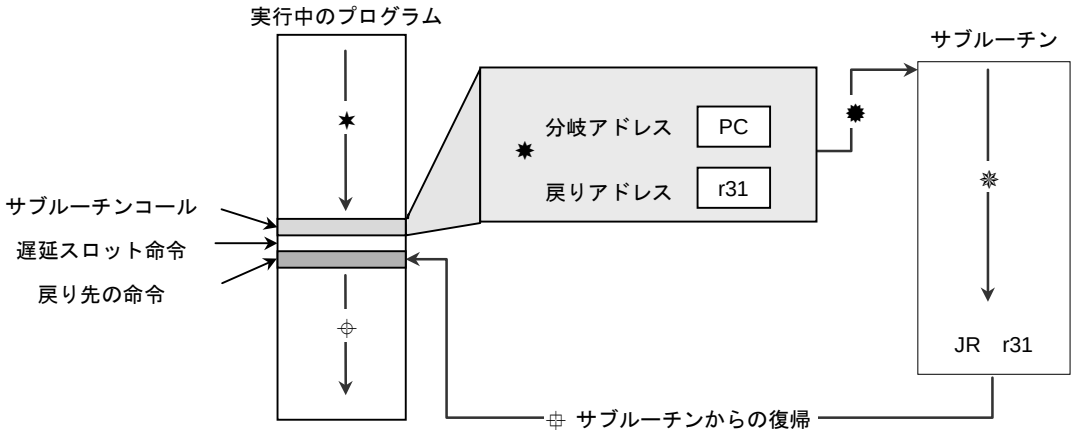


図 3-11　サブルーチンとコール

JAL 命令、JALX 命令を除くリンク機能付きジャンプ命令、分岐命令、分岐ライクリ命令は、ソースレジスタ (*rs*) フィールドをもっています。以下に例を示します。

```
BGEZAL r8,PSUB
```

この命令で、r8 はソースレジスタです。BGEZAL 命令は r8 の値が 0 以上かどうか調べ、その結果に基づき分岐します。

ジャンプ遅延スロットまたは分岐遅延スロットの命令を実行している最中に、例外や割り込みが発生すると、その命令の終了を待たずに例外処理が始まります。この場合、例外または割り込みが発生した命令の直前にあるジャンプ分岐、分岐命令、または分岐ライクリ命令のアドレスが例外プログラムカウンタ (EPC) に設定されます。例外ハンドラの実行後は、ジャンプ命令、分岐命令、または分岐ライクリ命令から処理を再開しなければなりません。そのため、r31 (ra) をソースレジスタとして使用してはいけません。例外処理の手順については、「9 章　例外処理」を参照してください。

3.5　コプロセッサ命令

TX19A は、システム制御コプロセッサ (CP0) のみを実装しており、その他の CP1、CP2 は接続することができません。したがって、32 ビット ISA で定義されている CP1、CP2 に関係する命令は、予約命令例外あるいはコプロセッサ使用不可例外として処理されます。

Status レジスタの CU ビットが 0 の状態でコプロセッサ命令 (CP1、CP2) を実行するとコプロセッサ使用不可例外が発生し、セットされた状態で実行すると予約命令例外が発生します。

32 ビット ISA にある LWC*z* (Load Word To Coprocessor) 命令と SWC*z* (Store Word From Coprocessor) 命令は、TX19A ではサポートされていません。これらのロード・ストア命令を実行しようとすると、予約命令例外が発生します。

システム制御コプロセッサ (CP0) 命令は、システム構成、メモリ管理、例外処理などの操作を行うための命令で、CP0 レジスタを操作します。そのため、CP0 にはある程度の特権保護が与えられています。ユーザーモードのときは、Statusレジスタの CU0 ビットが 1 に設定されていないと、CP0 レジスタにはアクセスできません。CU0 ビットが 0 のときにCP0 命令を実行しようとすると、コプロセッサ使用不可例外が発生します。ただし、カーネルモード、デバッグモードのときは、CU0 ビットの設定に関係なく、すべてのCP0 命令を実行できます。
表 3-6にCP0 命令を示します。

表 3-6　システム制御コプロセッサ (CP0) 命令

命令名	オペコード
Move To/From CP0	MTC0・MFC0
Exception Return	ERET
Debug Exception Return	DERET
Enter Standby Mode	WAIT

TX19A は、仮想アドレスから物理アドレスへの変換にダイレクトセグメントマッピング方式を採用しており、TLB (table lookaside buffer) はサポートしていません。

3.6　特殊命令

32 ビット ISA には 4 つの特殊命令が用意されていて、ソフトウエアにより例外を発生させることができます。特殊命令には SYSCALL (System Call)、BREAK (Breakpoint)、SDBBP (Software Debug Breakpoint)、Trap 命令があり、すべて R 形式です。特殊命令を実行すると、プログラムの処理は無条件に対応する例外ハンドラに移ります。例外処理の詳細については、9 章を参照してください。

3.7　命令の概要

この項では、32 ビット ISA の命令の概要を分類ごとに示します。

■　表記規則

この項では、命令中 *rt*、*rs*、*rd*、*immediate*、*sa* (シフト量: shift amount)などの小文字の斜体で示してある部分には、ユーザーが任意のレジスタや値などを指定できます。また、オペランドの意味がより明確になるように、例えば、ロード命令とストア命令では、*rs*、*immediate* と書かずに *base*、*offset* と記述してあります。HI と LO は、整数の乗算・除算の結果を格納する特殊レジスタです。

■　TX19A で拡張された命令

当社の TX19、TX39 のアーキテクチャにはなく、TX19A で拡張されている命令があります。それらの命令には、この項でただし書きを添えてあります。詳細は付録 D を参照してください。

表 3-7　ロード・ストア命令 (32 ビット ISA)

命令	形式	説明
Load Byte	LB *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。EA でアドレス指定されたバイトデータを符号拡張し、*rt* にロードします。
Load Byte Unsigned	LBU *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。EA でアドレス指定されたバイトデータをゼロ拡張し、*rt* にロードします。
Load Halfword	LH *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。EA でアドレス指定されたハーフワードデータを符号拡張し、*rt* にロードします。
Load Halfword Unsigned	LHU *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。EA でアドレス指定されたハーフワードデータをゼロ拡張し、*rt* にロードします。
Load Word	LW *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。EA でアドレス指定されたワードデータを、*rt* にロードします。
Load Word Left	LWL *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。*rt* の左側に、EA でアドレス指定されたメモリワードの上位部分を格納します。
Load Word Right	LWR *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。*rt* の右側に、EA でアドレス指定されたメモリワードの下位部分を格納します。
Store Byte	SB *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。*rt* の最下位バイトを、このアドレスにストアします。
Store Halfword	SH *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。*rt* の下位のハーフワードを、このアドレスにストアします。
Store Word	SW *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。*rt* の内容を、このアドレスにストアします。
Store Word Left	SWL *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。*rt* の左側の内容を、EA でアドレス指定されたメモリワードの上位の部分にストアします。
Store Word Right	SWR *rt*, *offset*(*base*)	実効アドレス (EA) は *base* + *offset* です。16 ビット *offset* を符号拡張します。*rt* の右側の内容を、EA でアドレス指定されたメモリワードの下位部分にストアします。
Sync	SYNC	直前に実行したロードまたはストア命令が完了するまで、パイプラインをインタロックします。

表 3-8　ALU 即値命令 (32 ビット ISA)

命令	形式	説明
Add Immediate	ADDI *rt*, *rs*, *immediate*	*rs* + *immediate* を *rt* に格納します。16 ビット *immediate* を符号拡張します。2 の補数のオーバフローで例外が発生します。
Add Immediate Unsigned	ADDIU *rt*, *rs*, *immediate*	*rs* + *immediate* を *rt* に格納します。16 ビット *immediate* を符号拡張します。2 の補数のオーバフローでも例外が発生しません。
Set On Less Than Immediate	SLTI *rt*, *rs*, *immediate*	*rs* が *immediate* より小さい場合は、*rt* に 1 を、そうでない場合は 0 を格納します。16 ビット *immediate* を符号拡張します。*rs* と *immediate* を符号付き整数として比較します。
Set On Less Than Immediate Unsigned	SLTIU *rt*, *rs*, *immediate*	*rs* が *immediate* より小さい場合は、*rt* に 1 を、そうでない場合は 0 を格納します。16 ビット *immediate* を符号拡張します。*rs* と *immediate* を符号なし整数として比較します。
AND Immediate	ANDI *rt*, *rs*, *immediate*	*rs* の内容と *immediate* の AND をとり、結果を *rt* に格納します。16 ビット *immediate* をゼロ拡張します。
OR Immediate	ORI *rt*, *rs*, *immediate*	*rs* の内容と *immediate* の OR をとり、結果を *rt* に格納します。16 ビット *immediate* をゼロ拡張します。
Exclusive-OR Immediate	XORI *rt*, *rs*, *immediate*	*rs* の内容と *immediate* の排他的 OR をとり、結果を *rt* に格納します。16 ビット *immediate* をゼロ拡張します。
Load Upper Immediate	LUI *rt*, *immediate*	16 ビット *immediate* を 16 ビット左にシフトし、下位 16 ビットの 0 と連結して、結果を *rt* に格納します。

表 3-9　2/3 オペランドレジスタタイプ命令 (32 ビット ISA)

命令	形式	説明
Add	ADD *rd*, *rs*, *rt*	*rs* + *rt* の和を *rd* に格納します。2 の補数のオーバフローで例外が発生します。
Add Unsigned	ADDU *rd*, *rs*, *rt*	*rs* + *rt* の和を *rd* に格納します。2 の補数のオーバフローでも例外が発生しません。
Subtract	SUB *rd*, *rs*, *rt*	*rs* − *rt* の差を *rd* に格納します。2 の補数のオーバフローで例外が発生します。
Subtract Unsigned	SUBU *rd*, *rs*, *rt*	*rs* − *rt* の差を *rd* に格納します。2 の補数のオーバフローでも例外が発生しません。
Set On Less Than	SLT *rd*, *rs*, *rt*	*rs* が *rt* より小さい場合は、*rd* に 1 を、そうでない場合は、*rd* に 0 を格納します。*rs* と *rt* を符号付き整数として比較します。
Set On Less Than Unsigned	SLTU *rd*, *rs*, *rt*	*rs* が *rt* より小さい場合は、*rd* に 1 を、そうでない場合は、*rd* に 0 を格納します。*rs* と *rt* を符号付なし整数として比較します。
AND	AND *rd*, *rs*, *rt*	*rs* の内容と *rt* の内容の AND をとり、結果を *rd* に格納します。
OR	OR *rd*, *rs*, *rt*	*rs* の内容と *rt* の内容の OR をとり、結果を *rd* に格納します。
Exclusive-R	XOR *rd*, *rs*, *rt*	*rs* の内容と *rt* の内容の排他的 OR をとり、結果を *rd* に格納します。
NOR	NOR *rd*, *rs*, *rt*	*rs* の内容と *rt* の内容の NOR をとり、結果を *rd* に格納します。
* Count Leading Ones in Word	CLO *rd, rs*	*rs* のビット 31 からビット 0 の方向へ何ビット連続して 1 が並んでいるかカウントし、その結果を *rd* に格納します。
* Count Leading Zeros in Word	CLZ *rd, rs*	*rs* のビット 31 からビット 0 の方向へ何ビット連続して 0 が並んでいるかカウントし、その結果を *rd* に格納します。
* Move Conditional on Not Zero	MOVN *rd, rs, rt*	*rt* ≠ 0 ならば、rs の内容を *rd* に格納します。
* Move Conditional on Zero	MOVZ *rd, rs, rt*	*rt* = 0 ならば、rs の内容を *rd* に格納します。

*　TX19 → TX19A で追加された命令

表 3-10　シフト命令 (32 ビット ISA)

命令	形式	説明
Shift Left Logical	SLL *rd*, *rt*, *sa*	*rt* の内容を *sa* ビット左へシフトし、右端の空いたビットを 0 で埋めます。結果を *rd* に格納します。
Shift Left Logical Variable	SLLV *rd*, *rt*, *rs*	*rt* の内容を *rs* の下位 5 ビットで指定されたビット数、左にシフトし、右端の空いたビットを 0 で埋めます。結果を *rd* に格納します。
Shift Right Logical	SRL *rd*, *rt*, *sa*	*rt* の内容を *sa* ビット右にシフトし、左端の空いたビットを 0 で埋めます。結果を *rd* に格納します。
Shift Right Logical Variable	SRLV *rd*, *rt*, *rs*	*rt* の内容を *rs* の最下位 5 ビットで指定されたビット数､右にシフトし、左端の空いたビットを 0 で埋めます。結果を *rd* に格納します。
Shift Right Arithmetic	SRA *rd*, *rt*, *sa*	*rt* の内容を *sa* ビット右にシフトし､左端の空いたビットを符号ビットで埋めます。結果を *rd* に格納します。
Shift Right Arithmetic Variable	SRAV *rd*, *rt*, *rs*	*rt* の内容を *rs* の下位 5 ビットで指定されたビット数、右にシフトし、左端の空いたビットを符号ビットで埋めます。結果を *rd* に格納します。

表 3-11　乗算・除算命令 (32 ビット ISA)

命令	形式	説明
* Multiply	MUL *rd*, *rs*, *rt*	被乗数は *rs* の符号付き整数です。乗数は *rt* の符号付き整数です。64 ビットの積 *rs* * *rt* の下位 32 ビットを *rd* に格納し、HI・LO レジスタは不定となります。
Multiply	MULT (*rd*,) *rs*, *rt*	被乗数は *rs* の符号付き整数です。乗数は *rt* の符号付き整数です。64 ビットの積 *rs* * *rt* を HI レジスタと LO レジスタに格納します。また、下位 32 ビットを任意で *rd* に格納します。
Multiply Unsigned	MULTU (*rd*,) *rs*, *rt*	被乗数は *rs* の符号なし整数です。乗数は *rt* の符号なし整数です。64 ビットの積 *rs* * *rt* を HI レジスタと LO レジスタに格納します。また、下位 32 ビットを任意で *rd* にコピーします。
Divide	DIV *rs*, *rt*	被除数は *rs* の符号付き整数です。除数は *rt* の符号付き整数です。商を LO レジスタに、剰余を HI レジスタに格納します。
Divide Unsigned	DIVU *rs*, *rt*	被除数は *rs* の符号なし整数です。除数は *rt* の符号なし整数です。商を LO レジスタに、剰余を HI レジスタに格納します。
Move From HI	MFHI *rd*	HI レジスタの内容を *rd* にロードします。
Move From LO	MFLO *rd*	LO レジスタの内容を *rd* にロードします。
Move To HI	MTHI *rs*	*rs* の内容を HI レジスタにロードします。
Move To LO	MTLO *rs*	*rs* の内容を LO レジスタにロードします

*　TX19 → TX19A で追加された命令

表 3-12　積和命令/積差命令 (32 ビット ISA)

命令	形式	説明
Multiply and Add	MADD　(*rd*,) *rs*, *rt*	被乗数は *rs* の符号付き整数です。乗数は *rt* の符号付き整数です。64 ビットの積 *rs* * *rt* を HI レジスタ・LO レジスタに格納されている 64 ビットの値に加算し、その結果を HI レジスタと LO レジスタに格納します。また、結果の下位 32 ビットを *rd* に格納します。
Multiply and Add Unsigned	MADDU　(*rd*,) *rs*, *rt*	被乗数は *rs* の符号なし整数です。乗数は *rt* の符号なし整数です。64 ビットの積 *rs* * *rt* を HI レジスタ・LO レジスタに格納されている 64 ビットの値に加算し、その結果を HI レジスタと LO レジスタに格納します。また、結果の下位 32 ビットを *rd* に格納します。
* Multiply and Substract	MSUB　(*rd*,) *rs*, *rt*	被乗数は rs の符号付整数です。乗数は *rt* の符号付整数です。HI レジスタ・LO レジスタに格納されている値から 64 ビットの積 *rs* * *rt* を減算し、その結果を HI レジスタと LO レジスタに格納し、また結果の下位 32 ビットを *rd* に格納します。
* Multiply and Substract Unsigned	MSUBU　(*rd*,) *rs*, *rt*	被乗数は rs の符号なし整数です。乗数は *rt* の符号なし整数です。HI レジスタ・LO レジスタに格納されている値から 64 ビットの積 *rs* * *rt* を減算し、その結果を HI レジスタと LO レジスタに格納し、また結果の下位 32 ビットを *rd* に格納します。

*　TX19 → TX19A で追加された命令

表 3-13　ジャンプ命令 (32 ビット ISA)

命令	形式	説明
Jump	J　*target*	ページ内絶対アドレッシングモードを使ってジャンプします。つまり、26 ビット *target* を 2 ビット左へシフトし、PC + 4 の上位 4 ビットと連結した結果がターゲットアドレスになります。
Jump And Link	JAL　*target*	ページ内絶対アドレッシングモードを使ってジャンプします。つまり、26 ビット *target* を 2 ビット左へシフトし、PC + 4 の上位 4 ビットと連結した結果がターゲットアドレスになります。また、遅延スロットの後続命令のアドレスを、r31 に格納します。
Jump And Link eXchange	JALX　*target*	ページ内絶対アドレッシングモードを使ってジャンプします。つまり、26 ビット *target* を 2 ビット左へシフトし、PC + 4 の上位 4 ビットと連結した結果がターゲットアドレスになります。また、遅延スロットの後続命令のアドレスを、r31 に保存します。PC の ISA モードビットが切り換わります。
Jump Register	JR　*rs*	*rs* の上位 31 ビットで指定されたアドレスへジャンプします。*rs* の最下位のビットの値によって、ISA モードが切り換わります。
Jump And Link Register	JALR　(*rd*,) *rs*	*rs* の上位 31 ビットで指定されたアドレスへジャンプします。*rs* の最下位のビットの値によって、ISA モードが切り換わります。また、遅延スロットの後続命令のアドレスを *rd* に保存します。*rd* を省略すると、デフォルトで r31 が使用されます。

表 3-14　分岐・分岐ライクラリ命令 (32 ビット ISA)

命令	形式	説明
Branch On Equal (Likely)	BEQ(L)　*rs*, *rt*, *offset*	*rs* = *rt* ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。
Branch On Not Equal (Likely)	BNE(L)　*rs*, *rt*, *offset*	*rs* ≠ *rt* ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。
Branch On Greater Than Zero (Likely)	BGTZ(L)　*rs*, *offset*	*rs* > 0 ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。
Branch On Greater Than or Equal to Zero (Likely)	BGEZ(L)　*rs*, *offset*	*rs* ≥ 0 ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。
Branch On Less Than Zero (Likely)	BLTZ(L)　*rs*, *offset*	*rs* < 0 ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。
Branch On Less Than or Equal to Zero (Likely)	BLEZ(L)　*rs*, *offset*	*rs* ≤ 0 ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。
Branch On Less Than Zero And Link (Likely)	BLTZAL(L)　*rs*, *offset*	*rs* < 0 ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。また、遅延スロットの後続命令のアドレスを r31 に格納します。
Branch On Greater Than or Equal to Zero And Link (Likely)	BGEZAL(L)　*rs*, *offset*	*rs* ≥ 0 ならば、PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。また、遅延スロットの後続命令のアドレスを r31 に格納します。
* Unconditional Branch	B　*offset*	常に PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。
* Branch And Link	BAL　*offset*	常に PC + 4 (遅延スロットのアドレス) に対して相対的に計算された 16 ビット *offset* に分岐します。また、遅延スロットの後続命令のアドレスを r31 に格納します。

*　TX19 → TX19A で追加された命令

■ カッコ内に示したオペコードの接尾辞「L」は分岐ライクリ命令を表します。

表 3-15　システム制御コプロセッサ (CP0) 命令 (32 ビット ISA)

命令	形式	説明
Move To CP0	MTC0　*rt*, *rd*	汎用レジスタ *rt* の内容を CP0 レジスタ *rd* にロードします。
Move From CP0	MFC0　*rt*, *rd*	CP0 レジスタ *rd* の内容を汎用レジスタ *rt* にロードします。
* Exception Return	ERET	Status レジスタの ERL ビットが 1 の状態で、本命令を実行すると Error EPC を戻り番地として復帰し、ERL ビットが 0 の状態で本命令を実行すると EPC を戻り番地として割り込みから復帰します。
Debug Exception Return	DERET	プログラム制御は、デバッグ例外処理プログラムよりユーザープログラムに戻ります。DEPC レジスタの戻りアドレスを PC に復元します。
* Enter Standby Mode	WAIT	Status レジスタの PR ビットの状態に応じて、HALT あるいは DOZE モードに遷移します。

*　TX19 → TX19A で追加された命令

表 3-16　特殊命令 (32 ビット ISA)

命令	形式	説明
System Call	SYSCALL *code*	システムコール例外が発生し、無条件で例外ハンドラの処理に移ります。
Breakpoint	BREAK *code*	ブレークポイント例外が発生し、無条件で例外ハンドラの処理に移ります。
Software Debug Breakpoint Exception	SDBBP *code*	デバッグブレークポイント例外が発生し、無条件で例外ハンドラの処理に移ります。
Trap If Equal	TEQ *rs, rt*	*rs = rt* ならば、Trap 例外を発生します。
* Trap If Equal Immediate	TEQI *rs*, *immediate*	16 ビット *immediate* を符号拡張し、*rs = immediate* の場合は、Trap 例外を発生します。*rs* と *immediate* を符合付き整数として比較します。
* Trap If Greater Than or Equal	TGE *rs, rt*	*rs >= rt* ならば、Trap 例外を発生します。*rs* と *rt* を符号付き整数として比較します。
* Trap If Greater Than or Equal Immediate	TGEI *rs, immediate*	16 ビット *immediate* を符号拡張し、*rs ≥ immediate* ならば、Trap 例外を発生します。*rs* と *immediate* を符号付き整数として比較します。
* Trap If Greater Than or Equal Immediate Unsigned	TGEIU *rs, immediate*	16 ビット *immediate* を符号拡張し、*rs ≥ immediate* ならば、Trap 例外を発生します。*rs* と *immediate* を符号付き整数として比較します。
* Trap If Greater Than or Equal Unsigned	TGEU *rs, rt*	*rs ≥ rt* ならば、Trap 例外を発生します。*rs* と *rt* を符号なし整数として比較します。
* Trap If Less Than	TLT *rs, rt*	*rs < rt* ならば、Trap 例外を発生します。*rs* と *rt* を符号付き整数として比較します。
* Trap If Less Than Immediate	TLTI *rs, immediate*	16 ビット *immediate* を符号拡張し、*rs < immediate* ならば、Trap 例外を発生します。*rs* と *immediate* を符号付き整数として比較します。
* Trap If Less Than Immediate Unsigned	TLTIU *rs, immediate*	16 ビット *immediate* を符号拡張し、*rs < immediate* ならば、Trap 例外を発生します。*rs* と *immediate* を符号なし整数として比較します。
* Trap If Less Than Unsigned	TLTU *rs, rt*	*rs < rt* ならば、Trap 例外を発生します。*rs* と *rt* を符号なし整数として比較します。
* Trap If Not Equal	TNE *rs, rt*	*rs ≠ rt* ならば、Trap 例外を発生します。
* Trap If Not Equal Immediate	TNEI *rs, immediate*	16 ビット *immediate* を符号拡張し、*rs ≠ immediate* の場合は、Trap 例外を発生します。*rs* と *immediate* を符号付き整数として比較します。

*　TX19 → TX19A で追加された命令

第4章　16 ビット ISA の概要

この章では、16 ビット ISA の命令とアドレッシングモードの概要を説明します。また、16 ビット ISA 命令を使った基本的なプログラミングについても説明します。16 ビット ISA の命令は以下のように分類されます。分岐ライクリ命令は 16 ビット ISA ではサポートされていません。

- ロード命令・ストア命令
- 演算命令
- ジャンプ命令・分岐命令
- ビット操作命令
- SAVE・RESTORE 命令
- システム制御コプロセッサ CP0 命令
- 特殊命令

MIPS16ASE のダブルワード命令は、TX19A では使用できません。

16 ビット ISA では、32 本の汎用レジスタのうち、通常、r2~r7、r16、r17 の 8 本のレジスタしか使用できません。ただし、CPU 自体には、32 本の汎用レジスタが含まれているので、16 ビット ISA には、通常アクセスできる 8 本のレジスタと 32 本のレジスタ間でデータを転送するための MOVE 命令があります。また、命令によっては、r24 (t8)、r28 (gp)、r29 (sp)、r30 (fp)、r31 (ra) を暗黙的に使用しています。r24 は比較結果を格納するレジスタで、16 ビット ISA では t8（コンディションコードレジスタ）と呼ばれています。r28 はグローバルポインタとして、r29 はスタックポインタとして、r30 はフレームポインタとして、r31 はリンクレジスタとして使われます。また、乗算・除算命令は特殊レジスタの HI レジスタと LO レジスタを使います。

4.1　命令形式

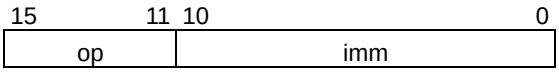
16 ビット長の命令には基本的に 図 4-1に示す 21 種類の命令形式があります。32 ビット長の命令は、図 4-2に示す 20 種類の命令形式があります。

16 ビット命令では、命令長が 16 ビットしかないため、即値フィールドは 3~11 ビットに制約されます。ただし、16 ビットISAには、これを補うための拡張命令があります。各命令形式で割り当てられているフィールドを超える即値を指定すると、アセンブラにより、その命令のまえにEXTEND命令が自動的に付加されます。
EXTEND命令はオペコードと即値のみで構成される命令と即値の部分がオペコードとして割り当てられている命令とあり、それ自体だけでは機械語命令は生成されません。即値フィールドを持っているEXTEND命令は、直後の命令の即値フィールドに連結することにより、32 ビットISAと同様の最大 16 ビットの即値を扱うことができるようになります。この場合、EXTENDが付加された命令（拡張命令）は全体で 32 ビットになり、図 4-2に示す命令形式となります。例えば、I形式の命令を拡張した命令はEXT-I形式になります。

16 ビット命令

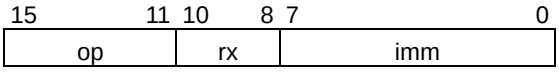
op	5 ビットの命令コード
rx	3 ビットのソースレジスタ番号・デスティネーションレジスタ番号
ry	3 ビットのソースレジスタ番号・デスティネーションレジスタ番号
immediate または imm または ximm3	3、4、5、8、11 ビットの即値、分岐オフセット値またはアドレスのオフセット値
rz	3 ビットのソースレジスタ番号・デスティネーションレジスタ番号
F	1、2、3、5 ビットの機能コード
r32	32 ビットの ISA 汎用レジスタ番号
ra	r31 レジスタ
s0	r16 レジスタ
s1	r17 レジスタ
pos3	メモリデータのビット番号
cpr32	コプロセッサレジスタ
hase	fp、sp、gp、r0 レジスタ
xsregs	退避・復元するレジスタの指定
aregs	退避・復元するレジスタの指定
framesize	領域確保のためのサイズ

I 形式



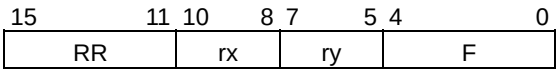
op:　B

RI 形式

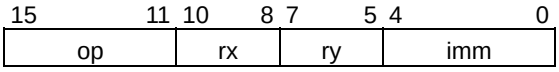


op:　ADDIU8・ADDIUPC・ADDIUSP・BEQZ・BNEZ・CMPI・LI・LWPC・LWSP・SLTI・SLTIU SWSP

RR 形式

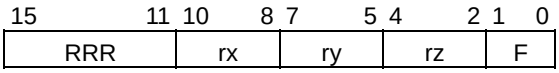


RRI 形式

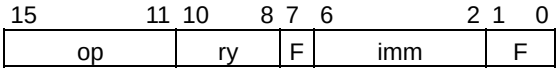


op:　LB・LBU・LH・LHU・LW・SB・SH・SW

RRR 形式①

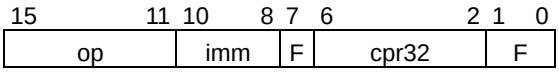


RRR 形式②



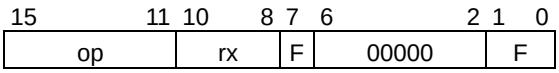
op:　SLL・SRL・SRA

RRR 形式③



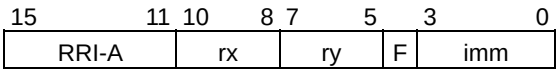
op:　AC0IU

RRR 形式④

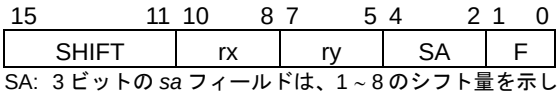


op:　MTHI・MTLO

RRI-A 形式

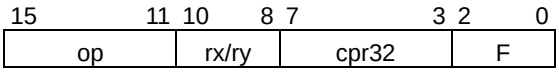


SHIFT 形式①



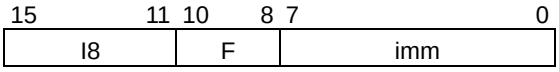
SA: 3 ビットの *sa* フィールドは、1 ~ 8 のシフト量を示します。16 ビット ISA では、
8 ビットのシフトのとき 0 を設定します。

SHIFT 形式②



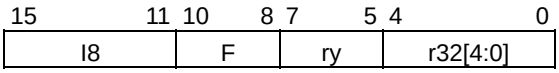
op: MTC0・MFC0

I8 形式

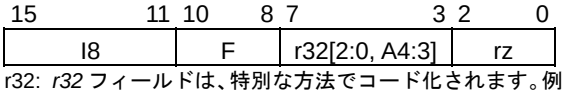


F: BTEQZ・BTNEZ・SWRASP・ADJSP・MOV32R・MOVR32・ADJFP

I8_MOVR32 形式

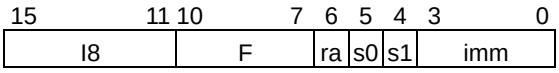


I8_MOV32R 形式

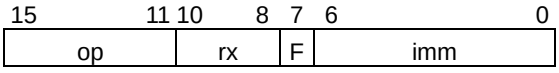


r32: *r32* フィールドは、特別な方法でコード化されます。例えば、レジスタ r7 (00111)をコード化すると、
r32 フィールドでは 11100 になります。

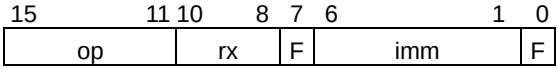
I8_SVRS 形式



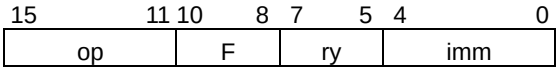
FP-B、SP-B 形式



FP-SP-H 形式

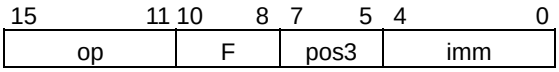


SPECIAL_SWFP、
SPECIAL_LWFP 形式



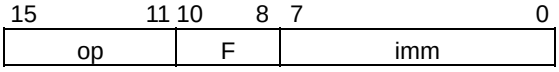
op: SWFP・LWFP

SPECIAL_BIT 形式



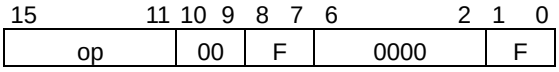
op: BTST・BEXT・BCLR・BSET・BINS

SPECIAL_BAL 形式



op: BAL

RRR_INT 形式

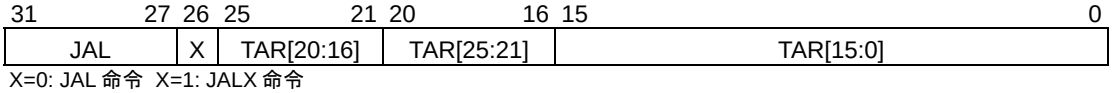


op: EI・DI

図 4-1　16 ビット命令形式

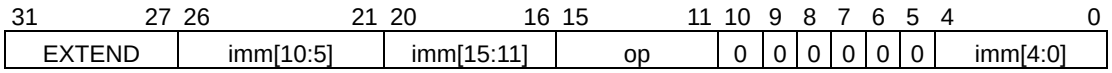
32 ビット命令

JAL・JALX 形式

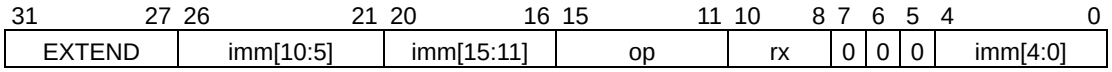


X=0: JAL 命令　X=1: JALX 命令

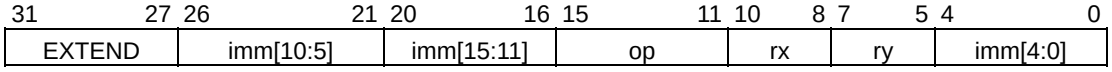
EXT-I 形式



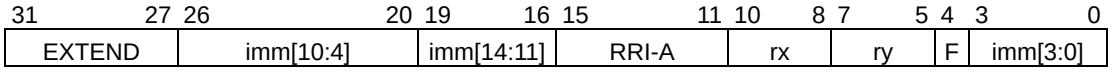
EXT-RI 形式



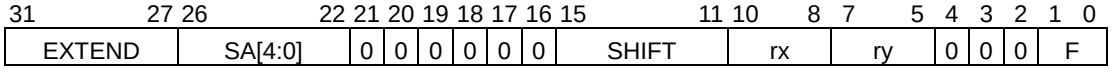
EXT-RRI 形式



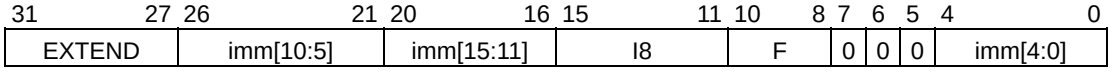
EXT-RRI-A 形式



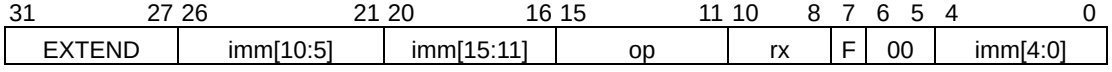
EXT-SHIFT 形式



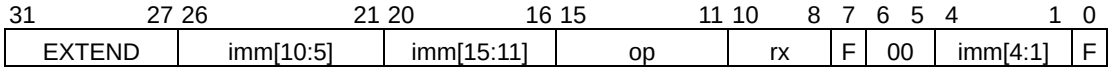
EXT-I8 形式



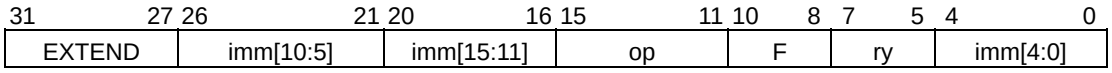
EXT-FP-B、EXT-SP-B 形式



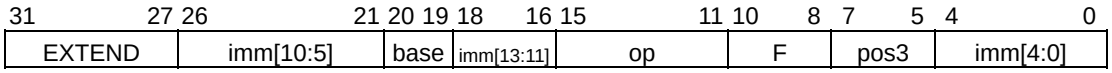
EXT-FP-SP-H 形式



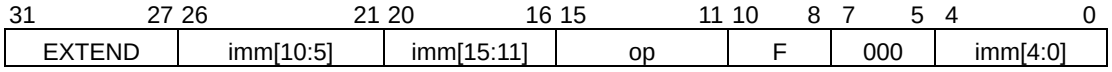
EXT-SPECIAL-SWFP、EXT-SPECIAL-LWFP 形式



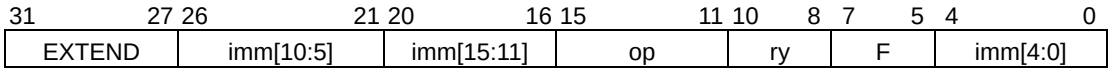
EXT-SPECIAL-BIT 形式



EXT-SPECIAL- BAL 形式

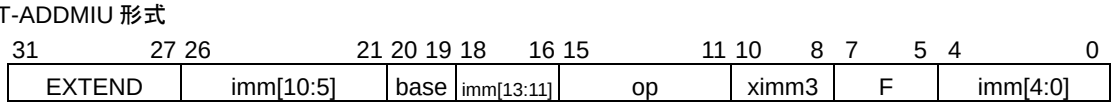


EXT-ADDIU8 形式



op:　ANDI・ORI・XORI・LUI

EXT-ADDMIU 形式



EXT-I8-SVRS 形式

31–27	26–24	23–20	19–16	15–11	10–8	7	6	5	4	3–0
EXTEND	xsregs	framesize	aregs	I8	SVRS	F	ra	s0	s1	framesize

EXT-RR 形式

31–27	26	25	24–16	15–11	10–5	4–0
EXTEND	0	1	000000000	11101	000000	op2

op2: ERET・DERET・WAIT

EXT-RR-SYSCALL 形式

31–27	26–22	21–16	15–11	10–5	4–0
EXTEND	imm[10:6]	imm[16:11]	11101	000000	01100

EXT-RR-BSIF 形式

31–27	26	25–16	15–11	10–8	7–5	4–0
EXTEND	1	0000000000	op	ry	rx	00111

EXT-RR-BFINS 形式

31–27	26	25–21	20–16	15–11	10–8	7–5	4–0
EXTEND	0	bit2	bit1	op	ry	rx	00111

EXT-RR-MAX/MIN 形式

31–27	26	25–24	23–19	18–16	15–11	10–8	7–5	4–0
EXTEND	M	00	00000	ry	op	rz	rx	00101

M=0: MAX 命令、M=1: MIN 命令

図 4-2　32 ビット命令形式

4.2　ロード・ストア命令

16 ビット ISA には、位置合わせされていないデータのロード・ストア命令はありません。16 ビット ISA のロード・ストア命令では、表現できる即値 (オフセット) の大きさが、符号なしの 5～8 ビットに制限されます。この制限を越えるオフセットを指定すると、オフセットフィールドは EXTEND 命令により、符号付きの 16 ビットに拡張されます。EXTEND 命令については、「4.5 特殊命令」を参照してください。また、16 ビット ISA にはコード中に埋め込まれた 32 ビットの定数をロードするためのアドレッシングモードが追加されています。

4.2.1項では 16 ビットロード・ストア命令でサポートされているアドレッシングモードについて説明します。
4.2.2項では、ロード・ストア命令の概要を説明します。
4.2.3項では、TX19Aで新しく追加されたアドレッシングモードを使って 32 ビットのアドレスを取得する方法を説明します。
4.2.4項では、SYNC命令について説明します。

4.2.1　アドレッシングモード

16 ビット ISA のロード・ストア命令では、以下の 4 つのアドレッシングモードがサポートされています。

- ◆ オフセット付きレジスタ間接アドレッシングモード
- ◆ オフセット付き SP 相対アドレッシングモード
- ◆ オフセット付き FP 相対アドレッシングモード
- ◆ オフセット付き PC 相対アドレッシングモード

■　**オフセット付きレジスタ間接アドレッシングモード**

16 ビット ISA では、ほとんどのロード・ストア命令はオフセット付きレジスタ間接アドレッシングモードを使います。命令形式は RRI (レジスタ・レジスタ・即値) 形式になります。このアドレッシングモードを使う命令は、ベースレジスタと符号なし 5 ビットオフセットフィールドをもっています。実効アドレスは、ベースレジスタとして指定した汎用レジスタの値に、5 ビットのオフセット値をゼロ拡張した値を加算することにより生成されます。ベースレジスタには、16 ビット ISA で通常アクセス可能な汎用レジスタ (r2～r7、r16、r17) ならどれでも使用できます。16 ビット ISA では、少しでもオフセットの範囲を広くとれるように、オフセット値は、ロードまたはストアするデータ形式にあわせて、左にシフトされます。すなわち、ワードアクセスの場合は、オフセットは 2 ビット左にシフトされ、ハーフワードアクセスの場合は、オフセットは 1 ビット左にシフトされます。

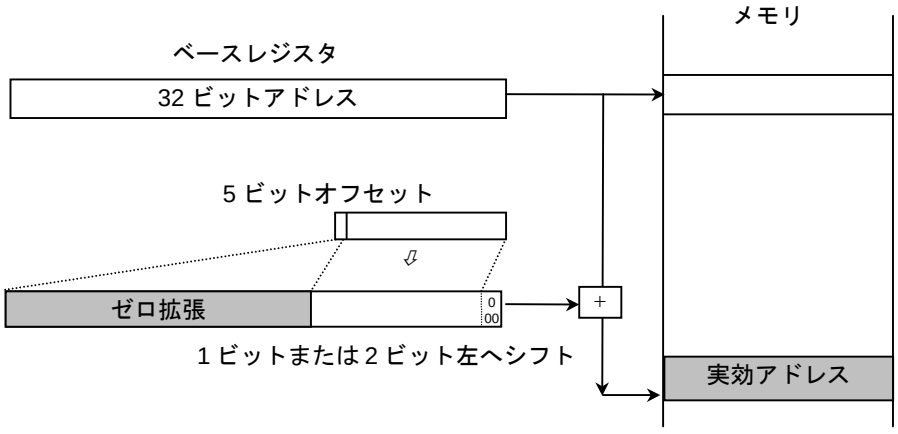


図 4-3　オフセット付きレジスタ間接アドレッシングモード (16 ビット ISA)

■　**オフセット付き SP 相対アドレス**

32 ビット ISA では、慣例的に r29 をスタックポインタレジスタとして使いますが、ハードウエア的には r0 以外の汎用レジスタならどのレジスタでも使うことができま

す。それに対して、16 ビット ISA では、r29 が常にスタックポインタとして使われます。r29 は別名で sp と呼ばれています。16 ビット ISA では、r29 を特殊な機能コードを使って暗黙的に参照しているため、命令コード中にベースレジスタフィールドがありません。このため、オフセットフィールドを 8 ビットとることができ、命令形式は RI (レジスタ・即値) 形式になります。実効アドレスは、スタックポインタレジスタ sp の内容に、8 ビットのオフセットを 2 ビット左へシフトし、ゼロ拡張した値を加算することにより生成されます (ワードアクセスの場合)。SP 相対アドレッシングモードは LBU、LHU、LW 命令と SB、SH、SW 命令で使用できます。これらの命令は、EXTEND 命令で拡張しなくても 1K バイト (2^{10}) の範囲までアドレッシングできます。

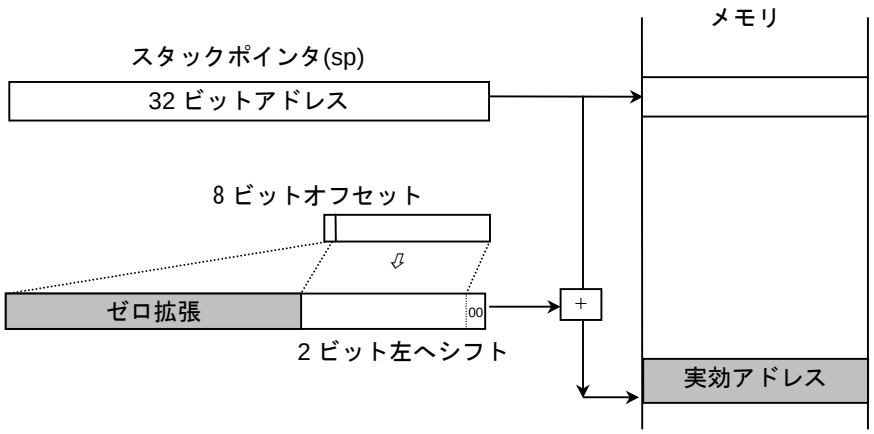


図 4-4　SP 相対アドレッシングモード (16 ビット ISA: ワードアクセス)

■　オフセット付き FP 相対アドレス

16 ビット ISA では、r30 を特殊な機能コードを使って暗黙的に参照しているため、命令コード中にベースレジスタフィールドがありません。このため、オフセットフィールドを 5 ビットとることができ、命令形式は RI (レジスタ・即値) 形式になります。実効アドレスは、フレームポインタ fp の内容に、5 ビットのオフセットを 2 ビット左へシフトし、ゼロ拡張した値を加算することにより生成されます (ワードアクセスの場合)。FP 相対アドレッシングモードは LBU、LHU、LW 命令と SB、SH、SW 命令で使用できます。これらの命令は、EXTEND 命令で拡張しなくても 128 バイト (27) の範囲までアドレッシングできます。

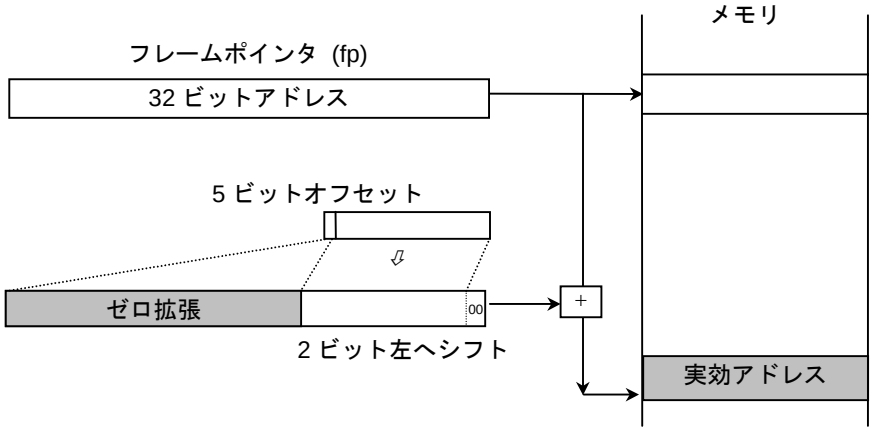


図 4-5　FP 相対アドレッシングモード (16 ビット ISA: ワードアクセス)

■　オフセット付き PC 相対アドレッシングモード

オフセット付き PC 相対は、LW (Load Word) 命令でサポートされているアドレッシングモードです。実効アドレスは、8 ビットのオフセットを 2 ビット左へシフトし、ゼロ拡張した値を、下位 2 ビットを 0 にマスクした PC の値に加算することにより生成されます。実効アドレスのメモリ位置に格納されている 32 ビットの定数が、レジスタにロードされます。これにより、32 ビットの定数を、コード中に埋めこむことができます。

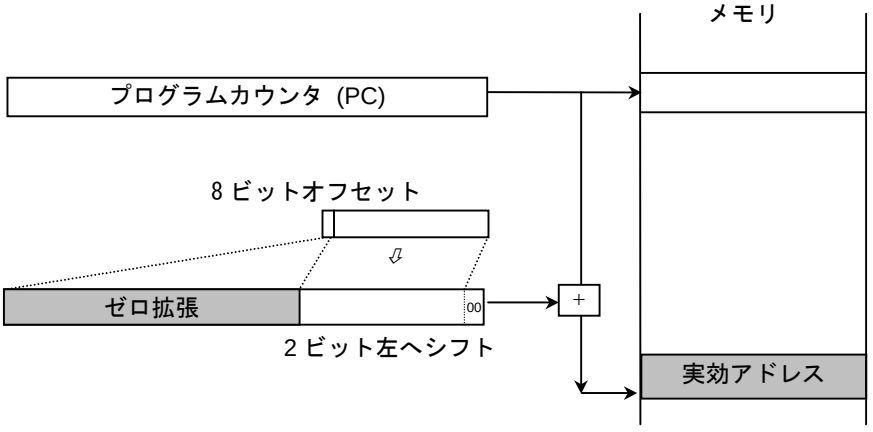


図 4-6　オフセット付き PC 相対アドレッシングモード (16 ビット ISA)

4.2.2　ロード・ストア命令の概要

表 4-1と 表 4-2にバイトアクセス、ハーフワードアクセス、ワードアクセス用のロード・ストア命令を示します。LB、LH命令では、ロードしたバイトまたはハーフワードは符号拡張されて、レジスタに格納されます。LBU、LHU命令では、ロードしたバイトまたはハーフワードはゼロ拡張されて、レジスタに格納されます。

表 4-1　ロード命令

データ形式	符号なしロード	符号付きロード	アドレッシング
バイト	LBU	LB	レジスタ間接/SP/FP 相対
ハーフワード	LHU	LH	レジスタ間接/SP/FP 相対
ワード	LW	—	レジスタ間接/SP/FP/PC 相対

表 4-2　ストア命令

データ形式	符号なしロード	アドレッシング
バイト	SB	レジスタ間接/SP/FP 相対
ハーフワード	SH	レジスタ間接/SP/FP 相対
ワード	SW	レジスタ間接/SP/FP 相対

4.2.3　32 ビットのアドレスの生成

16 ビット ISA のロード・ストア命令では、オフセットフィールドは 5 ~ 8 ビットしかありません。EXTEND 命令により拡張すると、32 ビット ISA と同様の符号付きの 16 ビットのオフセット値 (–32768 ~ +32767) が扱えるようになります。ただし、オフセットがこの範囲外の場合は、オフセットをいったん汎用レジスタに格納する必要があります。ワードロードの場合は、オフセット付き PC 相対アドレッシングモードを使用できます。3 つの例を以下に示します。

◆　例 1　ベースアドレス + 32 ビットオフセット
以下の命令では、ADDU (Add Unsigned) 命令によりレジスタ r5 に格納されているオフセットをレジスタ r4 のベースアドレスに加算し、その結果を r4 に書き戻しています。次に LW 命令で、r4 をベースレジスタとして指定しています。

```
ADDU    r4,r4,r5
LW      r6,0(r4)
```

◆ 例 2　ベースアドレス+ 32 ビットオフセット

3 章で説明したように、オフセット値が 16 ビット以上の場合、32 ビット ISA では、LUI (Load Upper Immediate) 命令を使って、指定した 16 ビットの即値をレジスタの上位 16 ビットにロードし、次に論理和命令により下位 16 ビットを連結します。TX19 では、16 ビット ISA に LUI 命令がなかったため PC 相対アドレッシングモードを使ってプログラムしていました。TX19A では、LUI 命令、ORI 命令がサポートされたので、32 ビット ISA と同様なプログラムで実行することができます。

– TX19 コード効率優先		– TX19A	
LW	r5,16(pc)	LUI	r5,0x0008
ADDU	r4,r4,r5	ORI	r5,0x0234
LW	r6,0(r4)	ADDU	r4,r4,r5
		LW	r6,0(r4)

◆ 例 3　任意の 32 ビットの絶対アドレス指定

以下の例で、最初の LW 命令は、PC 相対アドレッシングモードを使ってメモリから 32 ビットの絶対アドレスをロードしています。次の LW 命令では、r4 をベースレジスタとして使えば、オフセットをゼロとすることができます。

```
LW      r4,16(pc)
LW      r6,0(r4)
```

その他に LUI 命令と ORI 命令を使って絶対アドレスを指定することもできます。

```
LUI     r4,0x0008
ORI     r4,0x0234
LW      r6,0(r4)
```

4.2.4　SYNC 命令

SYNC 命令は、SYNC 命令の直前に実行したロード、ストア、命令フェッチまで命令パイプラインをインタロックし、後続のロード、ストアの実行を遅らせます。これにより SYNC 命令の前の命令と後続の命令の実行順序を守ることができます。

4.3　演算命令

この項では、16 ビットISAの演算命令について説明します。
4.3.1項では演算命令の分類と、TX19Aで追加された命令について説明します。
4.3.2項では 32 ビット定数を使用する演算について説明します。
64 ビットの加算、減算、ローテートについては、32 ビットISAと 16 ビットISAで同じ手法を使えるので、「3 章　32 ビットISA概要」を参照してください。

4.3.1　演算命令の分類

16 ビットISAの演算命令は、表 4-3に示す 4 つのグループに分類されます。演算命令には、算術、比較、論理、シフト、乗算、除算命令、積和命令があります。積差演算命令は、16 ビットISAにはありません。また、TX19Aの 16 ビットISAは、ダブルワード命令をサポートしていません。

表 4-3　演算命令

分類	命令	オペコード
ALU 即値	加算	ADDIU
	大小比較	SLTI・SLTIU
	一致比較	CMPI
	即値のロード	LI・LUI
	論理積	ANDI
	論理和	ORI
	排他的論理和	XORI
2/3 オペランド レジスタタイプ	加算	ADDU
	減算	SUBU
	飽和	SADD・SSUB
	大小比較	SLT・SLTU
	一致比較	CMP
	否定	NEG
	論理積	AND
	論理和	OR
	排他的論理和	XOR
	反転	NOT
	移動	MOVE
	ビットサーチ	BS1F
	ビットフィールド	BFINS
	最大/最小	MAX・MIN
	符号拡張/ゼロ拡張	SEB・SEH・ZEB・ZEH
シフト	論理シフト	SLL・SLLV・SRL・SRLV
	算術シフト	SRA・SRAV
乗算・除算	乗算/積和	MULT・MULTU・MADD・MADDU
	除算	DIV・DIVU・DIVE・DIVEU
	HI・LO レジスタと汎用レジスタ間の転送	MFHI・MFLO・MTHI・MTLO

ALU 即値命令のソースオペランドは、汎用レジスタと 4 ビットまたは 8 ビットの即値です。TX19A では、新規に ANDI、ORI、XORI、LUI 命令を追加していますが、EXTEND 命令によって拡張された命令コードしか存在しないため、即値は 16 ビットのみで、ゼロ拡張されて 32 ビットの符号なしデータとして扱われます (LUI 命令は除く)。ADDIU、LUI 命令を除き、ALU 即値命令の 8 ビットの即値はゼロ拡張されます。ただし、EXTEND 命令により拡張された場合は、ADDIU、SLTI、SLTIU 命令は 32 ビット ISA と同様に符号付き 16 ビットの即値として扱われます。その他の ALU 即値命令は、符号なし 16 ビットの即値として扱われます。

レジスタタイプ命令は、汎用レジスタに格納されている値を対象に演算をし、その結果を汎用レジスタに格納します。16 ビット ISA には、CMP、NEG、NOT 命令があります。CMP 命令は 2 つのレジスタの値を比較する命令です。NEG 命令はレジスタ値の 2 の補数をとる命令です。NOT 命令はレジスタ値の 1 の補数をとる命令です。また、16 ビット ISA には、通常使用できる 8 本のレジスタ+fp レジスタと、32 本のレジスタの間で値を移送するための MOVE 命令が用意されています。

CMP (Compare)、NEG (Negate)、NOT (Not) 命令のオペレーションは、32 ビット ISA では、ソースレジスタに r0 を使うことで他の命令で実現できます。ところが、16 ビット ISA では r0 を使えないため個別の命令として追加されています。一致比較命令 (CMP、CMPI) と大小比較命令 (SLTI、SLTIU、SLT、SLTU) ではデスティネーションレジスタとして t8 (r24) が暗黙的に使われます。

16 ビット ISA には、32 ビット ISA と同じ種類のシフト命令がありますが、TX19 では 16 ビット ISA での *sa* フィールドは 3 ビットで、シフト量は 1 ~ 8 しか指定できませんでした (000 は、8 ビットのシフトとして定義されています)。TX19A では、*sa* フィールドが 5 ビットまであり、シフト量は 1 ~ 31 まで指定できる命令を追加しました (この追加命令では、00000 は定義することができません)。シフト命令が EXTEND 命令により拡張された場合は、*sa* フィールドは 32 ビット ISA と同様の 5 ビットになります。

TX19A では、16 ビット ISA で 32 ビット ISA と同様の乗除算、積和命令を持っています。乗算、積和命令は、積の下位 32 ビットを汎用レジスタにロードする機能があります。また、HI・LO レジスタをアクセスする MTHI、MTLO、MFHI、MFLO 命令もあります。

TX19A では、オーバフローを検出する除算命令 (DIVE、DIVEU) が追加されました。符号付き除算命令では 0 除算もしくは、演算結果がオーバフローした場合はオーバフロー例外が発生します。符号なし除算命令では、0 除算を実行したときのみオーバフロー例外が発生します。

TX19A では、バイトデータ、ハーフワードデータを 32 ビットのデータにゼロ拡張したり、符号拡張したりする ZEB、ZEH、SEB、SEH 命令が追加されました。

TX19A では、飽和命令 (SADD、SSUB) が追加されました。SADD 命令の場合であれば、レジスタ rx とレジスタ ry の内容を加算し、もしオーバフローした場合レジスタ rx ≥ 0 のときは 0x7FFF_FFFF を、レジスタ rx < 0 のときは 0x8000_0000 をレジスタ ry に格納します。オーバフローしなかった場合は加算した結果をレジスタ ry に格納します。

TX19A では、レジスタとレジスタの大小関係を比較する MIN・MAX 命令が追加されました。MIN 命令の場合であれば、レジスタ rx とレジスタ ry の内容を符号付き整数として比較し、レジスタ rx が小さい場合はレジスタ rx をレジスタ rz に格納し、それ以外の場合は、レジスタ ry をレジスタ rz に格納します。

TX19A では、C コンパイラのコード効率を上げるためにビットフィールド命令 (BFINS) を追加しました。C 言語プログラムでは、ビットフィールドのデータを扱うことがあり、この命令を追加したことにより 1 命令でデータを転送することが可能です。この他に組み込み制御系のキースキャンなどのビットデータの検索処理に便利なビットサーチ命令 (BS1F) を追加しました。

4.3.2　32 ビットの定数

TX19 では、EXTEND 命令を使っても演算命令の即値フィールドは 16 ビットまでしか拡張できませんでした。TX19A では、LUI 命令、ORI 命令を追加したことで、32 ビット ISA と同じ方法で 32 ビットの定数を扱うことができます。

汎用レジスタの内容に 32 ビットの定数 (0x8000_1234) を加算する例を以下に示します。

```
LUI     r5,0x8000
ORI     r5,0x1234
ADDU    r6,r6,r5
```

もし、コード効率を優先するならば従来と同じ方法で 32 ビットの定数を扱います。32 ビットの定数は、コード中に、通常はサブルーチンとサブルーチンの間に埋め込み、LW 命令で PC 相対アドレッシングモードを使って参照します。定数を格納しておくエリアを考慮しても、LUI 命令と ORI 命令で必要とされるコードサイズよりもコードサイズが小さくなります。

以下の例では、LW 命令で 32 ビットの定数をメモリから rs にロードしています。そして ADDU 命令で r4 と r5 の内容を加算して、その結果を r6 に格納しています。

```
LW      r5,offset(PC)
ADDU    r6,r4,r5
```

■　ゼロ値

通常、16 ビット ISA の命令は、r0 に直接アクセスすることはできません。そこで、ゼロ値が必要なときは、LI (Load Immediate) 命令を以下のように使います。この命令は即値 (0) をゼロ拡張し、*rx* に格納します。

```
LI rx,0
```

また、LI 命令の代わりに、MOVE 命令を使ってゼロ値を得ることができます。MOVE 命令は、16 ビット ISA で通常アクセスできる 8 本のレジスタと 32 本のレジスタの間でデータを移送できるので、以下の命令でゼロ値を得ることができます。

```
MOVE ry,r0
```

4.4　ジャンプ・分岐命令

この項では、16 ビットISAで使用できるジャンプ・分岐命令について、32 ビット命令との違いを中心に説明します。
4.4.1項では、ジャンプ・分岐命令の概要を説明します。
4.4.2項では、大小関係に基づき分岐する方法について説明します。
4.4.3項では、32 ビットの絶対アドレスへジャンプする方法について説明します。

4.4.1　ジャンプ・分岐命令の概要

16 ビット ISA には、BEQ、BNE、BGEZ、BGTZ、BLEZ、BLTZ のような 2 値を比較し、分岐する命令がありません。これらの命令がない代わりに、16 ビット ISA には、2 つのレジスタ値、またはレジスタ値と即値が等しいかどうかを調べる一致比較命令 (CMP、CMPI) があります。これらの比較命令は 2 値の排他的 OR をとって、その結果をレジスタ t8 に格納します。つまり、2 値が等しい場合、t8 は 0 に設定されます。また、同様に大小比較命令 (SLT・SLTI・SLTIU・SLTU) でも、比較結果が t8 に設定されます。そこで、16 ビット ISA には、t8 の内容を調べ t8 が 0 かどうかによって分岐する分岐命令が用意されています。TX19A では、新規にリンク機能付きの分岐命令 (BAL 命令) が用意されました。

16 ビット ISA でも、ジャンプ先のアドレスの範囲を広くとれるように、JAL・JALX 命令だけは例外として 32 ビットの命令になっています。

表 4-4と 表 4-5に 16 ビットISAのジャンプ、分岐命令を示します。

表 4-4　ジャンプ命令 (16 ビット ISA)

オペコード	命令	アドレッシング
JAL	Jump And Link	ページ内絶対
JALX	Jump And Link Exchange	ページ内絶対
JR	Jump Register	レジスタ間接
JRC	Jump Register, Compact	レジスタ間接
JALR	Jump And Link Register	レジスタ間接
JALRC	Jump And Link Register, Compact	レジスタ間接

表 4-5　分岐・分岐ライクリ命令 (16 ビット ISA)

オペコード	命令	条件	アドレッシング
BEQZ	Branch On Equal to Zero	$rx = 0$	PC 相対
BNEZ	Branch On Not Equal Zero	$rx \neq 0$	PC 相対
BTEQZ	Branch On T8 Equal To Zero	$t8 > 0$	PC 相対
BTNEZ	Branch On T8 Not Equal To Zero	$t8 \neq 0$	PC 相対
B	Unconditional Branch	—	PC 相対
BAL	Branch And Link	—	PC 相対

リンク機能付きジャンプ命令または BAL 命令では、レジスタ r31 に戻りアドレスが格納されます。これらの命令は、サブルーチンコールで使われます。

16 ビット ISA の分岐命令は、32 ビット ISA と同じアドレッシングモードを使います。ただし、16 ビット命令はハーフワード境界にそろえられるため、アドレスのオフセット値は 2 ビットではなく、1 ビットのみのシフトになります。また、オフセット値は 8 ビットあるいは 11 ビットになります。

■　分岐遅延スロット

16 ビット ISA には、分岐遅延スロットはありません。分岐命令は常に後続の命令の前に実行されます。条件が成立して分岐した場合は、直後に置かれた命令は廃棄され、実行されません。そのため、分岐命令の直後に置く命令に制限がありません。

ジャンプ命令は、32 ビット ISA 同様、16 ビット ISA でも遅延スロットがありますが、TX19A で新規に追加した JRC、JALRC 命令には遅延スロットはありません。

■　ISA モードの実行中の切り換え

表 4-1に示したように、16 ビットISAには、32 ビットISAと同様にJALX、JR、JALR、JRC、JALRC命令があります。これらの命令によりプログラムカウンタ (PC) のISAモードビットを操作し、ISAモードを切り換えることができます。詳しくは、「3.4.3 ISAモードの切り換え」を参照してください。

■　サブルーチンコール

16 ビット ISA には、リンク機能付きジャンプ命令 (JALX・JALR)、リンク機能付き分岐命令 (BAL) があります。サブルーチンコールついては、「3.4.7 サブルーチンコール」を参照してください。

4.4.2　大小関係に基づく分岐

前項で説明したように、16 ビット ISA には「BEQ r10, r7, Equal」のように 2 つのレジスタの値を比較して、分岐する命令がありません。また、16 ビット ISA の大小比較命令 (SLT・SLTU) では、オペランドとしてレジスタを 2 つしか指定できません。16 ビット ISA の、SLT、SLTU 命令は、2 つのレジスタの値が等しいがどうかによって、暗黙的にレジスタ t8 を設定します。そのため 16 ビット ISA には、t8 レジスタが 0 かどうかを調べる命令 (BTEQZ・BTNEZ) があります。

「3.4.5　大小関係に基づく分岐」で説明したように、32 ビット ISA モードでは、ORI 命令と BEQ 命令 (または BNE 命令) を組み合わせて使うことにより、レジスタの内容と即値を比較しました。

```
ORI     r10,r0,0x1234
BEQ     r10,r7,Label
```

TX19A では､LUI 命令､ORI 命令といった即値を取り扱う命令が用意されたので、32 ビット ISA と同様なプログラミングが可能です。この他に 16 ビット ISA には CMPI 命令が用意されています。この命令は、レジスタの内容と即値を比較し、等しいかどうかに基づき t8 を設定します。

16 ビット ISA での比較・分岐の 3 つの例を以下に示します。

◆　例 1　r6 ≥ r7 の場合の分岐
以下に、r6 の値が r7 の値以上の場合に、分岐する例を示します。r6 が r7 より小さいと、SLT (Set On Less Than) 命令により t8 が 1 に設定されます。r6 が r7 以上の場合は、r8 は 0 に設定されます。t8 の値が 0 の場合、BTEQZ 命令で Label に分岐します。

```
SLT     r6,r7
BTEQZ   Label
```

◆　例 2　r7 ≥ 0x1234 の場合の分岐
以下に、r7 の値が 0x1234 以上の場合に分岐する例を示します。例 1 と同様、SLTI (Set On Less Than Immediate) 命令により、r7 の内容と 0x1234 の大小関係に基づき､t8 が暗黙的に設定されます｡t8 が 0 の場合､BTEQZ 命令は Label に分岐します。

```
SLTI    r7,0x1234
BTEQZ   Label
```

◆ 例 3　r7 = 0x1234 の場合の分岐
以下に、レジスタの値と即値が等しいかどうか調べて、それに基づき分岐する例を示します。この例では、CMPI (Compare Immediate) 命令は、r7 の値と 0x1234 を比較し、等しければ t8 を 0 に設定します。(CMPI は、実際には 2 値の排他的 OR をとっています。)

```
CMPI    r7,0x1234
BTEQZ   Label
```

4.4.3　32 ビットのアドレスへのジャンプ

TX19A では、LUI 命令および ORI 命令が追加されたので 32 ビット ISA と同様に LUI 命令と ORI 命令の組み合わせで 32 ビットのアドレスを生成することができます。コード効率を優先する場合は、従来どおりの PC 相対アドレッシングモードにより、LW 命令でメモリから 32 ビットの定数をロードしてください。

– コード効率優先

```
LW      r4,0(pc)
JR      r4
```

– 実行速度優先

```
LUI     r4,0x0008
ORI     r4,0x0234
JR      r4
```

また、PC 相対アドレスを計算し、その結果をレジスタに格納するための命令 (ADDIU, *rx,* pc, *immediate*) も用意されています。

4.5　ビット操作命令

TX19A では、新規にビット操作命令を追加しました。ビット操作命令は、メモリのビットデータに対して操作をする命令です。TX19 では、複数の命令を使ってメモリのビットデータを操作していましたが、この期間は割り込みを受け付けられないようにするための命令なども必要に応じて実行しなければなりませんでした。TX19A では、1 命令で実行が可能となり、コード効率、実行速度の面で改善されています。

表 4-6　ビット動作命令

ニーモニック	命令	デスティネーション
BTST	Bit Test	t8 レジスタ
BEXT	Bit Extract	t8 レジスタ
BCLR	Bit Clear	メモリ
BSET	Bit Set	メモリ
BINS	Bit Insert	メモリ
ADDMIU	Add Immediate to Memory Word	メモリ

4.6　SAVE・RESTORE 命令

TX19A では、新規に 1 命令で複数のレジスタに対してメモリからデータを復帰させたり、メモリへデータを退避させたりする SAVE、RESTORE 命令を追加しました。TX19 では、複数の命令によってレジスタの転送を実行していましたが、これらの命令により 1 命令で実行が可能になりました。それによりコード効率が改善されています。

表 4-7　SAVE・RESTORE 命令

ニーモニック	命令	対象レジスタ
SAVE	Save Registers and Set up Stack Frame	r4-r7, r16, r17, r18-r23, r30, r31
RESTORE	Restore Registers and Deal locate Stack Frame	r4-r7, r16, r17, r18-r23, r30, r31

4.7　システム制御コプロセッサ CP0 命令

TX19A では、新規にコプロセッサレジスタをアクセスできるシステム制御コプロセッサ CP0 命令を追加しました。TX19 ではなかった命令で、コプロセッサレジスタをアクセスするために JALX 命令などを使って 32 ビット ISA モードに切替えていました。

ただし、IER レジスタ、Config1 レジスタ、Config2 レジスタ、Config3 レジスタは、16 ビット ISA ではアクセスすることができません。Status レジスタの IE ビットを操作する場合、32 ビット ISA では IER レジスタに対して 0 もしくは 0 以外の値をセットすることで、IE ビットを操作することができました。16 ビット ISA では、IER レジスタをアクセスすることができないため、その代わりに IE ビットに対してセット/クリアする EI/DI 命令を追加しました。

表 4-8　システムコントロールコプロセッサ (CP0) 命令

ニーモニック	命令
MFC0	Move from Coprocessor 0
MTC0	Move to Coprocessor 0
AC0IU	Add Coprocessor 0 Immediate Unsigned.

4.8　特殊命令

特殊命令には、BREAK（Breakpoint）命令と SDBBP（Software Debug Breakpoint）命令と、さらに TX19A で追加した EI/DI（割り込み許可/禁止）命令、SYSCALL（SystemCall）命令、ERET 命令、DERET 命令、WAIT 命令があります。

また、16 ビットISAにはEXTEND命令があります。EXTEND命令は、それだけでは機械語の命令を生成しません。EXTEND命令は、5 ビットのオペコードと 11 ビットの即値だけで構成されており、自身の即値を後続の命令の即値に連結する場合（表 4-9）と、TX19Aで追加された一部の命令は 11 ビットの即値の部分をオペコードとして使用している命令もあります。後者の命令に該当するのが、SYNC、ERET、DERET、WAIT、BS1F、MAX、MIN命令でこれらはすべてTX19Aで追加した命令です。これらの命令は、32 ビットの拡張命令しかありません。

表 4-9　拡張命令

16 ビット命令		即値ビットサイズ	
		拡張前	拡張後
ロード・ストア	LB・LBU	5 (または 7)	16
	LH・LHU	5 (または 6)	16
	LW	5 (または 8)	16
	SB	5 (または 7)	16
	SH	5 (または 6)	16
	SW	5 (または 8)	16
演算	ADDIU	4	15
		8	16
	SLTI・SLTIU	8	16
	CMPI	8	16
	LI	8	16
	LUI	—	16
	SLL	3	5
	SRL	3	5
	SRA	3	5
	ANDI	—	16
	ORI	—	16
	XORI	—	16
	LUI	—	16
	ADDMIU	—	14
分岐	BEQZ	8	16
	BNEZ	8	16
	BTEQZ	8	16
	BTNEZ	8	16
	B	11	16
	BAL	8	16

16 ビット命令		即値ビットサイズ	
		拡張前	拡張後
ビット操作	BTST	5	14
	BEXT	5	14
	BCLR	5	14
	BSET	5	14
	BINS	5	14
SAVE・RESTORE	SAVE	4	8
	RESTORE	4	8
ビットフィールド	BFINS	—	5

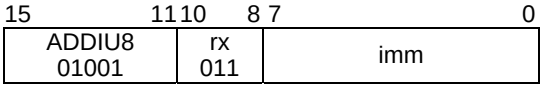
EXTEND 命令は、ワード境界から開始する必要はありません。EXTEND 命令には使用上 1 つだけ制約があり、ジャンプ遅延スロットに置いてはいけません。ジャンプ遅延スロットに置いた場合の動作は確定しません。

EXTEND 命令は、16 ビット命令の前に明示的に記述する必要はありません。即値フィールドをもつ 16 ビット ISA の命令で、即値フィールドで扱える範囲を超える値を指定すると、アセンブラにより自動的に EXTEND 命令を使って、即値が分解されます。例えば、以下のように記述したとします。

```
ADDIU r3,0x1234
```

この命令はRI形式で、扱える即値は、本来 表 4-9に示すように 8 ビットです。したがって、上記のADDIU命令はEXTEND命令により拡張され、32 ビットEXT-RI形式になります。

RI 形式 (即値)



EXT-RI 形式

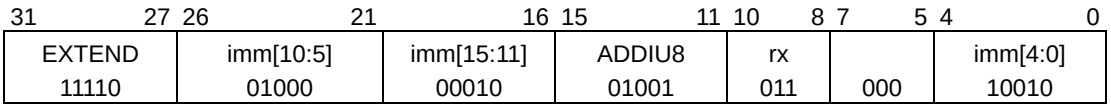


図 4-7　RI 形式と EXT-RI 形式

また、「ADDIU, *ry*, *rx*, *immediate*」という命令では即値フィールドは 4 ビットしかありません。EXTEND 命令の即値フィールドは 11 ビットしかないので、この命令だけは拡張しても、扱える即値は 32 ビット ISA と同様の 16 ビットにはならず、15 ビットまでです。

この他にEXTEND命令により拡張された場合でも、16 ビットの即値として扱えない命令もあります。SLL、SRL、SRA命令は 5 ビット、ビット操作命令は 14 ビット、SAVE・RESTORE命令は 8 ビット、BFINS命令は 5 ビットまでです (表 4-9参照)。

4.9　命令の概要

この項では、16 ビット ISA での命令の概要を分類ごとに示します。

■　命令表記規則

この項では、命令中 *rx*、*ry*、*rz*、*immediate*、*sa*（シフト量: shift amount）などの小文字の斜体で示してある部分には、ユーザーが任意のレジスタや値などを指定できます。オペランドの意味がより明確になるように、例えば、ロード命令とストア命令では、*rx*、*immediate* と書かずに *base*、*offset* と記述してあります。命令によっては、特定の目的のために r24 (t8)、r28 (gp)、r29 (sp)、r30 (fp)、r31 (ra) を使用します。これらのレジスタは、t8、gp、sp、fp、ra と示します。HI と LO は、整数の乗算・除算の結果を格納する特殊レジスタです。

■　TX19A で実現されていない命令

TX19A は、MIPS16eASE のダブルワード命令をサポートしていません。TX19 と TX19A と TX39 の比較は付録 D を参照してください。

表 4-10　ロード・ストア命令 (16 ビット ISA)

命令	形式	説明
Load Byte	LB　*ry*, *offset*(*base*)	5 ビット *offset* をゼロ拡張し、*base* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたバイトデータを符号拡張し、*ry* にロードします。
Load Byte Unsigned	LBU　*ry*, *offset*(*base*)	5 ビット *offset* をゼロ拡張し、*base* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたバイトデータをゼロ拡張し、*ry* にロードします。
	* LBU　*ry*, *offset*(*sp*)	7 ビット *offset* をゼロ拡張し、*sp* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたバイトデータをゼロ拡張し、*ry* にロードします。
	* LBU　*ry*, *offset*(*fp*)	7 ビット *offset* をゼロ拡張し、*fp* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたバイトデータをゼロ拡張し、*ry* にロードします。
Load Halfword	LH　*ry*, *offset*(*base*)	5 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、*base* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたハーフワードデータを符号拡張し、*ry* にロードします。
Load Halfword Unsigned	LHU　*ry*, *offset*(*base*)	5 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、*base* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたハーフワードデータをゼロ拡張し、*ry* にロードします。
	* LHU　*ry*, *offset*(*sp*)	6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、*sp* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたハーフワードデータをゼロ拡張し、*ry* にロードします。
	* LHU　*ry*, *offset*(*fp*)	6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、*fp* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたハーフワードデータをゼロ拡張し、*ry* にロードします。

*　TX19 → TX19A で追加された命令

命令	形式	説明
Load Word	LW *ry*, *offset*(*base*)	5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、*base* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたワードデータを *ry* にロードします。
	LW *rx*, *offset(pc)*	8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、マスクベース PC 値 (下位 2 ビットを 0 にマスクした PC 値) に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたワードデータを *rx* にロードします。
	LW *rx*, *offset(sp)*	8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、*sp* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたワードデータを *rx* にロードします。
	* LW *ry*, *offset(fp)*	5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、*fp* に加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたワードデータを *ry* にロードします。
Store Byte	SB *ry*, *offset*(*base*)	5 ビット *offset* をゼロ拡張し、*base* に加算した結果が実効アドレス (EA) になります。*ry* の最下位バイトをこのアドレスにストアします。
	* SB *ry*, *offset*(*sp*)	7 ビット *offset* をゼロ拡張し、*sp* に加算した結果が実効アドレス (EA) になります。*ry* の内容をこのアドレスに格納します。
	* SB *ry*, *offset*(*fp*)	7 ビット *offset* をゼロ拡張し、*fp* に加算した結果が実効アドレス (EA) になります。*ry* の内容をこのアドレスに格納します。
Store Halfword	SH *ry*, *offset*(*base*)	5 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、*base* に加算した結果がアドレス (EA) になります。*ry* の下位のハーフワードをこのアドレスに格納します。
	* SH *ry*, *offset*(*sp*)	6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、*sp* に加算した結果が実効アドレス (EA) になります。*ry* の下位のハーフワードをこのアドレスに格納します。
	* SH *ry*, *offset*(*fp*)	6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、*fp* に加算した結果が実効アドレス (EA) になります。*ry* の下位のハーフワードをこのアドレスに格納します。
Store Word	SW *ry*, *offset*(*base*)	5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、*base* に加算した結果が実効アドレス (EA) になります。*ry* の内容をこのアドレスに格納します。
	SW *rx*, *offset(sp)*	8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、*sp* に加算した結果が実効アドレス (EA) になります。*ry* の内容をこのアドレスに格納します。
	SW ra, *offset(sp)*	8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、*sp* に加算した結果が実効アドレス (EA) になります。ra の内容をこのアドレスに格納します。
	* SW ry, *offset(fp)*	5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、*fp* に加算した結果が実効アドレス (EA) になります。*ry* の内容をこのアドレスに格納します。

*　TX19 → TX19A で追加された命令

表 4-11　ALU 即値命令 (16 ビット ISA)

命令	形式	説明
Add Immediate	ADDIU *ry*, *rx*, *immediate*	4 ビット *immediate* を符号拡張して、*rx* に加算し、その結果を *ry* に格納します。2 の補数のオーバフローでも例外を発生しません。
	ADDIU *rx*, *immediate*	8 ビット *immediate* を符号拡張して、*rx* に加算し、その結果を *rx* に格納します。2 の補数のオーバフローでも例外を発生しません。
	ADDIU sp, *immediate*	8 ビット *immediate* を左へ 3 ビットシフトし、符号拡張します。その結果を sp に加算した和を sp に格納します。2 の補数のオーバフローでも例外を発生しません。
	* ADDIU fp, *immediate*	8 ビット *immediate* を左へ 2 ビットシフトし、符号拡張します。その結果を fp に加算した和を fp に格納します。2 の補数のオーバフローでも例外を発生しません。
	ADDIU *rx*, pc, *immediate*	8 ビット *immediate* を 2 ビット左へシフトし、ゼロ拡張します。その結果をマスクベース PC 値 (下位 2 ビットを 0 にマスクした PC 値) に加算した和を *rx* に格納します。2 の補数のオーバフローでも例外を発生しません。
	ADDIU *rx*, sp, *immediate*	8 ビット *immediate* を左へ 2 ビットシフトし、ゼロ拡張します。その結果を sp に加算した和を *rx* に格納します。2 の補数のオーバフローでも例外を発生しません。
Set On Less Than Immediate	SLTI *rx*, *immediate*	*rx* が *immediate* より小さい場合は、t8 に 1 を格納し、そうでない場合は 0 を格納します。8 ビット *immediate* をゼロ拡張した値と *rx* を符号付き整数として比較します。
Set On Less Than Immediate Unsigned	SLTIU *rx*, *immediate*	*rx* が *immediate* より小さい場合は、t8 に 1 を設定し、そうでない場合は 0 を格納します。8 ビット *immediate* をゼロ拡張した値と *rx* を符号なし整数として比較します。
Compare Immediate	CMPI *rx*, *immediate*	*rx* = *immediate* ならば、t8 に 0 を、rx ≠ *immediate* ならば、t8 に 0 以外の値を格納します。8 ビット *immediate* をゼロ拡張します。
Load Immediate	LI *rx*, *immediate*	8 ビット *immediate* をゼロ拡張し、*rx* に格納します。
* Logical AND Immediate	ANDI *ry*, *immediate*	16 ビット *immediate* をゼロ拡張し、その結果と *ry* の内容の AND をとり、結果を *ry* に格納します。
* Logical OR Immediate	ORI *ry*, *immediate*	16 ビット *immediate* をゼロ拡張し、その結果と *ry* の内容の OR をとり、結果を *ry* に格納します。
* Logical Exclusive_OR Immediate	XORI *ry*, *immediate*	16 ビット *immediate* をゼロ拡張し、その結果と *ry* の内容の排他的 OR をとり、結果を *ry* に格納します。
* Load Upper Immediate	LUI *ry*, *immediate*	16 ビット *immediate* を 16 ビット左へシフトし、下位 16 ビットの 0 と連結して、結果を *ry* に格納します。

*　TX19 → TX19A で追加された命令

表 4-12　レジスタタイプ命令 (16 ビット ISA)

命令	形式	説明
Add Unsigned	ADDU *rz*, *rx*, *ry*	*rx* + *ry* の和を *rz* に格納します。2 の補数のオーバフローでも例外を発生しません。
Subtract Unsigned	SUBU *rz*, *rx*, *ry*	*rx* – *ry* の差を *rz* に格納します。2 の補数のオーバフローでも例外を発生しません。
Set On Less Than	SLT *rx*, *ry*	*rx* が *ry* より小さい場合、t8 に 1 を、そうでない場合は、t8 に 0 を格納します。2 値を符号付き整数として比較します。
Set On Less Than Unsigned	SLTU *rx*, *ry*	*rx* が *ry* より小さい場合、t8 に 1 を、そうでない場合は、t8 に 0 を格納します。2 値を符号なし整数として比較します。
Compare	CMP *rx*, *ry*	*rx* が *ry* と等しい場合、t8 に 0 を格納します。そうでない場合、t8 に 0 以外の値を格納します。
Negate	NEG *rx*, *ry*	*rx* = 0 – *ry* (2 の補数)
AND	AND *rx*, *ry*	*rx* の内容と *ry* の内容の AND をとり、結果を *rx* に格納します。
OR	OR *rx*, *ry*	*rx* の内容と *ry* の内容の OR をとり、結果を *rx* に格納します。
Exclusive-OR	XOR *rx*, *ry*	*rx* の内容と *ry* の内容の排他的 OR をとり、結果を *rx* に格納します。
Not	NOT *rx*, *ry*	*ry* をビットごとに反転させ、結果を *rx* に格納します。(1 の補数)
Move	MOVE *ry, r32*	*r32* の内容を *ry* に格納します。
	MOVE *r32, rz*	*rz* の内容を *r32* に格納します。
	* MOVE *fp, r32*	*r32* の内容を *fp* に格納します。
* Bit Search One Forward	BS1F *ry, rx*	*rx* の内容を下位側から上位側に 1 をサーチしていき、1 が検出されたビット番号 + 1 が *ry* に格納されます。もし、1 が検出されなかった場合は、0 が *ry* に格納されます。
* Bit Field Insert	BFINS *ry, rx, bit2, bit1*	*rx* の [(bit2 – bit1) : 0] のビット列を *ry* の [bit2 : bit1] に格納します。
* Maximum Signed	MAX *rz, rx, ry*	*rx* の内容と *ry* の内容を符号付き整数として比較し、*rx* が大きい場合は *rx* を *rz* に格納し、それ以外の場合は *ry* を *rz* に格納します。
* Minimum Signed	MIN *rz, rx, ry*	*rx* の内容と *ry* の内容を符号付き整数として比較し、*rx* が小さい場合は *rx* を *rz* に格納し、それ以外の場合は *ry* を *rz* に格納します。
* Sign-Extend Byte	SEB *rx*	*rx* の下位バイトを符号拡張して、*rx* に格納します。
* Sign-Extend Halfword	SEH *rx*	*rx* の下位ハーフワードを符号拡張して、*rx* に格納します。
* Zero-Extend Byte	ZEB *rx*	*rx* の下位バイトをゼロ拡張して、*rx* に格納します。
* Zero-Extend Halfword	ZEH *rx*	*rx* の下位ハーフワードをゼロ拡張して、*rx* に格納します。
* Saturated Additional	SADD *ry, rx, ry*	*rx* の内容と *ry* の内容を加算し、もしオーバフローした場合、*rx* >= 0 のときは 0x7FFF_FFFF を、*rx* < 0 のときは 0x8000_0000 を *ry* に格納します。オーバフローしなかった場合は、*rx* の内容と *ry* の内容を加算した結果を *ry* に格納します。
* Saturated Subtraction	SSUB *ry, rx, ry*	*rx* の内容から *ry* の内容を減算し、もしオーバフローした場合、*rx* >= 0 のときは 0x7FFF_FFFF を、*rx* < 0 のときは 0x8000_0000 を *ry* に格納します。オーバフローしなかった場合は、*rx* の内容から *ry* の内容を減算した結果を *ry* に格納します。

*　TX19 → TX19A で追加された命令

表 4-13　シフト命令 (16 ビット ISA)

命令	形式	説明
Shift Left Logical	SLL *rx*, *ry*, *sa*	*ry* の内容を *sa* ビット左へシフトし、右端の空いたビットを 0 で埋め、結果を *rx* に格納します。
	* SLL *ry*, *sa*	*rx* の内容を *sa* ビット左へシフトし、右端の空いたビットを 0 で埋め、結果を *ry* に格納します。
Shift Left Logical Variable	SLLV *ry*, *rx*	*ry* の内容を *rx* の下位 5 ビットで指定されたビット数、左へシフトし、右端の空いたビットをゼロで埋めます。結果を *ry* に格納します。
Shift Right Logical	SRL *rx*, *ry*, *sa*	*ry* の内容を *sa* ビット右へシフトし、左端の空いたビットを 0 で埋め、結果を *rx* に格納します。
	* SRL *ry*, *sa*	*ry* の内容を *sa* ビット右へシフトし、左端の空いたビットを 0 で埋め、結果を *ry* に格納します。
Shift Right Logical Variable	SRLV *ry*, *rx*	*ry* の内容を *rx* の下位 5 ビットで指定されたビット数、右へシフトし、左端の空いたビットを 0 で埋めます。結果を *ry* に格納します。
Shift Right Arithmetic	SRA *rx*, *ry*, *sa*	*ry* の内容を *sa* ビット右へシフトし、左端の空いたビットを符号ビットで埋めます。結果を *rx* に格納します。
	* SRA *ry*, *sa*	*ry* の内容を *sa* ビット右へシフトし、左端の空いたビットを符号ビットで埋めます。結果を *ry* に格納します。
Shift Right Arithmetic Variable	SRAV *ry*, *rx*	*ry* の内容を *rx* の下位 5 ビットで指定されたビット数、右へシフトし、右端の空いたビットを符号ビットで埋めます。結果を *ry* に格納します。

*　TX19 → TX19A で追加された命令

表 4-14　SAVE/RESTORE 命令 (16 ビット ISA)

命令	形式	説明
* SAVE	SAVE *reg_list3, framesize4*	*reg_list3* で指定されたレジスタが、メモリにストアされ、*framesize4* の値によって SP の値が更新されます。
	SAVE *reg_list3, xsregs, aregs, framesize8*	*reg_list3, xsregs, aregs* で指定されたレジスタが、メモリにストアされ、*framesize8* の値によって SP の値が更新されます。
* RESTORE	RESTORE *reg_list3, framesize4*	*reg_list3* で指定されたレジスタに、メモリからデータがロードされ、*framesize4* の値によって SP の値が更新されます。
	RESTORE *reg_list3, xsregs, aregs, framesize8*	*reg_list3, xsregs, aregs* で指定されたレジスタに、メモリからデータがロードされ、*framesize8* の値によって SP の値が更新されます。

*　TX19 → TX19A で追加された命令

表 4-15　乗算・除算命令 (16 ビット ISA)

命令	形式	説明
Multiply	MULT *rx*, *ry*	被乗数は *rx* の符号付きの値です。乗数は *ry* の符号付きの値です。64 ビットの積 *rx* * *ry* を HI レジスタと LO レジスタに格納します。
	* MULT *ry, rx, ry*	被乗数は *rx* の符号付き整数です。乗数は *ry* の符号付き整数です。64 ビットの積 *rx* * *ry* を HI レジスタと LO レジスタに格納します。また、下位 32 ビットを *ry* に格納します。
Multiply Unsigned	MULTU *rx*, *ry*	被乗数は *rx* の符号なしの値です。乗数は *ry* の符号なしの値です。64 ビットの積 *rx* * *ry* を HI レジスタと LO レジスタに格納します。
	* MULTU *ry, rx, ry*	被乗数は *rx* の符号なし整数です。乗数は *ry* の符号なし整数です。64 ビットの積 *rx* * *ry* を HI レジスタと LO レジスタに格納します。また、下位 32 ビットを *ry* に格納します。
* Multiply And Add	MADD *rx*, *ry*	被乗数は *rx* の符号付き整数です。乗数は *ry* の符号付き整数です。64 ビットの積 *rx* * *ry* を HI レジスタ・LO レジスタに格納されている 64 ビットの値に加算し、その結果を HI レジスタと LO レジスタに格納します。
* Multiply And Add Unsigned	MADDU *rx*, *ry*	被乗数は *rx* の符号なし整数です。乗数は *ry* の符号なし整数です。64 ビットの積 *rx* * *ry* を HI レジスタ・LO レジスタに格納されている 64 ビットの値に加算し、その結果を HI レジスタと LO レジスタに格納します。
Divide	DIV *rx*, *ry*	被除数は *rx* の符号付きの値です。除数は *ry* の符号付きの値です。商を LO レジスタに、剰余を HI レジスタに格納します。
Divide Unsigned	DIVU *rx*, *ry*	被除数は *rx* の符号なしの値です。除数は *ry* の符号なしの値です。商を LO レジスタに、剰余を HI レジスタに格納します。
* Divide Exception	DIVE *rx*, *ry*	被除数は *rx* の符号付き整数です。除数は *ry* の符号付き整数です。商を LO レジスタに、剰余を HI レジスタに格納します。*ry* の内容が 0、もしくは演算結果がオーバフローした場合は、整数オーバフロー例外が発生します。
* Divide Exception Unsigned	DIVEU *rx*, *ry*	被除数は *rx* の符号なし整数です。除数は *ry* の符号なし整数です。商を LO レジスタに、剰余を HI レジスタに格納します。*ry* の内容が 0 の場合整数オーバフロー例外が発生します。
Move From HI	MFHI *rx*	HI レジスタの内容を *rx* にロードします。
Move From LO	MFLO *rx*	LO レジスタの内容を *rx* にロードします。
* Move to HI	MTHI *rx*	*rx* の内容を HI レジスタに格納します。
* Move to LO	MTLO *rx*	*rx* の内容を LO レジスタに格納します。

*　TX19 → TX19A で追加された命令

表 4-16　ビット操作命令 (16 ビット ISA)

命令	形式		説明
* Bit Test	BTST	*offset(base3), pos3*	14 ビット *offset* をゼロ拡張して、*base3* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を反転して、*t8* レジスタの LSB ビットに格納し、上位 31 ビットは 0 で埋められます。
	BTST	*offset*(r0), *pos3*	14 ビット *offset* を符号拡張して、*r0* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を反転して、*t8 レジスタの LSB ビットに格納し、上位 31 ビットは 0 で埋められます。*
	BTST	*offset(fp), pos3*	5 ビット *offset* をゼロ拡張して、*fp* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を反転して、*t8 レジスタの LSB ビットに格納し、上位 31 ビットは 0 で埋められます。*
* Bit Extract	BEXT	*offset(base3), pos3*	14 ビット *offset* をゼロ拡張して、*base3* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容 *t8* レジスタに格納します。
	BEXT	*offset*(r0), *pos3*	14 ビット *offset* を符号拡張して、*r0* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容 *t8* レジスタに格納します。
	BEXT	*offset(fp), pos3*	5 ビット *offset* をゼロ拡張して、*fp* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容 *t8* レジスタに格納します。
* Bit Clear	BCLR	*offset(base3), pos3*	14 ビット *offset* をゼロ拡張して、*base3* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を 0 にクリアします。
	BCLR	*offset*(r0), *pos3*	14 ビット *offset* を符号拡張して、*r0* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を 0 にクリアします。
	BCLR	*offset(fp), pos3*	5 ビット *offset* をゼロ拡張して、*fp* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を 0 にクリアします。
* Bit Set	BSET	*offset(base3), pos3*	14 ビット *offset* をゼロ拡張して、*base3* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を 1 にセットします。
	BSET	*offset*(r0), *pos3*	14 ビット *offset* を符号拡張して、*r0* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を 1 にセットします。
	BSET	*offset(fp), pos3*	5 ビット *offset* をゼロ拡張して、*fp* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を 1 にセットします。

命令	形式		説明
* Bit Insert	BINS	*offset(base3), pos3*	14 ビット *offset* をゼロ拡張して、*base3* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリに、*pos3* で指定されたビット番号の位置に、*t8* レジスタの LSB ビットの値を挿入する。
	BINS	*offset(*r0*), pos3*	14 ビット *offset* を符号拡張して、*r0* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリに、*pos3* で指定されたビット番号の位置に、*t8* レジスタの LSB ビットの値を挿入する。
	BINS	*offset(fp), pos3*	5 ビット *offset* をゼロ拡張して、*fp* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリに、*pos3* で指定されたビット番号の位置に、*t8* レジスタの LSB ビットの値を挿入する。
* Add Immediate to Memory Word	ADDMIU	*offset(base3), imm*	14 ビット *offset* を 2 ビット左シフトしてゼロ拡張して、*base3* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリの内容と *imm* との値を加算して、その結果をメモリにストアします。
	ADDMIU	*offset(*r0*), imm*	14 ビット *offset* を 2 ビット左シフトして符号拡張して、*r0* の内容と加算した結果が実効アドレス (EA) になります。EA でアドレス指定されたメモリの内容と *imm* との値を加算して、その結果をメモリにストアします。

*　TX19 → TX19A で追加された命令

表 4-17　システム制御コプロセッサ命令 (16 ビット ISA)

命令	形式		説明
* Move To Coprocessor 0	MTC0	*rx, cp0rd32*	汎用レジスタ *rx* の内容を CP0 レジスタ *cp0rd32* にロードします。
* Move From Coprocessor 0	MFC0	*ry, cp0rs32*	CP0 レジスタ *cp0rs32* の内容を汎用レジスタ *ry* にロードします。
* Add Coprocessor 0 Immediate Unsigned	AC0IU	*cp0rt32, immediate*	CP0 レジスタ *cp0rt32* の内容に *immediate* を加算して、その結果を CP0 レジスタ *cp0rt32* に格納します。

*　TX19 → TX19A で追加された命令

表 4-18　ジャンプ命令 (16 ビット ISA)

命令	形式	説明
Jump And Link	JAL *target*	ページ内絶対アドレッシングモードを使ってジャンプします。つまり、26 ビット *target* を 2 ビット左へシフトし、PC + 4 の上位 4 ビットと連結した結果がターゲットアドレスになります。遅延スロットの後続命令のアドレスを r31 に格納します。
Jump And Link eXchange	JALX *target*	ページ内絶対アドレッシングモードを使ってジャンプします。つまり、26 ビット *target* を 2 ビット左へシフトし、PC + 4 の上位 4 ビットと連結した結果がターゲットアドレスになります。遅延スロットの後続命令のアドレスを r31 に格納します。PC 内の ISA モードビットはトグルします。
Jump Register	JR *rx*	*rx* の上位 31 ビットで指定されたアドレスへジャンプします。*rx* の最下位のビットの値によって、ISA モードが切り換わります。
	JR ra	ra の上位 31 ビットで指定されたアドレスへジャンプします。ra の最下位のビットの値によって、ISA モードが切り換わります。
* Jump Register, Compact	JRC *rx*	*rx* の上位 31 ビットで指定されたアドレスへジャンプします。*rx* の最下位のビットの値によって、ISA モードが切り替わります。本命令には、ジャンプ遅延スロットを持っていません。
	JRC ra	ra の上位 31 ビットで指定されたアドレスへジャンプします。ra の最下位のビットの値によって、ISA モードが切り替わります。本命令には、ジャンプ遅延スロットを持っていません。
Jump And Link Register	JALR ra, *rx*	*rx* の上位 31 ビットで指定されたアドレスへジャンプします。*rx* の最下位のビットの値によって、ISA モードが切り替わります。また、遅延スロットの後続命令のアドレスを ra に格納します。
* Jump And Link Register, Compact	JALRC ra, *rx*	*rx* の上位 31 ビットで指定されたアドレスへジャンプします。*rx* の最下位のビットの値によって、ISA モードが切り替わります。また、本命令の後続命令のアドレスを ra に格納します。本命令には、ジャンプ遅延スロットを持っていません。

*　TX19 → TX19A で追加された命令

表 4-19　分岐命令 (16 ビット ISA)

命令	形式	説明
Branch On Equal To Zero	BEQZ *rx*, *offset*	*rx* = 0 ならば、PC + 2 に対して相対的に指定された 8 ビット *offset* に分岐します。拡張された場合は、PC + 4 となります。
Branch On Not Equal To Zero	BNEZ *rx*, *offset*	*rx* ≠ 0 ならば、PC + 2 に対して相対的に指定された 8 ビット *offset* に分岐します。拡張された場合は、PC + 4 となります。
Branch On T8 Equal To Zero	BTEQZ *offset*	t8 = 0 ならば、PC + 2 に対して相対的に指定された 16 ビット *offset* に分岐します。拡張された場合は、PC + 4 となります。
Branch On T8 Not Equal To Zero	BTNEZ *offset*	t8 ≠ 0 ならば、PC + 2 に対して相対的に指定された 16 ビット *offset* に分岐します。拡張された場合は、PC + 4 となります。
Unconditional Branch	B *offset*	PC + 2 に対して相対的に指定された 16 ビット *offset* に、無条件に分岐します。拡張された場合は、PC + 4 となります。
* Branch And Link	BAL *offset*	PC + 2 に対して相対的に指定された 16 ビット *offset* に、無条件に分岐します。分岐命令の後続命令のアドレスを *r31* に格納します。拡張された場合は、PC + 4 となります。

*　TX19 → TX19A で追加された命令

表 4-20　特殊命令 (16 ビット ISA)

Instruction	Format	Operation
Breakpoint	BREAK　*code*	ブレークポイント例外が発生し、無条件で例外ハンドラの処理に移ります。
Software Debug Breakpoint Exception	SDBBP　*code*	デバッグポイント例外が発生し、無条件で例外ハンドラの処理に移ります。
* Disable Interrupt	DI	Status レジスタの IE ビットを 0 にクリアします。
* Enable Interrupt	EI	Status レジスタの IE ビットを 1 にセットします。
* System Call	SYSCALL　*code*	システムコール例外が発生し、無条件で例外ハンドラの処理に移ります。
* Synchronize	SYNC	直前に実行したロードまたはストア命令が完了するまでパイプラインをインタロックします。
* Exception Return	ERET	Status レジスタの ERL ビットが 1 の状態で本命令を実行すると ErrorEPC を戻り番地として復帰し、ERL ビットが 0 の状態で本命令を実行すると EPC を戻り番地として例外から復帰します。
* Debug Exception Return	DERET	プログラム制御は、デバッグ例外処理プログラムよりユーザープログラムに戻ります。DEPC レジスタの戻りアドレスを PC に復元します。
* Enter Standby Mode	WAIT	Status レジスタの RP ビットの状態に応じて HALT あるいは DOZE モードに遷移します。

*　TX19 → TX19A で追加された命令

第5章　CPU パイプライン

5.1　概要

CPUの実行ユニットはステージといういくつかの部分で構成されているものと見ることができます。「2.5 パイプライン」で説明したように、各命令の処理は単純な処理に細分化され、それぞれの処理は各ステージで実行されます。あるステージでの処理結果は次のステージに渡され、パイプラインの各ステージは左から右へと流れていきます。TX19Aのパイプラインは、命令フェッチ (F)、デコード (D)、実行 (E)、メモリアクセス (M)、レジスタ書き込み (W) の 5 ステージで構成されます。例えばある命令がDステージを終了し、Eステージに進むと、後続の命令がDステージへと進んできます。各ステージは、約 1 クロックで実行されます。いったんパイプラインが満たされると、図 5-1に示すように 5 つの命令が同時に実行されます。

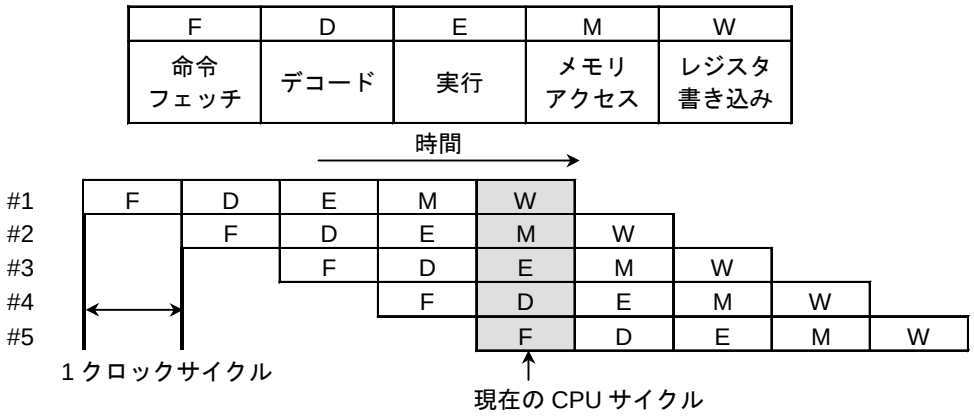


図 5-1　5 つの CPU パイプラインステージ

以下に代表的な命令が、各ステージでどのように処理されるかを示します。

命令フェッチ (F)　　命令メモリサブシステム(命令 ROM、命令 RAM)から命令がフェッチされます。命令は ISA モードに関らず、1 ワード単位でフェッチされます。

デコード (D)　　命令がデコードされ、必要なオペランドがレジスタファイルから読み出されます。

実行 (E)　　命令の種類により、以下の処理がALUにより実行されます。

- 整数の算術演算、論理演算、またはシフト処理を開始します。
- ロード、ストア命令の場合は、ベースレジスタの内容にオフセット値を加算して、実効アドレスを生成すると共にバスサイクルを起動します。
- ジャンプ命令の場合は、ターゲットアドレスを計算します。
- 分岐、分岐ライクリ命令の場合は、分岐条件が成立するかどうかを判定するとともに、ターゲットアドレスを計算します。

メモリアクセス (M)　ロード、ストア命令の場合データメモリがアクセスされます。

レジスタ書き込み (W)　命令の種類により、以下の処理が実行されます。

- 算術論理演算命令の場合は、E ステージでの ALU 操作の結果が、レジスタファイルに書き込まれます。
- リンク機能付きジャンプ、リンク機能付き分岐、リンク機能付き分岐ライクリ命令の場合は、戻りアドレスがレジスタ r31(ra)に書き込まれます。

TX19A のようなパイプラインを備えた CPU では、パイプラインのスムーズな流れを乱す命令があります。パイプラインの乱れをパイプラインハザードといいます。以下の項では、パイプラインハザードの原因、およびそれに対するハードウエア、ソフトウエア処理について説明します。

5.2　ロード・ストア・SYNC 命令

ソフトウエアの性能は、ソフトウエアの設計者、特にアセンブリ言語のプログラマが、プロセッサの基本的なハードウエアをどの程度理解しているかにより大きく左右されます。この項では、ロード遅延、ノンブロッキングロード、SYNC 命令などを CPU のパイプラインの観点から説明します。

5.2.1　ロード遅延

ロード命令は、図 5-2に示す順序で処理されます。

F	D	E	M	W
命令 フェッチ	デコード	実効アドレス生成 バスサイクル起動	メモリ アクセス	レジスタ 書き込み

図 5-2　ロード命令

ロード命令は、データ (オペランド) をメモリからCPUレジスタに読み込みます。高速内蔵メモリからのロードの場合は、ロード命令のMステージが終了した時点で、後続の命令はロードされたデータを使用できるようになります。Wステージでレジスタに書き込まれるまで待つ必要がありません。それでも、図 5-3に示すように、ロードされたデータは、ロード命令の直後の命令では使えません。なぜならば、直後の命令のEステージには間に合わないからです。この場合、ADD命令はLW命令に対してデータ依存関係があるといいます。図 5-3では、TX19Aが後続命令のEステージにウエイト (ストール) サイクルを挿入することにより、データの依存関係をなくしています。図 5-3では、1 サイクルの遅延 (レイテンシ) が発生します。ロード命令の直後の命令位置を「ロード遅延スロット」といいます。外部メモリからロードする必要がある場合は、さらに多くのストールサイクルが発生します。

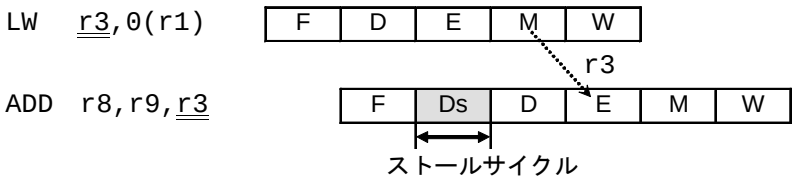


図 5-3　ロード命令のデータ依存

データの依存関係に対して、ハードウエアでストールサイクルを挿入するのは、効率的ではありません。コンパイラまたはアセンブラの最適化処理で、命令を並び替えることにより、ロード遅延スロットの命令が直前のロード命令でロードされるデータに依存しないようにできます。図 5-4に例を示します。この例は 2 つのメモリ位置の内容を入れ替えるプログラムの一部です。

- データ依存関係があるプログラム

```
LW r9,0(r8)
LW r10,4(r8)
SW r10,0(r8) ← ロード遅延スロット
SW r9,4(r8)
```

- データ依存関係がないプログラム

```
LW r9,0(r8)
LW r10,4(r8)
SW r9,4(r8) ← ロード遅延スロット
SW r10,0(r8)
```

図 5-4　データ依存関係をなくすための命令の並び替え

命令を並び替えたあとのプログラムでは、SW 命令と直前の LW 命令のデータとの依存関係はありません。このため、「LW r10,4(r8)」のロード遅延スロットの命令「SW r9,0(r8)」は、パイプラインをストールさせません。

5.2.2　ノンブロッキングロード

ロード命令の直後の命令がロード命令のターゲットレジスタ(*rt*)をアクセスしない場合、データの依存関係は発生しません。TX19A はデータの依存関係があるかどうか判断し、依存関係がない場合パイプラインをストールさせることなく後続の命令を実行します。これを「ノンブロッキングロード」といいます。ノンブロッキングロードにより、外部メモリアクセス中もパイプラインはストールしません。外部メモリアクセス中パイプラインの他のステージでは、データ依存関係のない命令が継続して実行されます。

図 5-5の例では、LW命令で外部メモリアクセスが発生してもストールせずに、データ依存関係のない命令「ADD r6,r4,r2」と「ADD r7,r5,r2」を継続して実行しています。ただし、LW命令とデータ依存関係のある命令「ADD r8,r9,r3」は、最初のLW命令のデータがロードされるまでパイプラインをストールします。

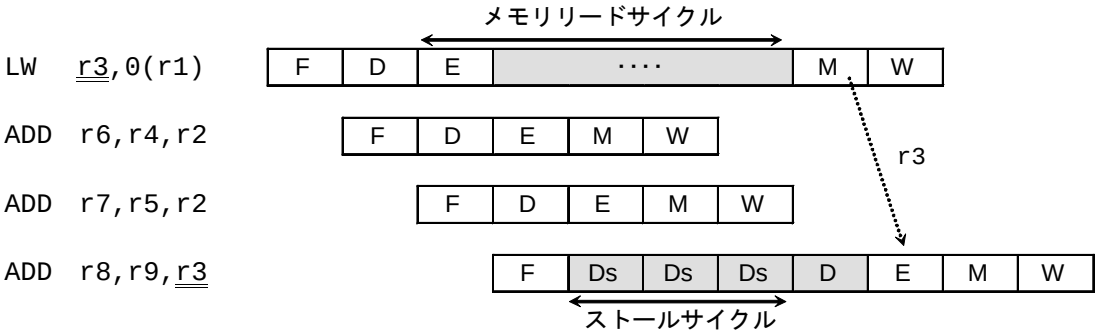


図 5-5　ノンブロッキングロード

ノンブロッキングロード機能を活用して、ロード命令を先行して実行し、ロードされたデータを使用する命令のまえにデータ依存関係のない命令を実行させることにより、実行時間を短縮できます。これを「プリフェッチ動作」といい、コンパイラによる最適化で命令の並び替えが試みられます。

5.2.3　ストア命令(32 ビット ISA / 16 ビット ISA)

ストア命令は、図 5-6に示す順序で実行されます。

F	D	E	M	W
命令 フェッチ	デコード	実効アドレス生成 バスサイクル起動	メモリアクセス （WAIT 時）	—

図 5-6　ストア命令

ストア命令は、CPU レジスタのデータをメモリに書きこむ命令です。

図 5-7に高速内蔵メモリにアクセスした場合の実行順序を示します。高速内蔵メモリへのストアはEステージで起動され、ライトバスサイクルは 1 クロックで終了します。その為M、Wステージでは何も実行されません。

図 5-8に外部メモリにアクセスした場合の命令実行順序を示します。外部メモリへのストアの場合、ライトバスサイクルは 2 クロック以上を必要とします。TX19Aは最大 4 つのライトデータを貯めることができるライトバッファを内蔵しているので、外部メモリをアクセスする命令が連続してもパイプラインがストールすることはありません。ただし、ライトバッファの空きがなくなった場合、後続のライトバッファを使用する命令はストールします。

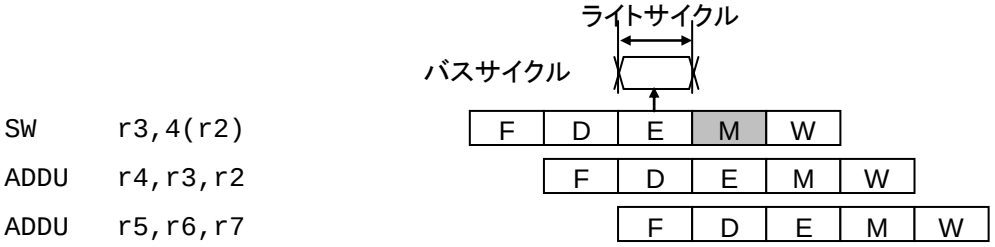


図 5-7　高速内蔵メモリへのアクセス

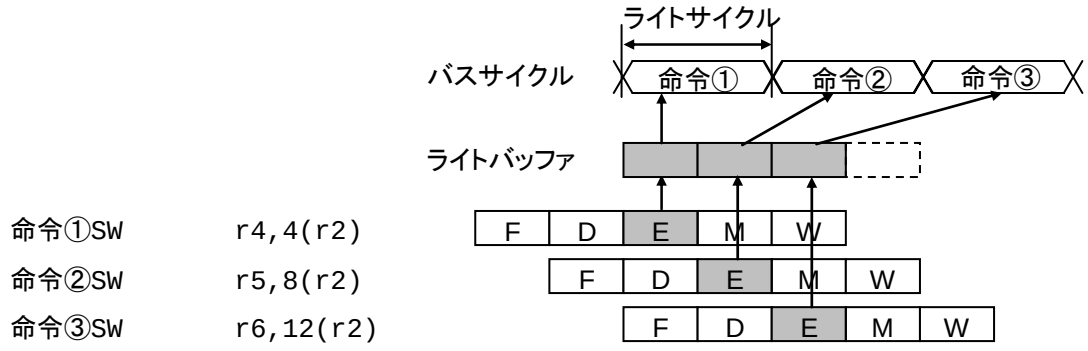


図 5-8　外部メモリを連続アクセス

5.2.4　SYNC 命令(32 ビット ISA / 16 ビット ISA)

ロード・ストア命令では、M ステージでメモリへのロード・ストアを行いますが、その間 TX19A は並行して他の命令を実行しています。

図 5-9にSYNC命令の処理手順を示します。SYNC命令は直前に実行したロード・ストア命令が完了するまでMステージでストールし、次の命令の実行を遅らせます。これにより、SYNC命令の前のロード・ストア命令とSYNC命令の後の命令の実行順序を守ることができます。

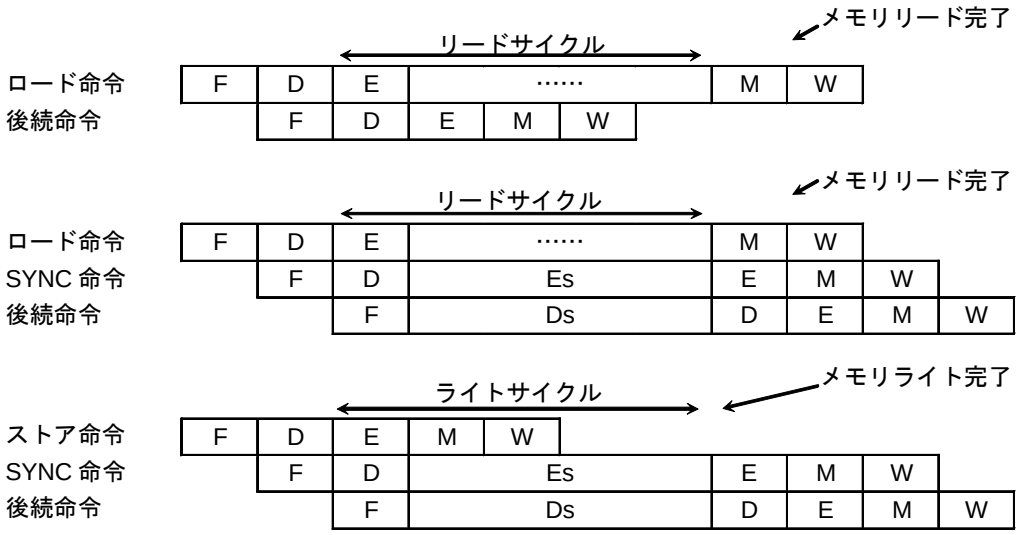


図 5-9　SYNC 命令

5.2.5　ビット操作命令(16 ビット ISA)

ビット操作命令には、メモリのビットデータに対して演算後メモリに再び書き戻す命令と、CPUレジスタに書き戻す命令の 2 通りの命令があります。それぞれの命令の実行手順 図 5-10、図 5-11に示します。

F	D	E	M	W
命令 フェッチ	デコード	実効アドレス生成 バスサイクル起動	メモリ アクセス	−

図 5-10　ビット操作命令(メモリに書き戻す命令)

F	D	E	M	Md	W
命令 フェッチ	デコード	実効アドレス生成 バスサイクル起動	メモリ アクセス	演算	レジスタ 書き込み

図 5-11　ビット操作命令(CPU レジスタに書き戻す命令)

メモリに書き戻す命令の場合は、ストア命令と同様に、バスオペレーションがライトバッファにエントリされ、パイプラインはストールせず命令を処理します。
図 5-12にメモリに書き戻す命令の場合の処理手順を示します。

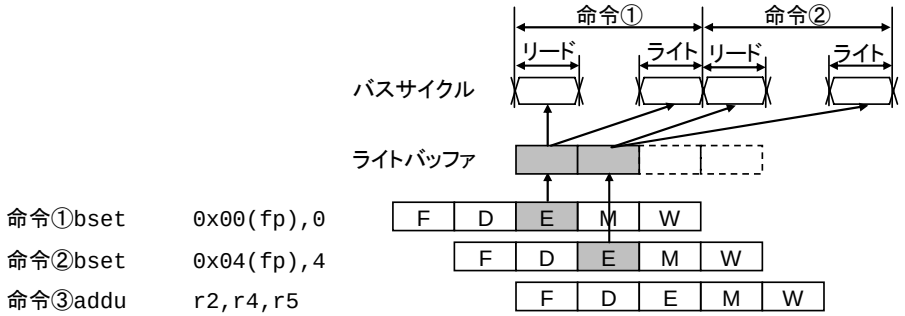


図 5-12　メモリに書き戻すビット操作命令の処理手順

CPUレジスタに書き戻す命令の場合は、命令の種類によって動作が異なります。ビット操作命令の直後の命令がビット操作命令のターゲットレジスタ(t8)をアクセスしない場合は、パイプラインをストールさせることなく後続の命令を実行します。このとき、ビット操作命令のBTST、BEXT命令はノンブロッキングロードで動作しています。

ターゲットレジスタ(t8)をアクセスする命令の場合は、後続の命令を実行する際にパイプラインはストールします。

図 5-13にCPUレジスタに書き戻す命令の実行順序を示します。この図の 2 つのADDU命令はターゲットレジスタをアクセスしないので、BTST命令の直後であってもストールすることなく実行します。しかし、4 番目のBTEGZ命令はターゲットレジスタを参照する必要があるためストールが発生します。

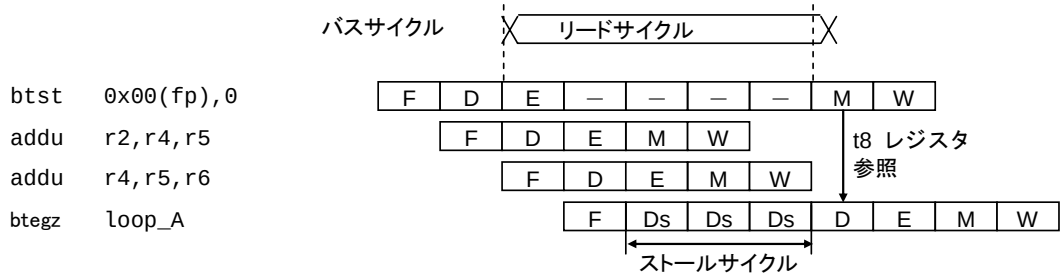


図 5-13　CPU に書き戻すビット操作命令の処理手順

ノンブロッキングロードで動作すると、ビット操作命令の W ステージと後続の命令の W ステージが衝突することがあります。この場合、後続命令はストールします。

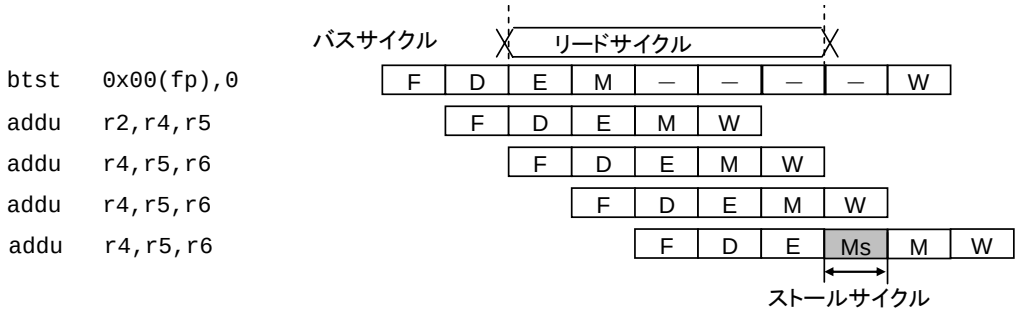


図 5-14　W ステージが衝突する場合

5.3　ジャンプ・分岐・分岐ライクリ命令

ジャンプ、分岐命令では、通常 2 命令サイクルの遅延 (レイテンシ) が発生します。ただし、コンパイラ、アセンブラは遅延スロットを利用することによって、これを 1 サイクルにすることができます。

この項では、ジャンプ、分岐遅延スロットについて、また、分岐ライクリ命令の処理について説明します。

5.3.1　ジャンプ・一般分岐命令 (32 ビット ISA)

ジャンプ命令、一般分岐命令は、図 5-15に示す順序で実行されます。

F	D	E	M	W
命令フェッチ	デコード	ターゲットアドレスの計算 分岐条件判定 PC 更新	なし	レジスタ 書き込み

図 5-15　ジャンプ・分岐命令

ジャンプ・分岐命令では、以下の処理を E ステージで実行します。

- ジャンプ命令の場合は、ALU によりジャンプターゲットアドレスを計算します。
- 一般分岐命令、分岐ライクリ命令の場合は、ALU により分岐条件が成立するかどうか判定するとともに、分岐ターゲットアドレスを計算します。

M ステージでは何も実行されません。また、リンク機能付きジャンプ命令またはリンク機能付き分岐命令の場合は、W ステージで戻りアドレスをレジスタ r31(ra)に書き込みます。

ジャンプ命令、一般目分岐命令の処理手順を 図 5-16に示します。ジャンプまたは分岐ターゲットアドレスは、Eステージで計算されます。ジャンプ先または分岐先の命令はターゲットアドレス計算後にフェッチされるため、2 命令サイクル分の遅延が発生します。しかし、ジャンプ命令または一般分岐命令の直後の命令(遅延スロット)は、常にジャンプ先または分岐先の命令より先に実行されます。このため、ジャンプ命令、一般分岐命令で発生する遅延サイクルは見かけ上 1 サイクルとなります。

遅延スロットはコンパイラの最適化処理により、有用な命令で埋められます。有用な命令がない場合は、コンパイラは遅延スロットに NOP 命令を挿入します。

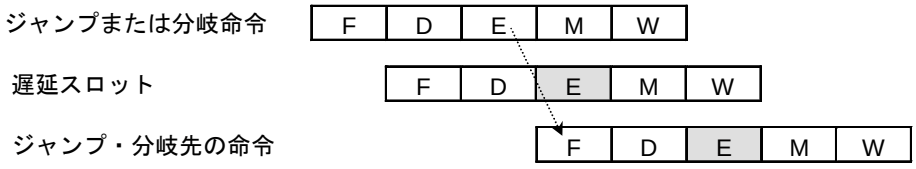


図 5-16　ジャンプ・分岐遅延スロット

(注 1)　ジャンプまたは分岐遅延スロットにジャンプ命令または分岐命令を置かないでください。
遅延スロットにジャンプ命令または分岐命令を置いた場合のハードウエアの動作は不定となります。
(注 2)　一般分岐命令では、分岐の成立、不成立にかかわらず常に遅延スロット内の命令を実行します。
そのため、分岐遅延スロット内にプログラムの論理に影響を与える命令を置かないでください。

5.3.2　分岐ライクリ命令 (32 ビット ISA)

一般分岐命令では、分岐が成立するかしないかにかかわらず常に遅延スロット内の命令を実行します。そのため、分岐遅延スロット内の命令はプログラムの論理に影響を与えるものであってはいけません。

これに対して、分岐ライクリ命令では、分岐が成立しない場合は遅延スロット内の命令をEステージで無効にし、分岐が成立した場合のみ遅延スロット内の命令を実行します。このため、コンパイラは分岐遅延スロット内に分岐先の命令を置くことができます。図 5-17に一般分岐命令と分岐ライクリ命令の実行手順の違いを示します。

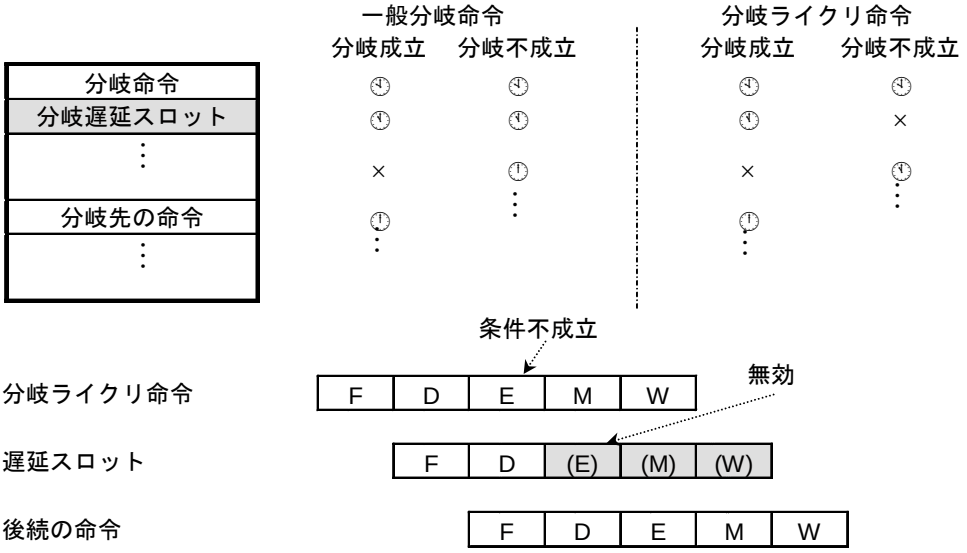


図 5-17　分岐ライクリ命令 (分岐不成立の場合)

5.3.3　ジャンプ命令 (16 ビット ISA)

16 ビットISAのJAL・JALX命令は例外的に 32 ビット長です。従来TX19 の 16 ビットISAでは、図 5-18に示すようにジャンプ命令を 2 段階に分けて実行しなければなりません。TX19 は最初のD、Eステージでは何も実行せず、後半のコードがフェッチされるのを待ってからジャンプ先のターゲットアドレスを計算します。アドレスの計算は、ジャンプ命令の後半のEステージで実行されます。その結果、16 ビットISAのJAL・JALX命令では 2 命令サイクルの遅延が発生します。

TX19Aでは、図 5-19に示すように従来 2 回に分けてデコードしていた処理を 1 回で実行することが可能です。そのため、実行サイクルとして従来 2 命令サイクルの遅延が発生していたのが、TX19Aでは 1 命令サイクルの遅延となり、処理能力が向上しています。

(注 1)　ジャンプ命令の遅延スロットにジャンプ命令または分岐命令を置かないでください。
遅延スロットにジャンプ命令または分岐命令を置いた場合のハードウエアの動作は不定となります。

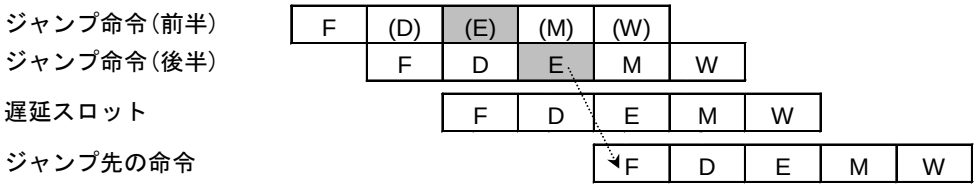


図 5-18　ジャンプ命令 (16 ビット ISA: TX19)

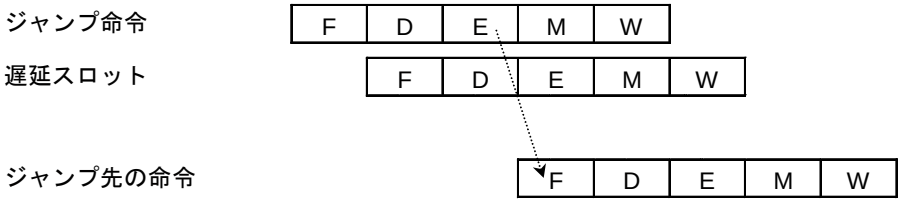


図 5-19　ジャンプ命令 (16 ビット ISA: TX19A)

5.3.4　分岐命令 (16 ビット ISA)

32 ビットISAとは異なり、16 ビットISAの分岐命令には遅延スロットがありません (図 5-20参照)。分岐はその直後の命令の前で有効になります。したがって分岐が成立した場合は、分岐命令の直後に置かれた命令は実行されません。そのため、分岐命令の直後に置く命令に制限はありません。

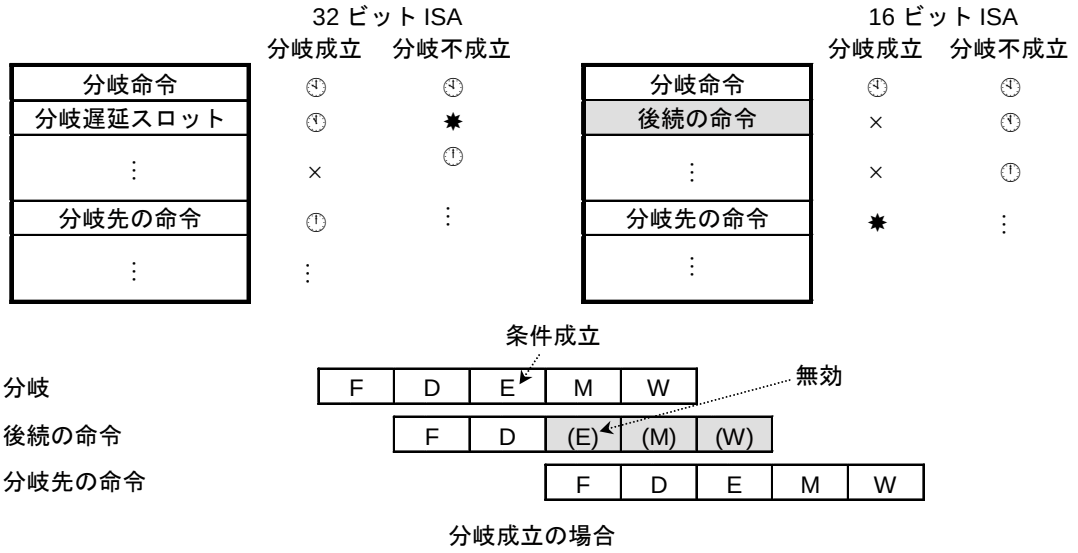


図 5-20　分岐命令 (16 ビット ISA)

5.4　SAVE ・ RESTOR 命令 (16 ビット ISA)

SAVE・RESTOR命令は、1 命令で複数のレジスタをメモリに退避したり、メモリからレジスタに復帰させたりする命令です。
図 5-21、図 5-22にSAVE命令、RESTOR命令の処理動作を示します。
最後のデータが復帰あるいは退避されスタックポインタレジスタ(r29) の更新がされるまで、後続の命令はストールします。

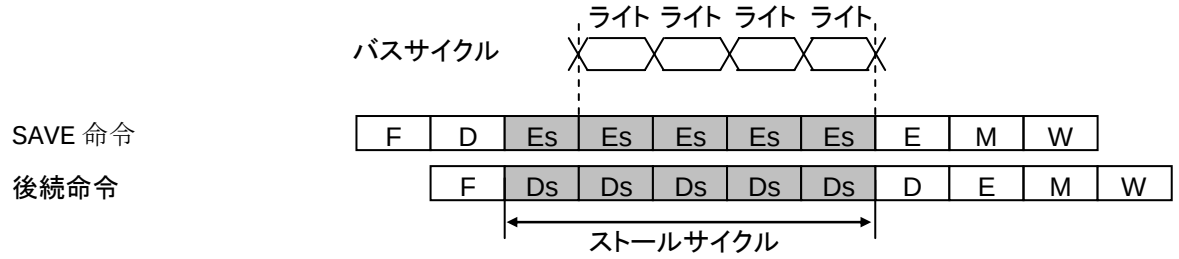


図 5-21　SAVE 命令

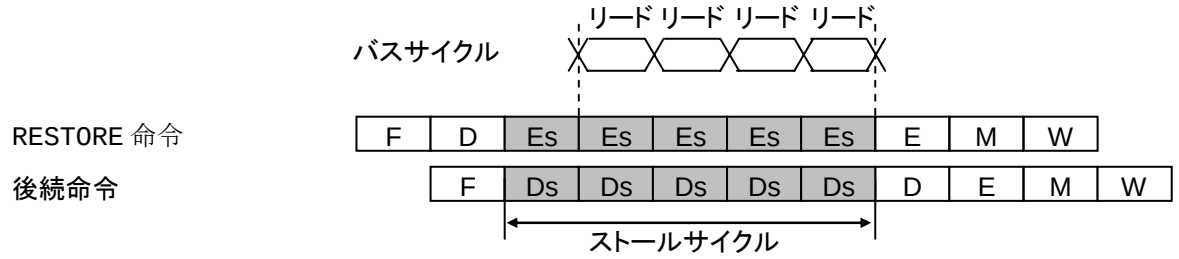


図 5-22　RESTOR 命令

5.5　除算命令

整数除算命令は、専用の除算ユニットで実行されるため、他の命令と並行して継続できます。除算ユニットは、遅延サイクル、例外が発生しても、命令の実行を継続します。除算命令の商と剰余は、LO レジスタと HI レジスタに格納されます。

除算は E ステージで実行を開始します。オペランドの値の大きさ、符号にかかわらず、命令の実行時間は 35 命令サイクルです。除算が完了するまえに、MFHI、MFLO、MADD、MADDU、MSUB、MSUBU 命令を実行すると、LO、HI レジスタが確定するまでパイプラインがストールします。

F	D	E	M	W
命令 フェッチ	デコード	実行	なし	なし

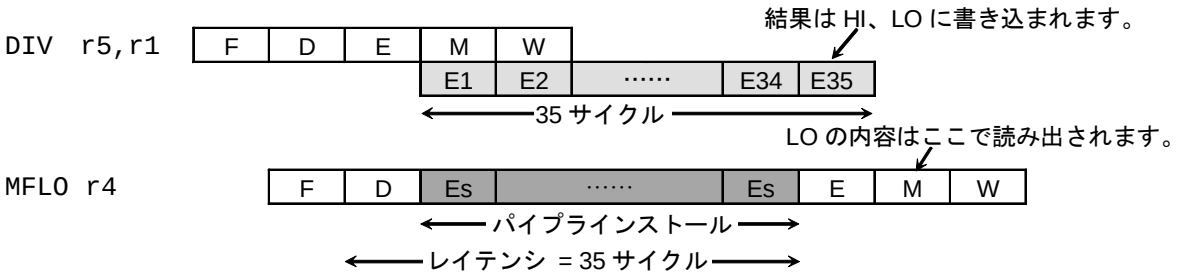


図 5-23　除算命令

5.6　乗算・積和命令/積差命令

整数の乗算、積和、積差命令は、専用 MAC ユニットで実行されるため、他の命令と並行して継続できます。乗算、積和、積差命令は、1 クロックで完了します。

乗算、積和、積差命令はEステージを 1 クロックで完了するので、複数の乗算、積和、積差命令を連続して実行してもパイプラインはストールしません。
(図 5-24参照)

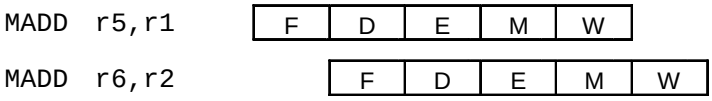


図 5-24　連続した積和命令

HI、LOレジスタの内容を読み出すには、MFHI、MFLO命令を使います。図 5-25 のように乗算、積和、積差命令の直後にMFHIまたはMFLO命令を置いても、パイプラインはストールしません。

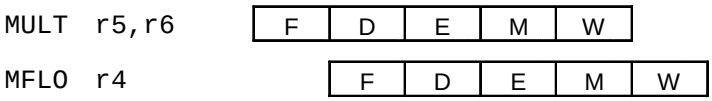


図 5-25　乗算命令と後続の MFLO 命令

乗算、積和、積差命令の結果は、E ステージではなく M ステージ完了後に使用可能になります。乗算、積和、または積差命令で、汎用レジスタをデスティネーションレジスタ(*rd*)として指定する場合、結果が *rd* に格納されるまでに後続の命令が *rd* にアクセスすると、パイプラインは *rd* に結果が格納されるまで D ステージでストールします。

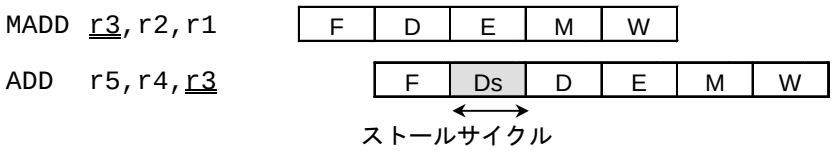


図 5-26　乗算命令での構造ハザード

5.7　拡張命令 (16 ビット ISA)

EXTENDが付加されると、16 ビットISAの命令は 32 ビットになります。拡張された命令のコードは、16 ビットのEXTENDコードと拡張される 16 ビット命令コードで構成されます。図 5-28に示すように、従来のTX19 は拡張された命令を 2 段階で実行していましたが、TX19Aは拡張命令でも 1 段階で実行が可能となり、性能を向上させています。

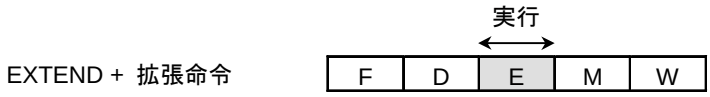


図 5-27　拡張命令 (16 ビット ISA: TX19A)

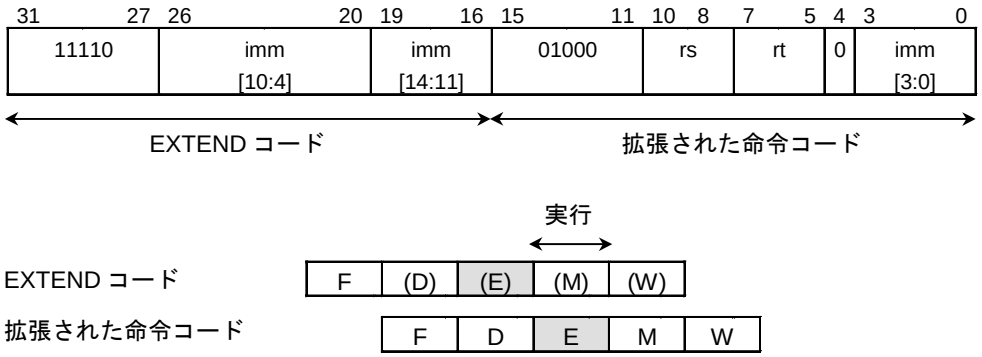


図 5-28　拡張命令 (16 ビット ISA: TX19)

第6章　メモリ管理

この章では、TX19A の動作モード、仮想アドレス空間、物理アドレス空間、アドレス変換について説明します。

6.1　動作モード

TX19A にはユーザーモードとカーネルモードの 2 種類の動作モードがあります。TX19A は例外が発生すると、自動的にカーネルモードに移行します。また、システムリセットがかかると、プロセッサはリセット例外により立ち上がるため、立ち上げ時はカーネルモードになります。ERET (Exception Return) 命令、または DERET (Debug Exception Return) 命令により、カーネルモードからユーザーモードへ復帰します。

■　ユーザーモード

動作モードにより、プログラムで使用できる仮想アドレス空間、レジスタ、命令が異なります。ユーザーモードではこれらの使用が制限されます。ユーザーモードで使用できる仮想アドレス空間は、0x0000_0000 から 2G バイトのリニアアドレス空間 (kuseg) に制限されます。また、CP0 レジスタへのアクセスは、Status レジスタの CU[0]ビットが 1 にセットされているときに制限されます。

ユーザーモードは、Status レジスタの UM ビット=1 のとき、かつ ERL ビット=EXL ビット=0 の状態を示します。

■　カーネルモード

カーネルモードはユーザーモードより高い特権レベルが与えられ、4G バイト仮想アドレス空間、すべてのレジスタ、命令を使用できます。オペレーティングシステムのルーチン、例外ハンドラ、デバックハンドラなどのプログラムは、カーネルモードで実行します。

カーネルモードは Debug レジスタの DM ビット=1 のとき、あるいは Status レジスタの UM ビット=0 のとき、あるいは ERL ビット=1 のとき、あるいは EXL ビット=1 の状態を示します。

(注意) TX19A では、カーネルモードのみ使用してください。

6.2　仮想アドレス空間

図 6-1にユーザーモード、カーネルモードでアクセスできる仮想アドレス空間を示します。ユーザーモードでは、2Gバイトの仮想アドレス空間 (kuseg) のみアクセスできます。これに対し、カーネルモードでは、4 つのセグメントの仮想アドレス空間 kuseg、kseg0、kseg1、kseg2 をすべてアクセスできます。

セグメントによりキャッシュ可/不可が決められていますが、TX19A ではキャッシュを実装しないので、この区別に意味はありません。

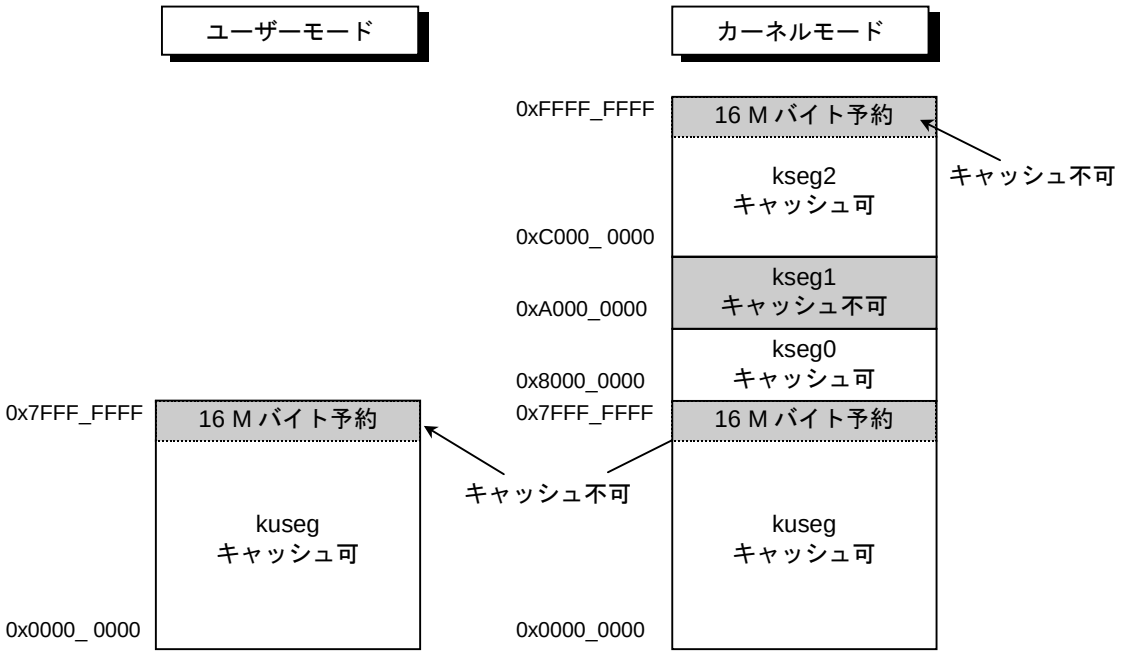


図 6-1　仮想アドレス空間

■ kuseg (カーネル・ユーザーセグメント)

カーネルモード、ユーザーモードでアクセスできる 2G バイトのセグメントです。アドレス 0x0000_0000 から 0x7FFF_FFFF までの仮想アドレス空間で、ユーザーモードの仮想アドレスの最上位ビットはかならず 0 になります。ユーザーモードのプログラムが、仮想アドレスの最上位ビットが 1 のアドレスをアクセスしようとすると、アドレスエラー例外が発生します。kuseg の上位 16 M バイトは、チップ上に内蔵された周辺回路用に予約されており、使用できません。

■ kseg0・kseg1・kseg2 (カーネルセグメント)

カーネルモード時のみにアクセスできる仮想アドレス空間は、仮想アドレス 0x8000_0000 から 0xFFFF_FFFF までの 2G バイトで、kseg0、kseg1、kseg2 の 3 つのセグメントに分割されています。

- kseg0 は、仮想アドレス 0x8000_0000 から始まる 512 M バイトのセグメントで、キャッシュ可能領域です。
- kseg1 は、仮想アドレス 0xA000_0000 から始まる 512 M バイトのセグメントです。kseg0 と異なり、キャッシュ不可領域です。
- kseg2 は、仮想アドレス 0xC000_0000 から始まる 1G バイトのリニアアドレス空間です。kseg2 の上位 16 M バイトはチップ上に内蔵された周辺回路用に予約されており、使用できません。0xFF20_0000 から 0xFF3F_FFFF

の 2 M バイトのアドレス空間は、デバッグ用に予約されています。上位 16 M バイトはキャッシュ不可領域で、それ以外の領域はキャッシュ可能領域です。

6.3　アドレス変換

仮想アドレスは、ダイレクトセグメントマッピング方式で物理アドレスに変換されます。カーネルモードのソフトウエアは、仮想ページの管理を必要とせずに、ユーザーモードアクセスから保護されます。図 6-2に、仮想アドレスと物理アドレスのマッピングを示します。

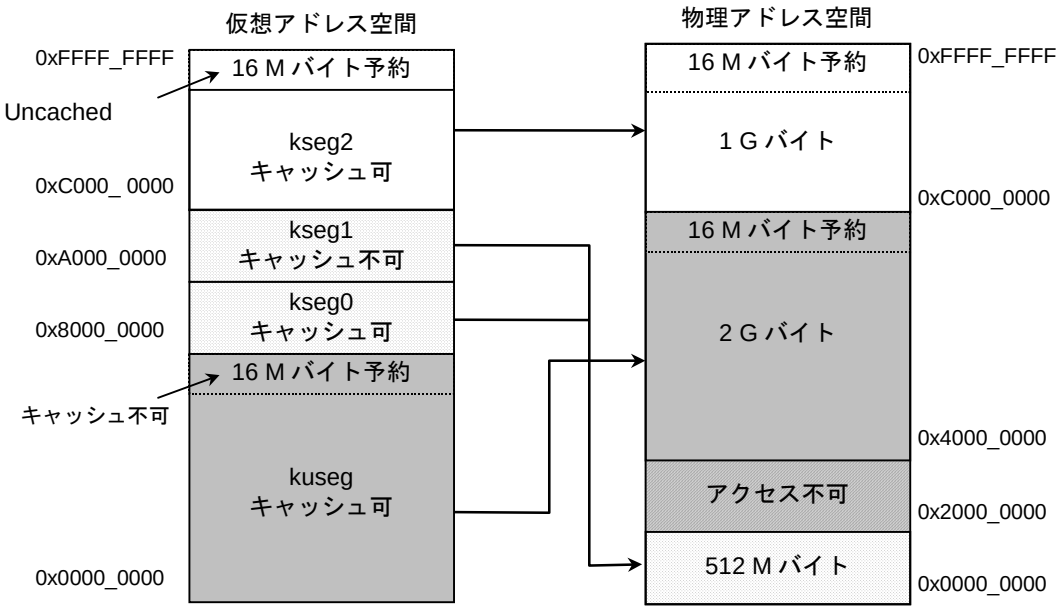


図 6-2　仮想アドレスから物理アドレスへの変換

図 6-3に、TX19Aで使用される仮想アドレスの構成を示します。上位 3 ビットによりセグメント番号を表し、この 3 ビットだけで仮想アドレスから物理アドレスへ変換されます。

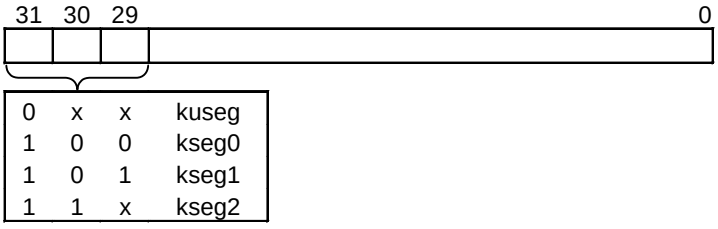


図 6-3　仮想アドレス構成

- kuseg は、アドレス 0x4000_0000 から始まる 2 G バイトの物理アドレス空間にマッピングされます。仮想アドレスの上位 2 ビット 0x を 01 に置き換えると物理アドレスになります。

- kseg0 と kseg1 の仮想アドレスは、アドレス 0x0000_0000 から始まる 512 Mバイトの物理アドレス空間にマッピングされます。仮想アドレスの上位 3 ビットが 100 のとき、仮想アドレスは kseg0 にあります。仮想アドレスの上位 3 ビットが 101 のとき、仮想アドレスは kseg1 にあります。仮想アドレスの上位 3 ビットを 000 に置き換えると物理アドレスになります。
- kseg2 の仮想アドレスは、物理アドレスとしてそのまま出力されます。

表 6-1　仮想アドレスと物理アドレスのマッピング

セグメント		仮想アドレス	物理アドレス	キャッシュ	動作モード
kseg2	予約	0xFF20_0000~0xFFFF_FFFF	0xFF00_0000~0xFFFF_FFFF	不可	カーネル
	自由	0xC000_0000~0xFEFF_FFFF	0xC000_0000~0xFEFF_FFFF	可	カーネル
kseg1		0xA000_0000~0xBFFF_FFFF	0x0000_0000~0x1FFF_FFFF	不可	カーネル
kseg0		0x8000_0000~0x9FFF_FFFF	0x0000_0000~0x1FFF_FFFF	可	カーネル
kuseg	予約	0x7F00_0000~0x7FFF_FFFF	0xBF00_0000~0xBFFF_FFFF	不可	カーネル・ユーザー
	自由	0x0000_0000~0x7EFF_FFFF	0x4000_0000~0xBEFF_FFFF	可	カーネル・ユーザー

セグメントをまたいで、プログラムを置くことは禁止されています。また、ジャンプ命令、分岐命令を使用して現在のセグメントの外に分岐することはできません。

第7章　内部 I/O バスオペレーション

7.1　内部メモリインタフェース

TX19Aコアのバスインタフェースの一例を 図 7-1に示します。TX19Aコアのメモリインタフェースは、基本的には、命令バスとオペランドバスとが独立したハーバーアーキテクチャです。それぞれのバスは互いに独立しており、しかも 1 クロック当たり 1 ワードのデータへアクセスできます。これらのバス仕様によりTX19Aコアは、1 クロック当たり 1 命令の実行を保証しています。

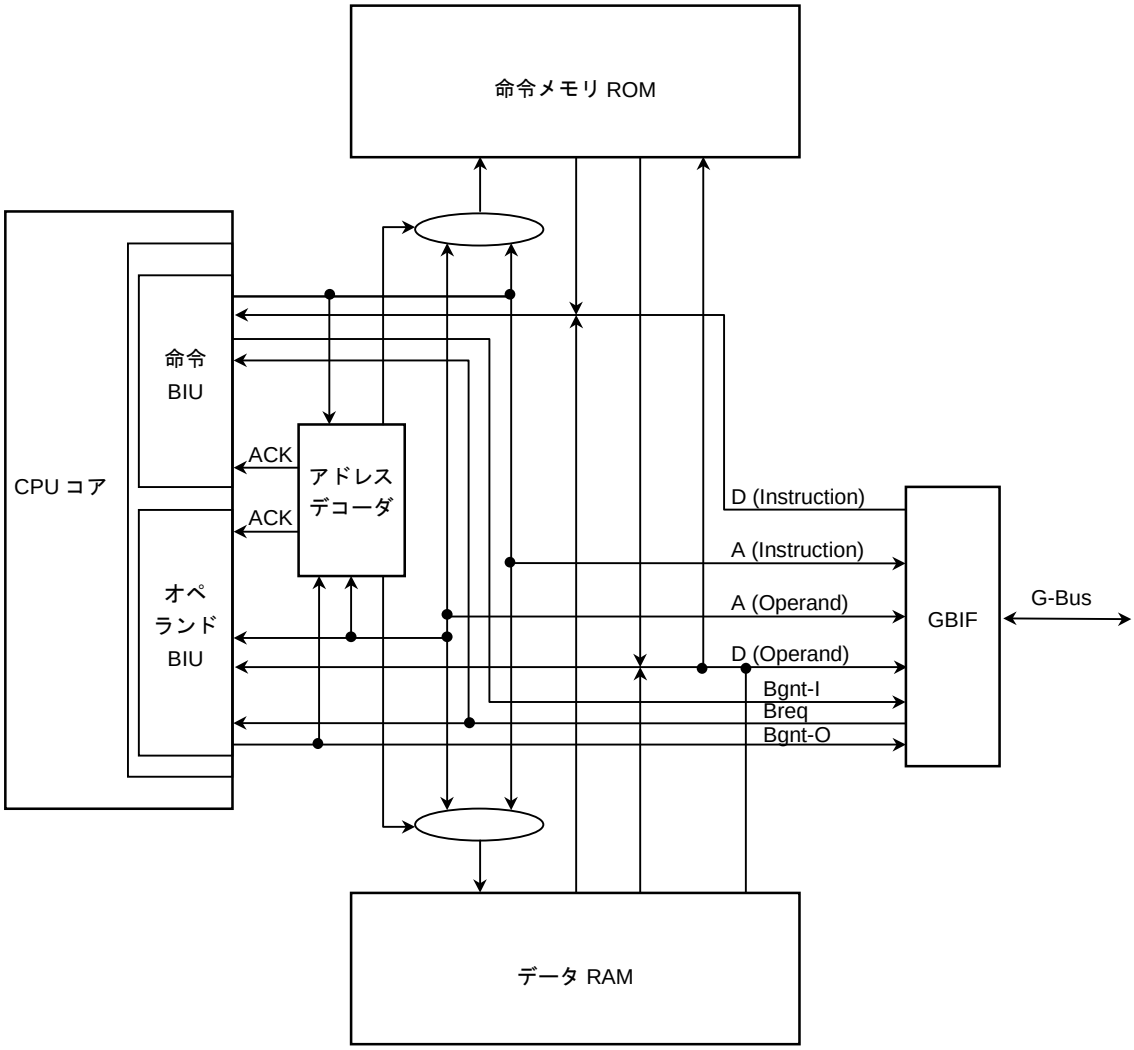


図 7-1　内部メモリインタフェースの概要

7.2　オペランドリード・命令フェッチオペレーション

TX19A コアのバスサイクルの特徴は、先行のバスサイクルからデータを読み込む動作が終了しないうちに後続する次のバスサイクルのアドレスを出力する、パイプラインバスサイクルになっていることです。これにより、フラッシュメモリのような動作が比較的遅いメモリを用いた場合でも、ゼロウエイトの動作が可能です。

オペランドリード時および命令フェッチ時のバスサイクルのタイミングチャート図を図 7-2、図 7-3に示します。

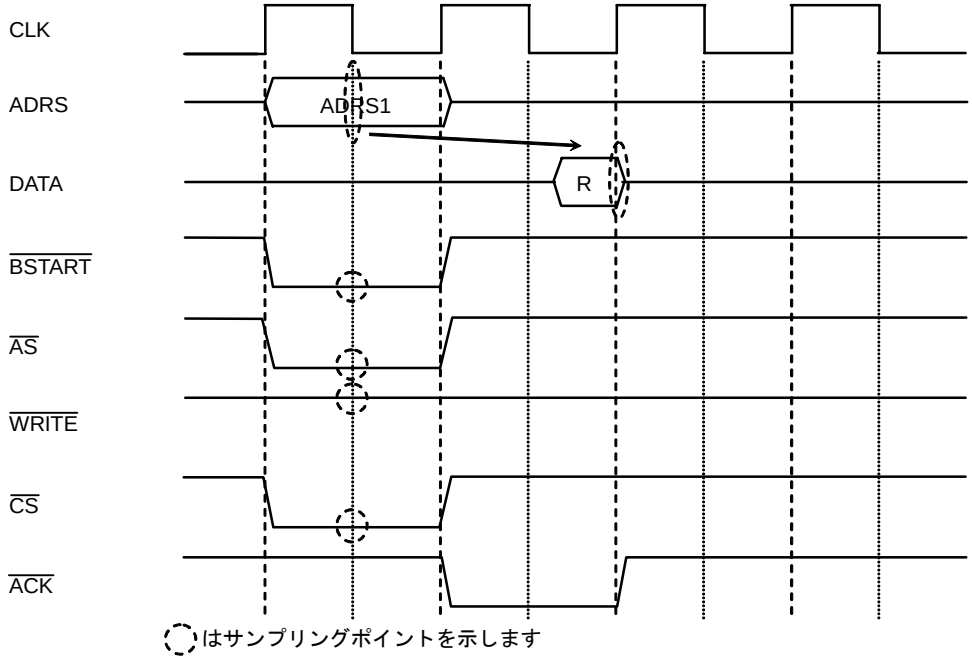


図 7-2　メモリリードのタイミング (ゼロウエイト)

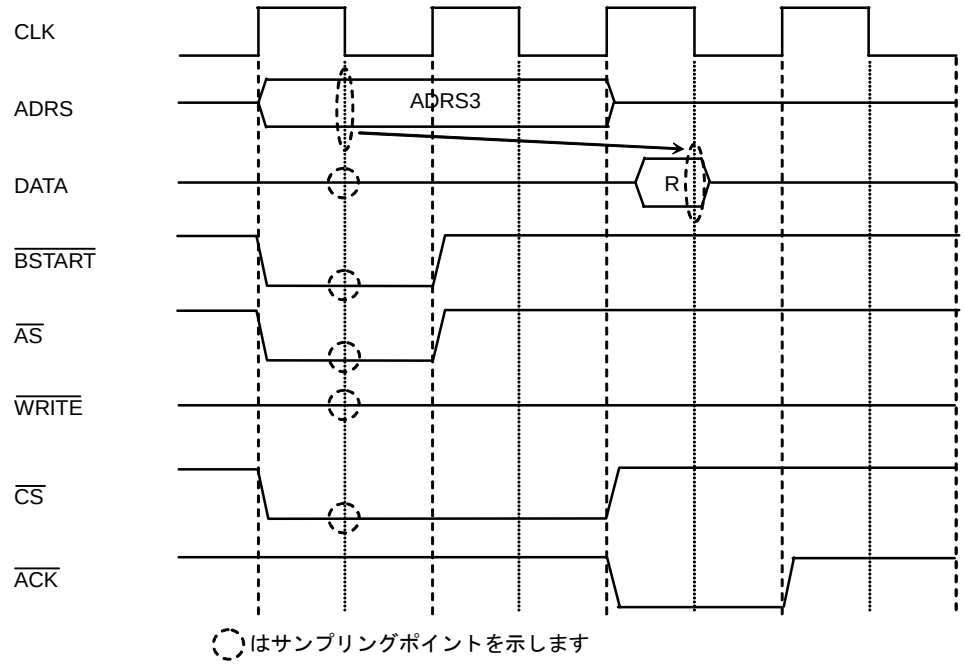


図 7-3　メモリリードタイミング (ADRS3 に対してウエイトを設定した場合)

7.3　ライトオペレーション

ライトオペレーションの動作は、基本的には、リードと同じです。

システムクロックの立ち上がりに同期してアドレスを出力します。これと同時にアサートする信号は、バイトイネーブル信号、バススタート信号($\overline{\text{BSTART}}$), アドレスストローブ信号($\overline{\text{AS}}$), ライト信号($\overline{\text{WRITE}}$) 、チップセレクト信号($\overline{\text{CS}}$)です。

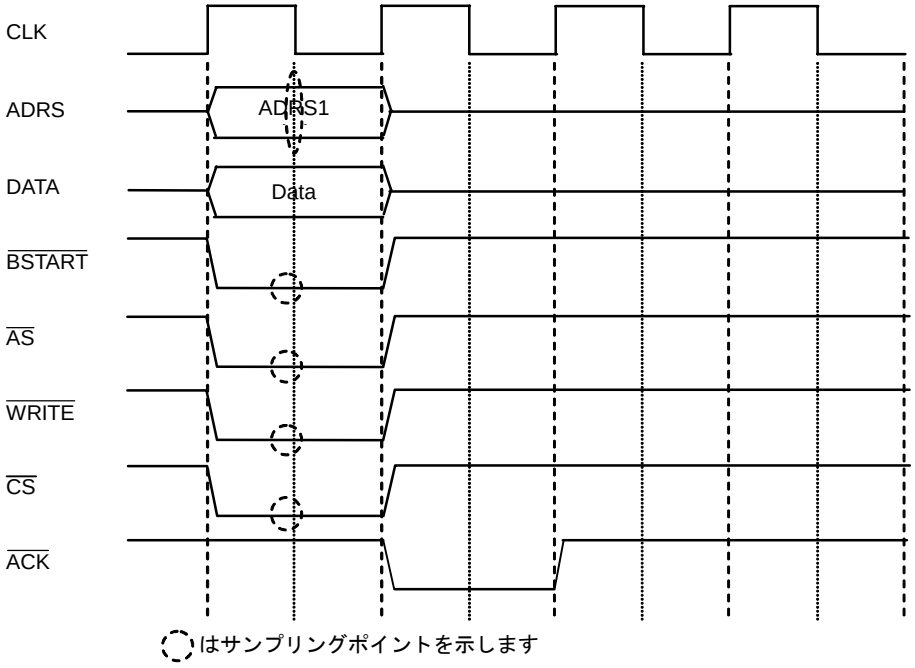


図 7-4　ライトタイミング (ゼロウエイト)

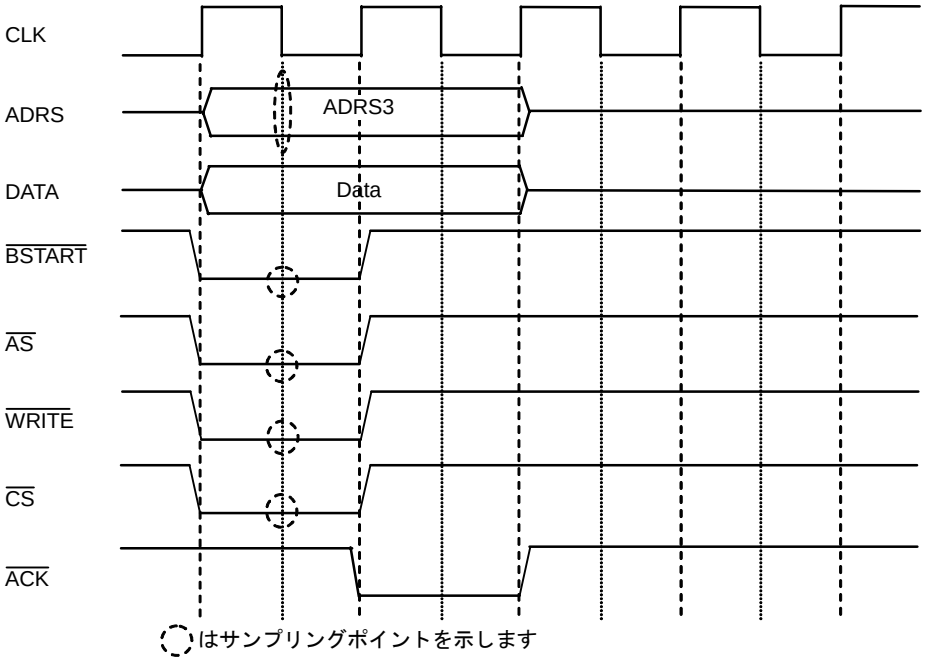


図 7-5　ライトオペレーションのタイミング (ADR3 に対してウエイトを設定した場合)

第8章　システム制御コプロセッサ (CP0) レジスタ

この章では、システム構成、メモリ管理、例外処理で使用するシステム制御コプロセッサ (CP0) について説明します。

プロセッサがカーネルモードのとき、システム制御コプロセッサ命令は、常に CP0 レジスタを使用できます。プロセッサがユーザーモードのときは、Status レジスタの CU0 ビットが 1 に設定されているときのみ CP0 レジスタを使用できます。

8.1　概要

表 8-1にCP0 レジスタの一覧を示します。レジスタ番号は、MFC0 (Move From CP0) 命令とMTC0 (Move To CP0) 命令で使われます。

表 8-1　CP0 レジスタ

分類	レジスタ名	レジスタ番号	説　明
システム構成	Config Config1 Config2 Config3	16 (SEL0) 16 (SEL1) 16 (SEL2) 16 (SEL3)	TX19A プロセッサのさまざまなコンフィグレーションです。
一般例外処理	BadVAddr	8 (SEL0)	仮想アドレスから物理アドレスへの変換でエラーを起こした仮想アドレスを示します。読み出し専用です。
	Count	9 (SEL0)	タイマのカウントアップレジスタです。
	Compare	11 (SEL0)	タイマのカウント比較レジスタです。
	Status	12 (SEL0)	動作モード (ユーザー・カーネル)、割り込み許可状態などのプロセッサの状態を保持します。
	Cause	13 (SEL0)	直前に発生した例外の原因を示します。
	EPC	14 (SEL0)	例外の原因となった命令のアドレス、すなわち、例外処理後のプログラムの戻りアドレスと、例外発生時の ISA モードを保持します。
	ErrorEPC	30 (SEL0)	Reset、NMI 専用の例外 PC 格納レジスタです。
	PRId	15 (SEL0)	TX19A プロセッサのリビジョンを示します。読み出し専用です。
	IER	9 (SEL7)	Status レジスタの割り込み許可・禁止ビットを操作します。
	SSCR	22 (SEL0) ／9 (SEL6)	Shadow Register の設定レジスタです。
デバッグ例外処理	Debug	23 (SEL0)	デバッグ例外の原因と現在の状態を示します。
	DEPC	24 (SEL0)	デバッグ例外の原因となった命令のアドレス、すなわち、デバッグ例外処理後のプログラムの戻りアドレスと、例外発生時の ISA モードを保持します。
	DESAVE	31 (SEL0)	ICE システムの専用レジスタです。

以下の項では、CP0 レジスタの構成とレジスタの各ビットの意味について説明します。見出し中で「8.2.1 Config Register (16:SEL0)」などレジスタ名の後の番号は、レジスタ番号を表します。

8.2　システム構成レジスタ

8.2.1　Config Register (16:SEL0)

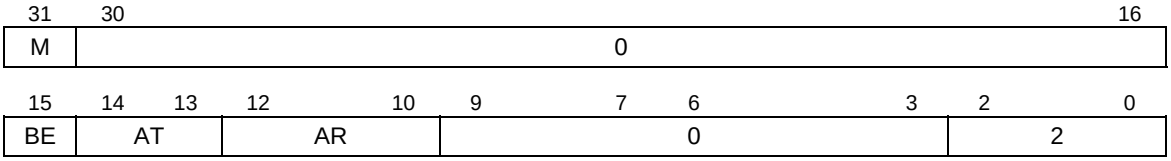


表 8-2　Config レジスタ

フィールド		説明	リード･ライト	リセット後
名前	ビット			
M	31	Config1 Register が存在することを示します。 1 に固定されています	R	1
―	30:16	0 に固定されています。	R	0
BE	15	Endian を決めます。製品によって、実装時にいずれかに固定します。 0: Little Endian 1: Big Endian	R	注 1
AT	14:13	MIPS アーキテクチャの実装レベルを示します。 0: MIPS32 1: MIPS64 with access only to 32-bit compatibility segments 2: MIPS64 with access to all address segments 3: Reserved 0 に固定されています。	R	0
AR	12:10	MIPS アーキテクチャのリヴィジョンを示します。 0: Revision 1 1-7: Reserved 0 に固定されています。	R	0
―	9:3	0 に固定されています	R	0
―	2:0	2 に固定されています。	R	2

注１　製品によって、実装時に０または１に固定されます。

8.2.2　Config1 Register (16:SEL1)

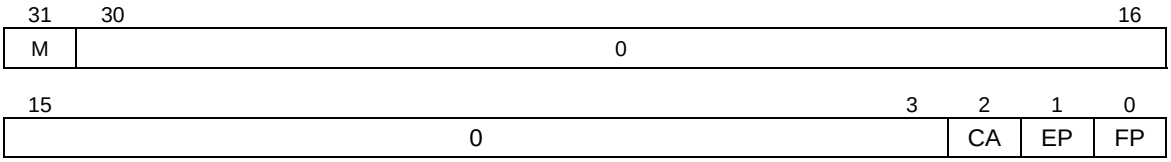


表 8-3　Config1 レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
M	31	Config2 Register が存在することを示します 1 に固定されています	R	1
―	30:3	0 に固定されています	R	0
CA	2	16 ビットのコードがインプリメントされていることを示します 0: MIPS16ASE not implemented 1: MIPS16ASE implemented 1 に固定されています。	R	1
EP	1	EJTAG デバッグ機構が実装されていることを示します 0: No EJTAG implemented 1: EJTAG implemented 1 に固定されています。	R	1
FP	0	FPU が実装されていることを示します 0: No FPU implemented 1: FPU implemented 0 に固定されています。	R	0

(注意) Config1 レジスタは読み出し専用です。

8.2.3　Config2 Register (16:SEL2)

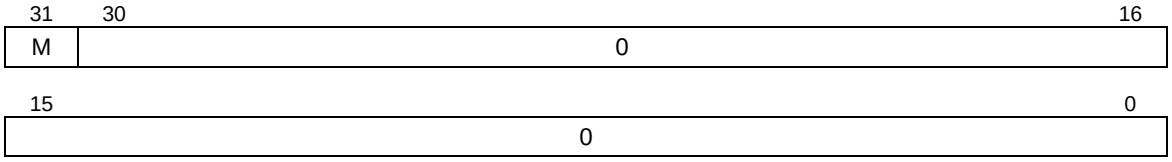


表 8-4　Config2 レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
M	31	Config3 Register が存在することを示します 0 に固定されています。	R	1
－	30:0	0 に固定されています。	R	0

(注意) Config2 レジスタは読み出し専用です。

8.2.4　Config3 Register (16:SEL3)

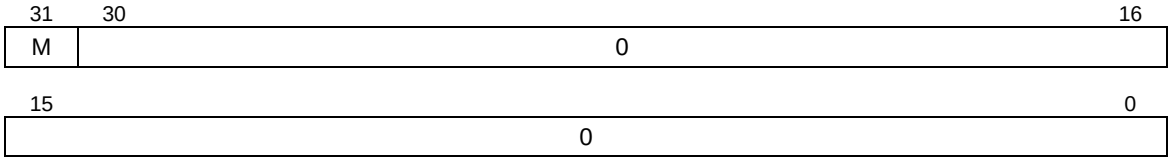


表 8-5　Config3 レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
M	31	Config4 Register Select field 0 に固定されています。	R	0
－	30:0	0 に固定されています。	R	0

(注意) Config3 レジスタは読み出し専用です。

8.3　一般例外処理レジスタ

この項では、一般例外処理で使用されるCP0 レジスタについて説明します。デバッグ例外処理で使用されるCP0 レジスタは、「8.4 デバッグ例外処理レジスタ」で説明します。

8.3.1　BadVAddr Register (8)

BadVaddr (Bad Virtual Address) レジスタは、読み出し専用のレジスタで、直前に仮想アドレスから物理アドレスへの変換でエラーを起こした仮想アドレスを表示します。アドレスエラー例外 (AdEL または AdES) が発生します。

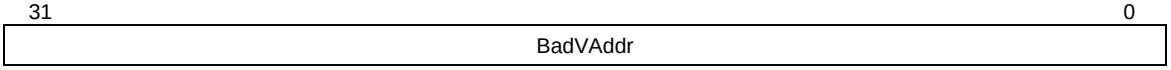


表 8-6　BadVAddr レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
BadVAddr	31:0	Bad Virtual Address	R	Undefined

(注意) BadVAddr レジスタは読み出し専用です。

8.3.2　Count Register (9:SEL0)

Count レジスタは、リード/ライトレジスタです。このレジスタはタイマとして作用し CPUCLK の 1/2 の速度で 1 ずつインクリメントされます。

プロセッサの入力端子 GTINTDIS 信号が “0” に設定されている場合はインクリメントされます。 “1” に設定されている場合はインクリメントされません。

このレジスタには、診断目的またはシステム初期設定のために書き込むことも可能です。

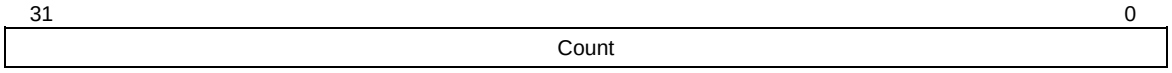


表 8-7　Count レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
Count	31:0	Interval counter	R/W	Undefined

8.3.3　Compare Register (11)

Count レジスタの値が Compare レジスタの値と等しくなると Cause レジスタ内の割り込みビット IP[7]がセットされます。割り込みがイネーブルのときただちに割り込み例外が発生します。

このレジスタに値を書き込むことによりタイマ割り込みがクリアされます。

このレジスタは、診断用としてはリード/ライトレジスタです。通常のオペレーションでは、このレジスタは書き込み専用です。

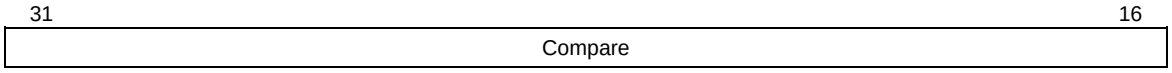


表 8-8　Compare レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
Compare	31:0	Interval count compare value	R/W	Undefined

8.3.4　Status Register (12)

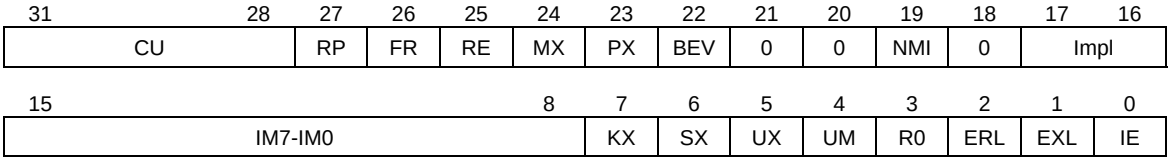


表 8-9　Status レジスタ (1/2)

フィールド		説明	リード・ライト	リセット後
名前	ビット			
CU (CU3, … CU0)	31:28	4 つのコプロセッサユニット番号のそれぞれの使用可否をコントロールします。カーネルモードの CP0 は、CU0 ビットの設定値に関係なく、常に使用可能です。CU3、CU2、CU1 は、0 を書き込んでください。 0: 使用不可能 1: 使用可能	R/W	Undefined
RP	27	低消費電力モード 0: Halt モード 1: Doze モード WAIT 命令が実行されたときの TX19A の低消費電力モードの種別を示します。TX19A は Halt モードと Doze モードの両方でパイプラインをストールします。Halt モードは、Doze モードよりも電力消費量を削減できます。	R/W	0
FR	26	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
RE	25	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
MX	24	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
PX	23	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
BEV	22	ブートストラップ例外ベクタ このビットは、リセット時に 1 にセットされます。BEV=1 のとき、ブートストラップ代替ベクタには例外ベクタが使用されます。ブートストラップ代替ベクタは、キャッシュ不可な kseg1 を参照するため、通常キャッシュ機能が有効になる前に診断テストを実行するのに使用されます。BEV=0 のとき、リセット、NMI 割り込み、デバッグ例外ベクタはキャッシュ不可な kseg1 を参照し、それ以外の例外ベクタはキャッシュ可能な kseg0 を参照します。	R/W	1
TS	21	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
SR	20			
NMI	19	NMI 割り込み信号が L にアサートされると、このビットは 1 にセットされます。このビットは、0 を書き込むことでクリアされます。1 を書き込んでも無視されます。この動作は、TX19 コアとは異なります。	R/W	0
ー	18	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
Impl	17:16	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
IM (IM7, … IM0)	15:8	割り込みマスク 外部割り込み、タイマ割り込み、ソフトウエア割り込みのそれぞれについてイネーブル/ディセーブルを制御します。割り込みが受け付けられるのは、割り込みがイネーブル (IE=1) されており、かつ IM フィールドと Cause レジスタの IP フィールドの対応ビットがともにセットされている場合です。 0: ディセーブル 1: イネーブル	R/W	0x00

表 8-9　Statusレジスタ (2/2)

フィールド		説明	リード・ライト	リセット後
名前	ビット			
KX	7	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
SX	6	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
UX	5	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
UM	4	動作モード 0: カーネルモード 1: ユーザーモード TX19A では、カーネルモードのみ使用してください。	R/W	0
ー	3	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
ERL	2	Error Level: このビットは、Reset、NMI が発生するとセットされます。 このビットがセットされているときは、 ・ カーネルモードで動作しています。 ・ 割り込みは禁止状態です。 ・ ERET 命令は、ErrorEPC を復帰 PC として使用。	R/W	1
EXL	1	Exception Level: このビットは、例外 (Reset、NMI を除く) が発生するとセットされます。 このビットがセットされている時は、 ・ カーネルモードで動作します。 ・ 割り込みは禁止状態です。 ・ 多重に例外が発生しても、EPC レジスタ、BDbit (Cause) は、更新されません。	R/W	0
IE	0	Interrupt Enable: 0: 割り込み禁止 1: 割り込み許可 このビットは、割り込み応答/ERET 命令によるセット/クリアは行われません (Reset 初期化を除く)。	R/W	0

8.3.5　Cause Register (13)

Cause レジスタは、最新の例外の原因を保持しています。このレジスタは、IP[1:0]、IV ビットを除いて読み出し専用です。

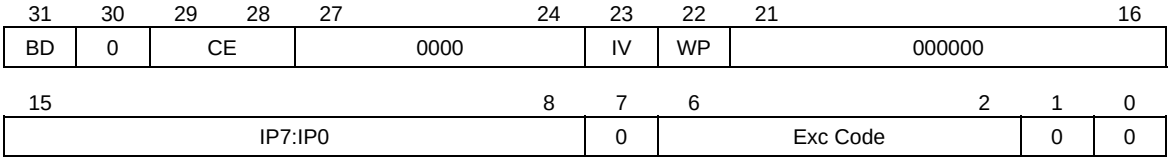


表 8-10　Cause レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
BD	31	ジャンプ命令、または分岐命令スロット内の命令を実行している時に、例外が発生すると 1 にセットされます。EXL ビットが 0 のときに割り込み、例外が発生した場合に更新されます。	R	Undefined
―	30	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
CE[1:0]	29:28	コプロセッサエラー: コプロセッサ使用不可例外が発生したときに参照されたコプロセッサを示します。 このビットは、コプロセッサ使用不可例外以外の例外が発生したときは不定です。	R	Undefined
―	27:24	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
IV	23	割り込みベクタ: このビットをセットすると割り込みベクタは他の例外と異なるベクタを生成します。 BEV (Status)　IV　割り込みベクタ 0　0　0x8000_0180 0　1　0x8000_0200 1　0　0xBFC0_0380 1　1　0xBFC0_0400	R/W	Undefined
WP	22	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
―	21:16	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
IP[7:2]	15:10	割り込み要求 (Hradware) IP[7]: Hardware interrupt 5 or timer interrupt IP[6]: Hardware interrupt 4 IP[5]: Hardware interrupt 3 IP[4]: Hardware interrupt 2 IP[3]: Hardware interrupt 1 IP[2]: Hardware interrupt 0 IP[7] の timer interrupt は、Count ($9)と Compare ($11) の値が一致したときに発生します。	R	Undefined
IP[1:0]	9:8	割り込み要求 (Software) IP[1]: Request software interrupt 1 IP[0]: Request software interrupt 0	R/W	Undefined
―	7	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0
ExcCode	6:2	例外要因コード (表 8-11参照: Exception Code Field)	R	Undefined
―	1:0	このビットは、常に 0 が読み出されます。 書き込み動作は無視されます。	R	0

表 8-11　例外要因コード (Exception Code Field)

例外要因コード		ニモニック	説明
10 進	16 進		
0	0x00	Int	割り込み (ソフトウエア、ハードウエア)
4	0x04	AdEL	アドレスエラー (命令フェッチまたはロード)
5	0x05	AdES	アドレスエラー (ストアアクセス)
6	0x06	IBE	バスエラー (命令フェッチ)
7	0x07	DBE	バスエラー (データアクセス: ロード)
8	0x08	Sys	システムコール例外
9	0x09	Bp	ブレイクポイント例外
10	0x0a	RI	予約命令例外
11	0x0b	CpU	コプロセッサ使用不可例外
12	0x0c	Ov	整数オーバーフロー例外
13	0x0d	Tr	トラップ例外
その他	(Reserved)		

8.3.6　EPC Register (14)

EPC レジスタは、リード/ライトレジスタです。このレジスタには、例外処理後に処理が再開するアドレスが格納されます。

同期例外では、以下のいずれかがレジスタに書き込まれます。

- その例外の直接原因となった命令の仮想アドレス
- 直前の分岐またはジャンプ命令の仮想アドレス (その命令が分岐遅延スロット内に存在する場合は、Cause レジスタ内の分岐遅延ビットが設定されます)。

Status レジスタの EXL ビットが 1 に設定された状態で検出した場合、EPC レジスタへの書き込みは行われません。

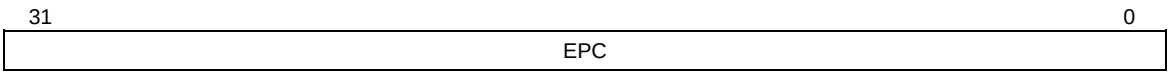


表 8-12　EPC レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
EPC	31:0	Exception Program Counter	R/W	Undefined

8.3.7　PRId Register (15)

PRId レジスタは、読み出し専用レジスタです。このレジスタは、CPU と CP0 の実装およびリビジョンレベルを識別する情報が含まれています。

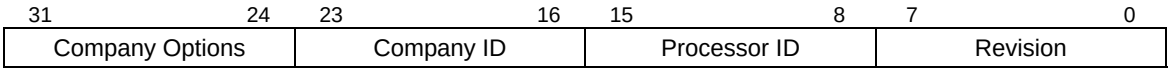


表 8-13　PRId レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
Company Options	31:24	カンパニーコードのオプション 0 に固定されています。	R	0x00
Company ID	23:16	カンパニー ID 東芝のカンパニーコードは 0x07 です。	R	0x07
Processor ID	15:8	Processor ID TX19A は 0x40 になります。	R	0x40
Revision	7:0	Revision TX19A は 0x00 が設定されています。	R	0x00

(注意) PRId レジスタは読み出し専用です。

8.3.8　ErrorEPC Register (30)

ErrorEPC レジスタは、リード/ライトレジスタで、EPC レジスタに似ています。このレジスタは、リセット、および NMI 例外発生時にプログラムカウンタ (PC) の値を格納するために使用されます。

ErrorEPC レジスタに格納されるアドレスは以下の通りです。

- その例外を発生させた命令の仮想アドレス
- このアドレスが分岐遅延スロット内に存在する場合には、直前の分岐またはジャンプ命令の仮想アドレス

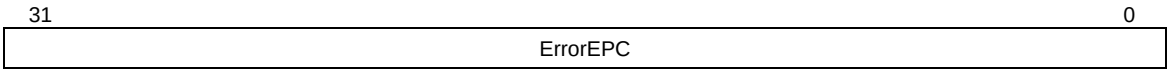


表 8-14　ErrorEPC レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
ErrorEPC	31:0	Error Exception Program Counter	R/W	Undefined

8.3.9　Shadow Register Set Control Register: SSCR (22　or　9：SEL6)

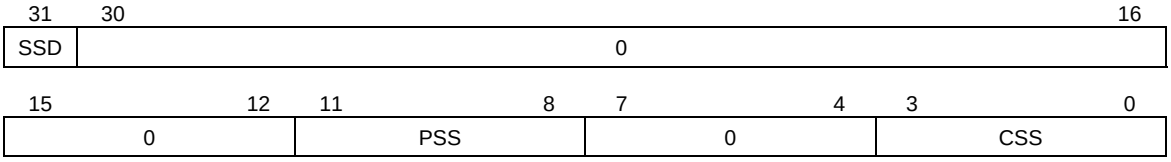


表 8-15　Shadow Register Set Control レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
SSD	31	Shadow Register Set Disable Signal 0: MIPS32 Version.1.0 with Shadow Register Set 1: MIPS32 Version.1.0	R/W	1
－	30:12	Reserved bits	R	0
PSS	11:8	Previous Shadow Register Set x000: Main GPRs x001: Shadow Register 1 x010: Shadow Register 2 x011: Shadow Register 3 x100: Shadow Register 4 x101: Shadow Register 5 x110: Shadow Register 6 x111: Shadow Register 7	R/W	Undefined
－	7:4	Reserved bits	R	0
CSS	3:0	Current Shadow Register Set x000: Main GPRs x001: Shadow Register 1 x010: Shadow Register 2 x011: Shadow Register 3 x100: Shadow Register 4 x101: Shadow Register 5 x110: Shadow Register 6 x111: Shadow Register 7	R/W	0000

注 1:　このレジスタは、リード/ライト可能です。

注 2:　CSS は割り込みコントローラからの割り込み要求 (レベル信号) を受け付けると、割り込みレベルと同じ値の Shadow Register Set 番号に書き換えられます。同時に、更新前の CSS の値が PSS に書かれます。

注 3:　ERET 命令が実行されると、PSS の値が CSS に書かれます。

注 4:　このレジスタを更新した場合、パイプラインハザードを回避するために後続に 2 つの NOP 命令を必ず置いてください。

```
例)  MTC0    r18, SSCR
     NOP
     NOP
     ADD     r19, r12, r13
```

注 5:　SSD ビットが 1 にセットされた状態では、割り込みが発生しても Shadow Register Set は更新されません。

注 6:　SSD ビットが 1 にセットされた状態では、アクセスできる Shadow Register Set は 0 番になり、CSS の値は無視されることになります。

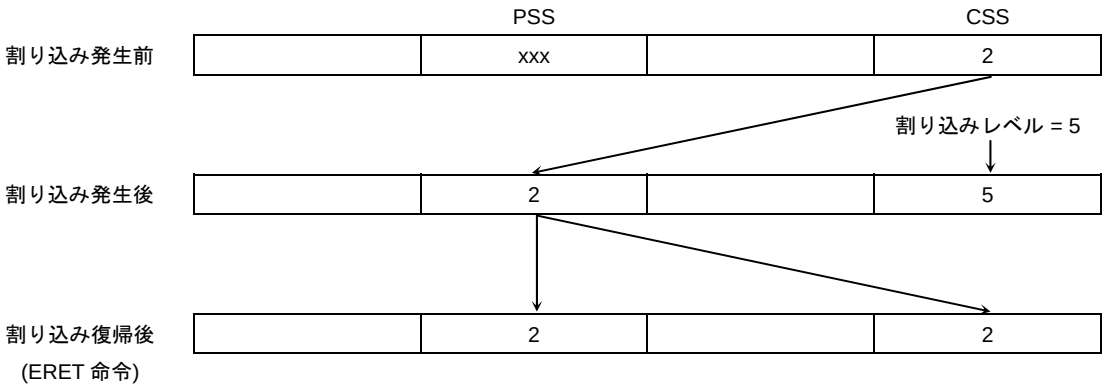


図 8-1 割り込み発生時、割り込み復帰時の SSCR レジスタの動作

8.3.10 IER Register (9:SEL7)

IER レジスタは、Status レジスタの IE ビットをセットまたはクリアするために使用します。IER レジスタに 0 を書き込むと Status レジスタの IE ビットがクリアされます。IER レジスタに 0 以外の値を書き込むと IE ビットがセットされます。割り込みをディセーブルするには、「MTC0 r0, IER」命令を使用します。割り込みをイネーブルするには、「MTC0 $sp, IER」のようにターゲットレジスタに 0 以外の値を使用します。

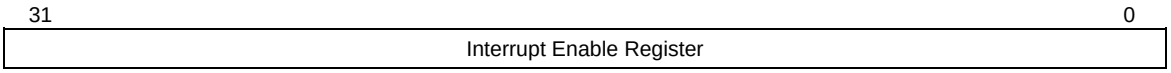


表 8-16　IER レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
IER	31:0	Interrupt Enable Register Status レジスタの IE ビットを操作します。 このレジスタに 0 を書き込むと IE ビットは、0 にクリアされます。0 以外の値を書き込むと IE ビットは 1 にセットされます。	R/W	Undefined

8.4　デバッグ例外処理レジスタ

TX19A ではデバッグのためにプログラムの実行を任意に停止させることができます。TX19A にはプログラムのデバッグを容易にするためのレジスタが用意されています。この項ではそれらのレジスタについて説明します。

8.4.1　Debug Register (23)

Debug レジスタは、デバッグ例外が発生したときの状態を保持するとともに、デバッグ処理の設定を行うことができます。プログラム中に SDBBP (Software Debug Breakpoint) 命令を埋め込むことにより、SDBBP 命令が実行された時点で、デバッグブレークポイント例外を発生させることができます。また、Debug レジスタの SSt ビットをセットすることで、シングルステップ実行を有効にできます。このとき、1 命令実行するごとにシングルステップ例外が発生します。

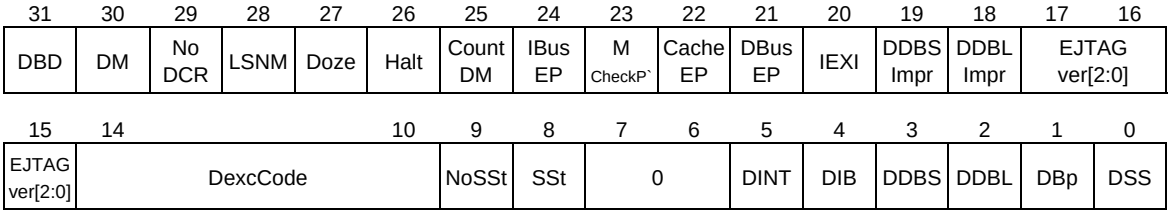


表 8-17　Debug レジスタ (1/3)

フィールド		説明	リード・ライト	リセット後
名前	ビット			
DBD	31	デバッグ分岐遅延: ジャンプ命令、または分岐命令スロット内に命令が存在するときにデバッグ例外が発生すると、このビットが 1 に設定されます。	R	Undefined
DM	30	デバッグモード: デバッグ例外が発生したことを示します。デバッグ例外の発生でセットされ、例外からの復帰 (DERET) でクリアされます。	R	0
NoDCR	29	dseg メモリセグメント: 0: 存在する 1: 存在しない	R	0
LSNM	28	Controls access of load/store between dseg and remaining memory when dseg is present: 0: Load/store in dseg address range go to dseg 1: Load/store in dseg address range go to system memory	R	0

表 8-17　Debugレジスタ (2/3)

フィールド		説明	リード・ライト	リセット後
名前	ビット			
Doze	27	Low Power Mode Flag (Doze): デバッグ例外が発生したときに、プロセッサ自身が Doze による Low Power Mode だった場合にこのビットはセットされます。	R	Undefined
Halt	26	Low Power Mode Flag (Halt): デバッグ例外が発生したときに、プロセッサ自身が Halt による Low Power Mode だった場合にこのビットはセットされます。	R	Undefined
CountDM	25	Count レジスタ制御: 0: Debug モード中は停止する。 1: Debug モード中は動作する。	R/W	0
IBusEP	24	命令バスエラー保留: このビットは、デバッグモード中にバスエラー (命令) が検出されるとき、あるいはソフトウエアによって 1 が書き込まれたときにセットされます。 このビットは、バスエラー例外に応答したときはクリアされます。 このビットがセットされている状態で、IEXI ビットをクリアすると、保留されていたバスエラー例外が発生し、このビットはクリアされます。 このビットは、0 書き込みは無視されます。	R/W1	0
McheckP	23	このビットは、TX19A では実装しない。 常に 0 が読み出されます。	R	0
CacheEP	22	このビットは、TX19A では実装しない。 常に 0 が読み出されます。	R	0
DBusEP	21	データバスエラー保留: このビットは、デバッグモード中にデータバスエラーが検出されるとき、あるいはソフトウエア 1 が書き込まれたときにセットされます。 このビットは、バスエラー例外に応答したときはクリアされます。 このビットがセットされている状態で、IEXI ビットをクリアすると、保留されていたバスエラー例外が発生し、このビットはクリアされます。 このビットは、0 書き込みは無視されます。	R/W1	0
IEXI	20	An Imprecise Error eXception Inhibit (IEXI) このビットは、デバッグ例外の発生あるいはデバッグモード中に発生した例外によってセットされ、DERET 命令の実行によってクリアされます。 その他にプログラムによてセット/クリアできます。 このビットがセットされているときに、バスエラー (命令、データ) が発生しても応答はしません。 このビットがクリアされたら応答します。	R/W	0
DDBS Impr	19	Debug Data Break Store Imprecise Exception: ライトバスサイクルに対してデータアドレスブレイクが発生すると 1 にセットされます。デバッグモード中に一般例外が発生するとクリアされます。	R	Undefined
DDBL Impr	18	Debug Data Break Load Imprecise Exception: リードバスサイクルに対してデータアドレスブレイクが発生すると 1 にセットされます。デバッグモード中に一般例外が発生するとクリアされます。	R	Undefined
EJATGver	17:15	EJTAG version: 0: Version 1 and 2.0 1: Version 2.5 2: Version 2.6 3-7: Reserved	R	010
DExcCode	14:10	デバッグモード中の一般例外 このビットは、デバッグ例外ハンドラの動作中 (DM=1) に一般例外が発生すると、その要因コードがセットされます。要因コードについては、8.3.5 CauseレジスタのExcCodeフィールドを参照してください。	R	Undefined
NoSSt	9	シングルステップ例外の実装: 0: 実装している 1: 実装していない TX19A は 0 です。	R	0

表 8-17　Debugレジスタ (3/3)

フィールド		説明	リード･ライト	リセット後
名前	ビット			
SSt	8	シングルステップ: このビットを1に設定するとシングルステップデバック機能が有効になります。0 に設定するとシングルステップ機能が無効になります。デバッグ例外ハンドラ動作中 (DM=1) のこの機能は無効となります。	R/W	0
0	7:6	書き込みは無視され、常に 0 が読み出されます。	0	0
DINT	5	Debug Interrupt exception: デバッグ割り込みが発生すると 1 にセットされます。 デバッグモード中に一般例外が発生するとクリアされます。	R	Undefined
DIB	4	Debug Instruction Break: 命令アドレスブレークが発生すると 1 にセットされます。デバッグモード中に一般例外が発生するとクリアされます。	R	Undefined
DDBS	3	Debug Data Break Store Exception: ストア操作時にデータアドレスブレイクが発生すると 1 にセットされます。デバッグモード中に一般例外が発生するとクリアされます。 TX19A はこの例外は実装していません。	R	Undefined
DDBL	2	Debug Data Break Load Exception: ロード操作時にデータアドレスブレイクが発生すると 1 にセットされます。この場合はデータの比較は行わず、アドレス比較 (ロードの) のみで行います。 デバッグモード中に一般例外が発生するとクリアされます。 TX19A はこの例外は実装していません。	R	Undefined
DBp	1	Debug Breakpoint Exception: このビットは、SDBBP 命令によるデバッグブレークポイント例外が発生すると 1 にセットされます。デバッグモード中に一般例外が発生するとクリアされます。	R	Undefined
DSS	0	Debug Single Step Exception: シングルステップ例外が発生すると 1 にセットされます。デバッグモード中に一般例外が発生するとクリアされます。	R	Undefined

8.4.2　DEPC Register (24)

DEPC レジスタは、デバッグ例外の処理後に処理を再開する命令のアドレスを格納します。DEPC レジスタ内に格納されるアドレスは、そのデバッグ例外を発生させる命令の仮想アドレスです。その命令が分岐遅延スロット内に存在する場合には、直前の分岐またはジャンプ命令の仮想アドレスがこのレジスタ内に格納されます。DERET 命令を実行すると、DEPC アドレスにジャンプします。DEPC レジスタは、リード/ライトレジスタです。

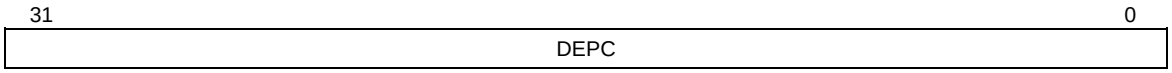


表 8-18　DEPC レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
DEPC	31:0	Debug Exception Program Counter	R/W	Undefined

8.4.3　DESAVE Register (31)

DESAVE レジスタは、デバッグ例外ハンドラによって GPR の 1 つを保存するために使用されます。DESAVE レジスタに値を保存した GPR を使用して、その Context の残りがプロセッサプローブなどのあらかじめ決められたメモリに保存されます。このレジスタを使用すると、例外ハンドラや、Context の保存に有効なスタックの存在を保証できないその他のタイプのコードを安全にデバッグすることができます。このレジスタは、ICE システム専用です。

表 8-19　DESAVE レジスタ

フィールド		説明	リード・ライト	リセット後
名前	ビット			
DESAVE	31:0	Debug Exception Save Register	R/W	Undefined

第9章　例外処理

この章では、TX19A アーキテクチャで規定されている例外処理について説明します。この章は以下の項で構成されます。

- 一般例外
- 割り込み
- デバッグ例外

9.1　一般例外

TX19A の例外は一般例外とデバッグ例外に分類されます。
この項では一般例外について、原因、処理などを説明します。

9.1.1　一般例外処理

例外とは、外部割り込み信号、エラー、または命令実行中に生じた異常状態の結果として、通常の命令シーケンスを変更する状態を示します。例外が発生すると、その時点におけるプロセッサの状態を保存し、カーネルモードに切り換えて、あらかじめ決められたアドレスに制御を移します。このアドレスを例外ベクタといい、例外ハンドラの開始アドレスを示します。

リセット例外、NMI例外以外の一般例外におけるTX19Aプロセッサの処理を 図 9-1に示します。リセット例外、NMI例外の処理 図 9-2に示します。

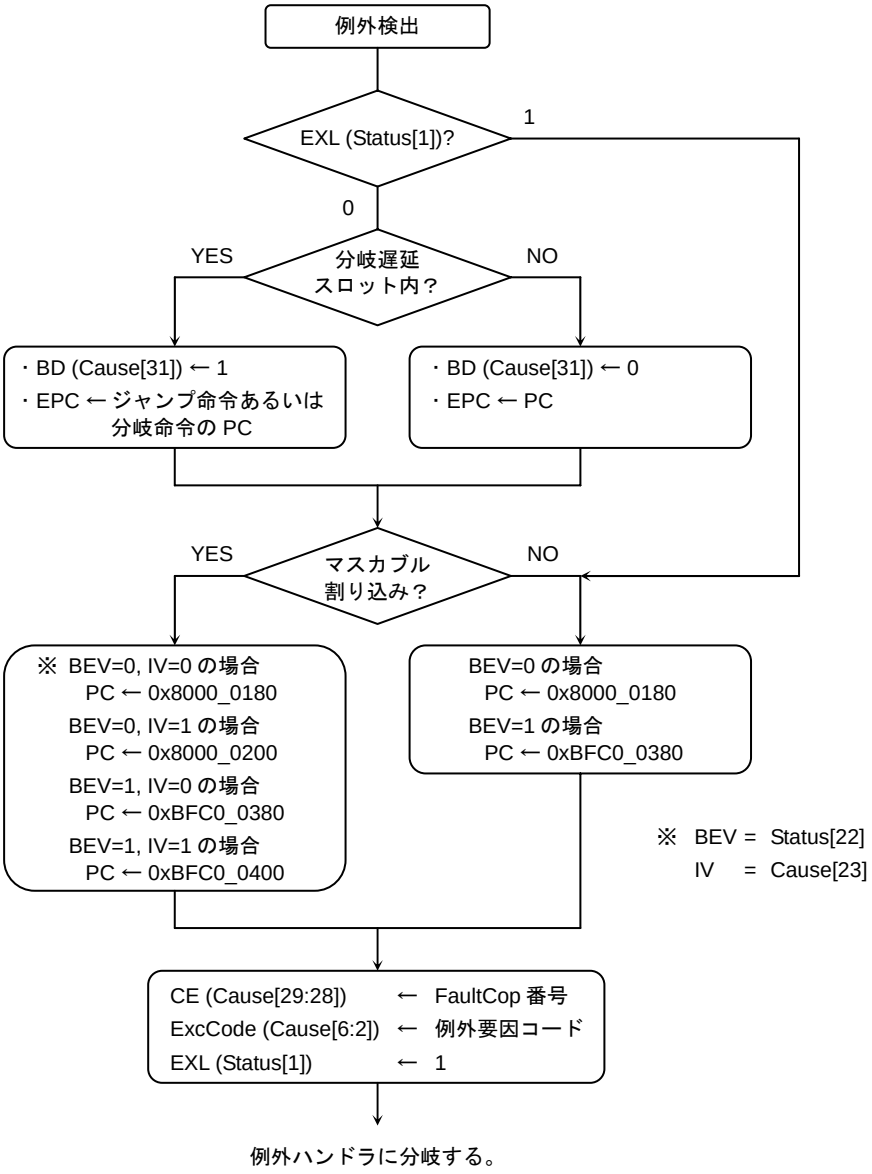


図 9-1　一般例外処理プロセス

CE フィールド(Cause レジスタのビット 28、ビット 29)は、コプロセッサ使用不可例外が発生したときのみ有効であり、それ以外の例外が発生したときのこのビットの状態は無効です。

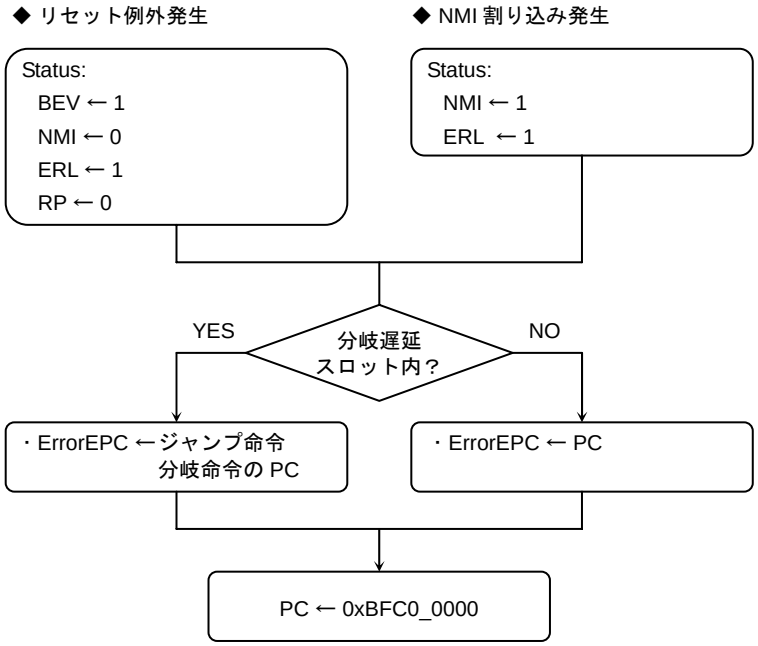


図 9-2　リセット例外、NMI 例外処理プロセス

9.1.2　例外の優先順位

1 つの命令に対して、複数の例外が同時に発生する可能性があります。この場合、表 9-1に示す優先順位に基づき、優先度の最も高い例外が 1 つだけ受け付けられます。

表 9-1　例外の優先順位

優先度	例外	ニモニック	タイプ
高い	リセット例外	Reset	Non_debug
	シングルステップ例外	DSS	Debug
	ノンマスカブル割り込み	Nmi	Non_debug
	マスカブル割り込み	Int	
	アドレスエラー例外 (命令フェッチ)	AdEL	Non_debug
	バスエラー (命令フェッチ)	IBE	
	デバッグブレークポイント例外 (SDBBP)	DBp	Debug
	コプロセッサ使用不可例外 (注 1)	CpU	Non_debug
	予約命令例外、整数オーバフロー、 トラップ、システムコール、 ブレークポイント	RI、Ov、 Tr、Sys、 Bp	
	アドレスエラー (ロード/ストア)	AdEL/AdES	Non_debug
低い	バスエラー (データアクセス)	DBE	

(注 1)　CU1=0 の状態で COP1(FPU 専用)命令を実行した場合、CpU 例外と RI 例外の両方を検出しますが、優先順位に従って CpU 例外として受け付けられます。

9.1.3　例外ベクタアドレス (Exception Vectors)

例外ベクタアドレスは、例外ハンドラの開始アドレスです。
リセット例外、ノンマスカブル割り込み例外の例外ベクタアドレスは 0xBFC0_0000 です。また、デバッグ例外の例外ベクタアドレスは 0xBFC0_0480 です。
その他の例外はStatusレジスタ[23]のBEVビット、Causeレジスタ[23]のIVビットの状態により異なります。
例外ベクタアドレスの一覧を 表 9-2に示します。

表 9-2　例外ベクタアドレス

例外	BEV=0		BEV=1	
	仮想アドレス	物理アドレス	仮想アドレス	物理アドレス
Reset、NMI	0xBFC0_0000	0x1FC0_0000	0xBFC0_0000	0x1FC0_0000
デバッグ例外	0xBFC0_0480	0x1FC0_0480	0xBFC0_0480	0x1FC0_0480
Interrupt (IV=0)	0x8000_0180	0x0000_0180	0xBFC0_0380	0x1FC0_0380
Interrupt (IV=1)	0x8000_0200	0x0000_0200	0xBFC0_0400	0x1FC0_0400
All others	0x8000_0180	0x0000_0180	0xBFC0_0380	0x1FC0_0380

9.1.4　リセット例外

■　原因

プロセッサのリセット信号がアサートされ、デアサートされると、リセット例外が発生します。

■　処理

1. すべての CP0 レジスタが初期化されます。
2. Status レジスタの ERL ビットがセットされます。
3. ErrorEPC に再開する PC が格納されます。
4. 例外ベクタアドレス 0xBFC0_0000 にジャンプして、例外ハンドラへ制御を移します。

※ バスサイクル中にリセット例外が発生すると、プロセッサはバスサイクルをただちに終了して、リセット例外を発生します。

9.1.5　ノンマスカブル割り込み (NMI)

■　原因

プロセッサのノンマスカブル割り込み信号 (GNMI) がアサートされたときに発生します。この割り込みはマスクできず、Status レジスタの EXL、ERL、および IE ビットとは無関係に発生します。

■　処理

1. Cause レジスタの ExcCode フィールド、CE ビットは不定になります。
2. Status レジスタの ERL、NMI ビットがセットされます。
3. ErrorEPC レジスタには、割り込みを発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、ErrorEPC レジスタには直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、ErrorEPC レジスタの最下位ビットには、例外が発生したときの ISA モードが保存されます。
4. プロセッサが低消費電力モード (HALT あるいは DOZE) のときは、解除して NMI の例外処理に移ります。
5. 16 ビット ISA モードで例外が発生した場合は、32 ビット ISA モードに切り換えられます。
6. 例外ベクタアドレス 0xBFC0_0000 にジャンプして、例外ハンドラへ制御を移します。

※ バスサイクル中にノンマスカブル割り込みが発生した場合は、リセット例外を除く例外と同様に、現在のバスサイクルの終端で割り込みを認識します。

9.1.6　アドレスエラー例外

■　原因

アドレスエラー例外は、以下の場合に発生します。

- ワード境界に位置合わせされていない 32 ビット ISA 命令をフェッチしようとしたとき (AdEL)。
- ハーフワード境界に位置合わせされていない 16 ビット ISA 命令をフェッチしようとしたとき (AdEL)。
- ワード境界に位置合わせされていないワードをロード、またはストアしようとしたとき (AdEL、AdES)。
- ハーフワード境界に位置合わせされていないハーフワードをロード、またはストアしようとしたとき (AdEL、AdES)。
- ユーザーモードにおいてカーネルセグメント (kseg0、kseg1、kseg2) を参照しようとしたとき (AdEL、AdES)。

■　処理

1. 例外が命令フェッチまたはロード (AdEL) により発生したのか、ストア (AdES) により発生したのかにより、Cause レジスタの ExcCode フィールドに AdDL (4)、AdES (5) が設定されます。
2. BadVAddr レジスタには、例外の要因となった仮想アドレス、または不正アクセスしたカーネルセグメントの仮想アドレスが保存されます。
3. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは “1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。
4. 16 ビット ISA モードで例外が発生した場合、32 ビット ISA モードに切り換えられます。
5. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

9.1.7　バスエラー例外

■　原因

バスエラー例外は、メモリリードバスサイクル時に GBUSERR 信号がアサートされると発生します。命令フェッチでは、命令の種類に関わらずバスエラー例外が発生する可能性があります。また、ロード命令、ビット操作命令はメモリリードバスサイクル中にバスエラー例外が発生する可能性があります。

TX19 コアでは、ストア命令によるバスサイクル中に GBUSERR 信号がアサートされてもバスエラー例外が発生していましたが、TX19A ではライトバッファを内蔵しているのでメモリライトバスサイクル中の GBUSERR 信号は無視されます。

この場合は、NMI を使ってプロセッサの動作を抑制する必要があります。

■　処理

1. 例外が命令フェッチ (IBE) により発生したのか、データのロード (DBE)により発生したのかにより、Cause レジスタの ExcCode フィールドに IBE (6)、または DBE (7) が設定されます。

2. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは“1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。

3. 以下の場合、EPC レジスタに例外が発生した時点のプログラムカウンタ (PC) の内容が保存されます。

   - ロード命令の直後に SYNC 命令を置いた場合
   - ロード直後の命令がロードされたデータに依存する場合

   これらの場合、ロード命令が終了するまで直後の命令はストールします。
   EPC レジスタにはロード命令直後の命令のアドレスが格納されます。

4. 16 ビット ISA モードで例外が発生した場合、32 ビット ISA モードに切り換えられます。

5. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

9.1.8　整数オーバフロー例外

■　原因

整数オーバフロー例外は、ADD、ADDI、SUB 命令 (32 ビット ISA)、DIVE 命令 (16 ビット ISA) の結果が 2 の補数のオーバフローになると発生します。その他に DIVE、DIVEU 命令 (16 ビット ISA) は、除数が “0” でもオーバフロー例外が発生します。

■　処理

1. Cause レジスタの ExcCode フィールドに Ov コード (12) が設定されます。
2. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは “1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。
3. 16 ビット ISA モードで例外が発生した場合、32 ビット ISA モードに切り換えられます。
4. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

9.1.9　トラップ例外

■　原因

トラップ例外は、TGE、TGEU、TLT、TLTU、TEQ、TNE、TGEI、TGEIU、TLTI、TLTIU、TEQI、または TNEI 命令によって条件が成立した場合に発生します。

■　処理

1. Cause レジスタの ExcCode フィールドに Tr コード (13) が設定されます。
2. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは “1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。
3. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

※ この例外は、32 ビット ISA モードでしか発生しません。

9.1.10 システムコール例外

■ 原因

システムコール例外は、SYSCALL 命令を実行すると発生します。

■ 処理

1. Cause レジスタの ExcCode フィールドに Sys コード (8) が設定されます。
2. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは “1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。
3. 16 ビット ISA モードで例外が発生した場合、32 ビット ISA モードに切り換えられます。
4. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

システムコール例外が発生すると、例外ハンドラに制御が移ります。SYSCALL 命令の未使用ビット (32 ビット ISA の場合はビット 25 ~ 6、16 ビット ISA の場合は、ビット 25 ~ 16、10 ~ 5) を使って、例外ハンドラに情報を渡すことができます。これらのビットを調べるには、EPC レジスタが指している命令をデータとしてロードします。ただし、例外を発生した命令がジャンプまたは分岐遅延スロット内にある場合 (Cause レジスタの BD ビットが “1” にセット) は、EPC レジスタの値に 4 を加えなければなりません。

例外ハンドラから戻る際には、SYSCALL 命令が再び実行されないように、EPC レジスタ内のアドレスに 4 を加えなければなりません。SYSCALL 命令がジャンプまたは分岐遅延スロット内にある (Cause レジスタの BD ビットが 1 にセットされている) 場合は、戻りアドレスの命令は、直前のジャンプまたは分岐命令になります。この場合は、ジャンプまたは分岐命令を解釈し、EPC レジスタを設定し直さなければなりません。

9.1.11 ブレイクポイント例外

■ 原因

ブレイクポイント例外は、BREAK 命令を実行すると発生します。

■ 処理

1. Cause レジスタの ExcCode フィールドに Bp コード (9) が設定されます。
2. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは “1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。
3. 16 ビット ISA モードで例外が発生した場合、32 ビット ISA モードに切り換えられます。
4. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

ブレイクポイント例外が発生すると、例外ハンドラに制御が移ります。BREAK 命令の未使用ビット (32 ビット ISA の場合はビット 25 ~ 16、16 ビット ISA の場合はビット 10 ~ 5) を使って、例外ハンドラに情報を渡すことができます。これらのビットを調べるには、EPC レジスタが指している命令をデータとしてロードします。ただし、例外を発生した命令がジャンプまたは分岐遅延スロット内にある (Cause レジスタの BD ビットが “1” にセット) 場合は、EPC レジスタの値に 4 (32 ビット ISA モード) あるいは 2 (16 ビット ISA モード) を加えなければなりません。

例外ハンドラから戻る際に、BREAK 命令が再び実行されないように、EPC レジスタ内のアドレスに 4 (32 ビット ISA モード) または 2 (16 ビット ISA モード) を加えなければなりません。BREAK 命令がジャンプまたは分岐遅延スロット内にある (Cause レジスタの BD ビットが 1 にセットされている) 場合は、戻りアドレスの命令は、直前のジャンプまたは分岐命令になります。この場合は、ジャンプまたは分岐命令を解釈し、EPC レジスタを設定し直さなければなりません。

9.1.12　予約命令例外

■　原因

予約命令例外は、以下の条件のいずれかにより発生します。

◆　32 ビット ISA モードの場合

- 未定義のメジャーオペコード (ビット 31～26) を持つ命令を実行しようとした。
- 未定義のマイナーオペコード (ビット 5～0) を持つ SPECIAL 命令を実行しようとした。
- 未定義のマイナーオペコード (ビット 5～0) を持つ SPECIAL2 命令を実行しようとした。
- 未定義のマイナーオペコード (ビット 20～16) を持つ REGIMM 命令を実行しようとした。
- 未定義のマイナーオペコード (ビット 25～21) を持つ COPz rs 命令 (z=1、2) を実行しようとした。
- LWCz、SWCz、LDCz、SDCz (z=1、2) 命令、MOVCI 命令を実行しようとした。

◆　16 ビット ISA モードの場合

- コードが 11101xxxxxx01001、11101xxxxxx10011、11101xxxx1100000、11101xxx01010001、11101xxx01110001、11101xxx11010001、11101xxx11110001 である未定義命令を実行しようとした。
- 予約命令 (LWU、LD、SD、DADDU、DSUBU、DADDIU、DMULT、DMULTU、DDIV、DDIVU、DSLL、DSRL、DSRA、DSLLV、DSRLV、DSRAV) を実行しようとした。
- 拡張できない命令に対して、EXTEND 命令を実行しようとした。
- 未定義の EXTEND+RR マイナーオペコード (ビット 4～0) を持つ命令を実行しようとした。
- 未定義の EXTEND+ADDIU8 マイナーオペコード (ビット 7～5:001,011) を持つ命令を実行しようとした。
- 未定義の EXTEND+INT マイナーオペコード ([7][1:0]=100 & [10:8]≠00x) を持つ命令を実行しようとした。

■　処理

1. Cause レジスタの ExcCode フィールドに RI コード (10) が設定されます。

2. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは “1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。

3. 16 ビット ISA モードで例外が発生した場合、32 ビット ISA モードに切り換えられます。

4. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

9.1.13 コプロセッサ使用不可例外

■ 原因

コプロセッサ使用不可例外は、以下のいずれかの場合にコプロセッサ命令を実行すると発生します。

- ユーザーモードで Status レジスタの CU0 ビットが “0” にクリアされているときに、CP0 命令を実行しようとした (カーネルモード、デバッグモードでは、CU0 ビットの設定に関わらず、CP0 命令が発行されても、例外は発生しません)。
- CU1 ビットを “0” にクリアした状態で COP1、LWC1、SWC1、LDC1、SDC1、MOVCI 命令を実行しようとした。
- CU2 ビットを “0” にクリアした状態で COP2、LWC2、SWC2、LDC2、SDC2 命令を実行しようとした。
- CU3 ビットを “0” にクリアした状態で COP3 命令を実行しようとした。

■ 処理

1. Cause レジスタの ExcCode フィールドに CpU コード (11) が設定されます。
2. Cause レジスタの CE フィールドは、該当するコプロセッサの番号が設定されます。
3. EXL ビットが 0 に設定されているときのみ、以下の操作が発生します。
   EXL ビットは “1” に設定され、EPC レジスタには、例外が発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、EPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Cause レジスタの BD ビットが 1 にセットされます。また、EPC レジスタの最下位ビットには例外が発生したときの ISA モードが保存されます。
4. 16 ビット ISA モードで例外が発生した場合、32 ビット ISA モードに切り換えられます。
5. 例外ベクタアドレスにジャンプし、例外ハンドラに制御が移ります。
   (表 9-2 例外ベクタアドレスを参照)

9.1.14 マスカブル割り込み (Interrupts)

■ 原因

マスカブル割り込み要求には、次の 3 つがあります。

- ソフトウエア割り込み（IP0 と IP1）　・・・　2 本
- ハードウエア割り込み（IP2, 3, 4, 5, 6, 7）・・・　6 本
- タイマ割り込み　（IP7）　・・・　1 本

上記マスカブル割り込みは、次の条件が成立したときに検出されます。

1. 割り込み要求 (IP[7:0]) がセットされている (Cause)
2. 割り込みマスクビット (IM[7:0]) がセットされている状態 (Status)
3. 割り込み許可フラグ (IE=1) がセットされている状態 (Status)
4. デバッグモード (DM=0) でない状態 (Debug)
5. エラーレベル（ERL=0)、例外レベル（EXL=0）がセットされていない状態 (Status)

これらの条件が全て揃ってなおかつ、マスカブル割り込みより優先度の高い例外要求がなければ、マスカブル割り込みに対して割り込み応答します。

IP7 はハードウエア割り込み（入力端子 GINT[5]）とタイマ割り込みのどちらかが選択されてアサートされます。選択の方法は、プロセッサの入力端子 GTINTDIS 信号が “0” に設定されている場合はタイマ割り込みが有効となり、“1” に設定されている場合は GINT[5]が有効となります。

■ 処理

ベクタアドレスはBEV (Status レジスタ)、IV (Causeレジスタ) の状態によって異なります。マスカブル割り込みベクタの一覧を 表 9-3に示します。

表 9-3　マスカブル割り込みベクタ

IV (Cause[23])	BEV (Status[22])	
	BEV=0	BEV=1
IV=0	0x8000_0180	0xBFC0_0380
IV=1	0x8000_0200	0xBFC0_0400

■　サービス

その割り込みが 2 つのソフトウエアによる例外 (IP1、IP0) の 1 つによって発生した場合には、対応する Cause レジスタのビットを 0 に設定することで解除できます。

その割り込みがハードウエアから発生した場合には、その割り込み端子がアサートされる原因となった条件を補正することによって解除できます。

タイマ割り込みが発生した場合は、その割り込み条件は Compare レジスタの値を書き替えることで解除できます。

9.2　割り込み

TX19A には、ノンマスカブル割り込み、マスカブルハードウエア割り込み、マスカブルソフトウエア割り込みがあります。この項では、割り込みの種類、優先度、割り込みの受け付けについて説明します。

9.2.1　割り込みの種類

TX19Aは、ノンマスカブル割り込み、6 本のマスカブルハードウエア割り込み、2 本のマスカブルソフトウエア割り込みを認識します。割り込み例外の処理は、ハードウエアで行われ、その後、割り込みハンドラに制御が移されます。割り込み例外の処理については、「9.1.14 マスカブル割り込み (Interrupts)」と「9.1.5 ノンマスカブル割り込み (NMI)」を参照してください。

ノンマスカブル割り込みは、外部端子からの入力またはウォッチドッグタイマなどの周辺回路からの要求により発生します。ノンマスカブル割り込みの発生原因については、お使いの製品のマニュアルを参照してください。ノンマスカブル割り込みは、緊急性の高い処理を行うための割り込みであり、ソフトウエアでマスクできません。ノンマスカブル割り込みは CPU の動作モードによらずかならず受け付けられ、強制的にノンマスカブル割り込み例外の例外ベクタアドレス 0xBFC0_0000 に移ります。

マスカブルハードウエア割り込みは、プロセッサの割り込み要求端子からの要求により発生します。割り込み要求は外部またはチップ上の周辺回路から出され、割り込みコントローラに入力されます。割り込みコントローラは割り込み要求を割り込みレベルを表す 3 ビットの値に変換し、TX19A プロセッサコアに入力します。割り込みレベルの 3 ビットの信号線は、プロセッサの IP4、IP3、IP2 に接続されています。したがって、マスカブルハードウエア割り込みを受け付け許可にするためには、IM[4:2]ビットをかならず 111 の状態に設定してください。TX19A は、割り込みに応答した直後の IP4、IP3、IP2 の状態に応じて Shadow Register Set を切り替えます。

ソフトウエア割り込みには、IP1、IP0 の 2 本があります。ソフトウエア割り込みは、Cause レジスタの対応するビットを 1 にセットすることにより発生します。アプリケーションプログラムは、これらのビットを使って割り込みを要求することができます。また、Status レジスタには、各ソフトウエア割り込みを個別にマスクするビットが用意されています。

9.2.2　マスカブル割り込みベクタ

マスカブル割り込みベクタは、各レジスタの設定により異なります。マスカブル割り込みベクタを 表 9-3に示します。TX19Aはハードウエア割り込みとソフトウエア割り込みの例外ベクタアドレスの区別はありません。割り込み例外ハンドラは割り込みコントローラを調べ、割り込み要因に対応した割り込みベクタアドレスを読み出し、制御をそのアドレスに移します。

9.2.3　マスカブル割り込みの受け付け

マスカブル割り込みは、以下の場合に発生します。

- 割り込み要求 (IP[7:0]) がセットされている (Cause)
- 割り込みマスクビット (IM[7:0]) がセットされている状態 (Status)
- 割り込み許可フラグ (IE=1) がセットされている状態 (Status)
- デバッグモード (DM=0) でない状態 (Debug)
- エラーレベル (ERL=0)、例外レベル (EXL=0) がセットされていない状態 (Status)

ハードウエア割り込みを許可にするためには、IM[4:2]=111 となるようにかならず設定してください。

ハードウエア割り込みとソフトウエア割り込みが同時に検出された場合は、ハードウエア割り込みが優先されます。

表 9-4　割り込みタイプと各レジスタ

割り込みタイプ	割り込み番号	Cause レジスタ		Status レジスタ	
		ビット番号	名前	ビット番号	名前
ソフトウエア割り込み	0	[8]	IP0	[8]	IM0
	1	[9]	IP1	[9]	IM1
ハードウエア割り込み	0	[10]	IP2	[10]	IM2
	1	[11]	IP3	[11]	IM3
	2	[12]	IP4	[12]	IM4
	3	[13]	IP5	[13]	IM5
	4	[14]	IP6	[14]	IM6
ハードウエア割り込みまたはタイマ割り込み	5	[15]	IP7	[15]	IM7

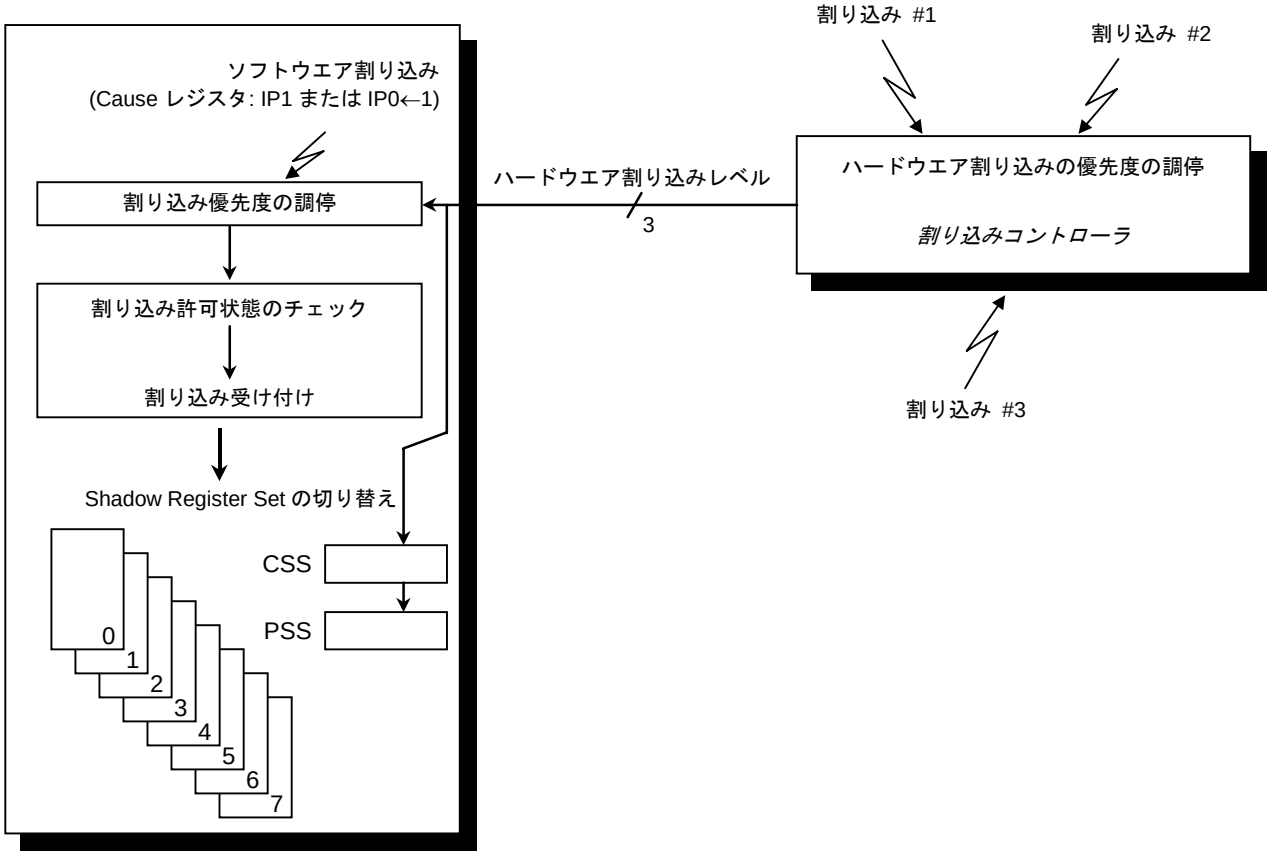


図 9-3　マスカブル割り込みの受け付け

9.2.4　Shadow Register Set の動作

ハードウエア割り込みが発生すると割り込みコントローラからの割り込みレベルに応じたShadow Register Setに切り替わり、そのときの割り込みレベルがSSCRレジスタのCSSビット(ビット 3～0)にセットされ、同時にPSSビット(ビット 11～8) には更新前のCSSビットがセットされます。割り込みコントローラからの割り込みレベルは、プロセッサのIP4、IP3、IP2 に接続されており、プロセッサが割り込みを認識したときにこれらの信号の状態に応じたShadow Register Setに切り替わります(表 9-5参照)。ソフトウエア割り込み、内部タイマ割り込み、その他の例外が発生してもShadow Register Setは切り替わりません。ただし、PSSビットの値は更新されます。

ERET 命令によって、割り込みハンドラから復帰する場合は、PSS ビットの内容が CSS ビットに上書きされます (図 8-1 参照)。

デバッグ例外発生時、あるいは DERET 命令によってデバッグ例外のハンドラから復帰する場合、CSS ビット、PSS ビットの内容は更新されません。

表 9-5　割り込み要求と Shadow Register Set

IP4	IP3	IP2	Shadow Register Set
0	0	0	–
0	0	1	1
0	1	0	2
0	1	1	3
1	0	0	4
1	0	1	5
1	1	0	6
1	1	1	7

9.3　デバッグ例外

TX19A のデバッグ例外には、シングルステップ例外とデバッグブレークポイント例外があります。この項では、デバッグ例外の要因と処理について説明します。

9.3.1　デバッグ例外処理プロセス

TX19A はデバッグのためプログラムの実行を任意に停止させることができます。ブレークポイント例外は、SDBBP (Software Debug Breakpoint) 命令を実行するときに発生します。シングルステップ例外は Debug レジスタの SSt ビットをセットしたときに発生します。

デバッグ例外処理は、図 9-4に示す順序で実行されます。

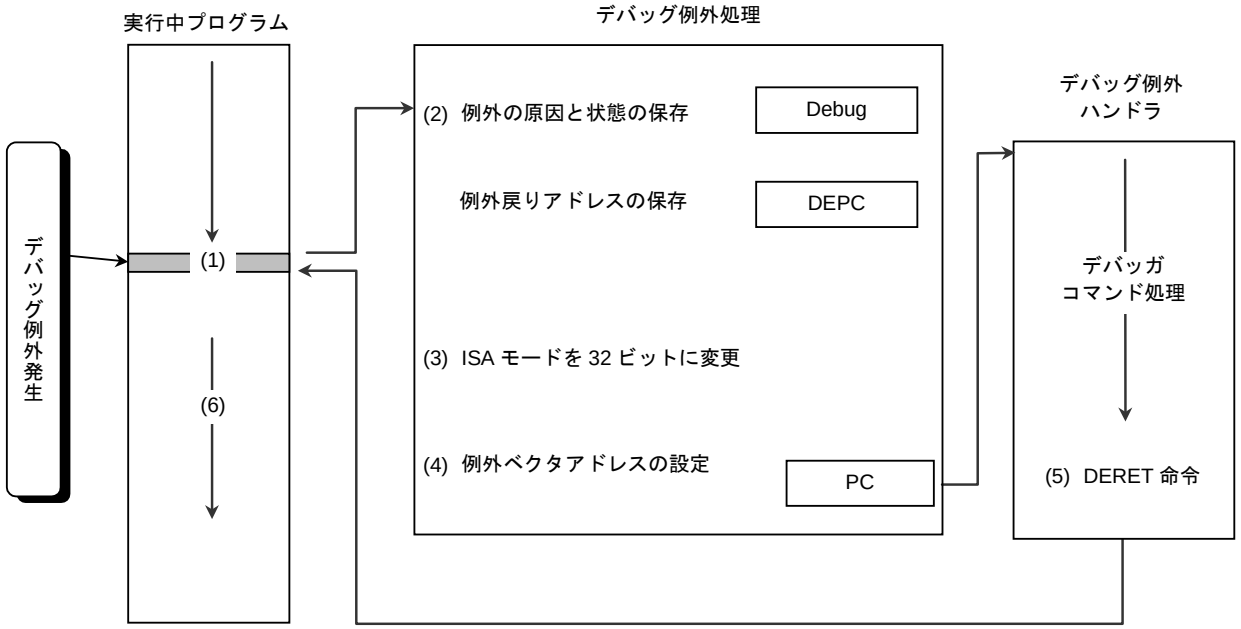


図 9-4　例外処理

1. 現在実行中の命令、およびすでに実行が開始されているパイプライン内の命令を中断します。

2. デバッグ例外用のレジスタは、デバッグに関する情報を保存します。

   - Debug 例外レジスタは、デバッグ例外の原因と、デバッグ例外が現在処理中であるかどうかを示します。

   - DEPC レジスタには、デバッグ例外の原因となった命令の仮想アドレスが格納されます。ただし、例外を発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、DEPC レジスタには直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Debug レジスタの DBD ビットが 1 にセットされます。また、DEPC レジスタの最下位ビットは、

例外が発生したときの ISA モードが保存されます。

3. プロセッサは例外処理用のカーネルモードに切り換えられ、Status レジスタの設定とは無関係に、割り込みが禁止されます。16 ビット ISA モードで例外が発生した場合は、PC の最下位ビット (ISA モード) は 0 にクリアされ、32 ビット ISA モードに切り換えられます。

4. デバッグ例外ベクタアドレスを PC に設定して、デバッグ例外ハンドラの開始アドレスにジャンプします。

5. デバッグ例外ハンドラの処理を終了したら、DERET 命令を実行して、DEPC レジスタに格納されている戻りアドレスへジャンプします。

6. 例外が発生したときプロセッサが実行を中断したアドレスから処理が再開されます。

9.3.2　デバッグ例外の種類

表 9-6に、TX19Aで発生するデバッグ例外の種類を示します。

表 9-6　デバッグ例外の種類

例外	説明
シングルステップ例外	Debug レジスタの SSt ビットが 1 にセットされているとき、次の命令の開始前に発生します。
デバッグブレークポイント例外	プログラムに埋め込まれた SDBBP 命令を実行すると発生します。ただし、Debug レジスタの SSt ビットが 1 にセットされている場合は、シングルステップ例外が優先されます。デバッグ例外の処理中 (Debug レジスタの DM ビットが 1 にセットされている) に、SDBBP 命令が実行されると再び Debug 例外が発生します。そのときは、Debug レジスタの DExcCode に Break 例外のコードがセットされます。

9.3.3　デバッグ例外の優先順位

デバッグ例外と一般例外は、同時に発生することがあります。この場合、プロセッサは「表 9-1 例外の優先順位」に従った応答をします。

9.3.4　例外マスク

デバッグ例外処理中（DM=1、IEXI=1)、プロセッサは他の例外をすべてマスクします。

- 命令バスエラーが発生すると、Debug レジスタの IBusEP ビットが 1 にセットされ、データバスエラーが発生すると Debug レジスタの DBusEP ビットが 1 にセットされます。
- デバッグ例外が処理されているあいだ、マスカブル割り込みは禁止状態になります(マスカブル割り込みは、DERET 命令の実行でマスクが解除されます)。
- ノンマスカブル割り込みは保留にされ、DERET 命令によるデバッグ例外からの復帰後に発生します。
- Debug レジスタの IEXI ビットを 0 にクリアすると､マスカブル割り込み、NMI 例外を除く一般例外の要求に応答します。この際に、一般例外を検出した場合でも、デバッグ例外処理を開始し、デバッグ例外ハンドラに分岐します。この場合には、Debug レジスタのデバッグ例外の要因を示すビット（DINT ビット、DIB ビット、DBp ビット、DSS ビット、DDBSImpr ビット、DDBLImpr ビット）はセットされないで、一般例外の要因を示す DExcCode フィールドがセットされます。このフィールドを見ることで、デバッグモード中にどの一般例外が発生したか確認することができます。

9.3.5　デバッグ例外ハンドラの実行

デバッグ例外ハンドラは、プログラムデバッグ用に制御された条件の下で、プロセッサを操作します。Debug レジスタの DSS ビット、DBp ビットにより、シングルステップ例外かデバッグブレークポイント例外かを判断して､対応する処理を行います。

9.3.6　デバッグ例外からの復帰

デバッグ例外ハンドラの処理が終わってから、元のプログラムに復帰するには、DERET 命令を実行します。DERET 命令は、次の処理を実行します。

1. DEPC レジスタの戻りアドレスをプログラムカウンタ（PC）に復元します。これにより、プロセッサは、デバッグ例外が発生したアドレスから処理を再開します。ただし、例外を発生した命令が、ジャンプまたは分岐遅延スロット内の命令であった場合、PC には直前のジャンプまたは分岐命令のアドレスが格納されます。また、PC の ISA モードビットは、DEPC レジスタのビット 0 から復元され、例外が発生したときの ISA モードに戻ります。
2. Debug レジスタの DM (Debug Mode) ビット､IEXI ビットをクリアします。
3. 強制的な動作モード「カーネルモード」が解除されます。

9.3.7　シングルステップ例外

■　原因

シングルステップ例外は、Debug レジスタの SSt ビットを 1 にセットすると発生します。

■　処理

シングルステップ例外は、次の命令を実行する前に実行します。図 9-5に、この例外の処理で使われるCP0 レジスタのフィールドを示します。

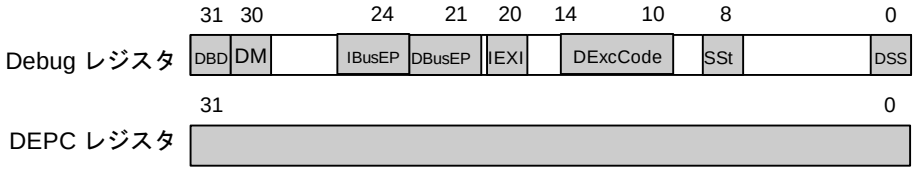


図 9-5　シングルステップ例外

1. Debug レジスタの DM ビットと DSS ビットが 1 にセットされます。シングルステップ例外が発生したということは、SSt ビットが 1 にセットされています。

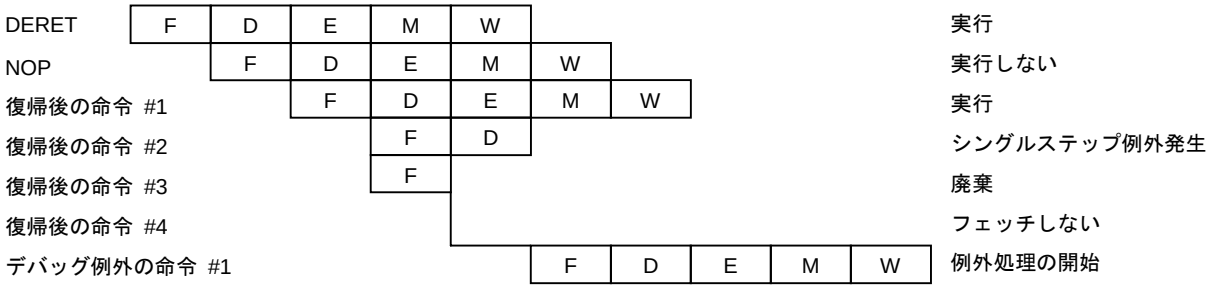
2. DEPC レジスタに、例外発生時のプログラムカウンタ (PC) の内容が保存されます。また、DEPC レジスタの最下位ビットは、例外発生時の ISA モードが保存されます。

3. Status レジスタの設定に関係なく、プロセッサはカーネルモードに切り換えられ、割り込みがすべて禁止されます。

4. 例外ベクタアドレス 0xBFC0_0480 にジャンプし、例外ハンドラに制御が移ります。

シングルステップ例外は以下の場合、発生しません。

- ジャンプまたは分岐遅延スロット内の命令
- デバッグ命令から、DERET命令で復帰した最初の命令 (図 9-6参照)
- デバッグ例外処理中 (Debug レジスタの DM ビットが 1 にセットされているとき)



DEPC レジスタは、復帰後の命令#2 のアドレスになります。

図 9-6　DERET 命令後の CPU パイプライン動作

9.3.8　デバッグブレークポイント例外

■　原因

デバッグポイント例外は、SDBBP 命令を実行すると発生します。

■　処理

図 9-7に、この例外の処理で使われるCP0 レジスタのフィールドを示します。

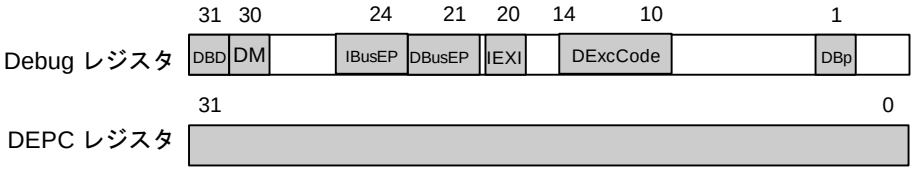


図 9-7　デバッグブレークポイント例外

1. Debug レジスタの DM ビットと DBp ビットが 1 にセットされます。デバッグブレークポイント例外が発生したということは、SSt ビットは 0 にクリアされています。

2. DEPC レジスタに、割り込みを発生した時点のプログラムカウンタ (PC) の内容を保存します。ただし、割り込みを発生した命令が、ジャンプまたは分岐遅延スロット内の命令である場合は、DEPC レジスタには、直前のジャンプまたは分岐命令のアドレスが格納されます。この場合、Debug レジスタの DBD ビットが 1 にセットされます。また、DEPC レジスタの最下位ビットには、例外が発生したときの ISA モードが保存されます。

3. Status レジスタの設定に関係なく、プロセッサはカーネルモードに切り換えられ、割り込みがすべて禁止されます。

4. 16 ビット ISA モードで例外が発生すると、32 ビット ISA モードに切り換えられます。

5. 例外ベクタアドレス 0xBFC0_0480 にジャンプし、例外ハンドラに制御が移ります。

SDBBP 命令の未使用ビット (32 ビット ISA の場合はビット 25~6、16 ビット ISA の場合はビット 10~5) を使って、例外ハンドラに情報を渡すことができます。これらのビットを調べるには、DEPC レジスタが指している命令をデータとしてロードします。ただし、例外を発生した命令が、ジャンプまたは分岐遅延スロット内にある (Debug レジスタの DBD ビットが 1 にセットされている) 場合は、DEPC レジスタの値に 4 を加えなければなりません。

例外ハンドラから戻る際に、SDBBP 命令が再び実行されないように、DEPC レジスタ内のアドレスに 4 (32 ビット ISA モード) または 2 (16 ビット ISA モード) を加えなければなりません。SDBBP 命令が、ジャンプまたは分岐遅延スロット内にある (Debug レジスタの DBD ビットが 1 にセットされている) 場合は、戻りアドレスの命令は、ジャンプまたは分岐命令になります。この場合、ジャンプまたは分岐命令を解釈し、DEPC レジスタに設定し直さなければなりません。

第10章 低消費電力モード

TX19A には、いくつかの低消費電力モードがあります。Halt モード、Doze モードへは、CP0 の Status レジスタの RP ビットを設定し、その後で WAIT 命令を実行することで移行できます。この章では、TX19A の低消費電力モードについて説明します。

10.1 各低消費電力モードの特徴

図 10-1にTX19Aの低消費電力モードを示します。

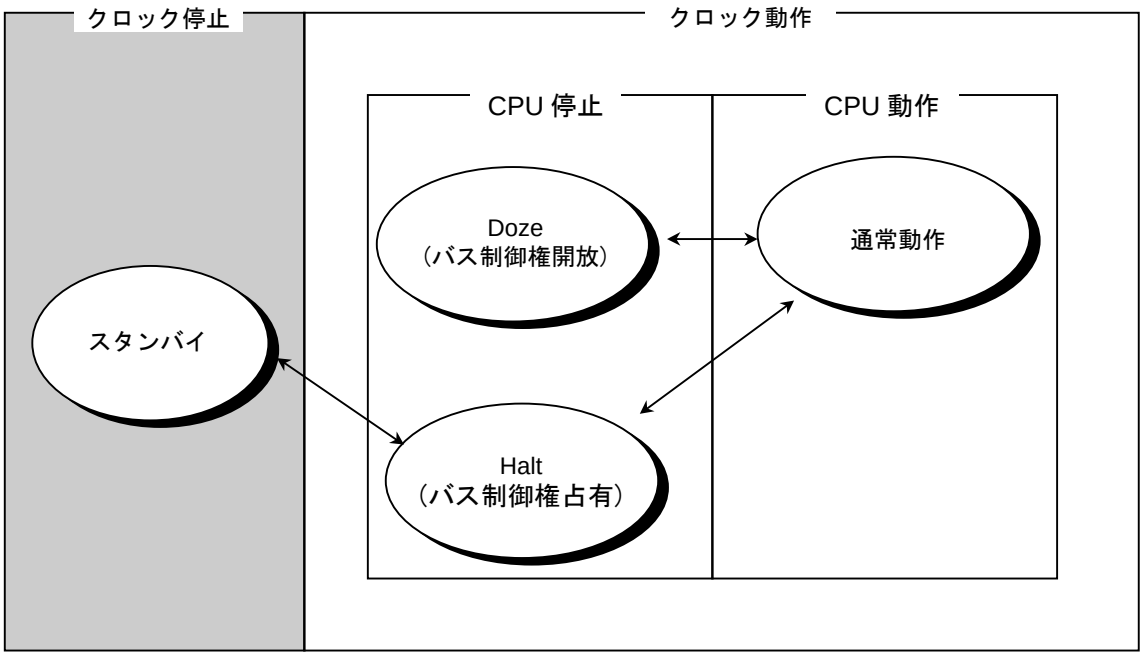


図 10-1　低消費電力モード

TX19Aには、動作中消費電力をダイナミックに管理するモードとして、表 10-1に示すモードがあります。

表 10-1　低消費電力モード

モード	説　明
スタンバイモード	消費電力を最小限にするために、プロセッサへのクロック供給を停止できます。スタンバイモードには、2 つのレベルのモードがあります。 1.　プロセッサと発振回路の両方を停止します。 2.　発振回路は継続して動作しますが、プロセッサへのクロック供給を停止します。 スタンバイモードの詳細については、各製品のマニュアルを参照してください。
Halt モード	Halt モードでは、プロセッサの動作はすべて停止し、外部からのバス制御権要求は受け付けません。TX19A は、バス制御権を占有した状態となります。Status レジスタの RP ビットを 0 に設定した状態で WAIT 命令を実行した場合、Halt モードに移行できます。
Doze モード	Doze モードでは、プロセッサの動作は停止しますが、外部からのバス制御権の要求は受け付け、バス制御権は開放した状態となります。Status レジスタの RP ビットを 1 にした状態で WAIT 命令を実行した場合、Doze モードに移行できます。
通常動作モード	TX19A のデフォルトの動作モードで、プロセッサが最大のクロック動作周波数で動作しているモードです。
その他のモード	例えば、時計用水晶 32.768 kHz で動作させる超低速モード、上記以外の低消費電力モードをもっている製品があります。各製品のマニュアルを参照してください。

10.2 Halt モード

図 10-2に、Haltモードに移行する経路を示します。

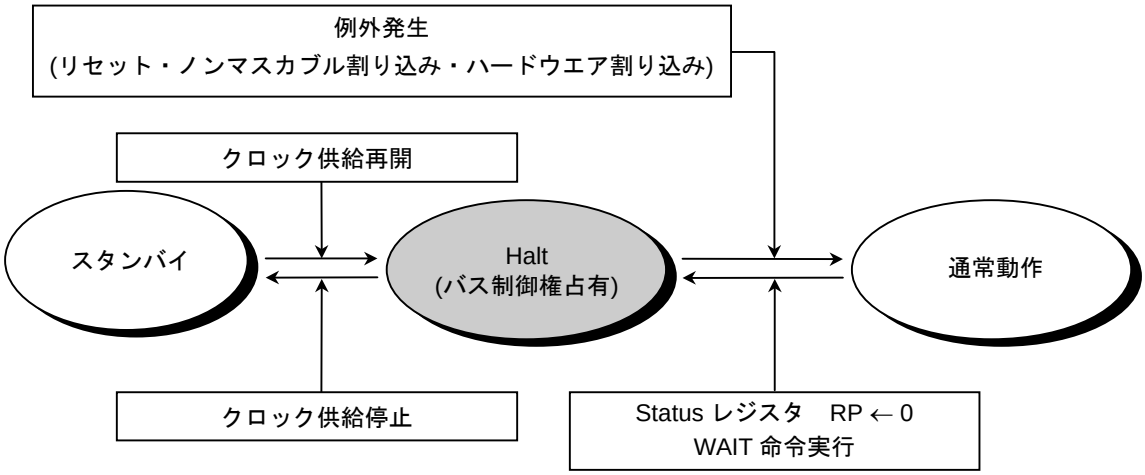


図 10-2　Halt モード

通常動作モード時に、Status レジスタの RP ビットを 0 に設定した状態で WAIT 命令を実行すると、Halt モードに移行します。Halt モード中、TX19A プロセッサコアはパイプラインの状態を保持したままプロセッサ動作を停止します。Halt モードではプロセッサは外部からのバス要求信号を無視し、バス制御権を占有したままの状態になります。

Halt モード中でもライトバッファユニットのデータを外部メモリに書き終わるまで、ライトバッファユニットは動作しつづけます。

Halt モード中にリセット例外、ノンマスカブル割り込み例外またはマスカブルハードウエア割り込み例外が発生した場合は Halt モードが解除され、例外処理を実行します。

Status レジスタでマスカブル割り込みがマスクされている状態でも、マスカブル割り込みを認識します。マスカブル割り込みを受け付けたとき、Halt モードに移行する前の状態から通常の処理が再開されます。

Halt モードの状態でスタンバイモードに移行することにより、クロックの供給を停止することができます。オシレータの発振停止、クロックの供給停止により、プロセッサはスタンバイモードに移行します。クロックが再開されると、スタンバイモードから Halt モードに復帰します。

10.3 Doze モード

図 10-3に、Dozeモードに移行する経路を示します。

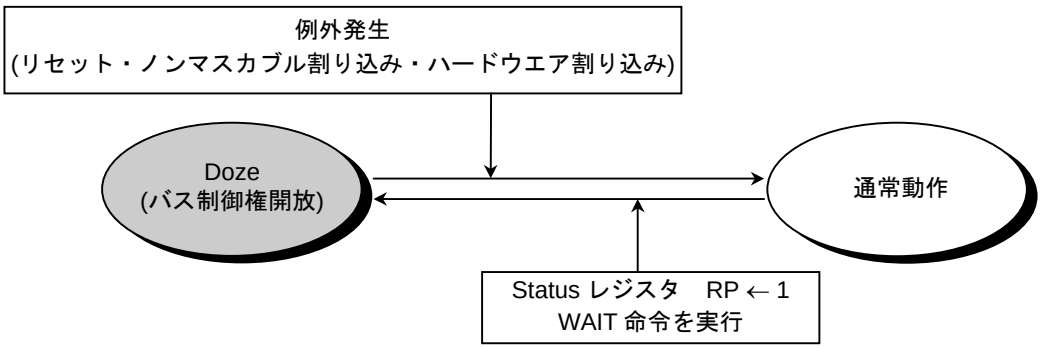


図 10-3　Doze モード

通常動作モード時に、Status レジスタの RP ビットを 1 に設定した状態で WAIT 命令を実行すると、Doze モードに移行します。Halt モードと同様に Doze モードでは、TX19A プロセッサコアはパイプラインの状態を保持したままプロセッサ動作を停止します。ただし、Doze モードではプロセッサは外部からのバス要求権を認識します。

Doze モード中でもライトバッファユニットのデータを外部メモリに書き終わるまで、ライトバッファユニットは動作しつづけます。

リセット例外、ノンマスカブル割り込み例外、またはハードウエアマスカブル割り込み例外が発生すると、Doze モードが解除され、例外処理を実行します。

Status レジスタでマスカブル割り込みがマスクされている状態でも、マスカブル割り込みは認識されます。この場合、Doze モードに移行する前の状態から通常の処理が再開されます。

付録A　32 ビット ISA の詳細

この章では 32 ビット ISA の命令について、シンタックス、命令形式、動作、命令の実行によって発生する可能性のある例外などを詳しく説明します。

命令はアルファベット順に記述されています。命令形式については「3.1　命令形式」を参照してください。

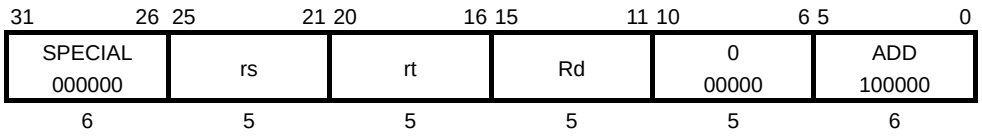
ADD *rd*, *rs*, *rt*

Add

動作

rd ⇐ *rs* + *rt*

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を加算し、その結果を汎用レジスタ *rd* に格納します。

c ⇐ a + b の計算において、a と b の符号が同一で、かつそれら符号と c の符号が異なる場合、計算結果がオーバフローしています。この場合、整数オーバフロー例外が発生します。整数オーバフロー例外が発生するとデスティネーションレジスタ (*rd*) の内容は変更されません。

例外

整数オーバフロー例外

使用例

1. レジスタ r2 の値が 0x0200_0000 で、レジスタ r3 の値が 0x0123_4567 の場合、以下の命令を実行すると、レジスタ r4 に和 (0x0323_4567) が格納されます。

```
ADD r4,r2,r3
```

2. レジスタ r2 の値が 0x7FFF_FFFF で、レジスタ r3 の値が 0x0000_0001 の場合、r2 と r3 を加算すると演算結果は、0x8000_0000 と負の値になり、2 の補数のオーバフローが発生します。この場合、以下の命令を実行すると、整数オーバフロー例外が発生し、レジスタ r4 の内容は変更されません。

```
ADD r4,r2,r3
```

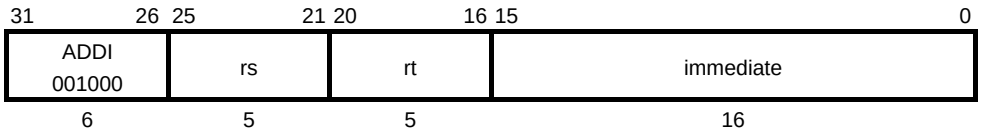
ADDI *rt, rs, immediate*

Add Immediate

動作

$rt \Leftarrow rs + ((immediate_{15})^{16} \parallel immediate_{15..0})$

コード



説明

16 ビット *immediate* を符号拡張して、汎用レジスタ *rs* の内容と加算し、その結果を汎用レジスタ *rt* に格納します。

2 の補数オーバフローで整数オーバフロー例外が発生します。整数オーバフロー例外が発生すると、デスティネーションレジスタ (*rt*) の内容は変更されません。

immediate は 16 ビットです。したがって、*immediate* で指定できる数値の範囲は、–32768 ~ +32767 です。この範囲外の値を扱いたい場合は、いったん汎用レジスタに格納してから、ADD または ADDU 命令を使います (「3.3.2　32 ビットの定数」を参照)。

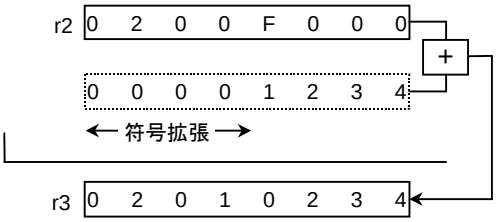
例外

整数オーバフロー例外

使用例

レジスタ r2 の値が 0x0200_F000 の場合、以下の命令を実行すると、和 (0x0201_0234) がレジスタ r3 に格納されます。

```
ADDI r3,r2,0x1234
```



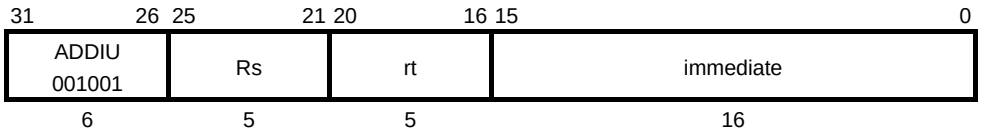
ADDIU *rt*, *rs*, *immediate*

Add Immediate Unsigned

動作

$rt \Leftarrow rs + ((immediate_{15})^{16} \parallel immediate_{15..0})$

コード



説明

ADDIU は Add Immediate Unsigned を表しますが、16 ビット *immediate* を「符号拡張」して、汎用レジスタ *rs* の内容に加算し、その結果を汎用レジスタ *rt* に格納します。

ADDI 命令と ADDIU 命令の違いは、ADDUI 命令では整数オーバフロー例外が発生しないという点です。

例外

なし

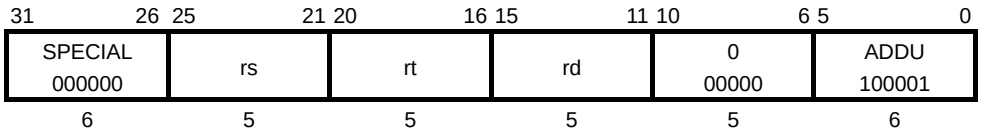
ADDU *rd*, *rs*, *rt*

Add Unsigned

動作

rd ⇐ *rs* + *rt*

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を加算し、その結果を汎用レジスタ *rd* に格納します。

ADDU 命令と ADD 命令の違いは、ADDU 命令では整数オーバフロー例外が発生しないという点です。

例外

なし

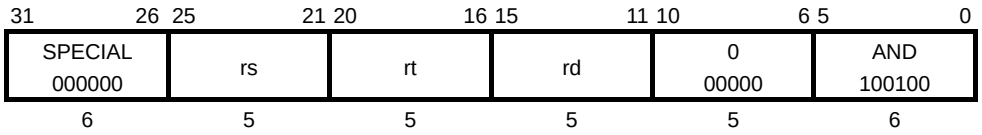
AND *rd*, *rs*, *rt*

AND

動作

rd ⇐ *rs* AND *rt*

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容の論理積演算を行い、その結果を汎用レジスタ *rd* に格納します。

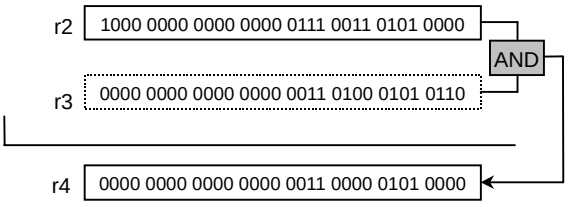
例外

なし

使用例

レジスタ r2 の値が 0x8000_7350 で、レジスタ r3 の値が 0x0000_3456 の場合、以下の命令を実行すると、図示したように、r2 と r3 の論理積 (0x0000_3050) がレジスタ r4 に格納されます。

```
AND r4,r2,r3
```



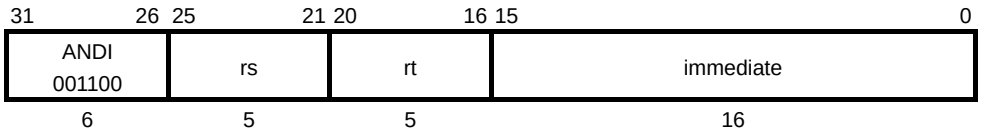
ANDI *rt*, *rs*, *immediate*

Logical AND Immediate

動作

$rt \Leftarrow rs$ AND $(0^{16} \parallel immediate_{15..0})$

コード



説明

16 ビット *immediate* をゼロ拡張し、汎用レジスタ *rs* の内容と論理積演算を行い、その結果を汎用レジスタ *rt* に格納します。

immediate は 16 ビットです。これを超える値を扱いたい場合は、いったん汎用レジスタに格納してから、AND 命令を使います (「3.3.2　32 ビットの定数」を参照)。

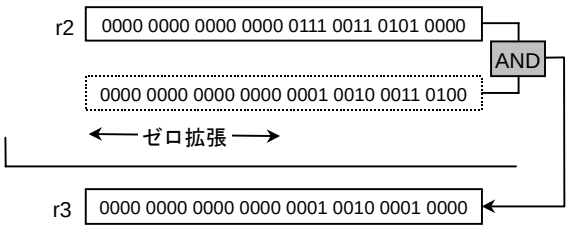
例外

なし

使用例

レジスタ r2 の値が 0x0000_7350 の場合、以下の命令を実行すると、図示したように、0x0000_7350 と 0x0000_1234 の論理積 (0x0000_1210) がレジスタ r3 に格納されます。

```
ANDI r3,r2,0x1234
```



B *offset*

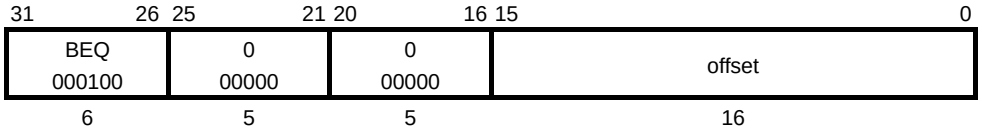
Assembly Idiom

Unconditional Branch

動作

pc ⇐ pc + 4 + sign-extend(*offset* ‖ 00)

コード



説明

1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。ターゲットアドレスは、分岐遅延スロット内の命令アドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

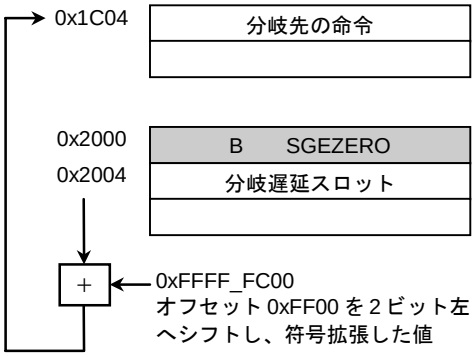
なし

使用例

```
B  SGEZERO
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SGEZERO が 0x1C04 に絶対アドレス化される場合、以下に図示するように、SGEZERO はアセンブラ・リンカによりオフセット 0xFF00 に変換されます。

プログラムの処理はアドレス 0x1C04 に分岐します。この場合、遅延スロット内の命令は、分岐の前に実行されます。



BAL *offset*

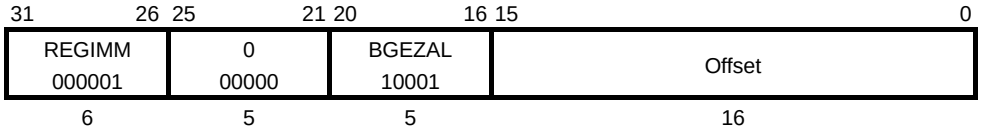
Assembly Idiom

Branch And Link

動作

r31 ⇐ pc + 8; pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐し、分岐遅延スロットの後続命令のアドレス (PC+8) を無条件にリンクレジスタ r31 (ra) に格納します。ターゲットアドレスは、分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

使用例

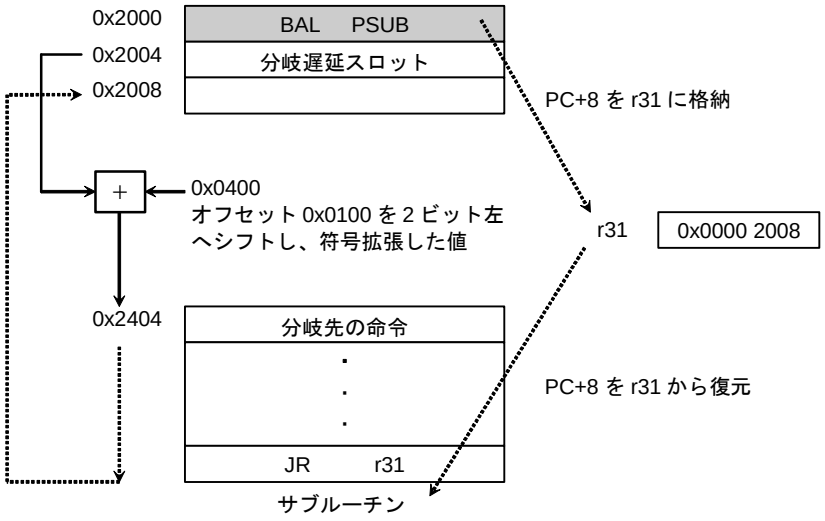
```
BAL  PSUB
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル PSUB が 0x2404 に絶対アドレス化される場合、以下に図示するように、PSUB はアセンブラ・リンカによりオフセット 0x0100 に変換されます。

プログラムの処理はアドレス 0x2404 に分岐します。この場合、分岐遅延スロット内の命令は、分岐の前に実行されます。

コールされたサブルーチンの終わりで JR 命令を実行することにより、分岐遅延スロットの直後の命令 (PC+8) に戻ることができます。

```
JR r31
```


0x2000
BAL　PSUB
0x2004
分岐遅延スロット
0x2008
PC+8 を r31 に格納
+
0x0400
オフセット 0x0100 を 2 ビット左
へシフトし、符号拡張した値
r31
0x0000 2008
0x2404
分岐先の命令
PC+8 を r31 から復元
JR　r31
サブルーチン

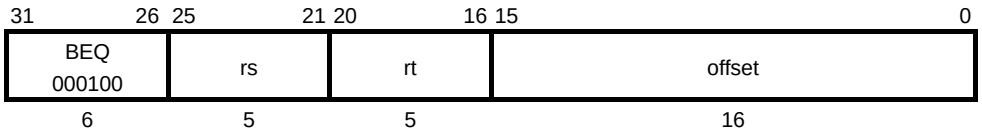
BEQ *rs*, *rt*, *offset*

Branch On Equal

動作

if *rs* = *rt* then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を比較し、両者が等しい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは、分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

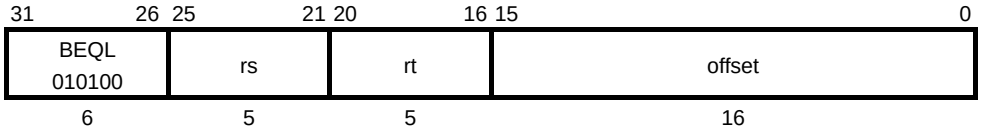
BEQL *rs*, *rt*, *offset*

Branch On Equal Likely

動作

if *rs* = *rt* then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を比較し、両者が等しい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した場合は、遅延スロット内の命令は、分岐の前に実行されます。分岐が成立しない場合は、分岐遅延スロット内の命令は無効になります。ターゲットアドレスは、分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

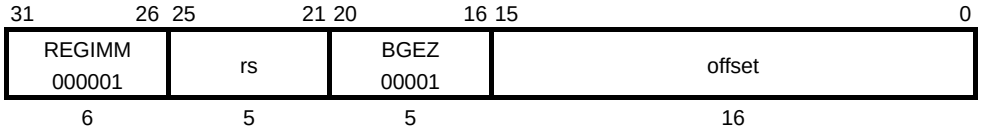
BGEZ *rs*, *offset*

Branch On Greater Than Or Equal To Zero

動作

if *rs* ≥ 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 以上の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは、分岐遅延スロット内の命令アドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

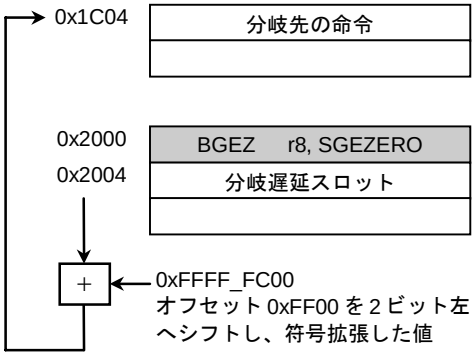
なし

使用例

```
BGEZ r8,SGEZERO
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SGEZERO が 0x1C04 に絶対アドレス化される場合、以下に図示するように、SGEZERO はアセンブラ・リンカによりオフセット 0xFF00 に変換されます。

レジスタ r8 の内容が 0 以上のとき (符号ビットが 0 であるとき)、プログラムの処理はアドレス 0x1C04 に分岐します。この場合、遅延スロット内の命令は、分岐の前に実行されます。



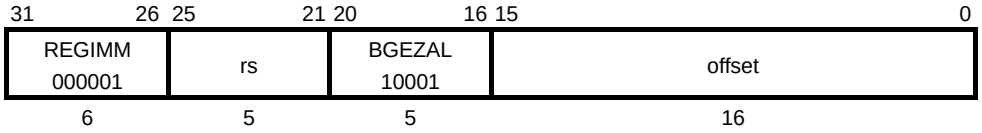
BGEZAL *rs*, *offset*

Branch On Greater Than or Equal To Zero And Link

動作

r31 ⇐ pc +8; if *rs* ≥ 0 then pc ⇐ pc + 4 + sign-extend(*offset* ‖ 00)

コード



説明

汎用レジスタ *rs* の内容が 0 以上の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐し、分岐遅延スロットの後続命令のアドレス (PC+8) を、無条件にリンクレジスタ r31 に格納します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは、分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

汎用レジスタ *rs* に r31 を指定できません。*rs* として r31 を指定すると、戻りアドレスによって *rs* の内容が破壊されてしまいます。すると、例外や割り込みにより、分岐遅延スロット内の命令が完了しなかった場合、例外処理後、分岐命令から実行を再開できなくなってしまうからです。

例外

なし

使用例

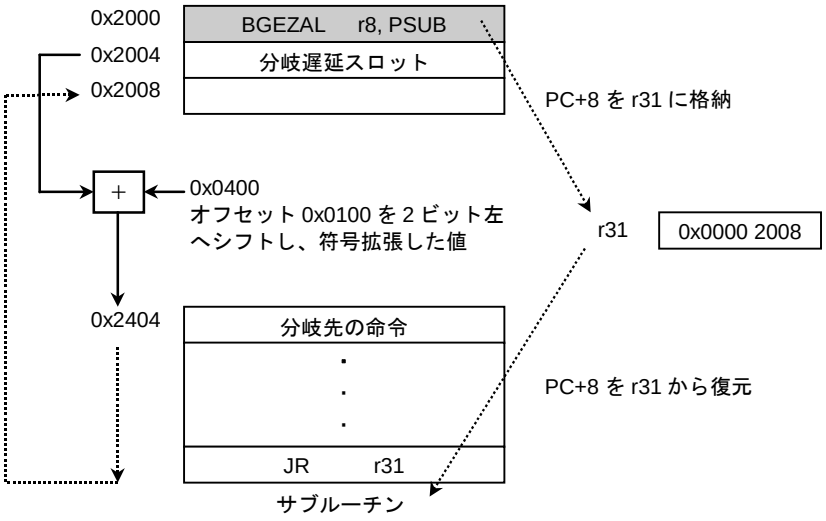
```
BGEZAL r8,PSUB
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル PSUB が 0x2404 に絶対アドレス化される場合、以下に図示するように、PSUB はアセンブラ・リンカによりオフセット 0x0100 に変換されます。

レジスタ r8 の内容が 0 以上のとき (符号ビットが 0 であるとき)、プログラムの処理はアドレス 0x2404 に分岐します。この場合、分岐遅延スロット内の命令は、分岐の前に実行されます。

コールされたサブルーチンの終わりで JR 命令を実行することにより、分岐遅延スロットの直後の命令 (PC+8) に戻ることができます。

```
JR r31
```


0x2000
0x2004
0x2008
BGEZAL　r8, PSUB
分岐遅延スロット
PC+8 を r31 に格納
+
0x0400
オフセット 0x0100 を 2 ビット左
へシフトし、符号拡張した値
r31
0x0000 2008
0x2404
分岐先の命令
PC+8 を r31 から復元
JR　r31
サブルーチン

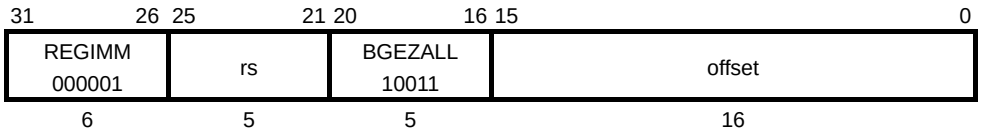
BGEZALL *rs*, *offset*

Branch On Greater Than Or Equal To Zero And Link Likely

動作

r31 ⇐ pc +8; if *rs* ≥ 0 then pc ⇐ pc + 4 + sign-extend(*offset* ‖ 00)

コード



説明

汎用レジスタ *rs* の内容が 0 以上の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐し、分岐遅延スロットの後続命令のアドレス (PC + 8) を、無条件にリンクレジスタ r31 に格納します。分岐が成立した場合は、遅延スロット内の命令は分岐の前に実行されます。分岐が成立しない場合は、分岐遅延スロット内の命令は無効になります。ターゲットアドレスは、分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

汎用レジスタ *rs* に r31 を指定できません。*rs* として r31 を指定すると、戻りアドレスによって *rs* の内容が破壊されてしまいます。すると、例外や割り込みにより、分岐遅延スロット内の命令が完了しなかった場合、例外処理後、分岐命令から実行を再開できなくなってしまうからです。

例外

なし

使用例

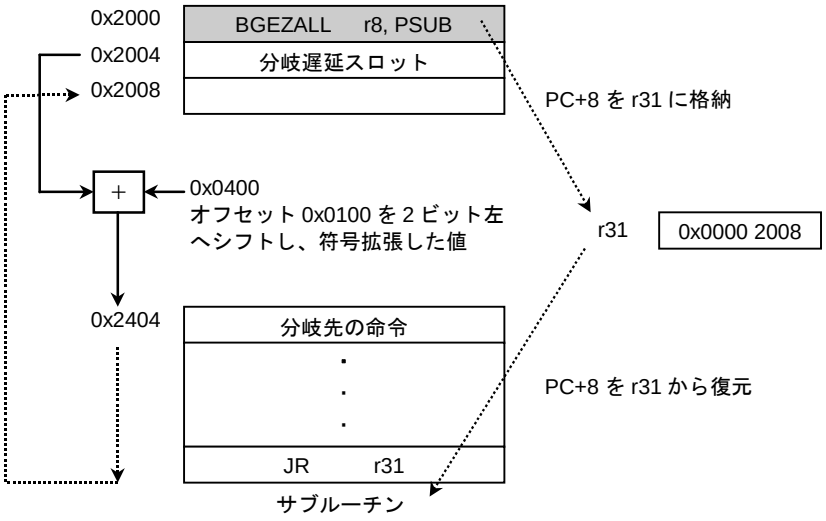
```
BGEZALL r8,PSUB
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル PSUB が 0x2404 に絶対アドレス化される場合、以下に図示するように PSUB はアセンブラ・リンカによりオフセット 0x0100 に変換されます。

レジスタ r8 の内容が 0 以上のとき (符号ビットが 0 であるとき)、プログラムの処理はアドレス 0x2404 に分岐します。この場合、分岐遅延スロット内の命令は、分岐の前に実行されます。分岐条件が成立しない場合は、分岐遅延スロット内の命令は無効になります。

コールされたサブルーチンの終わりで JR 命令を実行することにより、分岐遅延スロットの直後の命令 (PC+8) に戻ることができます。

```
JR r31
```


0x2000
0x2004
0x2008
BGEZALL　r8, PSUB
分岐遅延スロット
PC+8 を r31 に格納
+
0x0400
オフセット 0x0100 を 2 ビット左
へシフトし、符号拡張した値
r31
0x0000 2008
0x2404
分岐先の命令
PC+8 を r31 から復元
JR　r31
サブルーチン

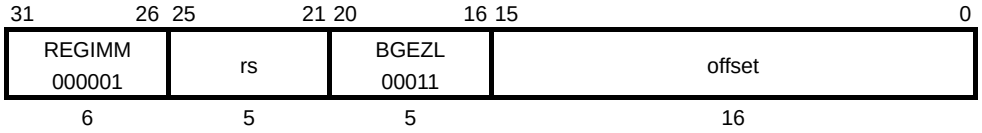
BGEZL *rs*, *offset*

Branch On Greater Than Or Equal To Zero Likely

動作

if *rs* ≥ 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 以上の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスへ分岐します。分岐が成立した場合は、遅延スロット内の命令は分岐の前に実行されます。分岐が成立しない場合は、分岐遅延スロットの命令は無効になります。ターゲットアドレスは、分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

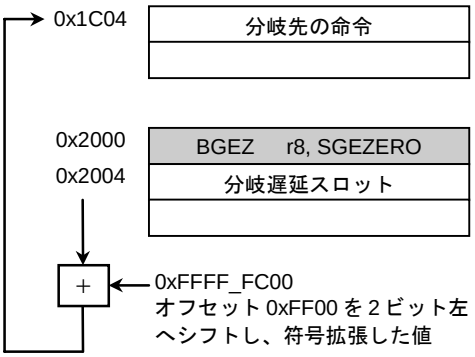
なし

使用例

```
BGEZL r8,SGEZERO
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SGEZERO が 0x1C04 に絶対アドレス化される場合、以下に図示するように、SGEZERO はアセンブラ・リンカによりオフセット 0xFF00 に変換されます。

レジスタ r8 の内容が 0 以上のとき (符号ビットが 0 であるとき)、プログラムの処理はアドレス 0x1C04 に分岐します。この場合、分岐遅延スロット内の命令は、分岐の前に実行されます。分岐条件が成立しない場合は、分岐遅延スロット内の命令は無効になります。



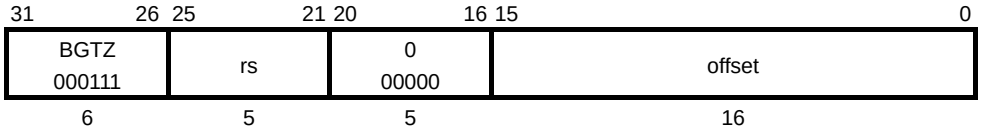
BGTZ *rs*, *offset*

Branch On Greater Than Zero

動作

if *rs* > 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 より大きい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

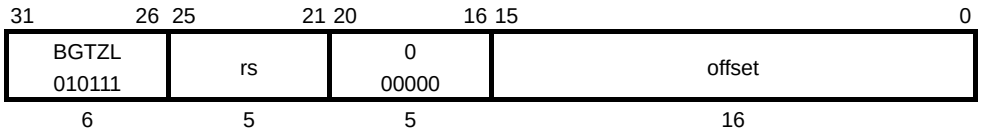
BGTZL *rs*, *offset*

Branch On Greater Than Zero Likely

動作

if *rs* > 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 より大きい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した場合は、遅延スロット内の命令は分岐の前に実行されます。分岐が成立しない場合は、分岐遅延スロット内の命令は無効になります。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

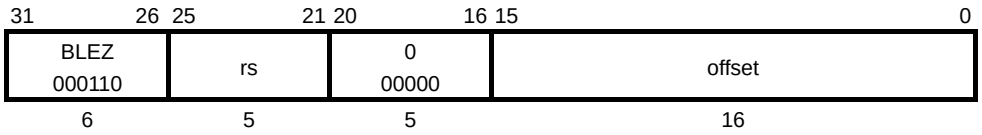
BLEZ *rs*, *offset*

Branch On Less Than Or Equal To Zero

動作

if *rs* ≤ 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 以下の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

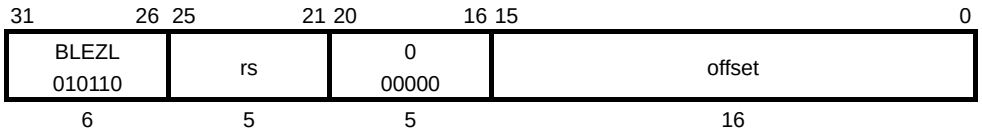
BLEZL *rs*, *offset*

Branch On Less Than Or Equal To Zero Likely

動作

if *rs* ≤ 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 以下の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した場合は、遅延スロット内の命令は分岐の前に実行されます。分岐が成立しない場合は、分岐遅延スロット内の命令は無効になります。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

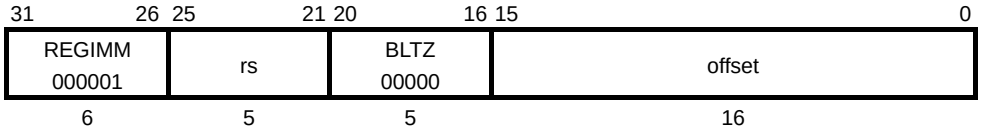
BLTZ *rs*, *offset*

Branch On Less Than Zero

動作

if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 より小さい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

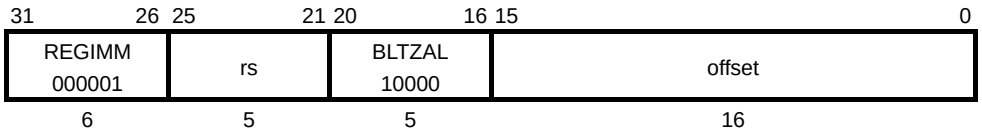
BLTZAL *rs*, *offset*

Branch On Less Than Zero And Link

動作

r31 ⇐ pc +8; if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* ‖ 00)

コード



説明

汎用レジスタ *rs* の内容が 0 より小さい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐し、分岐遅延スロットの後続命令のアドレス (PC+8) を、無条件にリンクレジスタ r31 に格納します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

汎用レジスタ *rs* に r31 を指定できません。*rs* として r31 を指定すると、戻りアドレスによって *rs* の内容が破壊されてしまいます。すると、例外や割り込みにより、分岐遅延スロット内の命令が完了しなかった場合、例外処理後、分岐命令から実行を再開できなくなってしまうからです。

例外

なし

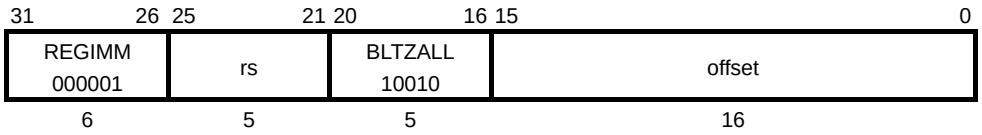
BLTZALL *rs*, *offset*

Branch On Less Than Zero And Link Likely

動作

r31 ⇐ pc +8; if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* ‖ 00)

コード



説明

汎用レジスタ *rs* の内容が 0 より小さい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐し、分岐遅延スロットの後続命令のアドレス (PC+8) を、無条件にリンクレジスタ r31 に格納します。分岐が成立した場合は、遅延スロット内の命令は分岐の前に実行されます。分岐が成立しない場合は、遅延スロット内の命令は無効になります。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算さます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

汎用レジスタ *rs* に r31 を指定できません。*rs* として r31 を指定すると、戻りアドレスによって *rs* の内容が破壊されてしまいます。すると、例外や割り込みにより、分岐遅延スロット内の命令が完了しなかった場合、例外処理後、分岐命令から実行を再開できなくなってしまうからです。

例外

なし

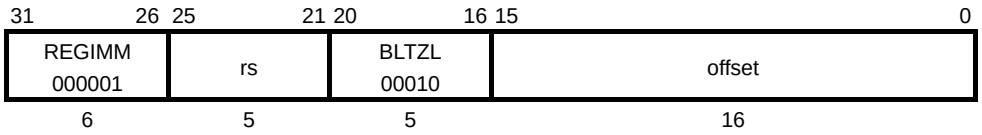
BLTZL *rs*, *offset*

Branch On Less Than Zero Likely

動作

if *rs* < 0 then pc ⇐ pc + 4 + sign-extend(*offset* || 00)

コード



説明

汎用レジスタ *rs* の内容が 0 より小さい場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した場合は、遅延スロット内の命令は分岐の前に実行されます。分岐が成立しない場合は、分岐遅延スロットの命令は無効になります。ターゲットアドレスは分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

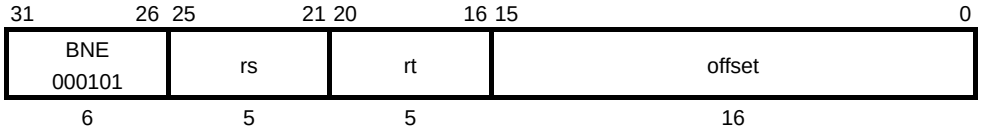
BNE *rs*, *rt*, *offset*

Branch On Not Equal

動作

if *rs* ≠ *rt* then pc ⇐ pc + 4 + sign-extend(*offset* ‖ 00)

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を比較し、両者が等しくない場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した、しないに関わらず遅延スロット内の命令は実行されます。ターゲットアドレスは、分岐遅延スロット内の命令アドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

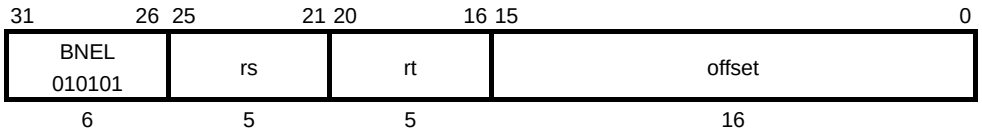
BNEL *rs*, *rt*, *offset*

Branch On Not Equal Likely

動作

if *rs* ≠ *rt* then pc ⇐ pc + 4 + sign-extend(*offset* ‖ 00)

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を比較し、両者が等しくない場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。分岐が成立した場合は、遅延スロット内の命令は分岐の前に実行されます。分岐が成立しない場合は、分岐遅延スロット内の命令は無効になります。ターゲットアドレスは、分岐遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。16 ビット *offset* を 2 ビット左へシフトし、符号拡張した値を PC+4 に加算した結果がターゲットアドレスになります。

例外

なし

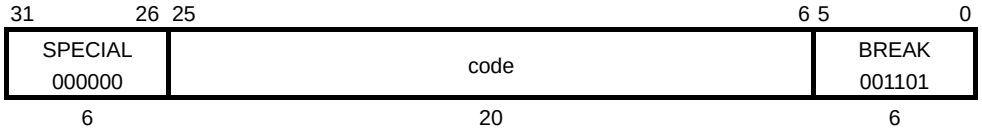
BREAK *code*

Breakpoint

動作

ブレークポイント例外

コード



説明

この命令を実行すると、無条件にブレークポイント例外が発生し、制御を例外ハンドラへ渡します。

命令内の *code* フィールドを使用して、例外ハンドラにパラメータを渡すことができます。例外ハンドラがこのパラメータを使用する場合には、命令を含むメモリワードの内容をデータとしてロードする必要があります。詳細は「9.1.11　ブレークポイント例外」を参照してください。

例外

ブレークポイント例外

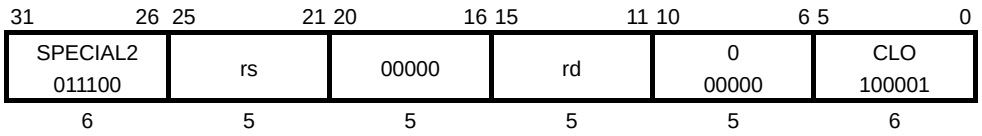
CLO *rd*, *rs*

Count Leading Ones in Word

動作

rd ⇐ count_leading_ones *rs*

コード



説明

汎用レジスタ *rs* の内容がビット 31 からビット 0 の方向へ何ビット連続して 1 が並んでいるかカウントし、その結果を汎用レジスタ *rd* に格納します。もし、*rs* の全ビットがセットされていたら、32 を *rd* に格納します。

例外

なし

使用例

汎用レジスタ r2 に 0xFE23_DE67 が格納されているとします。

```
CLO     r4, r2
```

このとき、上記の命令を実行すると、汎用レジスタ r4 には、0x0000_0007 が格納されます。

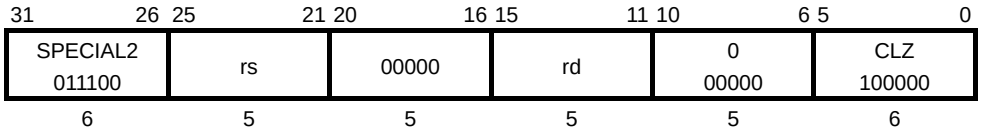
CLZ *rd*, *rs*

Count Leading Zeros in Word

動作

rd ⇐ count_leading_zeros *rs*

コード



説明

汎用レジスタ *rs* の内容がビット 31 からビット 0 の方向へ何ビット連続して 0 が並んでいるかカウントし、その結果を汎用レジスタ *rd* に格納します。もし、*rs* の全ビットがクリアされていたら、32 を *rd* に格納します。

例外

なし

使用例

汎用レジスタ r2 に 0x07EF_45CD が格納されているとします。

```
CLZ     r4, r2
```

このとき、上記の命令を実行すると、汎用レジスタ r4 には、0x0000_0005 が格納されます。

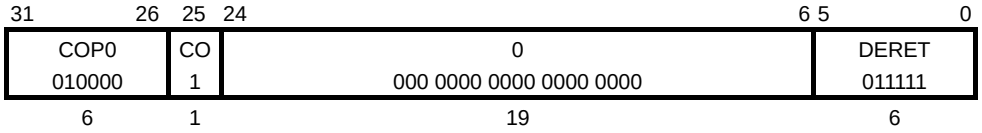
DERET

Debug Exception Return

動作

pc ⇐ DEPC, Debug[DM] ⇐ 0, Debug[IEXI] ⇐ 0

コード



説明

DERET 命令は、デバッグ例外処理から復帰するための命令です。DEPC レジスタの内容をプログラムカウンタ (PC) にロードすることにより実行されます。詳細については、「9.3.6 デバッグ例外からの復帰」を参照してください。

DERET 命令には、分岐遅延スロットがありません。1 命令 (2 命令サイクル) の遅延後、実行されます。

DERET 命令は、DEPC レジスタのビット 0 を PC の ISA モードビット (ビット 0) に復元し、デバッグ例外が実行される前の ISA モードになります。

DERET 命令を、ジャンプまたは分岐遅延スロットに置いてはいけません。

デバッグモードでないとき (Debug レジスタの DM ビットが 0 のとき) は、DERET 命令の動作は保証されません。

通常、デバッグ例外が発生すると、例外の原因となった命令のアドレスが自動的に DEPC レジスタに保存されます。MTC0 命令を使って DEPC レジスタに戻りアドレスを設定したい場合は、デバッグ例外ハンドラで少なくとも 2 つの命令を実行してから、DERET 命令を実行しなければなりません。

例外

なし

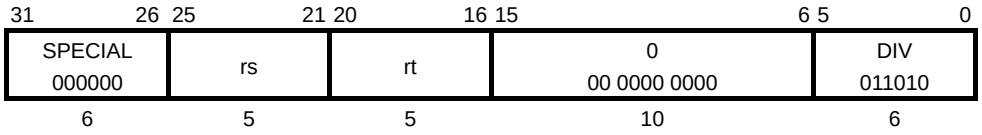
DIV *rs*, *rt*

Divide

動作

LO ⇐ *rs* ÷ *rt*;
HI ⇐ *rs* MOD *rt*

コード



説明

汎用レジスタ *rs* の内容を汎用レジスタ *rt* の内容で除算します。両オペランドとも、符号付き整数として扱われます。商は LO に格納され、剰余は HI レジスタに格納されます。整数オーバフロー例外は発生しません。

除数が 0 の場合、DIV 命令の結果は確定しません。通常、DIV 命令の後に、ゼロ除算とオーバフローを検査する命令を置きます。

除算命令は、専用の除算ユニットで実行されるため、他の命令の実行を並行して継続できます。除算ユニットは、遅延サイクル、例外が起きたときでも、実行を継続します。

除算命令が完了する前に、MFHI、MFLO、MADD、 MADDU、MSUB、MSUBU 命令で除算結果を読もうとすると、パイプラインがストールします(「5.4　除算命令」を参照)。

例外

なし

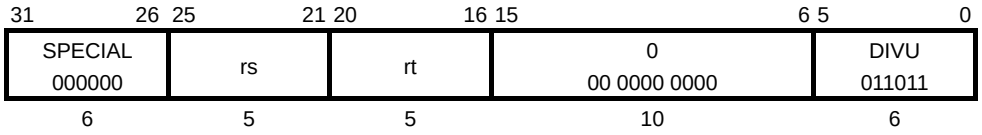
DIVU *rs*, *rt*

Divide Unsigned

動作

LO ⇐ *rs* ÷ *rt*;
HI ⇐ *rs* MOD *rt*

コード



説明

汎用レジスタ *rs* の内容を汎用レジスタ *rt* の内容で除算します。商は LO レジスタに格納され、剰余は HI レジスタに格納されます。整数オーバフロー例外を発生しません。DIV 命令との違いは、DIVU 命令ではオペランドが符号なし整数として扱われるという点です。

例外

なし

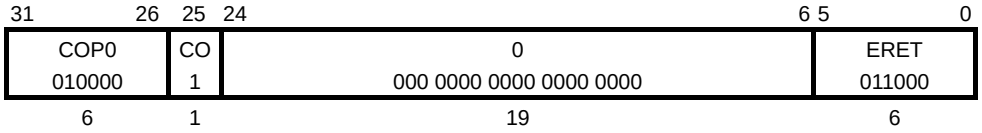
ERET

Exception Return

動作

```
if Status[ERL] = 1 then pc ⇐ ErrorEPC
                        Status[ERL] ⇐ 0
                   else pc ⇐ EPC
                        Status[EXL] ⇐ 0
SSCR[CSS] ⇐ SSCR[PSS]
```

コード



説明

ERET 命令は、割り込み、例外、エラートラップから復帰するための命令です。

ERET 命令には分岐遅延スロットがありません。1 命令 (2 命令サイクル) の遅延後、分岐が実行されます。

ERET 命令は ErrorEPC レジスタのビット 0 を PC の ISA モードビット (ビット 0) に復元し、例外が発生する前の ISA モードになります。

ERET 命令は、ユーザーモードで Status レジスタの CU0 ビットを許可しないで実行すると、コプロセッサ使用不可例外が発生します。MTC0 命令を使って ErrorEPC レジスタあるいは EPC レジスタに戻りアドレスを設定したい場合および Status レジスタの値を書き換えた場合は、例外ハンドラで少なくとも 2 つの命令を実行してから、ERET 命令を実行しなければなりません。

本命令を実行した場合、Status レジスタの ERL ビット＝1 のときは、ErrorEPC レジスタから PC をロードし、同時に ERL ビットが 0 にクリアされます。そうでないときは、EPC レジスタから PC をロードし、同時に EXL ビットが 0 にクリアされます。

本命令を実行した場合、SSCR レジスタの PSS フィールドの値が CSS フィールドに書かれます。

本命令自体をジャンプ命令または分岐命令の遅延スロットに置いてはなりません。

例外

コプロセッサ使用不可例外

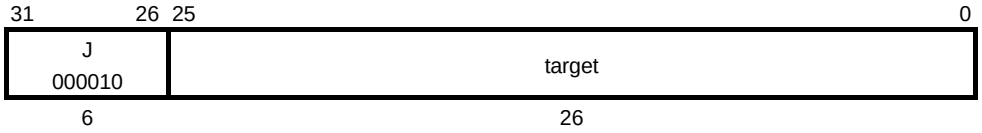
J *target*

Jump

動作

pc ⇐ pc[31:28] ‖ *target* ‖ 00

コード



説明

1 命令 (2 命令サイクル) の遅延後、無条件にターゲットアドレスへジャンプします。ターゲットアドレスは、ジャンプ遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。26 ビット *target* を 2 ビット左へシフトし、PC+4 の上位 4 ビットと連結した結果がターゲットアドレスになります。

J 命令では、ターゲットアドレスは、2^{28} バイトセグメント内でなければなりません。任意の 32 ビットのアドレスへジャンプするには、アドレスをいったんレジスタに格納してから、JR 命令を使います (「3.4.6　32 ビットのアドレスへのジャンプ」を参照)。

例外

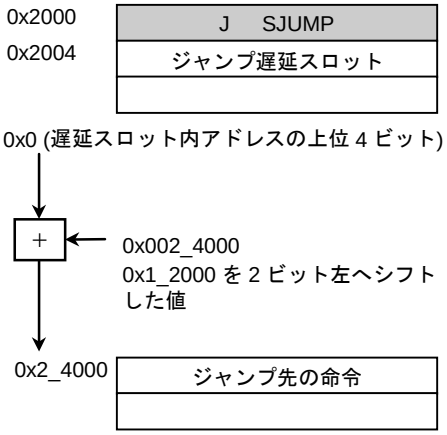
なし

使用例

```
J SJUMP
```

上記のジャンプ命令がアドレス 0x2000 にあり、ラベル SJUMP が 0x2_4000 に絶対アドレス化される場合、SJUMP はアセンブラ・リンカにより 0x1_2000 に変換されます。

プログラムの処理は、無条件にアドレス 0x2_4000 にジャンプします。ジャンプ遅延スロット内の命令はジャンプの前に実行されます。


0x2000
0x2004
J　SJUMP
ジャンプ遅延スロット
0x0 (遅延スロット内アドレスの上位 4 ビット)
+
0x002_4000
0x1_2000 を 2 ビット左へシフトした値
0x2_4000
ジャンプ先の命令

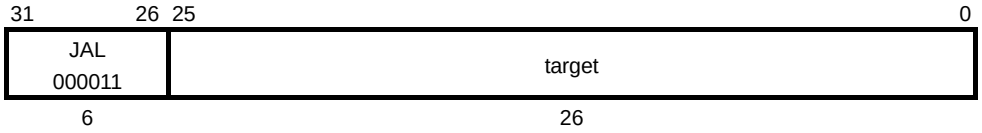
JAL *target*

Jump And Link

動作

r31 ⇐ pc + 8; pc ⇐ pc[31:28] ‖ *target* ‖ 00

コード



説明

1 命令 (2 命令サイクル) の遅延後、無条件にターゲットアドレスにジャンプします。ターゲットアドレスは、ジャンプ遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。26 ビット *target* を 2 ビット左へシフトし、PC+4 の上位 4 ビットと連結した結果がターゲットアドレスになります。プログラムカウンタ (PC) の ISA モードビットは変化しません。

ジャンプ遅延スロットの次の命令のアドレス (PC+8) を、リンクレジスタ r31 (ra) に格納します。また、r31 の最下位ビットに、ジャンプ前の ISA モードビットを格納します。

JAL 命令では、ターゲットアドレスは、2^{28} バイトセグメント内でなければなりません。任意の 32 ビットのアドレスへジャンプするには、アドレスをいったんレジスタに格納してから、JR 命令を使います (「3.4.6　32 ビットのアドレスへのジャンプ」を参照)。

例外

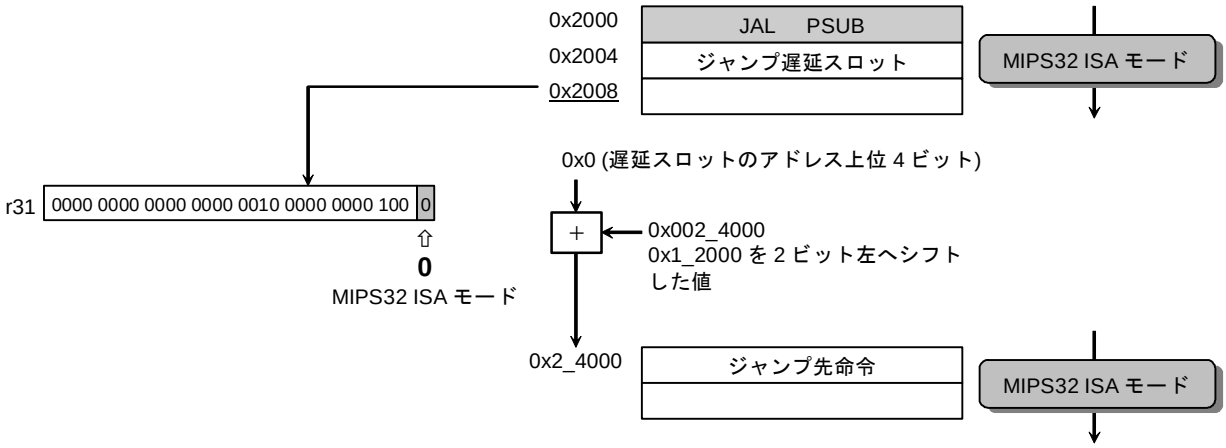
なし

使用例

```
JAL  PSUB
```

上記のジャンプ命令がアドレス 0x2000 にあり、ラベル PSUB が 0x2_4000 に絶対アドレス化される場合、以下に図示するように、PSUB はアセンブラ・リンカにより 0x1_2000 に変換されます。

プログラムの処理は、無条件にアドレス 0x2_4000 にジャンプします。ジャンプ遅延スロット内の命令は、ジャンプの前に実行されます。また、ジャンプ遅延スロットの次の命令のアドレスが、リンクレジスタ r31 に格納されます。


0x2000
0x2004
0x2008
JAL　PSUB
ジャンプ遅延スロット
MIPS32 ISA モード
0x0 (遅延スロットのアドレス上位 4 ビット)
r31
0000 0000 0000 0000 0010 0000 0000 100
0
0
MIPS32 ISA モード
+
0x002_4000
0x1_2000 を 2 ビット左へシフト
した値
0x2_4000
ジャンプ先命令
MIPS32 ISA モード

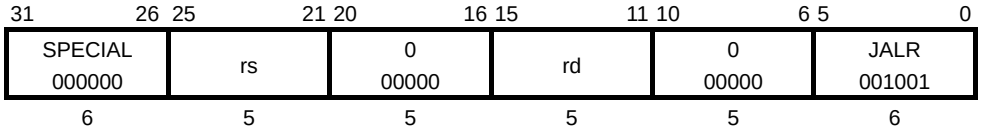
JALR (*rd*,) *rs*

Jump And Link Register

動作

rd or r31 ⇐ pc + 8; pc ⇐ *rs*

コード



説明

1 命令 (2 命令サイクル) の遅延後、汎用レジスタ *rs* の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。*rs* の最下位ビットの値によって、ISA モードが切り換わります。また、ジャンプ遅延スロットの次の命令のアドレス (PC+8) を、汎用レジスタ (rd) に格納します。オペランド *rd* を省略すると、デフォルトで r31 (ra) が使用されます。

rs と *rd* に、同じレジスタを指定できません。*rd* として *rs* を指定すると、戻りアドレスによって *rs* の内容が破壊されてしまいます。すると、例外や割り込みにより、ジャンプ遅延スロットの命令が完了しなかった場合、例外処理後、ジャンプ命令から実行を再開できなくなってしまうためです。

32 ビット ISA では、命令はすべてワード境界で位置合わせされなければなりません。したがって、ジャンプ後の ISA モードを 32 ビットに指定する場合、ターゲットレジスタ (*rs*) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。

例外

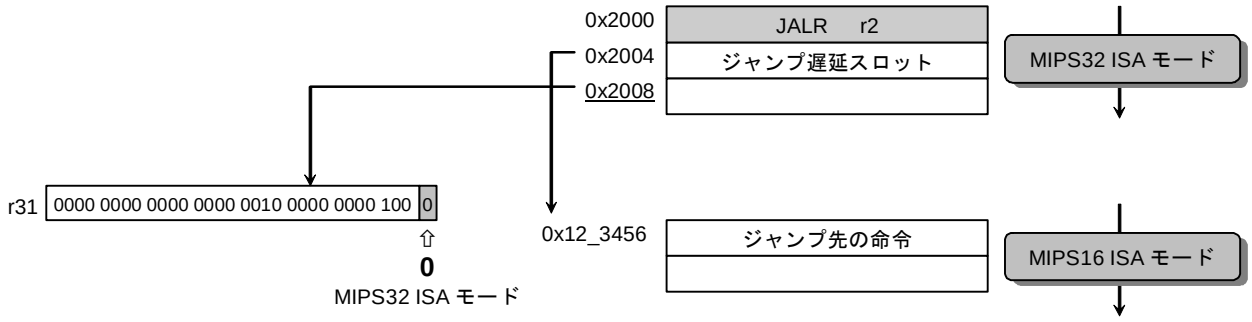
なし

使用例

レジスタ r2 の内容が 0x0012_3457 で、以下のジャンプ命令がアドレス 0x0000_2000 に置かれているとします。

```
JALR r2
```

上記の命令を実行すると、プログラムの処理は、0x0012_3457 の最下位ビットを 0 にマスクしたアドレス 0x0012_3456 にジャンプします。ジャンプ遅延スロット内の命令はジャンプの前に実行されます。レジスタ r2 の最下位ビットは 1 に設定されているので、ジャンプ後の ISA モードビットは 1 に変化し、16 ビット ISA モードになります。戻りアドレス 0x0000_2008 は、ISA モードビットと共にリンクレジスタ r31 に格納されます。


0x2000
JALR　r2
0x2004
ジャンプ遅延スロット
0x2008
MIPS32 ISA モード
r31
0000 0000 0000 0000 0010 0000 0000 100
0
0
MIPS32 ISA モード
0x12_3456
ジャンプ先の命令
MIPS16 ISA モード

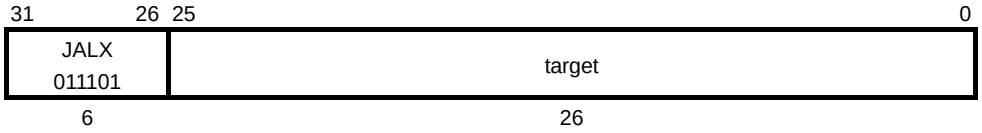
JALX *target*

Jump And Link eXchange

動作

r31 ⇐ pc + 8; pc[31:1] ⇐ pc[31:28] ‖ *target* ‖ 00; pc[0] ⇐ NOT pc[0]

コード



説明

1 命令 (2 命令サイクル) の遅延後、無条件にターゲットアドレスにジャンプします。ターゲットアドレスは、ジャンプ遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。26 ビット *target* を 2 ビット左へシフトし、PC+4 の上位 4 ビットを連結した結果がターゲットアドレスになります。プログラムカウンタ (PC) の ISA モードビットは無条件に変化します。

ジャンプ遅延スロットの次の命令のアドレス (PC+8) を、リンクレジスタ r31 (ra) に格納します。また、r31 の最下位ビットに、ジャンプ前の ISA モードビットを格納します。

例外

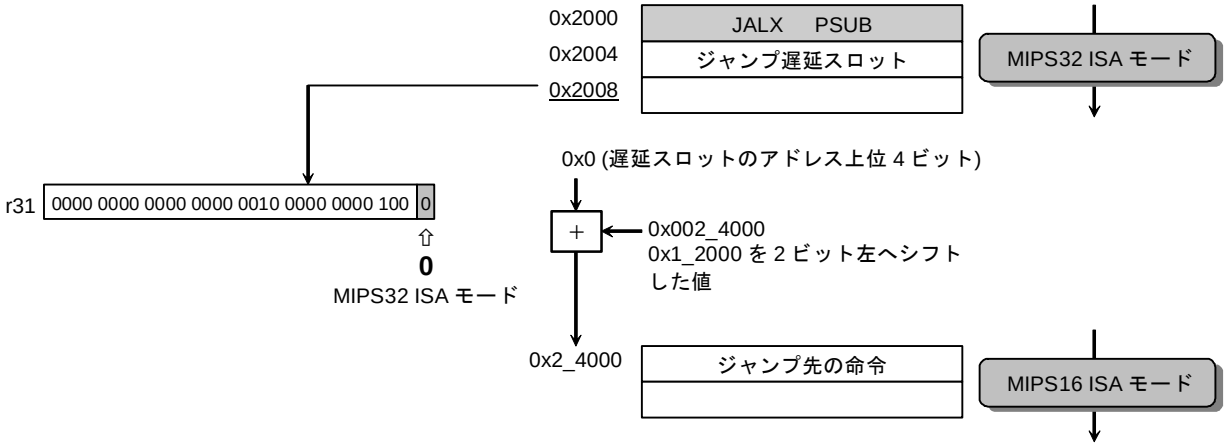
なし

使用例

```
JALX  PSUB
```

上記のジャンプ命令がアドレス 0x0000_2000 にあり、ラベル PSUB が 0x2_4000 に絶対アドレス化される場合、以下に図示するように、PSUB はアセンブラ・リンカにより 0x1_2000 に変換されます。

プログラムの処理は、無条件にアドレス 0x2_4000 にジャンプします。ジャンプ遅延スロット内の命令は、ジャンプの前に実行されます。ISA モードビットは無条件に変化し、16 ビット ISA モードになります。また、戻りアドレス 0x0000_2008 が、ISA モードビットと共にリンクレジスタ r31 に格納されます。


0x2000
0x2004
0x2008
JALX　PSUB
ジャンプ遅延スロット
MIPS32 ISA モード
0x0 (遅延スロットのアドレス上位 4 ビット)
r31
0000 0000 0000 0000 0010 0000 0000 100
0
0
MIPS32 ISA モード
+
0x002_4000
0x1_2000 を 2 ビット左へシフト
した値
0x2_4000
ジャンプ先の命令
MIPS16 ISA モード

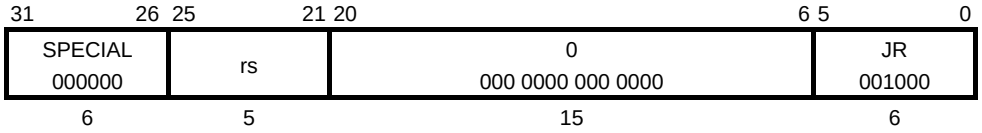
JR *rs*

Jump Register

動作

pc ⇐ *rs*

コード



説明

1 命令 (2 命令サイクル) の遅延後、汎用レジスタ *rs* の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。*rs* の最下位ビットの値によって ISA モードが切り換わります。

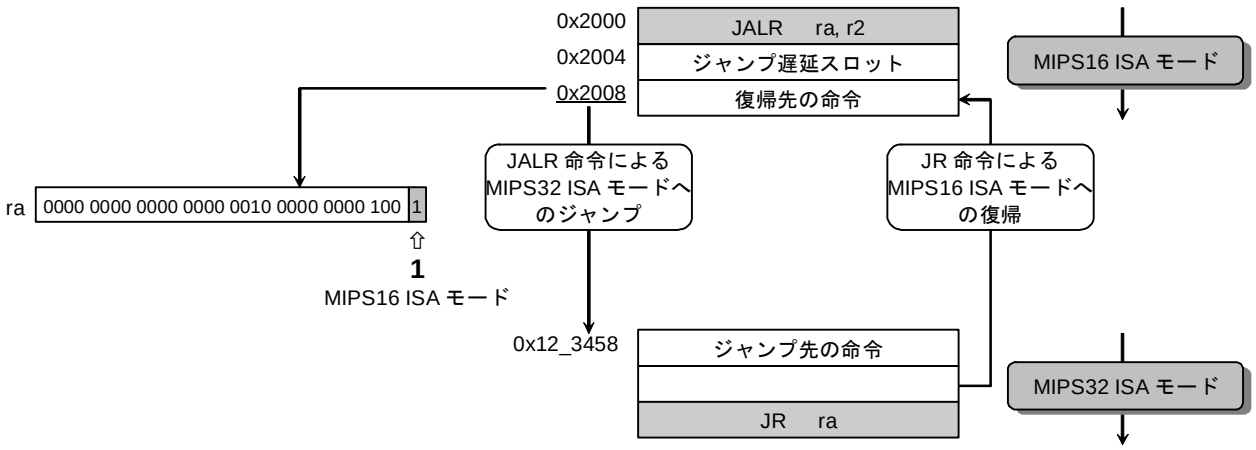
32 ビット ISA では、命令はすべてワード境界に位置合わせされていなければなりません。そのため、32 ビットのモードにジャンプするとき、ターゲットレジスタ (*rs*) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、プロセッサがジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。

例外

なし

使用例

以下の例では、16 ビットの JALR 命令により 32 ビットモードのルーチンにジャンプします。32 ビットのルーチンの最後で、JR 命令により戻りアドレスをリンクレジスタ r31 (ra) からプログラムカウンタ (PC) に復元しています。JALR 命令により、ISA モードが ra の最下位ビットに保存されているので、32 ビットルーチンの終わりで JR 命令を実行すると、16 ビット ISA モードに戻ります。



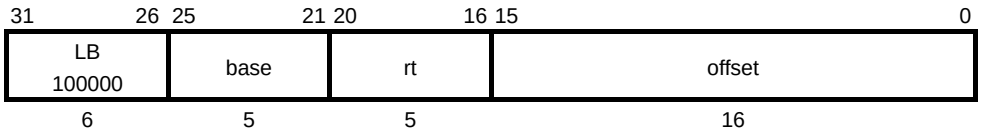
LB *rt*, *offset* (*base*)

Load Byte

動作

rt ⇐ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張して、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータを符号拡張して汎用レジスタ *rt* にロードします。

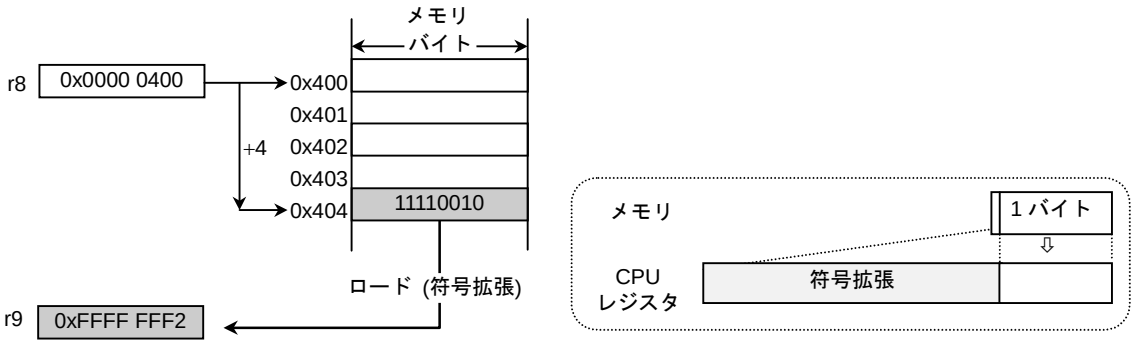
例外

アドレスエラー例外

使用例

レジスタ r8 の値が 0x0000_0400 で、アドレス 0x404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r9 に 0xFFFF_FFF2 がロードされます。

```
LB r9,4(r8)
```



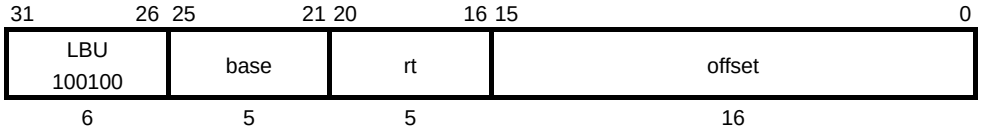
LBU *rt*, *offset* (*base*)

Load Byte Unsigned

動作

rt ⇐ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張して、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータをゼロ拡張して、汎用レジスタ *rt* にロードします。

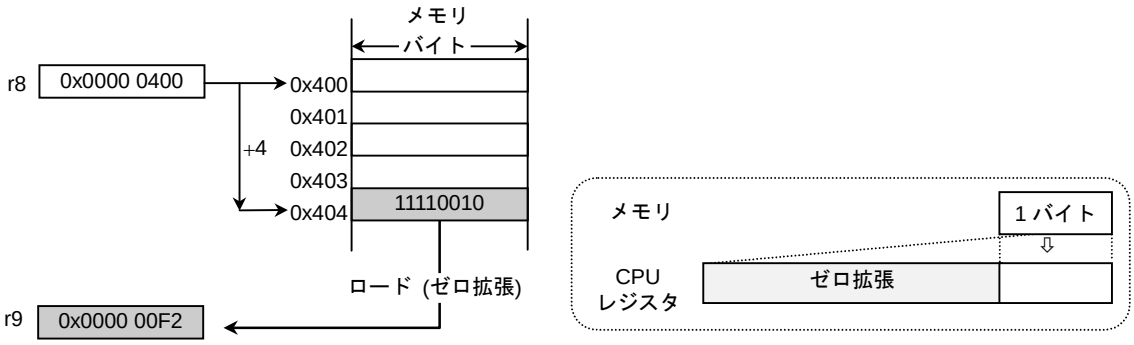
例外

アドレスエラー例外

使用例

レジスタ r8 の値が 0x0000_0400 で、アドレス 0x404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r9 には 0x0000_00F2 がロードされます。

```
LBU r9,4(r8)
```



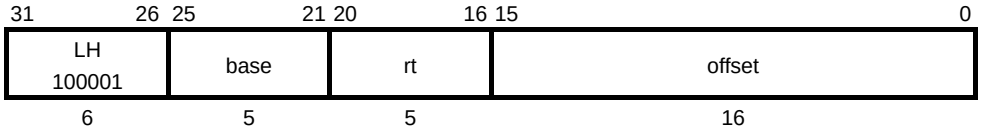
LH *rt*, *offset* (*base*)

Load Halfword

動作

rt ⇐ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張して汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のハーフワードデータを符号拡張して、汎用レジスタ *rt* にロードします。

実効アドレスの最下位ビットが 0 でない (実効アドレスがハーフワード境界でない) 場合、アドレスエラー例外が発生します。

例外

アドレスエラー例外

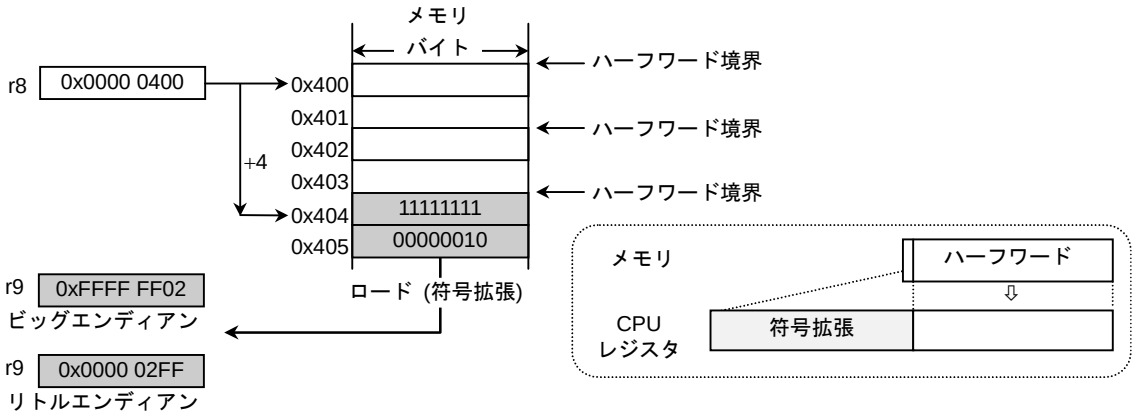
使用例

レジスタ r8 の値が 0x0000_0400 で、アドレス 0x404 と 0x405 の内容がそれぞれ 0xFF と 0x02 の場合、以下の命令を実行すると、レジスタ r9 にはビッグエンディアンのときは 0xFFFF_FF02 がロードされ、リトルエンディアンのときは 0x0000_02FF がロードされます。

```
LH r9,4(r8)
```

また、以下の命令を実行すると、0x403 はハーフワード境界でないので、アドレスエラー例外が発生します。

```
LH r9,3(r8)
```


メモリ
バイト
ハーフワード境界
r8
0x0000 0400
0x400
0x401
0x402
0x403
0x404
0x405
+4
11111111
00000010
ハーフワード境界
ハーフワード境界
メモリ
ハーフワード
CPU
レジスタ
符号拡張
r9
0xFFFF FF02
ビッグエンディアン
ロード (符号拡張)
r9
0x0000 02FF
リトルエンディアン

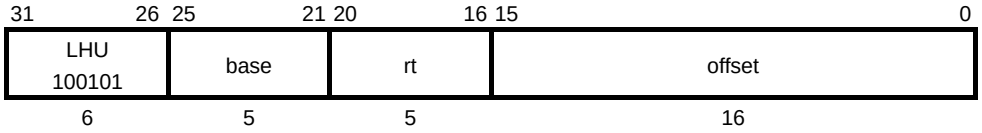
LHU *rt*, *offset* (*base*)

Load Halfword Unsigned

動作

rt ⇐ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張して、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のハーフワードデータをゼロ拡張し、汎用レジスタ *rt* にロードします。

実効アドレスの最下位ビットが 0 でない (実効アドレスがハーフワード境界でない) 場合、アドレスエラー例外が発生します。

例外

アドレスエラー例外

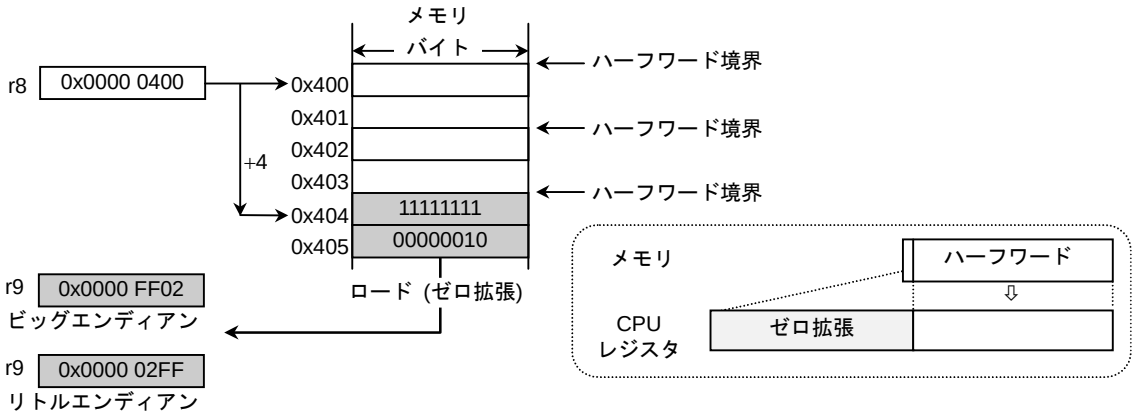
使用例

レジスタ r8 の値が 0x0000_0400 で、アドレス 0x404 と 0x405 の内容がそれぞれ 0xFF と 0x02 の場合、以下の命令を実行すると、レジスタ r9 にはビッグエンディアンのときは 0x0000_FF02 がロードされ、リトルエンディアンのときは 0x0000_02FF がロードされます。

```
LHU r9,4(r8)
```

また、以下の命令を実行すると、0x403 はハーフワード境界でないので、アドレスエラー例外が発生します。

```
LH r9,3(r8)
```


メモリ
バイト
ハーフワード境界
r8
0x0000 0400
0x400
0x401
0x402
0x403
+4
0x404
0x405
11111111
00000010
ハーフワード境界
ハーフワード境界
ロード (ゼロ拡張)
r9
0x0000 FF02
ビッグエンディアン
r9
0x0000 02FF
リトルエンディアン
メモリ
ハーフワード
CPU
レジスタ
ゼロ拡張

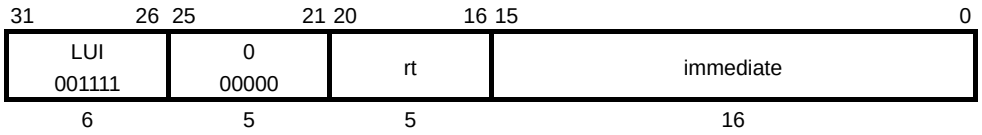
LUI *rt*, *immediate*

Load Upper Immediate

動作

rt ⇐ *immediate* ‖ 0x0000

コード



説明

16 ビット *immediate* を 16 ビット左へシフトし、下位 16 ビットのを 0 で埋めた値を汎用レジスタ *rt* にロードします。

例外

なし

使用例

以下の命令は、レジスタ r9 に 0x1234_0000 をロードします。

```
LUI r9,0x1234
```

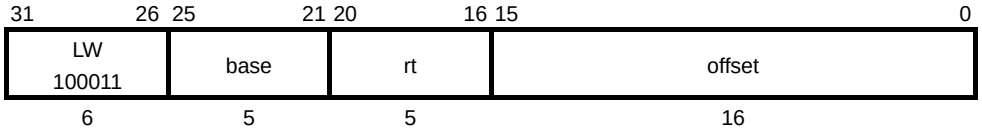
LW *rt*, *offset* (*base*)

Load Word

動作

rt ⇐ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張して汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のワードデータを、汎用レジスタ *rt* にロードします。

実効アドレスの下位 2 ビットが 0 でない (実効アドレスがワード境界でない) 場合、アドレスエラーが発生します。

例外

アドレスエラー例外

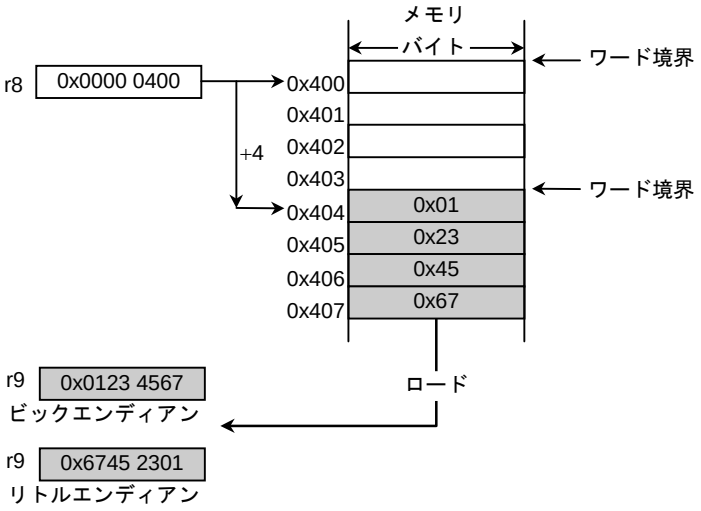
使用例

レジスタ r8 の値が 0x0000_0400 で、アドレス 0x404 から 0x407 の内容が 0x01、0x23、0x45、0x67 の場合、以下の命令を実行すると、レジスタ r9 にはビッグエンディアンモードのときは 0x0123_4567 がロードされ、リトルエンディアンモードのときは 0x6745_2301 がロードされます。

```
LW r9,4(r8)
```

また、以下の命令を実行すると、0x405 はワード境界でないので、アドレスエラー例外が発生します。

```
LW r9,5(r8)
```


メモリ
バイト
ワード境界
r8
0x0000 0400
0x400
0x401
0x402
0x403
ワード境界
+4
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ロード
r9
0x0123 4567
ビックエンディアン
r9
0x6745 2301
リトルエンディアン

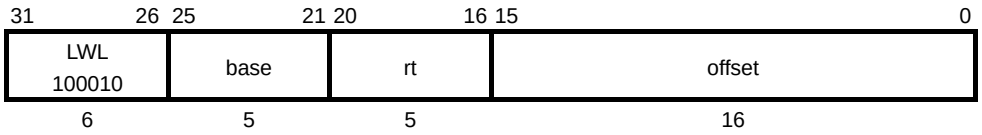
LWL *rt*, *offset* (*base*)

Load Word Left

動作

rt ⇐ *rt* MERGE {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張し、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。ワード境界にない、メモリ中のワードデータの上位部分を汎用レジスタ *rt* に左詰めでロードします。

実効アドレスがワード境界に位置していないことによる、アドレスエラー例外は発生しません。

直前のロード命令と後続の LWL 命令は、同じ汎用レジスタを *rt* として指定できます。汎用レジスタ *rt* の内容は、プロセッサコア内で内部的にバイパシング (フォワーディング) されるので、両命令の間に NOP 命令を入れる必要はありません。

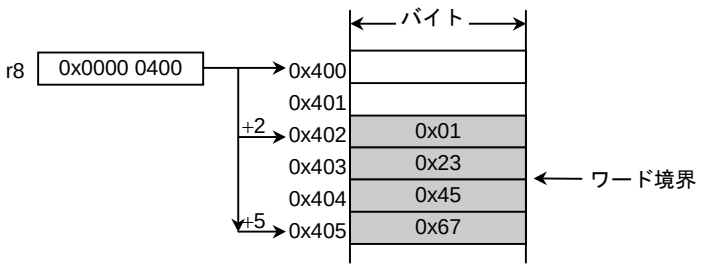
ワード境界に位置合わせされていないメモリ中のワードを、汎用レジスタにロードするときに、LWL 命令と LWR 命令を組み合わせて使用します。

例外

アドレスエラー例外

使用例

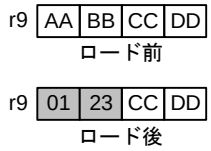
レジスタ r8 の値が 0x0000_0400 で、アドレス 0x402 ~ 0x405 の内容が 0x01、0x23、0x45、0x67 であるとします。



- ビッグエンディアンのとき

```
LWL r9,2(r8)
```

上記の命令は、アドレス 0x402 のバイト位置から、上位アドレス方向へワード境界に達するまで、データを読み出し、r9 の最上位バイトから順にロードします。その結果を以下に示します。



(a) ビッグエンディアン

- リトルエンディアンのとき

```
LWL r9,5(r8)
```

上記の命令は、アドレス 0x405 のバイト位置から、下位アドレス方向へワード境界に達するまで、データを読み出し、r9 の最上位バイトから順にロードします。その結果を以下に示します。

(b) リトルエンディアン

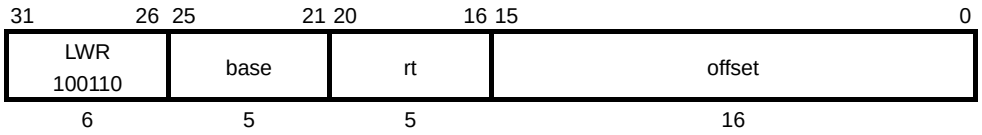
LWR *rt*, *offset* (*base*)

Load Word Right

動作

rt ⇐ *rt* MERGE {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張し、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。ワード境界にない、メモリ中のワードデータの下位部分を汎用レジスタ *rt* に右詰めでロードします。

実効アドレスがワード境界に位置していないことによる、アドレスエラー例外は発生しません。

直前のロード命令と後続の LWLR 命令は、同じ汎用レジスタを *rt* として指定できます。汎用レジスタ *rt* の内容は、プロセッサコア内で内部的にバイパシング (またはフォワーディング) されるので、両命令の間に NOP 命令を入れる必要はありません。

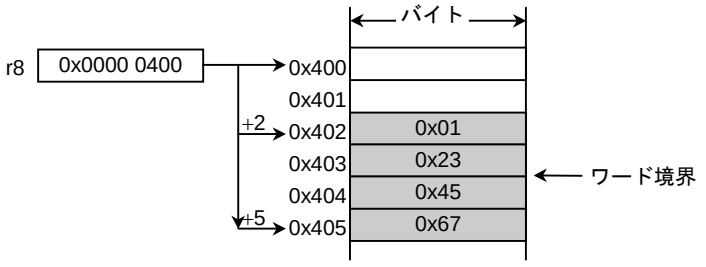
ワード境界に位置合わせされていないメモリ中のワードを、汎用レジスタにロードするときに、LWL 命令と LWR を組み合わせて使用します。

例外

アドレスエラー例外

使用例

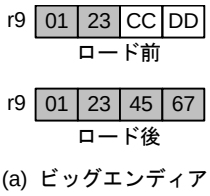
レジスタ r8 の値が 0x0000_0400 で、アドレス 0x402 ~ 0x405 の内容が 0x01、0x23、0x45、0x67 であるとします。



- ビッグエンディアンのとき

```
LWR r9,5(r8)
```

上記の命令は、アドレス 0x405 のバイト位置から、下位アドレス方向へワード境界に達するまで、データを読み出し、r9 の最下位バイトから順にロードします。その結果を以下に示します。



(a) ビッグエンディアン

- リトルエンディアンのとき

```
LWR r9,2(r8)
```

上記命令は、アドレス 0x402 のバイト位置から、上位アドレス方向へワード境界に達するまで、データを読み出し、r9 の最下位バイトから順にロードします。その結果を以下に示します。

(b) リトルエンディアン

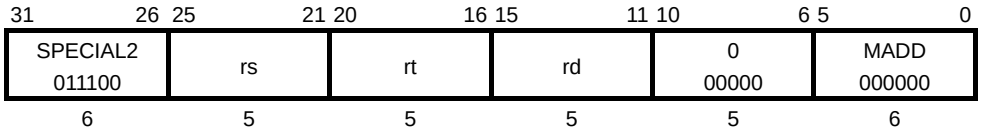
MADD (*rd*,) *rs*, *rt*

Multiply and Add

動作

HI ⇐ (HI ‖ LO) + (*rs* × *rt*) の上位ワード
LO ⇐ (HI ‖ LO) + (*rs* × *rt*) の下位ワード
rd ⇐ (HI ‖ LO) + (*rs* × *rt*) の下位ワード

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を乗算し、その積を HI・LO レジスタに格納されているダブルワードの値に加算します。*rs* と *rt* を符号付き整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。デスティネーションレジスタ *rd* が指定されている場合、結果の下位ワードを *rd* にも格納します。

rd を省略すると、デフォルトで r0 になり、その結果、汎用レジスタに結果は格納されません。

いかなる場合でも、整数オーバフロー例外は発生しません。

例外

なし

使用例

HI レジスタには 0x0000_0000 が、LO レジスタには 0xFFFF_FFFF が格納されていて、汎用レジスタ r2 には 0x0123_4567 が、r3 に 0x89AB_CDEF が格納されているとします。

```
MADD r4,r2,r3
```

このとき、上記の命令は、以下の演算を実行します。

0x0000_0000_FFFF_FFFF + (0x0123_4567 × 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0xFF79_5E36_C94E_4629
= 0xFF79_5E37_C94E_4628

結果の上位ワード 0xFF79_5E37 が HI レジスタに格納され、下位ワード 0xC94E_4628 が LO レジスタと r4 に格納されます。

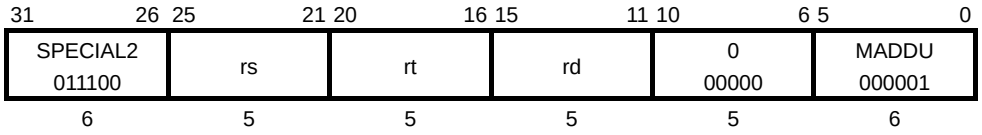
MADDU (*rd*,) *rs*, *rt*

Multiply and Add Unsigned

動作

HI ⇐ (HI ‖ LO) + (*rs* × *rt*) の上位ワード
LO ⇐ (HI ‖ LO) + (*rs* × *rt*) の下位ワード
rd ⇐ (HI ‖ LO) + (*rs* × *rt*) の下位ワード

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を乗算し、その積を HI・LO レジスタに格納されているダブルワードの値に加算します。*rs* と *rt* を符号なし整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。デスティネーションレジスタ *rd* が指定されている場合、結果の下位ワードを *rd* にも格納します。

rd を省略すると、デフォルトで r0 になり、その結果、汎用レジスタに結果は格納されません。

いかなる場合でも、整数オーバフロー例外は発生しません。

例外

なし

使用例

HI レジスタには 0x0000_0000 が、LO レジスタには 0xFFFF_FFFF が格納されていて、汎用レジスタ r2 には 0x0123_4567 が、r3 には 0x89AB_CDEF が格納されているとします。

```
MADDU r4,r2,r3
```

このとき、上記の命令は、以下の演算を実行します。

0x0000_0000_FFFF_FFFF + (0x0123_4567 × 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0x009C_A39D_C94E_4629
= 0x009C_A39E_C94E_4628

結果の上位ワード 0x009C_A39E が HI レジスタに格納され、下位ワード 0xC94E_4628 が LO レジスタと r4 に格納されます。

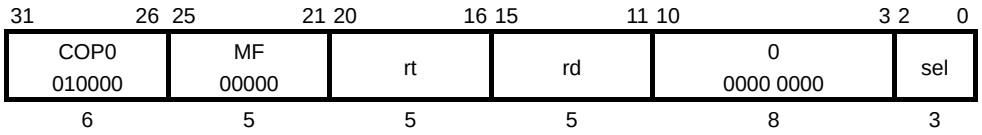
MFC0 *rt*, *rd*

Move From Coprocessor 0

動作

rt ⇐ CP0 のコプロセッサレジスタ *rd*

コード



説明

CP0 レジスタ *rd* の内容を汎用レジスタ *rt* にロードします。

例外

コプロセッサ使用不可例外

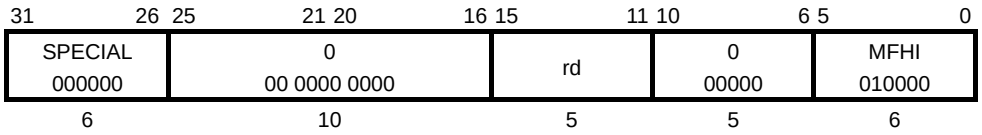
MFHI *rd*

Move From HI

動作

rd ⇐ HI

コード



説明

HI レジスタの内容を汎用レジスタ *rd* にロードします。

例外

なし

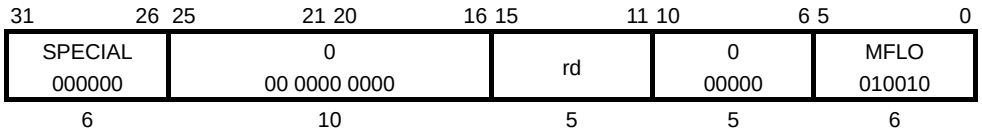
MFLO *rd*

Move From LO

動作

rd ⇐ LO

コード



説明

LO レジスタの内容を汎用レジスタ *rd* にロードします。

例外

なし

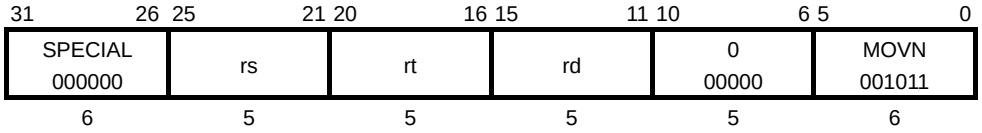
MOVN *rd*, *rs*, *rt*

Move Conditional on Not Zero

動作

if $rt \neq 0$ then $rd \Leftarrow rs$

コード



説明

汎用レジスタ rt の内容が 0 でなければ、汎用レジスタ rs の内容が汎用レジスタ rd へ転送されます。

例外

なし

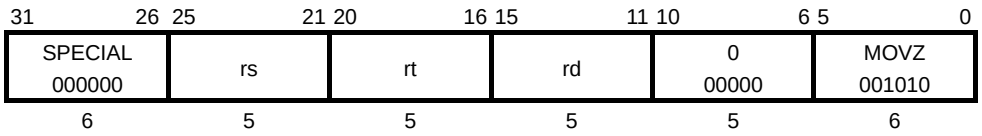
MOVZ *rd*, *rs*, *rt*

Move Conditional on Zero

動作

if *rt* = 0 then *rd* ⇐ *rs*

コード



説明

汎用レジスタ rt の内容が 0 であれば、汎用レジスタ rs の内容が汎用レジスタ rd へ転送されます。

例外

なし

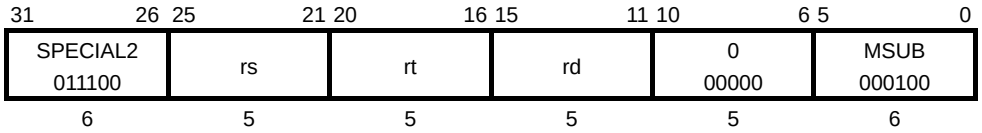
MSUB (*rd*), *rs*, *rt*

Multiply and Subtract

動作

HI ⇐ (HI ‖ LO) − (*rs* × *rt*) の上位ワード
LO ⇐ (HI ‖ LO) − (*rs* × *rt*) の下位ワード
rd ⇐ (HI ‖ LO) − (*rs* × *rt*) の下位ワード

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を乗算し、その積を HI・LO レジスタに格納されているダブルワードの値から減算します。*rs* と *rt* を符号付き整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。デスティネーションレジスタ *rd* が指定されている場合、結果の下位ワードを *rd* にも格納します。

rd を省略すると、デフォルトは r0 になり、その結果汎用レジスタに結果は格納されません。

いかなる場合でも、整数オーバフロー例外は発生しません。

例外

なし

使用例

HI レジスタには 0xFF79_5E37 が、LO レジスタには 0xC94E_4628 が格納されていて、汎用レジスタ r2 には 0x0123_4567 が、r3 に 0x89AB_CDEF が格納されているとします。

```
MSUB r2,r3
```

このとき、上記の命令は、以下の演算を実行します。

0xFF79_5E37_C94E_4628 − (0x0123_4567 × 0x89AB_CDEF)
= 0xFF79_5E37_C94E_4628 − 0xFF79_5E36_C94E_4629
= 0x0000_0000_FFFF_FFFF

結果の上位ワード 0x0000_0000 が HI レジスタに格納され、下位ワード 0xFFFF_FFFF が LO レジスタに格納されます。

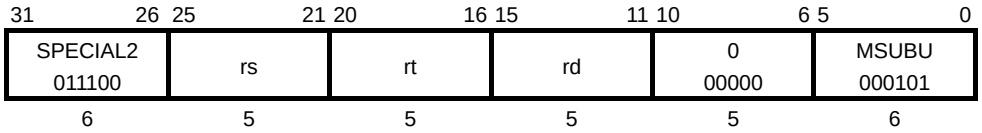
MSUBU (*rd*), *rs*, *rt*

Multiply and Subtract Unsigned

動作

HI ⇐ (HI ‖ LO) − (*rs* × *rt*) の上位ワード
LO ⇐ (HI ‖ LO) − (*rs* × *rt*) の下位ワード
rd ⇐ (HI ‖ LO) − (*rs* × *rt*) の下位ワード

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を乗算し、その積を HI・LO レジスタに格納されているダブルワードの値から減算します。*rs* と *rt* を符号なし整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。デスティネーションレジスタ *rd* が指定されている場合、結果の下位ワードを *rd* にも格納します。

rd を省略すると、デフォルトは r0 になり、その結果汎用レジスタに結果は格納されません。

いかなる場合でも、整数オーバフロー例外は発生しません。

例外

なし

使用例

HI レジスタには 0x009C_A39E が、LO レジスタには 0xC94E_4628 が格納されていて、汎用レジスタ r2 には 0x0123_4567 が、r3 には 0x89AB_CDEF が格納されているとします。

```
MSUBU r2,r3
```

このとき、上記の命令は、以下の演算を実行します。

0x009C_A39E_C94E_4628 − (0x0123_4567 × 0x89AB_CDEF)
= 0x009C_A39E_C94E_4628 − 0x009C_A39D_C94E_4629
= 0x0000_0000_FFFF_FFFF

結果の上位ワード 0x0000_00000 が HI レジスタに格納され、下位ワード 0xFFFF_FFFF が LO レジスタに格納されます。

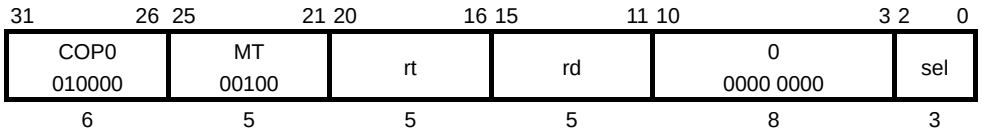
MTC0 *rt*, *rd*

Move To Coprocessor 0

動作

CP0 のコプロセッサレジスタ *rd* ⇐ *rt*

コード



説明

汎用レジスタ *rt* の内容を CP0 レジスタ *rd* にロードします。

ERET 命令の直前または 2 命令前に、MTC0 命令により Status、EPC または ErrorEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、動作は不定になります。

同様に、DERET 命令の直前または 2 命令前に MTC0 命令により DEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、動作は不定になります。

MTC0 命令により、仮想アドレス変換システムの状態が変わることがあるので、直前や直後のロード・ストア命令の動作は確定しません。

MTC0 命令により、SSCR レジスタを書き換えた場合、後続に 2 つの NOP 命令をかならず置いてください。

例外

コプロセッサ使用不可例外

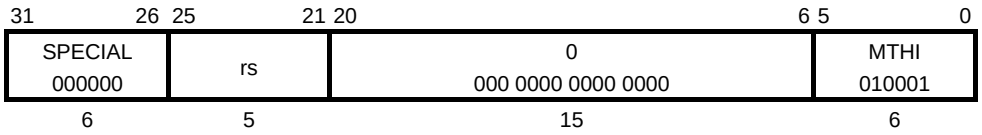
MTHI *rs*

Move To HI

動作

HI ⇐ *rs*

コード



説明

汎用レジスタ *rs* の内容を HI レジスタにロードします。

例外

なし

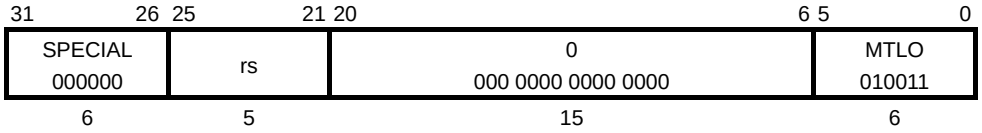
MTLO *rs*

Move To LO

動作

LO ⇐ *rs*

コード



説明

汎用レジスタ *rs* の内容を LO レジスタにロードします。

例外

なし

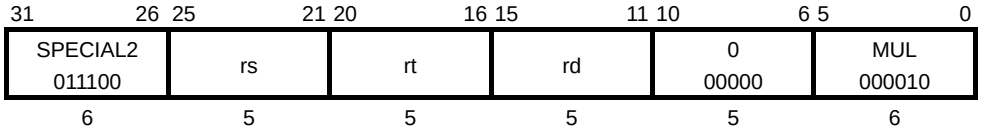
MUL *rd*, *rs*, *rt*

Multiply

動作

rd ⇐ (*rs* × *rt*) の下位ワード

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を乗算します。*rs* と *rt* は符号付き整数として扱います。結果の下位ワードを *rd* に格納します。HI・LO レジスタの内容は、不定となります。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

汎用レジスタ r2 に 0x0123_4567 が、r3 に 0x89AB_CDEF が格納されているとします。

```
MUL r4,r2,r3
```

このとき、上記の命令は、以下の演算を実行します。

(0x0123_4567 × 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629

結果の下位ワード 0xC94E_4629 が r4 に格納されます。

MULT (*rd*,) *rs*, *rt*

Multiply

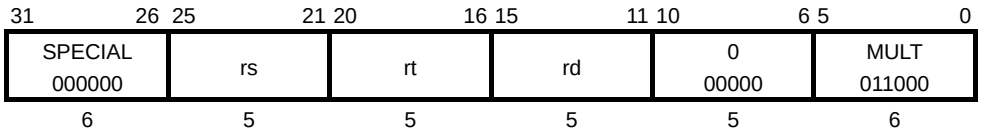
動作

HI ⇐ (*rs* × *rt*) の上位ワード;

LO ⇐ (*rs* × *rt*) の下位ワード;

rd ⇐ (*rs* × *rt*) の下位ワード

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を乗算します。*rs* と *rt* は符号付き整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。デスティネーションレジスタ *rd* が指定されている場合、結果の下位ワードを *rd* にも格納します。

rd を省略すると、デフォルトは r0 になり、その結果、汎用レジスタに結果は格納されません。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

汎用レジスタ r2 に 0x0123_4567 が、r3 に 0x89AB_CDEF が格納されているとします。

```
MULT r4,r2,r3
```

このとき、上記の命令は、以下の演算を実行します。

(0x0123_4567 × 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629

結果の上位ワード 0xFF79_5E36 が HI レジスタに格納され、下位ワード 0xC94E_4629 が LO レジスタと r4 に格納されます。

MULTU (*rd*,) *rs*, *rt*

Multiply Unsigned

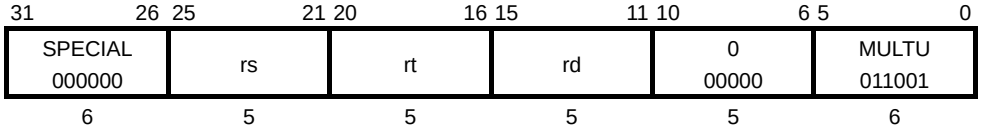
動作

HI ⇐ (*rs* × *rt*) の上位ワード;

LO ⇐ (*rs* × *rt*) の下位ワード;

rd ⇐ (*rs* × *rt*) の下位ワード

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を乗算します。*rs* と *rt* は符号なし整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。デスティネーションレジスタ *rd* が指定されている場合、結果の下位ワードを *rd* にも格納します。

rd を省略すると、デフォルトは r0 となり、その結果、汎用レジスタに結果は格納されません。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

汎用レジスタ r2 に 0x0123_4567 が、r3 に 0x89AB_CDEF が格納されているとします。

```
MULTU r4,r2,r3
```

このとき、上記の命令は、以下の演算を実行します。

(0x0123_4567 × 0x89AB_CDEF)
= 0x009C_A39D_C94E_4629

結果の上位ワード 0x009C_A39D が HI レジスタに格納され、下位ワード 0xC94E_4629 が LO レジスタと r4 に格納されます。

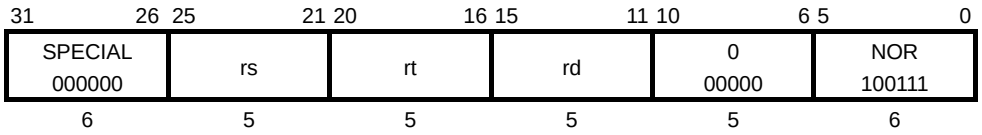
NOR *rd*, *rs*, *rt*

NOR

動作

rd ⇐ *rs* NOR *rt*

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容の否定論理和 (NOR) をとり、結果を汎用レジスタ *rd* に格納します。

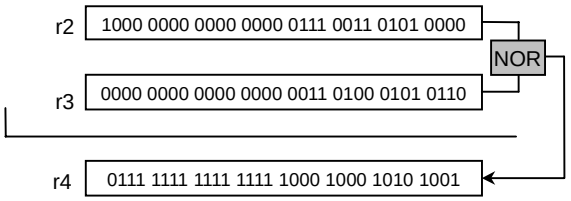
例外

なし

使用例

レジスタ r2 の値が 0x8000_7350 で、r3 の値が 0x0000_3456 のとき、以下の命令を実行すると、図示したように、r2 と r3 の否定論理和 0x7FFF_88A9 がレジスタ r4 に格納されます。

```
NOR r4,r2,r3
```



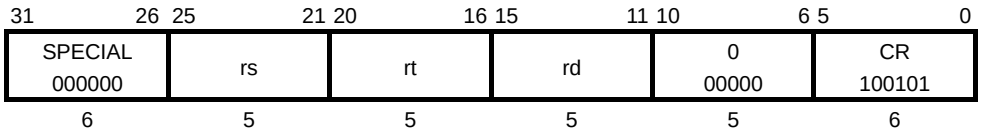
OR *rd*, *rs*, *rt*

OR

動作

rd ⇐ *rs* OR *rt*

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容の論理和 (OR) をとり、結果を汎用レジスタ *rd* に格納します。

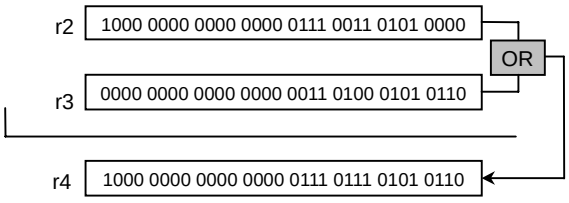
例外

なし

使用例

レジスタ r2 の値が 0x8000_7350 で、r3 の値が 0x0000_3456 のとき、以下の命令を実行すると、図示したように r2 と r3 の論理和 0x8000_7756 がレジスタ r4 に格納されます。

```
OR r4,r2,r3
```



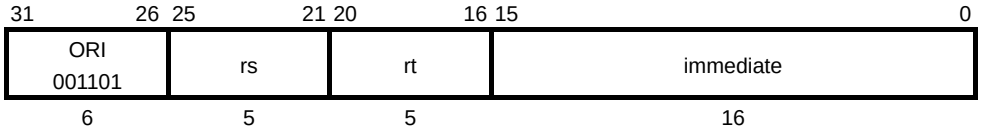
ORI *rt*, *rs*, *immediate*

OR Immediate

動作

$rt \Leftarrow rs$ OR $(0^{16} \parallel immediate_{15..0})$

コード



説明

16 ビット *immediate* をゼロ拡張した値と汎用レジスタ *rs* の内容との論理和 (OR) をとり、結果を汎用レジスタ *rt* に格納します。

immediate は 16 ビットです。これを超える値を扱いたい場合は、いったん汎用レジスタに格納してから、OR 命令を使います (「3.3.2　32 ビットの定数」を参照)。

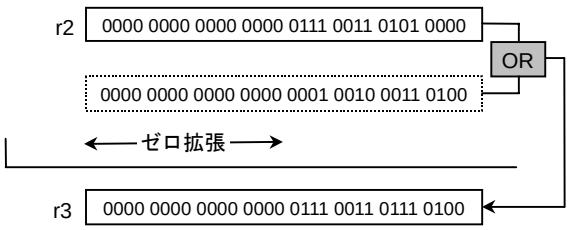
例外

なし

使用例

レジスタ r2 の値が 0x0000_7350 の場合、以下の命令を実行すると、図示したように、0x0000_7350 と 0x0000_1234 の論理和 0x0000_7374 がレジスタ r3 に格納されます。

```
ORI r3,r2,0x1234
```



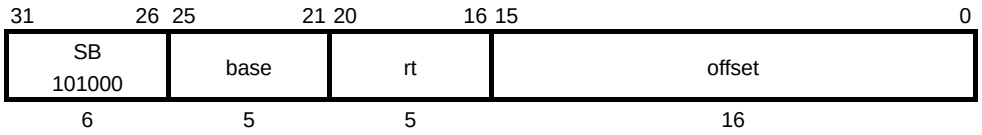
SB *rt*, *offset* (*base*)

Store Byte

動作

rt ⇒ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張した値と汎用レジスタ *base* の内容を加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *rt* の最下位バイトを、このアドレスにストアします。

rt の上位 3 バイトは無視されるため、符号付きと符号なしの区別がなくなります。

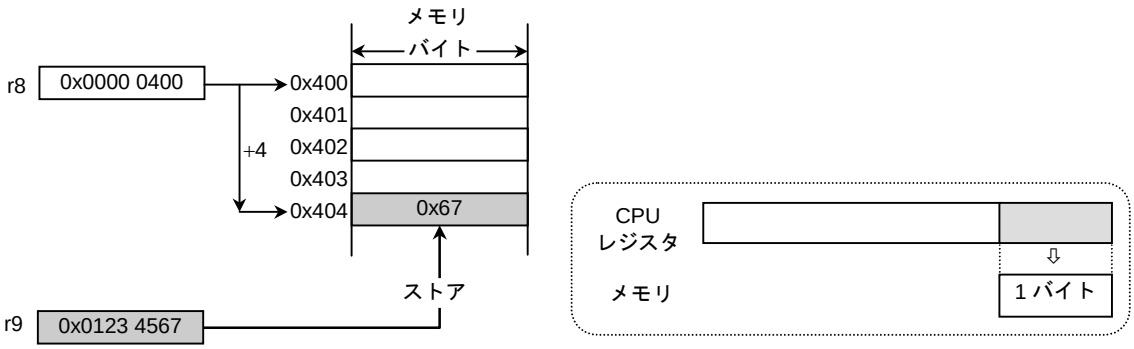
例外

アドレスエラー例外

使用例

レジスタ r8 の値が 0x0000_0400 で、r9 の値が 0x0123_4567 の場合、以下の命令を実行すると、0x67 がアドレス 0x404 に格納されます。

```
SB r9,4(r8)
```



SDBBP *code*

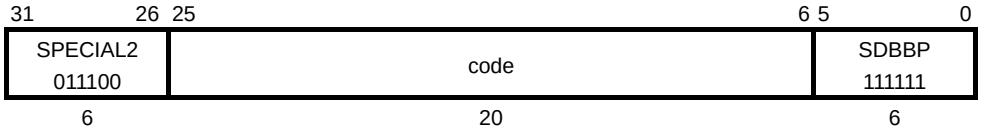
EJTAG

Software Debug Breakpoint Exception

動作

ソフトウエアデバッグブレークポイント例外

コード



説明

デバッグブレークポイントが発生し、無条件に制御を例外ハンドラに移します。

SDBBP 命令の *code* フィールドは、例外ハンドラに情報を渡すために使用できます。例外ハンドラが code フィールドを取り出すには、命令を含むメモリワードの内容をデータとしてロードする必要があります。詳細は「9.3　デバッグ例外」を参照してください。

SDBBP 命令は、開発ツールで使用しますので、ユーザープログラム中では記述しないでください。EJTAG を実装しない製品で実行すると予約命令例外が発生します。

例外

デバッグブレークポイント例外

予約命令例外

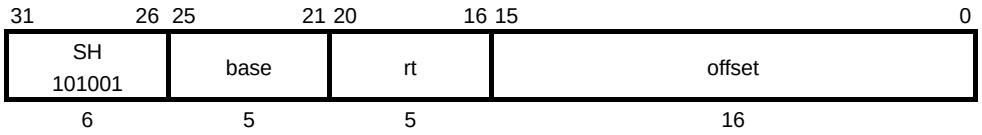
SH *rt*, *offset* (*base*)

Store Halfword

動作

rt ⇒ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張して、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *rt* の下位ハーフワードをこのアドレスにストアします。

rt の上位ハーフワードは無視されるので、符号付き、符号なしの区別はありません。

実効アドレスの最下位ビットが 0 でない (実効アドレスがハーフワード境界でない) 場合、アドレスエラー例外が発生します。

例外

アドレスエラー例外

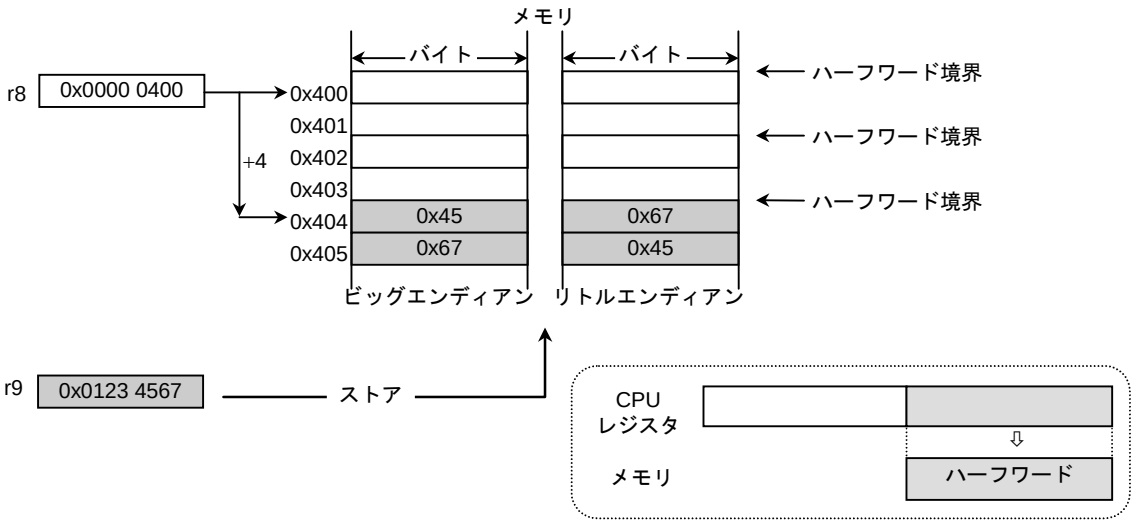
使用例

レジスタ r8 の値が 0x0000_0400 で、r9 の値が 0x0123_4567 の場合、以下の命令を実行すると、ビッグエンディアンのときは、アドレス 0x404 に 0x45 が、アドレス 0x405 に 0x67 がストアされます。リトルエンディアンのときは、アドレス 0x404 に 0x67 が、アドレス 0x405 に 0x45 がストアされます。

```
SH r9,4(r8)
```

また、以下の命令を実行すると、0x403 はハーフワード境界でないので、アドレスエラー例外が発生します。

```
SH r9,3(r8)
```


メモリ
バイト
バイト
ハーフワード境界
ハーフワード境界
ハーフワード境界
r8
0x0000 0400
+4
0x400
0x401
0x402
0x403
0x404
0x405
0x45
0x67
0x67
0x45
ビッグエンディアン
リトルエンディアン
r9
0x0123 4567
ストア
CPU
レジスタ
メモリ
ハーフワード

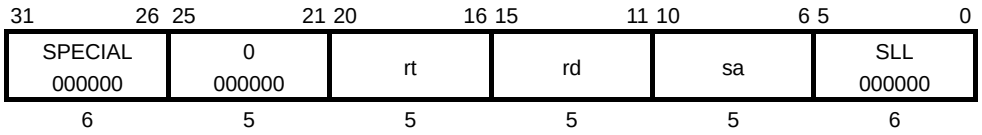
SLL *rd*, *rt*, *sa*

Shift Left Logical

動作

rd ⇐ *rt* << *sa*

コード



説明

汎用レジスタ *rt* の 32 ビットの内容を *sa* ビット左へシフトし、右端の空いたビットを 0 で埋め、結果を汎用レジスタ *rd* に格納します。

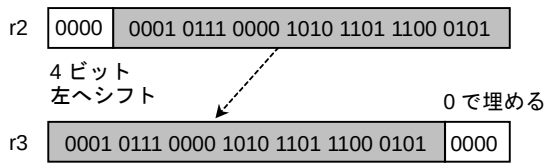
例外

なし

使用例

レジスタ r2 の内容が 0x2170_ADC5 の場合、以下の命令を実行すると、レジスタ r3 に 0x170A_DC50 が格納されます。

```
SLL r3,r2,4
```



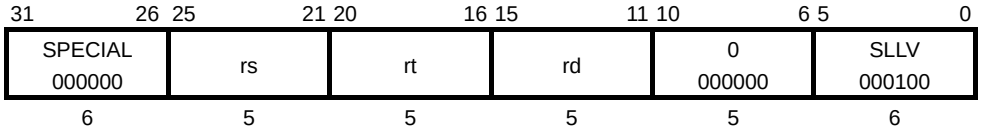
SLLV *rd*, *rt*, *rs*

Shift Left Logical Variable

動作

rd ⇐ *rt* << *rs* の下位 5 ビット

コード



説明

汎用レジスタ *rt* の 32 ビットの内容を、汎用レジスタ *rs* の下位 5 ビットで指定されたビット数、左にシフトし、右端の空いたビットをゼロで埋めます。結果を汎用レジスタ *rd* に格納します。

例外

なし

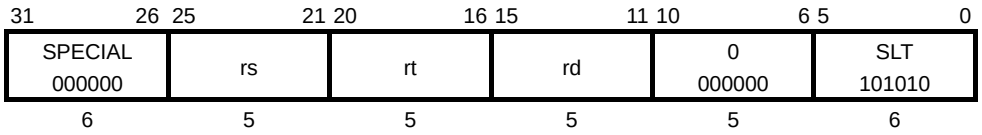
SLT *rd*, *rs*, *rt*

Set On Less Than

動作

if *rs* < *rt* then *rd* ⇐ 1; else *rd* ⇐ 0

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を符号付き整数として比較します。*rs* が *rt* より小さい場合は、汎用レジスタ *rd* は 1 を、そうでない場合は、*rd* は 0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

例外

なし

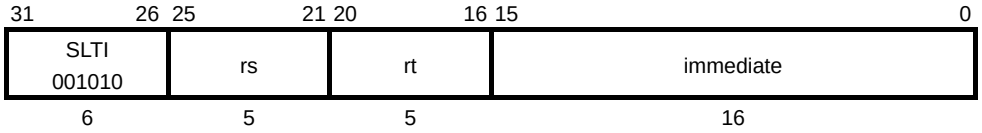
SLTI *rt*, *rs*, *immediate*

Set On Less Than Immediate

動作

if *rs* < ((*immediate*$_{15}$)^{16} || *immediate*$_{15..0}$) then *rt* ⇐ 1; else *rt* ⇐ 0

コード



説明

16 ビット *immediate* を符号拡張した値と汎用レジスタ *rs* の内容を符号付き整数として比較します。*rs* が *immediate* より小さい場合は、汎用レジスタ *rt* に 1 を、そうでない場合は、0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

immediate は 16 ビットです。したがって *immediate* で指定できる数値範囲は、-32768 ~ +32767 です。この範囲外の値を扱いたい場合は、いったん汎用レジスタに格納してから、SLT 命令を使います(「3.3.2　32 ビットの定数」を参照)。

例外

なし

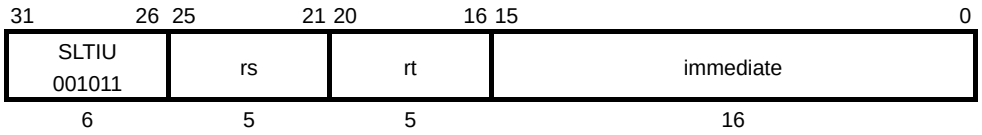
SLTIU *rt*, *rs*, *immediate*

Set On Less Than Immediate Unsigned

動作

if (0 ‖ *rs*) < ((*immediate*$_{15}$)^{17} ‖ *immediate*$_{15..0}$) then *rt* ⇐ 1; else *rt* ⇐ 0

コード



説明

16 ビット *immediate* を符号拡張した値と汎用レジスタ *rs* の内容を符号なし整数として比較します。*rs* が *immediate* より小さい場合は、汎用レジスタ *rt* を 1 に、そうでない場合は、0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

immediate は 16 ビットです。この範囲外の場合は、いったん汎用レジスタに格納してから、SLTU 命令を使います (「3.3.2　32 ビットの定数」を参照)。

例外

なし

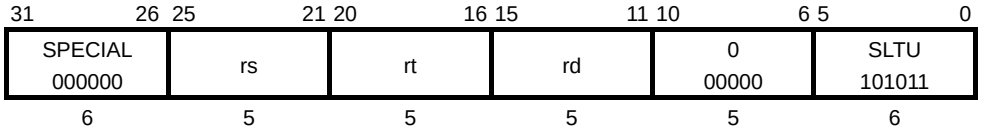
SLTU *rd*, *rs*, *rt*

Set On Less Than Unsigned

動作

if (0 ‖ *rs*) < (0 ‖ *rt*) then *rd* ⇐ 1; else *rd* ⇐ 0

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容を符号なし整数として比較します。*rs* が *rt* より小さい場合、汎用レジスタ *rd* に 1 を、そうでない場合は、0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

例外

なし

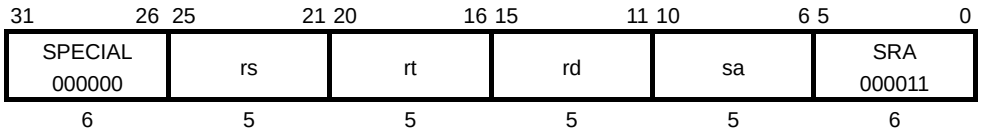
SRA *rd*, *rt*, *sa*

Shift Right Arithmetic

Operand

rd ⇐ *rt* >> *sa*

コード



説明

汎用レジスタ *rt* の 32 ビットの内容を *sa* ビット右へシフトし､左端の空いたビットを符号ビットで埋めます。結果を汎用レジスタ *rd* に格納します。

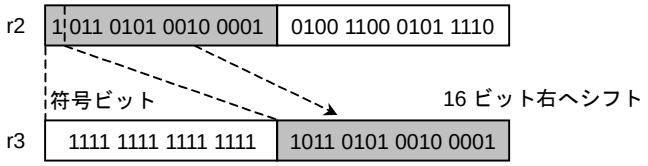
例外

なし

使用例

レジスタ r2 の値が 0xB521_4C5E の場合、以下の命令を実行すると、レジスタ r3 に 0xFFFF_B521 が格納されます。

```
SRA r3,r2,16
```



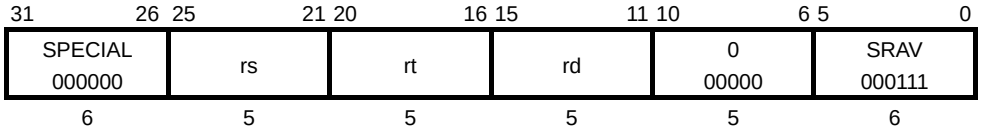
SRAV *rd*, *rt*, *rs*

Shift Right Arithmetic Variable

動作

rd ⇐ *rt* >> *rs* の下位 5 ビット

コード



説明

汎用レジスタ *rt* の 32 ビットの内容を、汎用レジスタ *rs* の下位 5 ビットで指定されたビット数、右にシフトし、左端の空いたビットを符号ビットで埋めます。結果を汎用レジスタ *rd* に格納します。

例外

なし

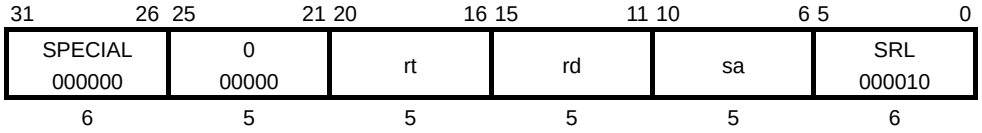
SRL *rd*, *rt*, *sa*

Shift Right Logical

動作

rd ⇐ *rt* >> *sa*

コード



説明

汎用レジスタ *rt* の 32 ビットの内容を *sa* ビット右へシフトし、左端の空いたビットを 0 で埋めます。結果を汎用レジスタ *rd* に格納します。

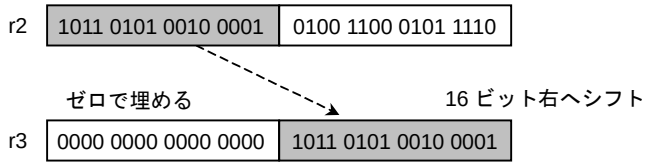
例外

なし

使用例

レジスタ r2 の内容が 0xB521_4C5E の場合、以下の命令を実行すると、レジスタ r3 に 0x0000_B521 が格納されます。

```
SRL r3,r2,16
```



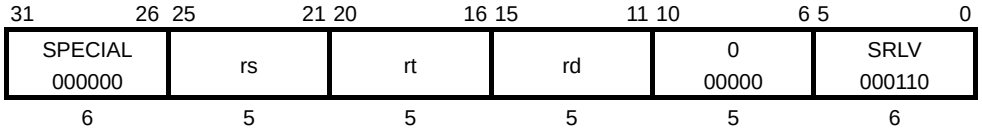
SRLV *rd*, *rt*, *rs*

Shift Right Logical Variable

動作

rd ⇐ *rt* >> *rs* の下位 5 ビット

コード



説明

汎用レジスタ *rt* の 32 ビットの内容を、汎用レジスタ *rs* の下位 5 ビットで指定されたビット数、右にシフトし、左端の空いたビットを 0 で埋めます。結果を汎用レジスタ *rd* に格納します。

例外

なし

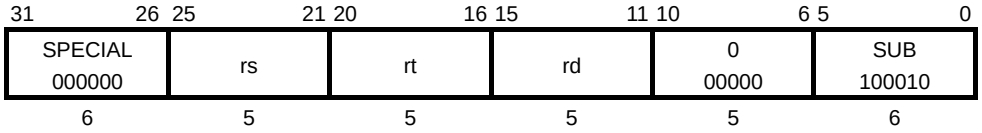
SUB *rd*, *rs*, *rt*

Subtract

動作

rd ⇐ *rs* – *rt*

コード



説明

汎用レジスタ *rs* の内容から汎用レジスタ *rt* の内容を減算し、減算結果を汎用レジスタ *rd* に格納します。*rs* と *rt* は符号付き整数として扱われます。

2 の補数のオーバフローが発生した場合、整数オーバフロー例外が発生します。オーバフローは、c ⇐ a – b の計算において a と b の符号が異なり、かつ a と c の符号が異なる場合に発生します。整数オーバフロー例外が発生すると、デスティネーションレジスタ (*rd*) の内容は変更されません。

例外

整数オーバフロー例外

使用例

1. レジスタ r2 の値が 0x7654_3210 で、レジスタ r3 の値が 0x5000_0000 のとき、以下の命令を実行すると、減数結果 (0x2654_3210) が r4 に格納されます。

```
SUB r4,r2,r3
```

2. レジスタ r2 の値が 0x7FFF_FFFF で、レジスタ r3 の値が 0x8FFF_FFFF のとき、r2 から r3 を減算すると、結果は 0xF000_0000 になります。つまり、r2 と r3 の符号が異なり、r2 の符号と減算結果の符号が異なっているため 2 の補数のオーバフローが発生します。

```
SUB r4,r2,r3
```

このとき、上記の命令を実行すると、整数オーバフロー例外が発生します。レジスタ r4 は変更されません。

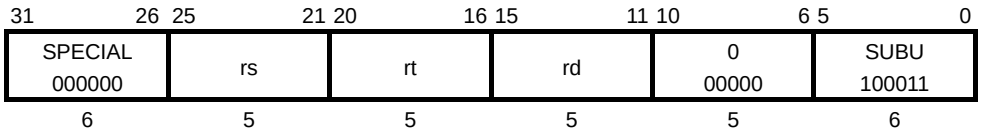
SUBU *rd*, *rs*, *rt*

Subtract Unsigned

動作

rd ⇐ *rs* – *rt*

コード



説明

汎用レジスタ *rs* の内容から汎用レジスタ *rt* の内容を減算し、減算結果を汎用ジレスタ *rd* に格納します。

SUB 命令と SUBU 命令の唯一の違いは、SUBU 命令では整数オーバフロー例外が発生しないという点だけです。

例外

なし

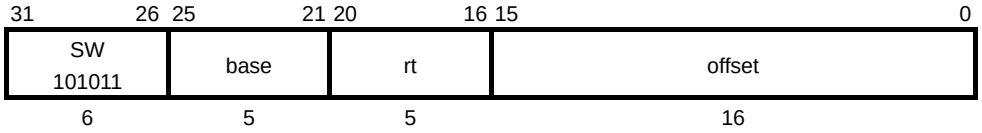
SW *rt*, *offset* (*base*)

Store Word

動作

rt ⇒ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張して、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *rt* の内容をこのアドレスにストアします。

実効アドレスの下位 2 ビットが 0 でない (実効アドレスが境界でない) 場合、アドレスエラー例外が発生します。

例外

アドレスエラー例外

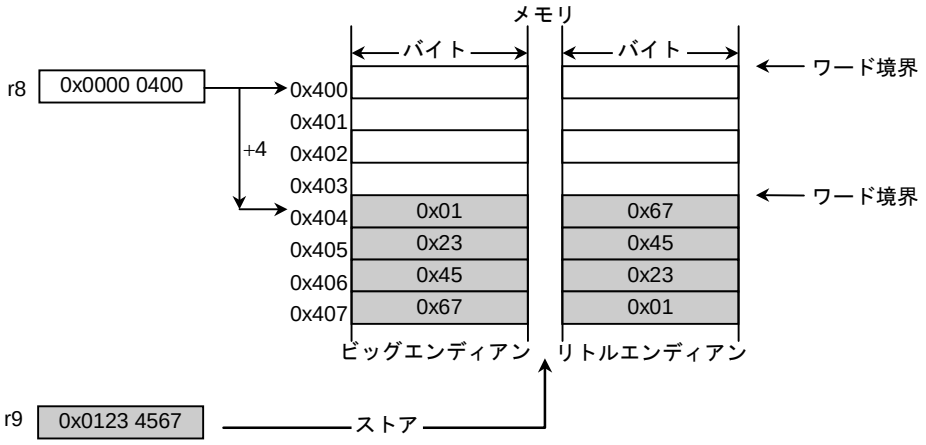
使用例

レジスタ r8 の値が 0x0000_0400 で、レジスタ r9 の値が 0x0123_4567 の場合、以下の命令を実行すると、ビッグエンディアンのときは、アドレス 0x404~0x407 に 0x01、0x23、0x45、0x67 がストアされます。リトルエンディアンのときは、アドレス 0x404~0x407 に 0x67、0x45、0x23、0x01 がストアされます。

```
SW r9,4(r8)
```

また、以下の命令を実行すると、0x405 はワード境界でないので、アドレスエラー例外が発生します。

```
SW r9,5(r8)
```


メモリ
バイト
バイト
ワード境界
r8
0x0000 0400
0x400
0x401
0x402
0x403
+4
ワード境界
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
0x67
0x45
0x23
0x01
ビッグエンディアン
リトルエンディアン
r9
0x0123 4567
ストア

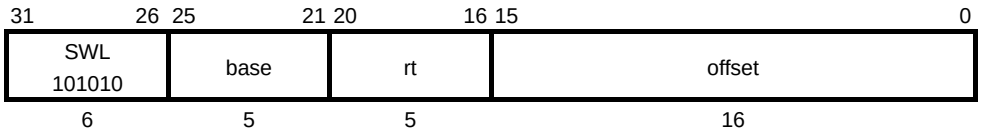
SWL *rt*, *offset* (*base*)

Store Word Left

動作

rt ⇒ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張し、汎用レジスタ *base* の内容に加算することにより実効アドレス (EA) を生成します。汎用レジスタ *rt* の左部分を、ワード境界をまたがったワード位置の上位にストアします。

実効アドレスがワード境界に位置していないことによるアドレスエラー例外は発生しません。

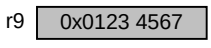
ワード境界をまたがってレジスタ中のワードデータをメモリにストアするとき、SWL 命令と SWR 命令を組み合わせて使います。

例外

アドレスエラー例外

使用例

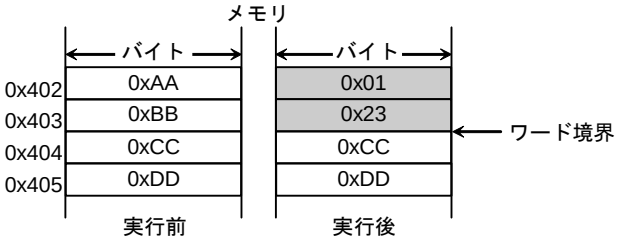
レジスタ r8 の値が 0x0000_0400 で、レジスタ r9 の値が 00123_4567 であるとします。



- ビッグエンディアンのとき

```
SWL r9,2(r8)
```

上記の命令は、レジスタ r9 の左部分をアドレス 0x0402 から上位アドレス方向へ、ワード境界に達するまでストアします。その結果を以下に示します。

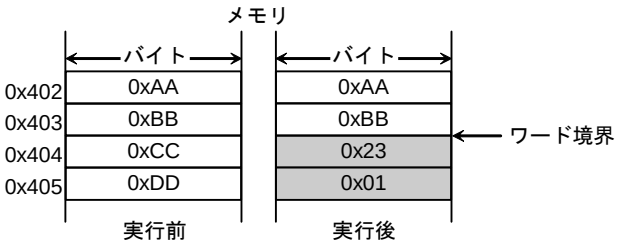


(a) ビッグエンディアン

- リトルエンディアンのとき

```
SWL r9,5(r8)
```

上記の命令は、レジスタ r9 の左部分をアドレス 0x0405 から下位アドレス方向へ、ワード境界に達するまでストアします。その結果を以下に示します。



(b) リトルエンディアン

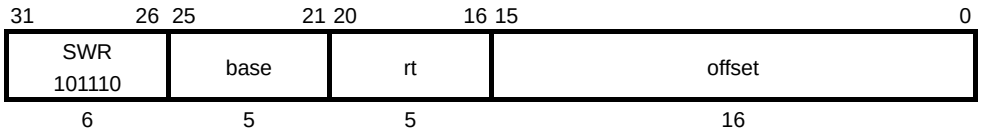
SWR *rt*, *offset* (*base*)

Store Word Right

動作

rt ⇒ {sign-extend(*offset*) + (*base*)}

コード



説明

16 ビット *offset* を符号拡張し、汎用レジスタ *base* の内容に加算することにより実効アドレス (EA) を生成します。汎用レジスタ *rt* の右部分を、ワード境界をまたがったワード位置の下位にストアします。

ワード境界に位置していないことによるアドレスエラー例外は発生しません。

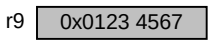
ワード境界をまたがってレジスタ中のワードデータをメモリにストアするときに、SWL 命令と SWR 命令を組み合わせて使います。

例外

アドレスエラー例外

使用例

レジスタ r9 の値が 0x123_4567 とします。

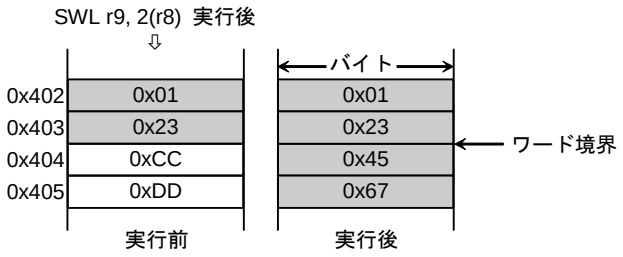


前ページの SWL 命令で説明したように、汎用レジスタの左部分の内容をストアした後に、右部分をストアする方法を以下に示します。

- ビッグエンディアン

```
SWR r9,5(r8)
```

上記の命令は、レジスタ r9 の右部分をアドレス 0x0405 から下位アドレス方向へ、ワード境界に達するまでストアします。その結果を以下に示します。

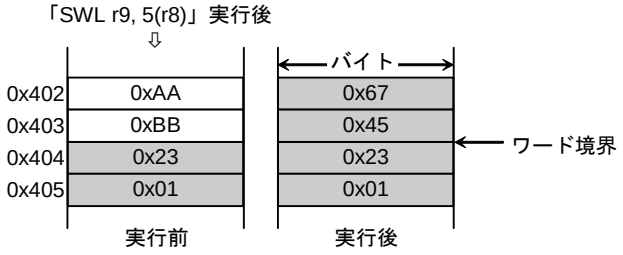


(a) ビッグエンディアン

- リトルエンディアン

```
SWR r9,2(r8)
```

上記の命令は、レジスタ r9 の右部分をアドレス 0x0402 から上位アドレス方向へ、ワード境界に達するまでストアします。その結果を以下に示します。



(b) リトルエンディアン

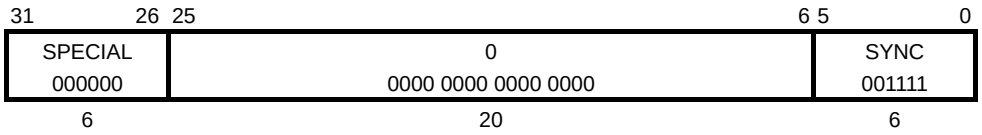
SYNC

Synchronize

動作

メモリ同期操作

コード



説明

SYNC 命令は、SYNC 命令の直前に実行したロード、ストアが完了するまで、命令パイプラインをインタロックし、後続の命令の実行を遅らせます。これにより、SYNC 命令の前の命令と後続の命令の実行順序を守ることができます。「5.2.4　SYNC 命令 (32 ビット ISA/16 ビット ISA)」を参照してください。

ロード命令の後続命令がそのロード結果を使用しない場合、パイプラインをストールせず実行が継続されます。この機能をノンブロッキングロードといいます。パイプラインの他の部分は、データと依存関係のない命令の実行を継続します。

例外

なし

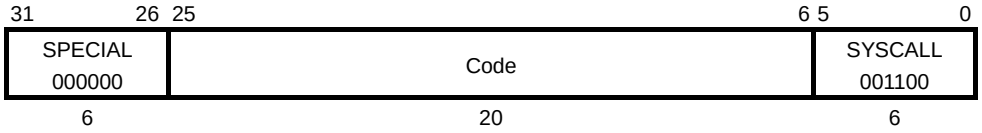
SYSCALL *code*

System Call

動作

システムコール例外

コード



説明

SYSCALL 命令を実行すると、システムコール例外が発生し、無条件に制御を例外ハンドラに渡します。

SYSCALL 命令の *code* フィールドを使用して、例外ハンドラにパラメータを送ることができます。これらのビットを調べるには、EPC レジスタが示す命令の内容をロードします。システムコール例外の詳細は、「9.1.10　システムコール例外」を参照してください。

例外

システムコール例外

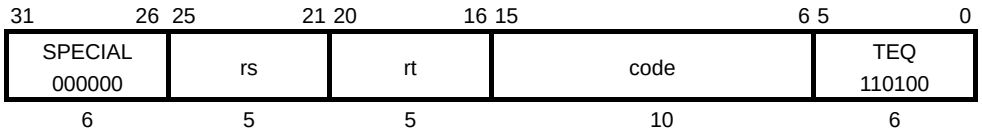
TEQ *rs*, *rt*, *code*

Trap If Equal

動作

if *rs* = *rt* then Trap Exception; else Next Instruction

コード



説明

汎用レジスタ *rt* の内容と汎用レジスタ *rs* の内容と比較します。汎用レジスタ *rt* の内容が汎用レジスタ *rs* の内容と等しい場合にトラップ例外が発生します。code フィールドは、ソフトウエアパラメータとして使用できますが、code フィールドを例外ハンドラが取り出す場合は、本命令を含むメモリワードの内容をロードしてください。

例外

トラップ例外

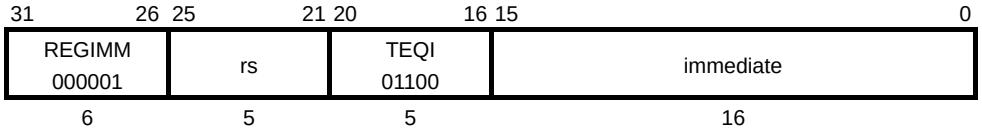
TEQI *rs*, *immediate*

Trap If Equal Immediate

動作

if *rs* = (*immediate*$_{15}$)^{16} ‖ *immediate*$_{15..0}$ then Trap Exception else Next Instruction

コード



説明

16 ビットの *immediate* は、符号拡張されてから汎用レジスタ *rs* の内容と比較されます。汎用レジスタ *rs* の内容が、符号拡張された *immediate* と等しい場合に、トラップ例外が発生します。

例外

トラップ例外

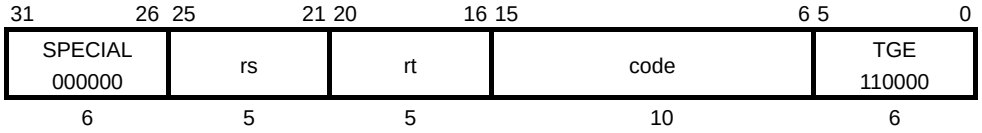
TGE *rs*, *rt*, *code*

Trap If Greater Than or Equal

動作

if *rs* ≧ *rt* then Trap Exception; else Next Instruction

コード



説明

汎用レジスタ *rt* の内容が、汎用レジスタ *rs* の内容と比較されます。どちらの値も符号つき整数として扱われて、汎用レジスタ *rs* の内容が汎用レジスタ *rt* の内容以上の場合に、トラップ例外が発生します。code フィールドは、ソフトウエアパラメータとして使用できますが、code フィールドを例外ハンドラが取り出す場合は、本命令を含むメモリワードの内容をロードしてください。

例外

トラップ例外

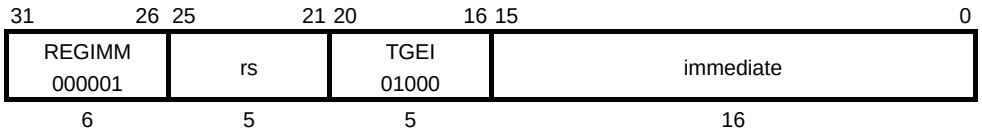
TGEI *rs*, *immediate*

Trap If Greater Than Or Equal Immediate

動作

if *rs* ≧ $(immediate_{15})^{16}$ ‖ $immediate_{15..0}$ then Trap Exception else Next Instruction

コード



説明

16 ビットの *immediate* は、符号拡張されてから汎用レジスタ *rs* の内容と比較されます。どちらの値も符号つき整数として扱われて、汎用レジスタ *rs* の内容が、符号拡張された *immediate* 以上の場合に、トラップ例外が発生します。

例外

トラップ例外

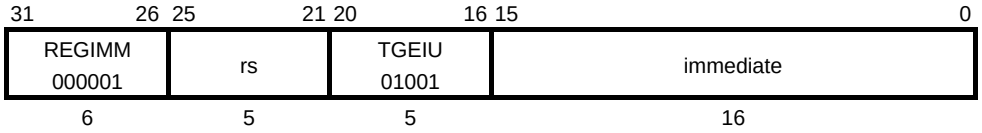
TGEIU *rs*, *immediate*

Trap If Greater Than Or Equal Immediate Unsigned

動作

if $(0 \parallel rs) \geqq 0 \parallel (immediate_{15})^{16} \parallel immediate_{15..0}$ then Trap Exception else Next Instruction

コード



説明

16 ビットの *immediate* は、符号拡張されてから汎用レジスタ *rs* の内容と比較されます。どちらの値も符号なし整数として扱われて、汎用レジスタ *rs* の内容が、符号拡張された *immediate* 以上の場合に、トラップ例外が発生します。

例外

トラップ例外

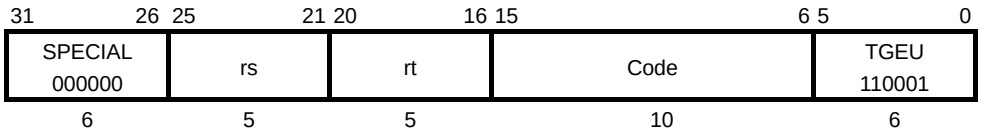
TGEU *rs*, *rt*, *code*

Trap If Greater Than or Equal Unsigned

動作

if (0 ‖ *rs*) ≧ (0 ‖ *rt*) then Trap Exception; else Next Instruction

コード



説明

汎用レジスタ *rt* の内容が、汎用レジスタ *rs* の内容と比較されます。どちらの値も符号なし整数として扱われて、汎用レジスタ *rs* の内容が汎用レジスタ *rt* の内容以上の場合に、トラップ例外が発生します。code フィールドは、ソフトウエアパラメータとして使用できますが、code フィールドを例外ハンドラが取り出す場合は、本命令を含むメモリワードの内容をロードしてください。

例外

トラップ例外

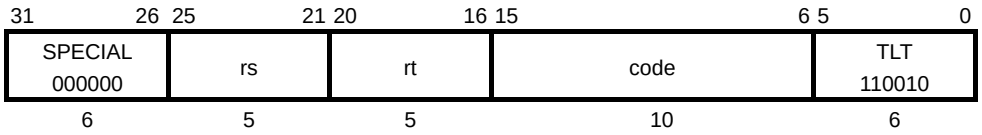
TLT *rs*, *rt*, *code*

Trap If Less Than

動作

if *rs* < *rt* then Trap Exception; else Next Instruction

コード



説明

汎用レジスタ *rt* の内容が、汎用レジスタ *rs* の内容と比較されます。どちらの値も符号つき整数として扱われて、汎用レジスタ *rs* の内容が汎用レジスタ *rt* の内容より小さい場合に、トラップ例外が発生します。code フィールドは、ソフトウエアパラメータとして使用できますが、code フィールドを例外ハンドラが取り出す場合は、本命令を含むメモリワードの内容をロードしてください。

例外

トラップ例外

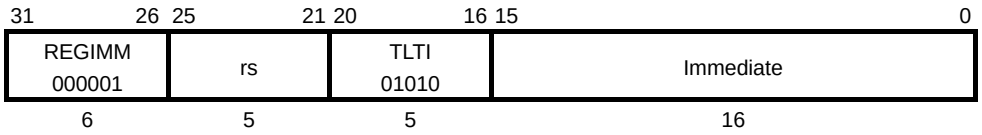
TLTI *rs*, *immediate*

Trap If Less Than Immediate

動作

if *rs* < (*immediate*$_{15}$)^{16} ‖ *immediate*$_{15..0}$ then Trap Exception else Next Instruction

コード



説明

16 ビットの *immediate* は、符号拡張されてから汎用レジスタ *rs* の内容と比較されます。どちらの値も符号つき整数として扱われて、汎用レジスタ *rs* の内容が、符号拡張された *immediate* より小さい場合に、トラップ例外が発生します。

例外

トラップ例外

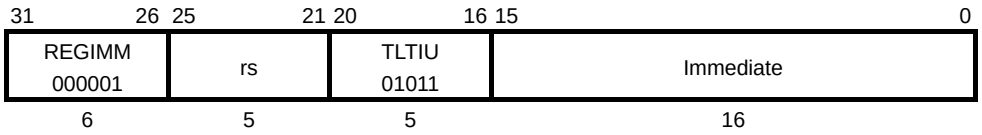
TLTIU *rs*, *immediate*

Trap If Less Than Immediate Unsigned

動作

if (0 ‖ *rs*) < 0 ‖ ($immediate_{15}$)^{16} ‖ $immediate_{15..0}$ then Trap Exception else Next Instruction

コード



説明

16 ビットの *immediate* は、符号拡張されてから汎用レジスタ *rs* の内容と比較されます。どちらの値も符号なし整数として扱われて、汎用レジスタ *rs* の内容が、符号拡張された *immediate* より小さい場合に、トラップ例外が発生します。

例外

トラップ例外

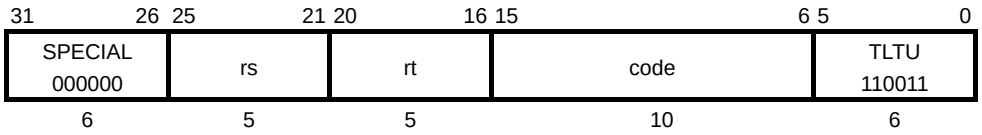
TLTU *rs*, *rt*, *code*

Trap If Less Than Unsigned

動作

if (0 || *rs*) < (0 || *rt*) then Trap Exception; else Next Instruction

コード



説明

汎用レジスタ *rt* の内容が、汎用レジスタ *rs* の内容と比較されます。どちらの値も符号なし整数として扱われて、汎用レジスタ *rs* の内容が汎用レジスタ *rt* より小さい場合に、トラップ例外が発生します。code フィールドは、ソフトウエアパラメータとして使用できますが、code フィールドを例外ハンドラが取り出す場合は、本命令を含むメモリワードの内容をロードしてください。

例外

トラップ例外

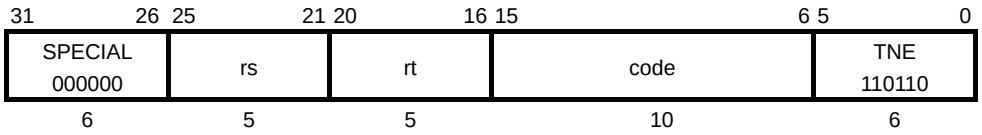
TNE *rs*, *rt*, *code*

Trap If Not Equal

動作

if *rs* ≠ *rt* then Trap Exception; else Next Instruction

コード



説明

汎用レジスタ *rt* の内容が、汎用レジスタ *rs* の内容と比較されます。汎用レジスタ *rt* の内容が汎用レジスタ *rs* の内容と等しくない場合にトラップ例外が発生します。code フィールドは、ソフトウエアパラメータとして使用できますが、code フィールドを例外ハンドラが取り出す場合は、本命令を含むメモリワードの内容をロードしてください。

例外

トラップ例外

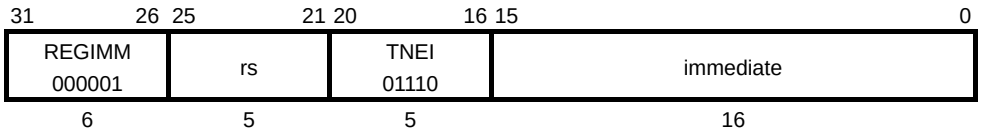
TNEI *rs*, *immediate*

Trap If Not Equal Immediate

動作

if $rs \neq (immediate_{15})^{16} \parallel immediate_{15..0}$ then Trap Exception else Next Instruction

コード



説明

16 ビットの *immediate* は、符号拡張されてから汎用レジスタ *rs* の内容と比較されます。汎用レジスタ *rs* の内容が、符号拡張された *immediate* と等しくない場合に、トラップ例外が発生します。

例外

トラップ例外

WAIT

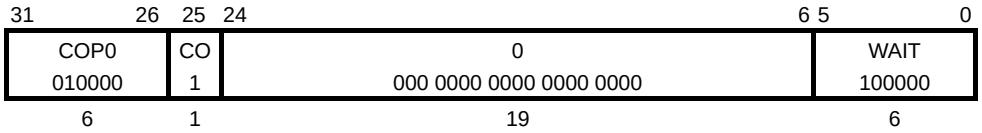
Enter Standby Mode

動作

if Status[RP] = 1 then DOZE モード

else HALT モード

コード



説明

WAIT 命令は、内部パイプラインを停止するために使用され、その結果 CPU の消費電力を低減させます。本命令を実行後、Status レジスタの RP ビット＝1 のときは DOZE モードに、RP ビット＝0 のときは HALT モードに遷移します。詳細は「第 10 章　低消費電力モード」を参照してください。

WAIT 命令を分岐命令やジャンプ命令の遅延スロットに置くことは禁止されています。

また、WAIT 命令の直前または 2 命令前にて、MTC0 命令を使用し Status レジスタに書きこみを行うことは禁止されています。Status レジスタに書きこみを行った場合動作は不定となります。

例外

コプロセッサ使用不可例外

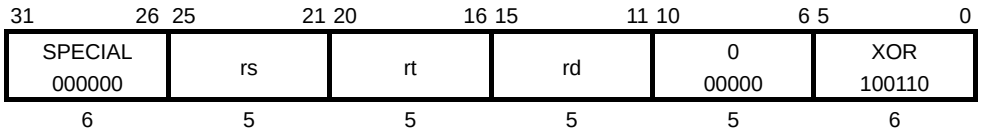
XOR *rd*, *rs*, *rt*

Exclusive OR

動作

rd ⇐ *rs* XOR *rt*

コード



説明

汎用レジスタ *rs* の内容と汎用レジスタ *rt* の内容との排他的論理和 (XOR) をとり、結果を汎用レジスタ *rd* に格納します。

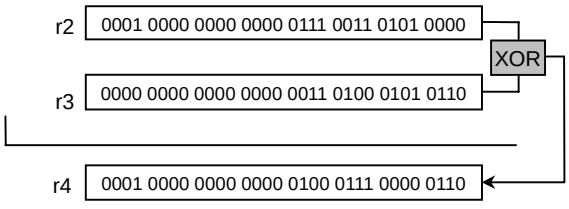
例外

なし

使用例

レジスタ r2 の値が 0x1000_7350 で、レジスタ r3 の値が 0x0000_3456 のとき、以下の命令を実行すると、結果の 0x1000_4706 がレジスタ r4 に格納されます。

```
XOR r4,r2,r3
```



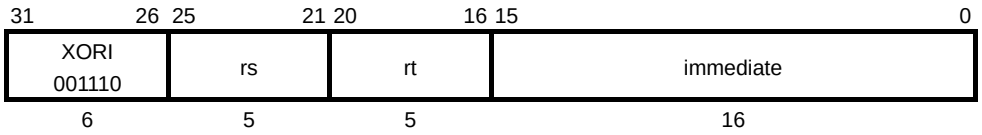
XORI *rt*, *rs*, *immediate*

Exclusive OR Immediate

動作

$rt \Leftarrow rs$ XOR $(0^{16} \parallel immediate_{15..0})$

コード



説明

16 ビット *immediate* をゼロ拡張し、汎用レジスタ *rs* の内容との排他的論理和 (XOR) をとり、結果を汎用レジスタ *rt* に格納します。

immediate は 16 ビットです。これを超える値はいったん汎用レジスタに格納し、XOR 命令を使う必要があります (「3.3.2　32 ビットの定数」を参照)。

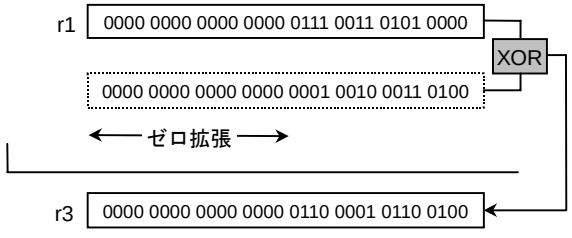
例外

なし

使用例

レジスタ r2 の値が 0x0000_7350 の場合、以下の命令を実行すると、結果の 0x0000_6164 がレジスタ r3 に格納されます。

```
XORI r3,r2,0x1234
```



付録B　16 ビット ISA の詳細

この章では、16 ビット ISA の命令について、シンタックス、命令形式、動作、命令の実行によって発生する可能性のある例外などを詳しく説明しています。命令はアルファベット順に記述されています。命令形式については「4.1　命令形式」を参照してください。

16 ビット ISA の命令では各レジスタフィールド (*rx*、*ry*、*rz*、*base*) は 3 ビットしかなく、32 本の汎用レジスタのうち、r2~r7、r16、r17 の 8 本のレジスタしか使用できません。16 ビット ISA におけるレジスタのコードを以下に示します。

コード	レジスタ	コード	レジスタ
000	r16	100	r4
001	r17	101	r5
010	r2	110	r6
011	r3	111	r7

ただし、命令によっては、r24 (t8)、r28 (gp)、r29 (sp)、r30 (fp)、r31 (ra) を使用しています。r24 は比較結果を格納するコンディションコードレジスタです。r28 はグローバルポインタです。r29 はスタックポインタです。r30 は、フレームポインタです。r31 はサブルーチンの戻りアドレスを格納するリンクレジスタです。これらのレジスタは、命令により暗黙的に使用されます。

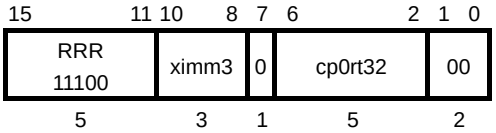
AC0IU *cp0rt32*, *imm3*

Add Coprocessor 0 Immediate Unsigned

動作

cp0rt32 ⇐ *cp0rt32* + *imm3*

コード



説明

3 ビット *imm3* を下表に従ってデコードを行い、cp0 レジスタ cp0rt32 の内容に加え、結果を CP0 レジスタ cp0rt32 に格納します。本命令の imm3 は、−8、−4、−2、−1、+1、+2、+4、+8 しか取り扱いません。

ximm3	imm3
1 1 1	−8
1 1 0	−4
1 0 1	−2
1 0 0	−1
0 0 0	+1
0 0 1	+2
0 1 0	+4
0 1 1	+8

計算の結果による整数オーバフロー例外は発生しません。

ERET 命令の直前または 2 命令前で、AC0IU 命令を使用して Status レジスタ、EPC レジスタ、ErrorEPC レジスタへ書き込むことは禁止されています。ERET 命令の直前または 2 命令前にこれらのレジスタへの書き込みを行った場合、動作は不定となります。また、AC0IU 命令を使用して SSCR レジスタを書き換える場合、後続に 2 つの NOP 命令を必ず置いてください。

例外

コプロセッサ使用不可例外

使用例

コプロセッサレジスタの EPC の値が、0x8001_0060 のときに、

```
AC0IU  EPC, -8
```

を実行すると、EPC には、0x8001_0058 の値が格納されます。

ADDIU fp, *immediate*

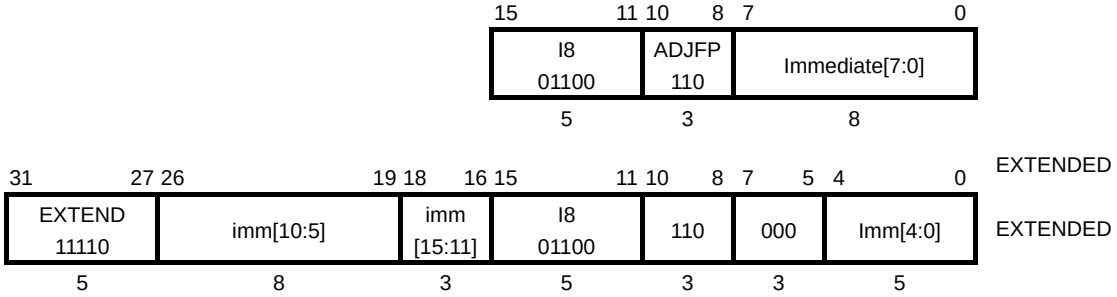
Add Immediate Unsigned

動作

$$r30 \Leftarrow r30 + (immediate_7)^{22} \parallel (immediate_{7..0}) \parallel 00$$

(EXTENDED)　$r30 \Leftarrow r30 + (immediate_{15})^{16} \parallel (immediate_{15..0})$

コード



説明

ADDIU は Add Immediate Unsigned を表しますが、8 ビット *immediate* を 2 ビット左へシフトし、「符号拡張」します。その結果を fp レジスタ (r30) の内容に加算します。

整数オーバフロー例外は発生しません。

immediate は 8 ビットで、2 ビット左へシフトすることにより、扱うことのできる数値の範囲は、4 刻みで–512 ~ +504 になります。この範囲外の値を指定すると、ADDIU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*immediate* はシフトされません。

例外

なし

使用例

フレームポインタ fp の値が 0x0000_2000 の場合、以下の命令を実行すると、図示したように、結果 0x0000_2008 がレジスタ fp に格納されます。

```
ADDIU fp,8
```

ADDIU *rx*, *immediate*

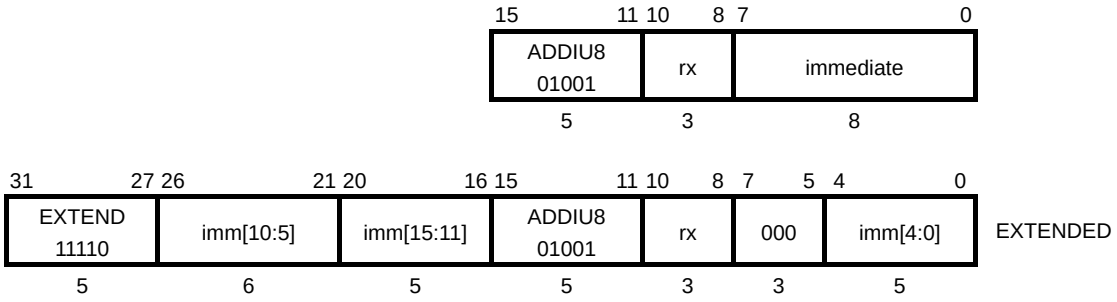
Add Immediate Unsigned

動作

$$rx \Leftarrow rx + (immediate_{7})^{24} \parallel (immediate_{7..0})$$

(EXTENDED)　$$rx \Leftarrow rx + (immediate_{15})^{16} \parallel (immediate_{15..0})$$

コード



説明

ADDIU は Add Immediate Unsigned を表しますが、8 ビット *immediate* を「符号拡張」して、汎用レジスタ *rx* の内容に加算し、その結果を汎用レジスタ *rx* に格納します。

整数オーバフロー例外は発生しません。

immediate は 8 ビットで、扱えることのできる数値の範囲は、–128 ~ +127 です。この範囲外の値を指定すると、ADDIU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。

例外

なし

ADDIU *rx*, pc, *immediate*

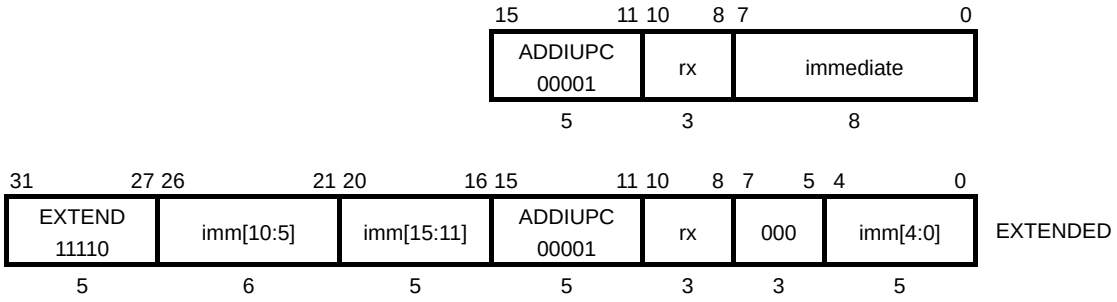
Add Immediate Unsigned

動作

$rx \Leftarrow$ マスクベース PC + 0^{22} ‖ ($immediate_{7..0}$) ‖ 00

(EXTENDED)　$rx \Leftarrow$ マスクベース PC + ($immediate_{15}$)^{16} ‖ ($immediate_{15..0}$)

コード



説明

アドレス計算のベースとして使用される PC 値を、ベース PC 値といい、ベース PC 値の下位 2 ビットを 0 にマスクした値を、マスクベース PC 値といいます。この命令は、8 ビット *immediate* を 2 ビット左へシフトし、ゼロ拡張します。その結果をマスクベース PC 値に加算した和を、仮想アドレスとして汎用レジスタ *rx* に格納します。この命令は、この命令の近くに置かれた命令やデータの PC 相対アドレスを算出するための専用の命令です。

整数オーバフロー例外は発生しません。

伸長後の 32 ビット命令コードは、有効な 32 ビット ISA 命令ではなく、この命令の機能は、32 ビット ISA の ADDIU 命令とは異なります。

immediate は 8 ビットで、2 ビット左へシフトすることにより、扱うことのできる数値の範囲は、4 刻みで 0~1020 になります。この範囲外の値を指定すると、ADDIU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*immediate* はシフトされません。

命令が遅延スロットに置かれているか、拡張されているかにより、ベース PC 値は以下のように異なります。

ADDIUPC	ベース PC 値
JR・JALR 命令の遅延スロット	JR・JALR 命令のアドレス
JAL・JALX 命令の遅延スロット	JAL・JALX 命令の上位ハーフワードのアドレス
拡張されている	EXTEND 命令のアドレス
拡張されていない	ADDIUPC 命令のアドレス

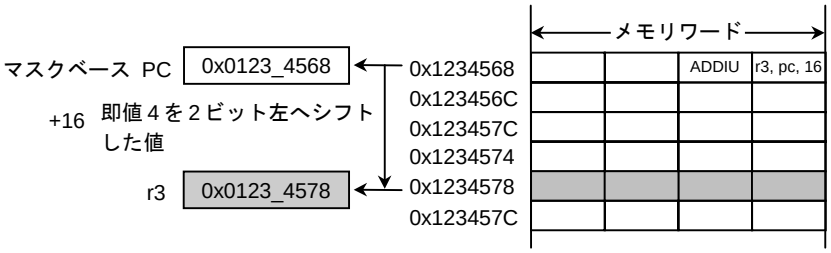
例外

なし

使用例

```
ADDIU r3,pc,16
```

上記の命令がアドレス 0x0123_456A に置かれており、このアドレスが遅延スロットでないものとします。このとき、PC の下位 2 ビットを 0 にマスクすると、マスクベース PC は、0x0123_4568 になります。即値は 2 ビット左へシフトされるため、指定したオペランド (16) はアセンブラにより 4 に変換されます。したがって、上記の ADDIU 命令の命令コードは 0x0B04 になります。下図に示すように、オフセットがマスクベース PC に付加され、その結果がレジスタ r3 に格納されます。



ADDIU *rx*, sp, *immediate*

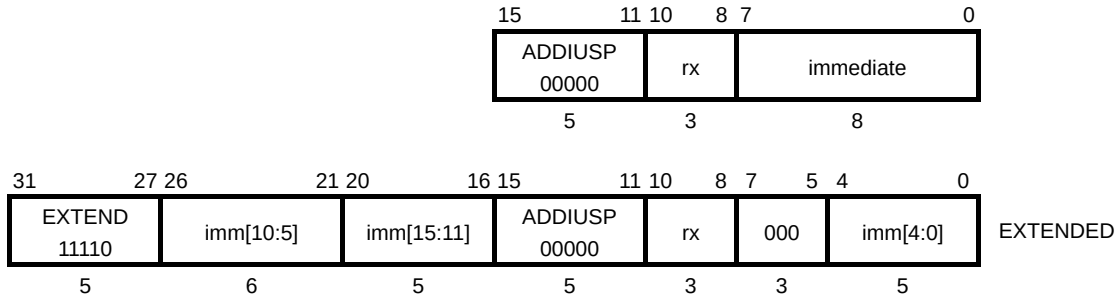
Add Immediate Unsigned

動作

$$rx \Leftarrow \mathrm{sp} + 0^{22} \parallel (immediate_{7..0}) \parallel 00$$

(EXTENDED)　$rx \Leftarrow \mathrm{sp} + (immediate_{15})^{16} \parallel (immediate_{15..0})$

コード



説明

8 ビット *immediate* を 2 ビット左へシフトし、ゼロ拡張します。その結果をスタックポインタ sp (r29) の内容に加算した和を、汎用レジスタ *rx* に格納します。

整数オーバフロー例外は発生しません。

immediate は 8 ビットで、2 ビット左へシフトすることにより、扱うことのできる数値の範囲は、4 刻みで 0~1020 になります。この範囲外の値を指定すると、ADDIU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*immediate* はシフトされません。

例外

なし

ADDIU *ry*, *rx*, *immediate*

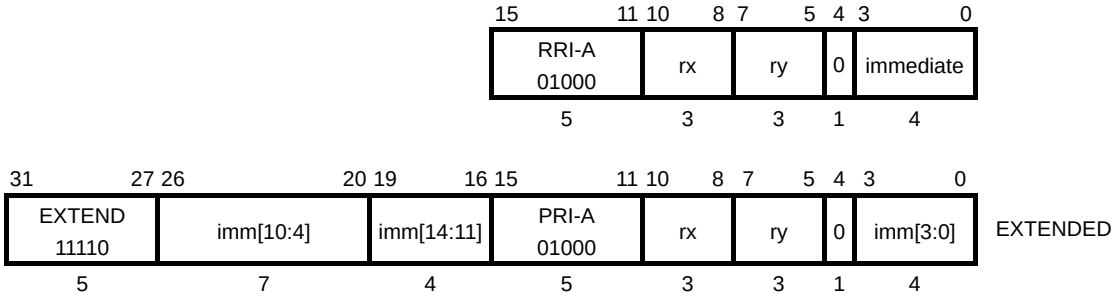
Add Immediate Unsigned

動作

$$ry \Leftarrow rx + (immediate_3)^{28} \parallel (immediate_{3..0})$$

(EXTENDED)　　$ry \Leftarrow rx + (immediate_{14})^{17} \parallel (immediate_{14..0})$

コード



説明

ADDIU は Add Immediate Unsigned を表しますが、4 ビット *immediate* を「符号拡張」して、汎用レジスタ *rx* の内容に加算し、その結果を汎用レジスタ *ry* に格納します。

整数オーバフロー例外は発生しません。

immediate は 4 ビットで、扱うことのできる数値の範囲は、–8 ~ +7 です。*immediate* がこの範囲外の場合、EXTEND に命令が自動的に付加されます。これにより、ALU 即値命令の即値フィールドは、符号付きの 15 ビットに拡張されます。したがって、*immediate* で指定できる数値の範囲は、–16384 ~ +16383 になります。

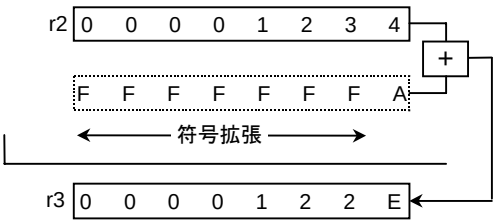
例外

なし

使用例

レジスタ r2 の値が 0x0000_1234 の場合、以下の命令を実行すると、図示したように、和 (0x0000_122E) がレジスタ r3 に格納されます。

```
ADDIU r3,r2,-6
```



ADDIU sp, *immediate*

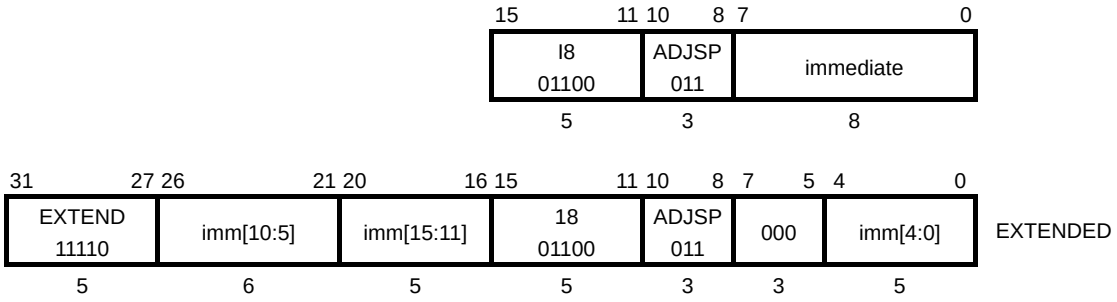
Add Immediate Unsigned

動作

$$sp \Leftarrow sp + (immediate_7)^{21} \parallel (immediate_{7..0}) \parallel 000$$

(EXTENDED)　$sp \Leftarrow sp + (immediate_{15})^{16} \parallel (immediate_{15..0})$

コード



説明

ADDIU は Add Immediate Unsigned を表しますが、8 ビット *immediate* を 3 ビット左へシフトし、「符号拡張」します。その結果をスタックポインタ sp (r29) の内容に加算します。

整数オーバフロー例外は発生しません。

immediate は 8 ビットで、3 ビット左へシフトすることにより、扱うことのできる数値の範囲は、8 刻みで–1024 ~ +1016 になります。この範囲外の値を指定すると、ADDIU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*immediate* はシフトされません。

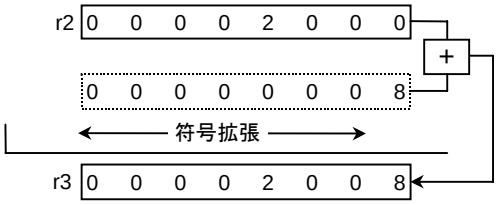
例外

なし

使用例

スタックポインタ sp の値が 0x0000_2000 の場合、以下の命令を実行すると、図示したように、結果 0x0000_2008 がスタックポインタ sp に格納されます。

```
ADDIU sp,8
```


r2
0 0 0 0 2 0 0 0
+
0 0 0 0 0 0 0 8
符号拡張
r3
0 0 0 0 2 0 0 8

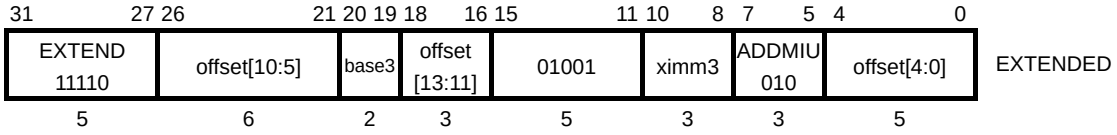
ADDMIU *offset* (*base3*), *imm3*

Add Immediate to Memory Word

動作

{zero-extend (*offset* || 00) + (*base3*)} ⇐ {zero-extend (*offset* || 00) + (*base3*)} + *imm3*

コード



説明

14 ビット *offset* を 2 ビット左にシフトし、ゼロ拡張します。その値と汎用レジスタ *base3* の内容を加算することにより、実行アドレス (EA) を生成します。3 ビットの *imm3* は、以下の表のようにデコードされ、EA で指定されたメモリ中のワードデータと加算されます。その結果をメモリに書き戻します。

base3	GPR
01	r28 (gp)
10	r29 (sp)
11	r30 (fp)

本命令の即値は、–8, –4, –2, –1, +1, +2, +4, +8 を取り扱うことができます。

ximm3	imm3
1　1　1	−8
1　1　0	−4
1　0　1	−2
1　0　0	−1
0　0　0	+1
0　0　1	+2
0　1　0	+4
0　1　1	+8

本命令では、整数オーバフロー例外は発生しません。

14 ビット *offset* を 2 ビット左へシフトして取り扱うことのできる数値の範囲は、4 刻みで 0～65532 となります。

例外

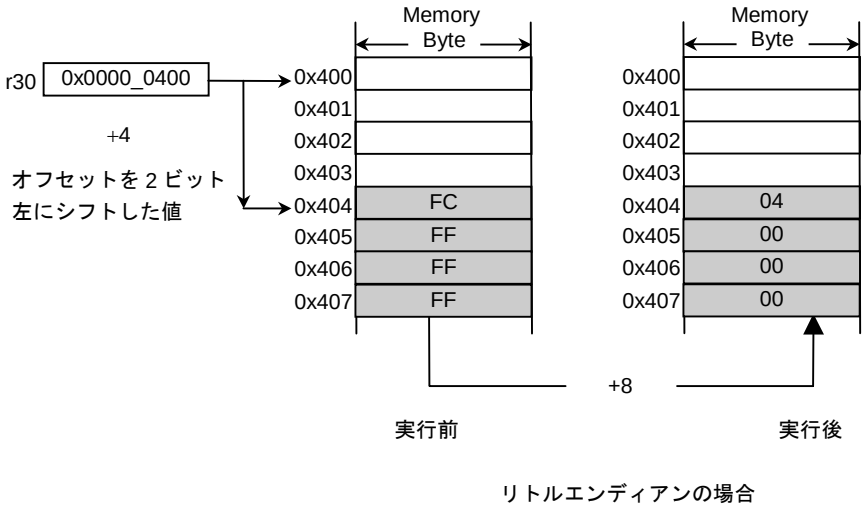
アドレスエラー例外

使用例

fp レジスタの値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xFFFF_FFFC であるとします。

```
ADDMIU 4(fp), 8
```

上記命令を実行すると、アドレス 0x0404 には 0x0000_0004 が格納されます。



リトルエンディアンの場合

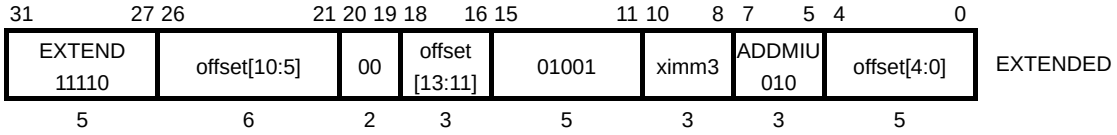
ADDMIU *offset* (r0), *imm3*

Add Immediate to Memory Word

動作

{sign-extend (*offset* ‖ 00)} ⇐ {sign-extend (*offset* ‖ 00)} + *imm3*

コード



説明

14 ビット *offset* を 2 ビット左にシフトし、符号拡張します。その値と汎用レジスタ r0 の内容を加算することにより、実行アドレス (EA) を生成します。3 ビットの *imm3* は、以下の表のようにデコードされ、EA で指定されたメモリ中のワードデータと加算されます。その結果をメモリに書き戻します。

本命令の即値は、–8, –4, –2, –1, +1, +2, +4, +8 を取り扱うことができます。

ximm3	imm3
1　1　1	−8
1　1　0	−4
1　0　1	−2
1　0　0	−1
0　0　0	+1
0　0　1	+2
0　1　0	+4
0　1　1	+8

本命令では、整数オーバフロー例外は発生しません。

14 ビット *offset* を 2 ビット左へシフトして取り扱うことのできる数値の範囲は、4 刻みで-32768～+32764 となります。

例外

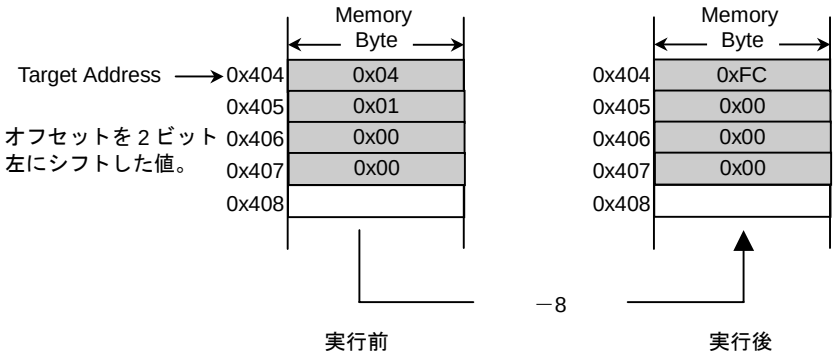
アドレスエラー例外

使用例

アドレス 0x0404 の内容が、0x0000_0104 とします。

```
ADDMIU 0x404(r0), -8
```

上記命令を実行すると、アドレス 0x0404 には 0x0000_00FC が格納されます。



リトルエンディアンの場合

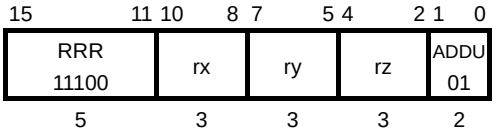
ADDU *rz*, *rx*, *ry*

Add Unsigned

動作

rz ⇐ *rx* + *ry*

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を加算し、その結果を汎用レジスタ *rz* に格納します。整数オーバフロー例外は絶対に発生しません。

例外

なし

使用例

レジスタ r2 の値が 0x0200_0000 で、レジスタ r3 の値が 0x0123_4567 の場合、以下の命令を実行すると、和 (0x0323_4567) がレジスタ r4 に格納されます。

```
ADDU r4,r2,r3
```

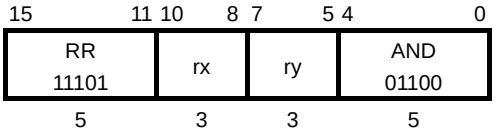
AND *rx*, *ry*

AND

動作

rx ⇐ *rx* AND *ry*

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容の論理積演算を行い、その結果を汎用レジスタ *rx* に格納します。

例外

なし

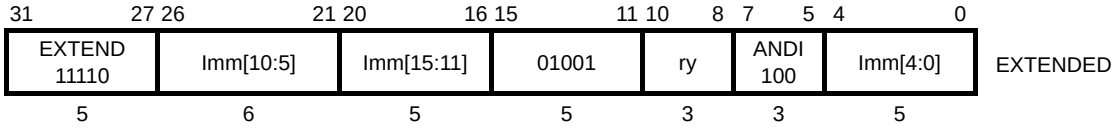
ANDI *ry*, *immediate*

Logical AND Immediate

動作

$ry \Leftarrow ry$ AND $(0^{16} \parallel immediate_{15..0})$

コード



説明

16 ビット *immediate* をゼロ拡張し、汎用レジスタ *ry* の内容と論理積演算を行い、その結果を汎用レジスタ *ry* に格納します。

immediate フィールドは 16 ビットです。これを超える値を扱いたい場合は、いったん汎用レジスタに格納してから、AND 命令を使います(「4.3.2　32 ビットの定数」を参照)。

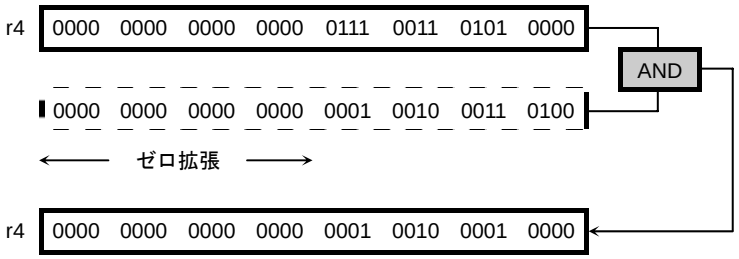
例外

なし

使用例

レジスタ r4 の値が 0x0000_7350 の場合、以下の命令を実行すると、図示したように、0x0000_7350 と 0x0000_1234 の論理積 (0x0000_1210) がレジスタ r4 に格納されます。

```
ANDI r4,0x1234
```



B *offset*

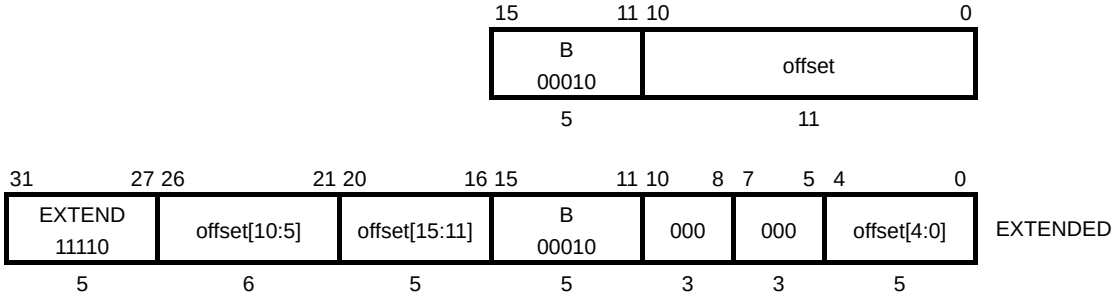
Unconditional Branch

動作

pc ⇐ pc + 2 + sign-extend (*offset* ‖ 0)

(EXTENDED)　pc ⇐ pc + 4 + sign-extend (*offset* ‖ 0)

コード



説明

1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスへ無条件に分岐します。パイプラインの遅延については、「5.3.4　分岐命令 (16 ビット ISA)」を参照してください。この命令には分岐遅延スロットは存在しません。分岐する場合に直後の命令は実行されません。ターゲットのアドレスは直後の命令のアドレスに対して相対的に計算されます。通常は PC + 2、EXTEND 命令では PC + 4 がベースになります。

offset は 11 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、–2048 ~ +2046 です。この範囲外の値を指定すると、B 命令は EXTEND 命令により拡張され、符号付きの 17 ビットの即値 (–65536 ~ +65534) を扱えるよになります。この場合も、ターゲットアドレスは、拡張しない場合と同様に計算されます。

例外

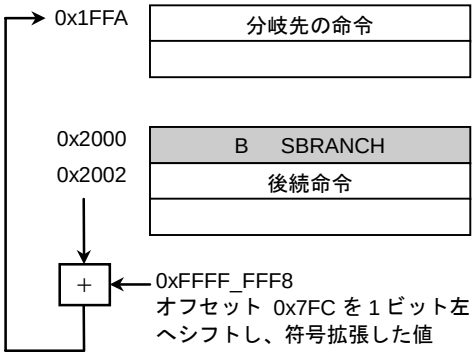
なし

使用例

```
B  SBRANCH
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SBRANCH が 0x1FFA に絶対アドレス化される場合、以下に図示するように、SBRANCH はアセンブラ・リンカによりオフセット 0x7FC に変換されます。したがって、命令コードは 0x17FC になります。

上記の命令を実行すると、プログラムの処理はターゲットアドレス 0x1FFA に分岐します。この場合、B 命令の後続の命令は実行されません。


0x1FFA
分岐先の命令
0x2000
B　SBRANCH
0x2002
後続命令
+
0xFFFF_FFF8
オフセット 0x7FC を 1 ビット左
へシフトし、符号拡張した値

BAL *offset*

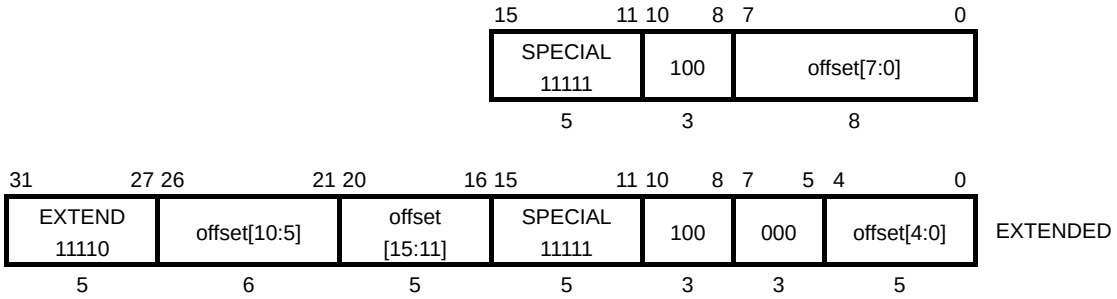
Branch And Link

動作

r31 ⇐ pc + 3;　pc ⇐ pc + 2 + sign-extend (*offset* ‖ 0)

(EXTENDED)　r31 ⇐ pc + 5;　pc ⇐ pc + 4 + sign-extend (*offset* ‖ 0)

コード



説明

1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐し、後続命令のアドレス (PC+2) を、無条件にリンクレジスタ r31 に格納します。パイプラインの遅延については、「5.3.4　分岐命令 (16 ビット ISA)」を参照してください。この命令には分岐遅延スロットは存在しません。分岐する場合に直後の命令は実行されません。ターゲットのアドレスは直後の命令のアドレスに対して相対的に計算されます。通常は PC + 2、EXTEND 命令では PC + 4 がベースになります。

BAL 命令の次のアドレスがリンクレジスタ ra (r31) に格納されます。また、ra の最下位ビットに ISA モードビット (16 ビット ISA モード = 1) を格納します。

offset は 8 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は–256 ~ +254 です。この範囲外の指定をすると、BAL 命令は EXTEND 命令により拡張され、符号付きの 17 ビットの即値 (–65536 ~ +65534) を扱えるようになります。この場合も、ターゲットアドレスは拡張しない場合と同様に計算されます。

例外

なし

使用例

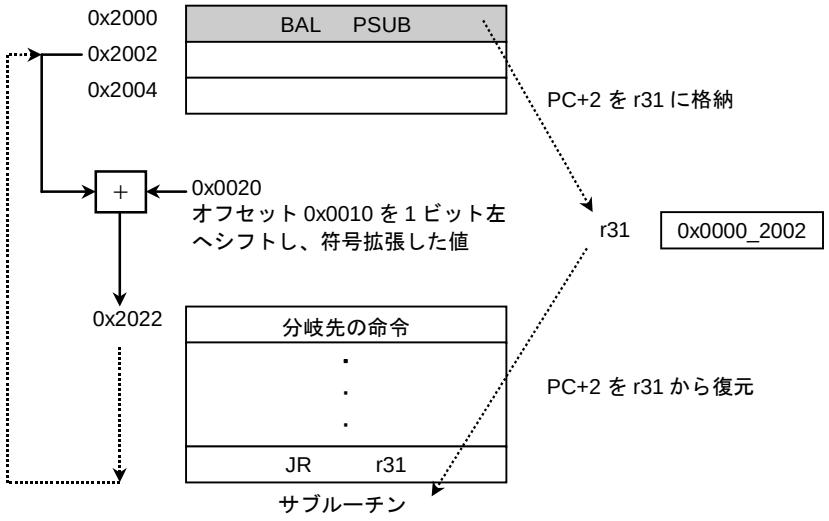
```
BAL  PSUB
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル PSUB が 0x2022 に絶対アドレス化される場合、以下に図示するように、PSUB はアセンブラ・リンカによりオフセット 0x0010 に変換されます。

プログラムの処理はアドレス 0x2022 に分岐します。

コールされたサブルーチンの終わりで JR 命令を実行することにより、分岐命令の次の命令 (PC+2) に戻ることができます。

```
JR r31
```



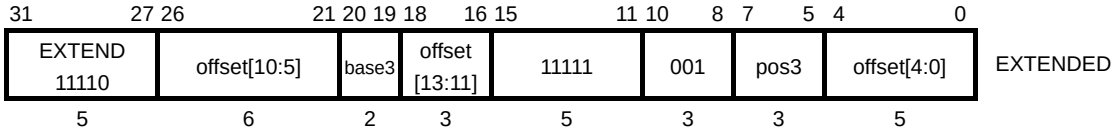
BCLR *offset* (*base3*), *pos3*

Bit Clear

動作

{zero-extend (*offset*) + (*base3*)} [*pos3*] ⇐ 0

コード



説明

14 ビット *offset* をゼロ拡張して、*base3* で指定された汎用レジスタの内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、0 にクリアします。

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

offset は 14 ビットで、扱うことのできる数値の範囲は、0～ + 16383 です。

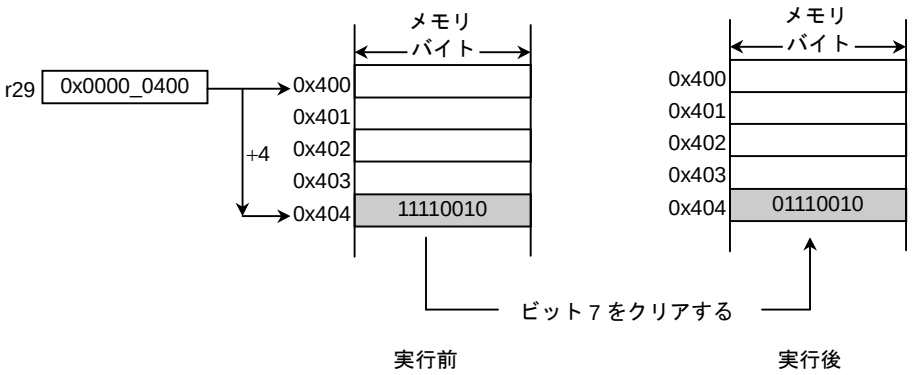
例外

アドレスエラー例外

使用例

sp レジスタ (r29) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、アドレス 0x0404 の内容が 0x72 になります。

```
BCLR 4(sp),7
```


メモリ
バイト
r29
0x0000_0400
+4
0x400
0x401
0x402
0x403
0x404
11110010
ビット 7 をクリアする
実行前
メモリ
バイト
0x400
0x401
0x402
0x403
0x404
01110010
実行後

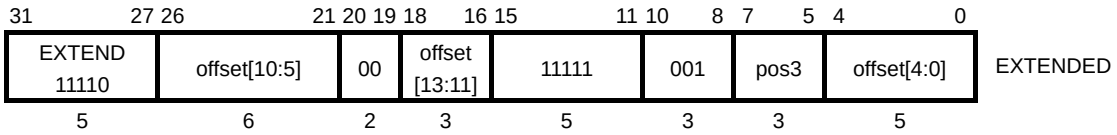
BCLR *offset* (r0), *pos3*

Bit Clear

動作

{sign-extend (*offset*)} [*pos3*] ⇐ 0

コード



説明

14 ビット *offset* を符号拡張して、汎用レジスタ r0 の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、0 にクリアします。

offset は 14 ビットで、扱うことのできる数値の範囲は、−8192 ～ + 8191 です。

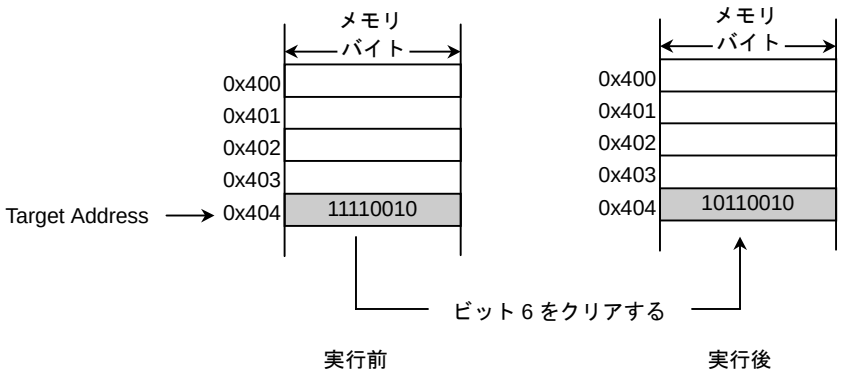
例外

アドレスエラー例外

使用例

アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、アドレス 0x404 の内容が 0xB2 になります。

```
BCLR 0x404(r0), 6
```



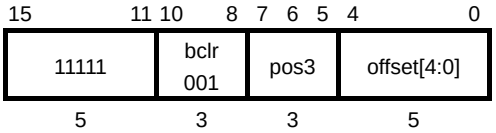
BCLR *offset* (fp), *pos3*

Bit Clear

動作

{zero-extend (*offset*) + (fp)} [*pos3*] ⇐ 0

コード



説明

5 ビット *offset* をゼロ拡張して、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、0 にクリアします。

offset は 5 ビットで、扱うことのできる数値の範囲は、0～ + 31 です。

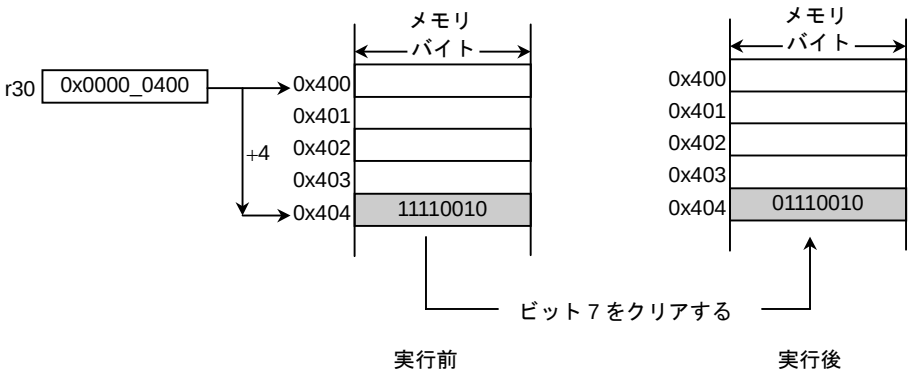
例外

アドレスエラー例外

使用例

fp レジスタ (r30) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、アドレス 0x0404 の内容が 0x72 になります。

```
BCLR 4(fp), 7
```



BEQZ *rx*, *offset*

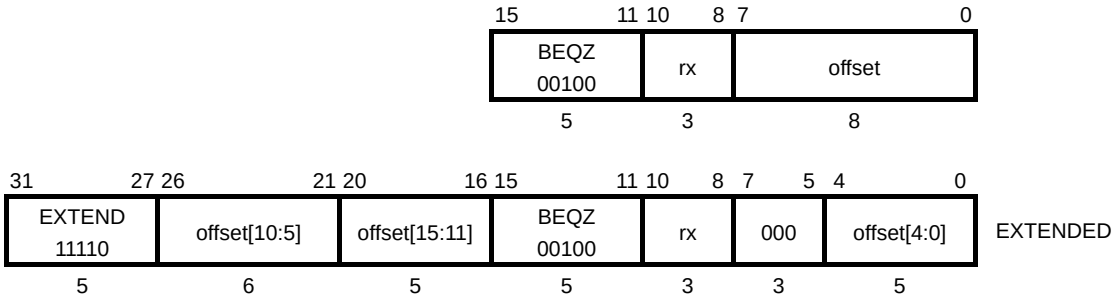
Branch On Equal To Zero

動作

if *rx* = 0 then pc ⇐ pc + 2 + sign-extend (*offset* || 0)

(EXTENDED)　if *rx* = 0 then pc ⇐ pc + 4 + sign-extend (*offset* || 0)

コード



説明

汎用レジスタ *rx* の内容が 0 の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。パイプライン遅延については「5.3.4 分岐命令 (16 ビット ISA)」を参照してください。この命令には分岐遅延スロットは存在しません。分岐する場合に直後の命令は実行されません。ターゲットのアドレスは直後の命令のアドレスに対して相対的に計算されます。通常は PC + 2、EXTEND 命令では PC + 4 がベースになります。

offset は 8 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、–256 ~ +254 です。この範囲外の値を指定すると、BEQZ 命令は EXTEND 命令により拡張され、符号付きの 17 ビットの即値 (–65536 ~ +65534) を扱えるようになります。この場合も、ターゲットアドレスは、拡張しない場合と同様に計算されます。

例外

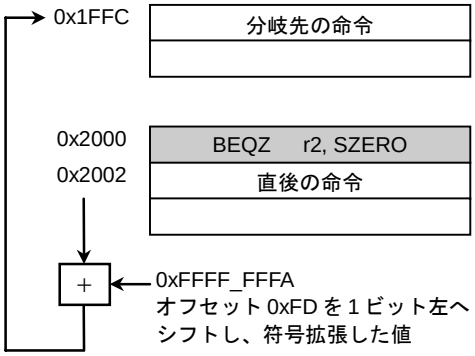
なし

使用例

```
BEQZ r2,SZERO
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SZERO が 0x1FFC に絶対アドレス化される場合、以下に図示するように、SZERO はアセンブラ・リンカによりオフセット 0xFD に変換されます。したがって、命令コードは 0x22FD になります。

r2 の内容が 0 のとき、プログラムの処理はアドレス 0x1FFC に分岐します。r2 の内容が 0 以外の場合は、分岐せずに次の命令 (アドレス 0x2002) の実行に移ります。


0x1FFC
分岐先の命令
0x2000
BEQZ　r2, SZERO
0x2002
直後の命令
+
0xFFFF_FFFA
オフセット 0xFD を 1 ビット左へ
シフトし、符号拡張した値

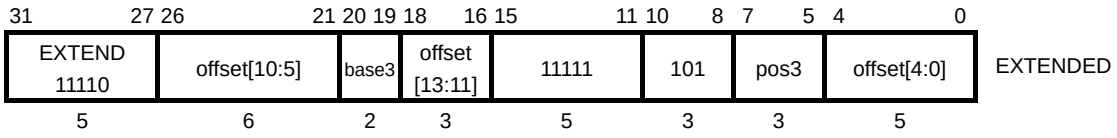
BEXT *offset* **(***base3***),** *pos3*

Bit Extract

動作

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 ‖ {zero-extend (*offset*) + (*base3*)} [*pos3*]

コード



説明

14 ビット *offset* をゼロ拡張して、*base3* で指定された汎用レジスタの内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、汎用レジスタ t8 (r24) の LSB ビットに格納し、残りのビットは 0 が格納されます。

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

offset は 14 ビットで、扱うことのできる数値の範囲は、0～ + 16383 です。

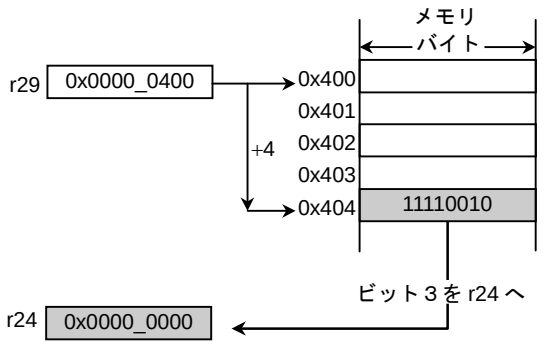
例外

アドレスエラー例外

使用例

sp レジスタ (r29) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r24 に 0x0000_0000 がロードされます。

```
BEXT 4(sp), 3
```



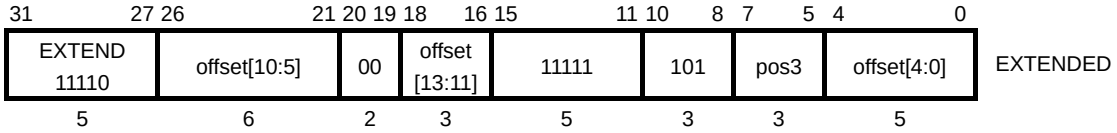
BEXT *offset* (r0), *pos3*

Bit Extract

動作

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 ‖ {sign-extend (*offset*)} [*pos3*]

コード



説明

14 ビット *offset* を符号拡張して、汎用レジスタ r0 の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、汎用レジスタ t8 (r24) の LSB ビットに格納し、残りのビットは 0 が格納されます。

offset は 14 ビットで、扱うことのできる数値の範囲は、−8192 ～ + 8191 です。

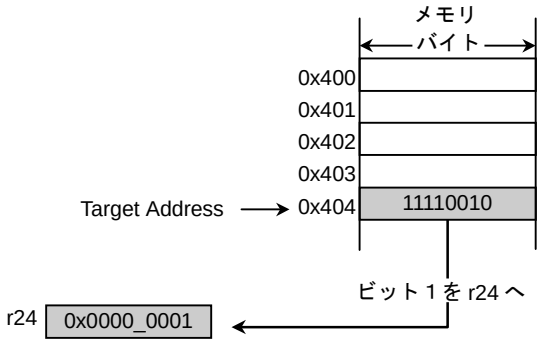
例外

アドレスエラー例外

使用例

アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r24 に 0x0000_0001 がロードされます。

```
BEXT 0x404(r0), 1
```



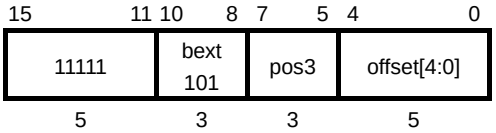
BEXT *offset* **(fp),** *pos3*

Bit Extract

動作

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 ‖ {zero-extend (*offset*) + (fp)} [*pos3*]

コード



説明

5 ビット *offset* をゼロ拡張して、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、汎用レジスタ t8 (r24) の LSB ビットに格納し、残りのビットは 0 が格納されます。

offset は 5 ビットで、扱うことのできる数値の範囲は、0～ + 31 です。

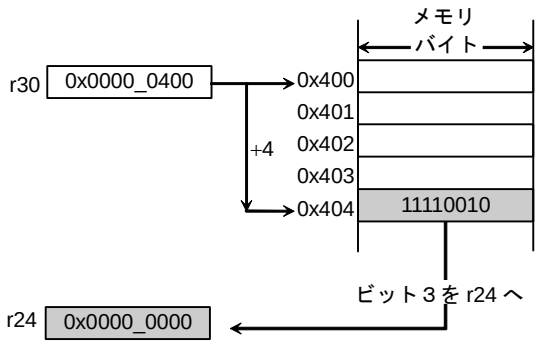
例外

アドレスエラー例外

使用例

fp レジスタ (r30) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r24 に 0x0000_0000 がロードされます。

```
BEXT 4(fp), 3
```



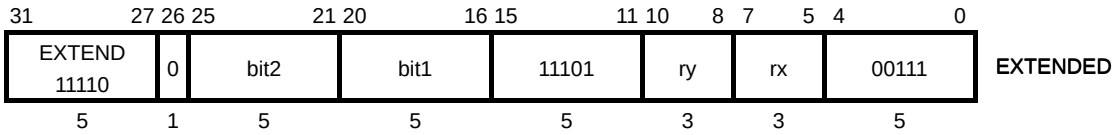
BFINS *ry*, *rx*, *bit2*, *bit1*

Bit Field Insert

動作

ry[*bit2*:*bit1*] ⇐ *rx*[*bit2*-*bit1*：*0*]；

コード



説明

汎用レジスタ *rx* の[(*bit2*-*bit1*):*0*]のビット列を汎用レジスタ *ry* の[*bit2*:*bit1*]へ格納します。

例外

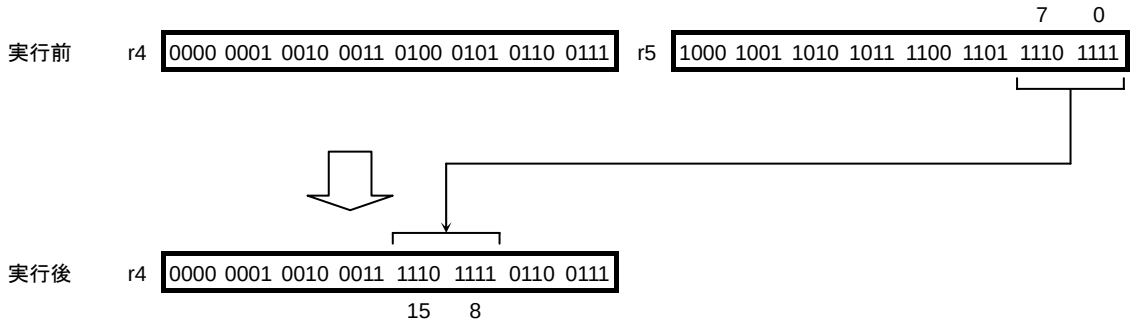
なし

使用例

汎用レジスタ r4 の値が 0x0123_4567 で、汎用レジスタ r5 の値が 0x89AB_CDEF の場合、

```
bfins  r4, r5, 15, 8
```

を実行すると、r4 に 0x0123_EF67 が格納されます。



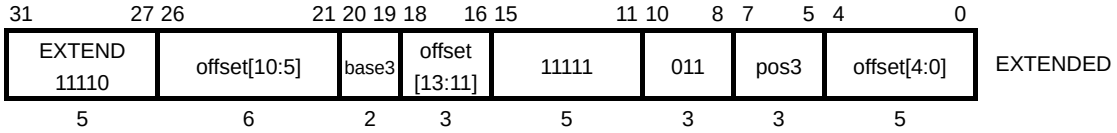
BINS *offset* **(***base3***),** *pos3*

Bit Insert

動作

{zero-extend (*offset*) + (*base3*)} [*pos3*] ⇐ t8[0]

コード



説明

14 ビット *offset* をゼロ拡張して、*base3* で指定された汎用レジスタの内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリのバイトデータに *pos3* で指定されたビット番号の位置に、汎用レジスタ t8 (r24) の LSB ビットを挿入する。

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

offset は 14 ビットで、扱うことのできる数値の範囲は、0～ + 16383 です。

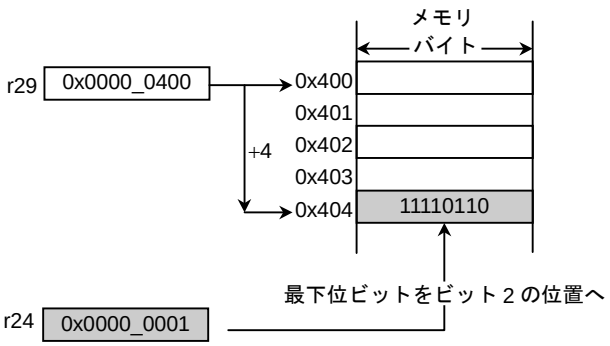
例外

アドレスエラー例外

使用例

sp レジスタ (r29) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2、汎用レジスタ r24 の内容が 0x0000_0001 の場合、以下の命令を実行すると、アドレス 0x0404 の内容が 0xF6 となります。

```
BINS 4(sp), 2
```



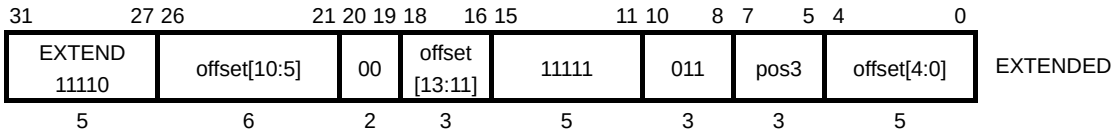
BINS *offset* (r0), *pos3*

Bit Insert

動作

{sign-extend (*offset*)} [*pos3*] ⇐ t8[0]

コード



説明

14 ビット *offset* を符号拡張して、汎用レジスタ r0 の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリのバイトデータに *pos3* で指定されたビット番号の位置に、汎用レジスタ t8 (r24) の LSB ビットを挿入する。

offset は 14 ビットで、扱うことのできる数値の範囲は、−8192 ～ + 8191 です。

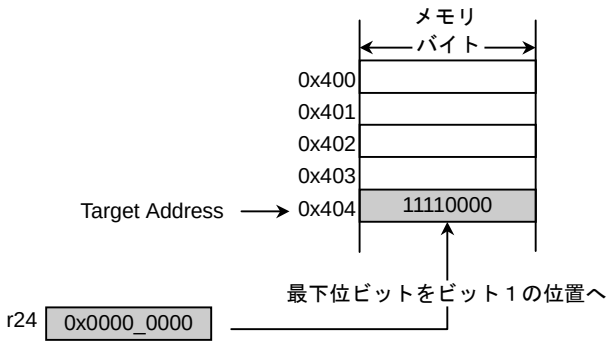
例外

アドレスエラー例外

使用例

アドレス 0x0404 の内容が 0xF2、汎用レジスタ r24 の内容が 0x0000_0000 の場合、以下の命令を実行すると、アドレス 0x0404 の内容は 0xF0 になります。

```
BINS 0x404(r0), 1
```



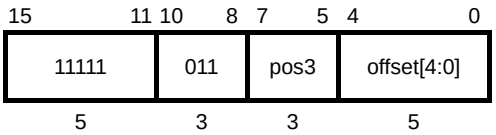
BINS *offset* (fp), *pos3*

Bit Insert

動作

{zero-extend (*offset*) + (fp)} [*pos3*] ⇐ t8[0]

コード



説明

5 ビット *offset* をゼロ拡張して、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリのバイトデータに *pos3* で指定されたビット番号の位置に、汎用レジスタ t8 (r24) の LSB ビットを挿入する。

offset は 5 ビットで、扱うことのできる数値の範囲は、0～ + 31 です。

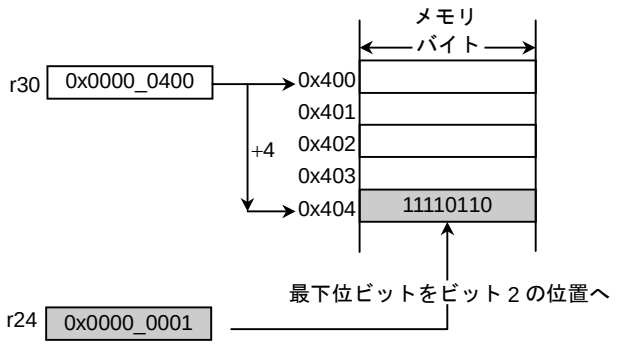
例外

アドレスエラー例外

使用例

fp レジスタ (r30) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2、汎用レジスタ r24 の内容が 0x0000_0001 の場合、以下の命令を実行すると、アドレス 0x0404 の内容が 0xF6 となります。

```
BINS 4(fp), 2
```



BNEZ *rx*, *offset*

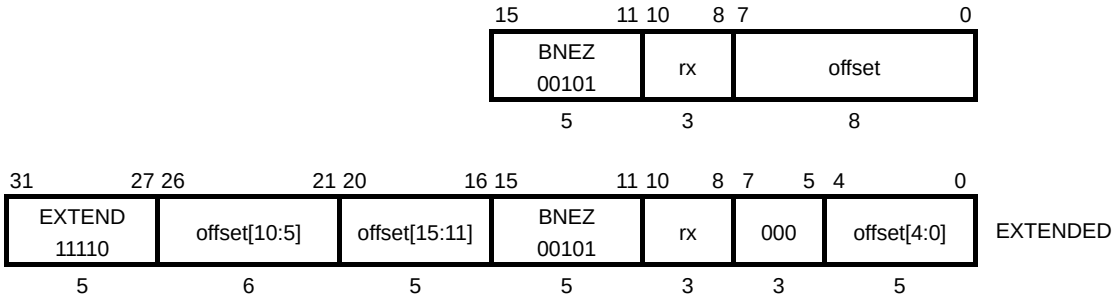
Branch On Not Equal To Zero

動作

if *rx* ≠ 0 then pc ⇐ pc + 2 + sign-extend (*offset* ‖ 0)

(EXTENDED)　if *rx* ≠ 0 then pc ⇐ pc + 4 + sign-extend (*offset* ‖ 0)

コード



説明

汎用レジスタ *rx* の内容が 0 でない場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。パイプライン遅延については「5.3.4 分岐命令 (16 ビット ISA)」を参照してください。この命令には分岐遅延スロットは存在しません。分岐する場合に直後の命令は実行されません。ターゲットのアドレスは直後の命令のアドレスに対して相対的に計算されます。通常は PC + 2、EXTEND 命令では PC + 4 がベースになります。

offset は 8 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、–256 ~ +254 です。この範囲外の値を指定すると、BNEZ 命令は EXTEND 命令により拡張され、符号付きの 17 ビットの即値 (–65536 ~ +65534) を扱えるようになります。この場合も、ターゲットアドレスは、拡張しない場合と同様に計算されます。

例外

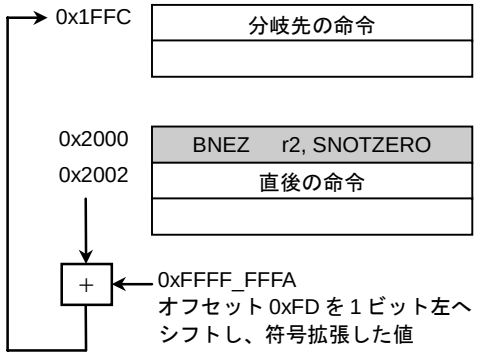
なし

使用例

```
BNEZ r2,SNOTZERO
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SNOTZERO が 0x1FFC に絶対アドレス化される場合、以下に図示するように、SNOTZERO はアセンブラ・リンカによりオフセット 0xFD に変換されます。したがって、命令コードは 0x2AFD になります。

r2 の内容が 0 以外のとき、プログラムの処理はアドレス 0x1FFC に分岐します。r2 の内容が 0 の場合は、分岐せずに次の命令 (アドレス 0x2002) の実行に移ります。


0x1FFC
分岐先の命令
0x2000
BNEZ　r2, SNOTZERO
0x2002
直後の命令
+
0xFFFF_FFFA
オフセット 0xFD を 1 ビット左へ
シフトし、符号拡張した値

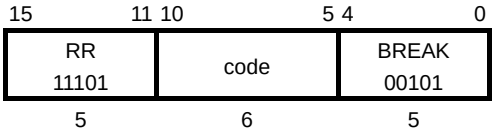
BREAK *code*

Breakpoint Exception

動作

Breakpoint exception

コード



説明

この命令を実行すると、無条件にブレークポイント例外が発生し、制御を例外ハンドラへ渡します。

命令内の *code* フィールドを使用して、例外ハンドラにパラメータを渡すことができます。例外ハンドラがこのパラメータを使用する場合には、命令を含むメモリハーフワードの内容をデータとしてロードする必要があります。詳細は「9.1.11　ブレークポイント例外」を参照してください。

例外

ブレークポイント例外

BS1F *ry*, *rx*

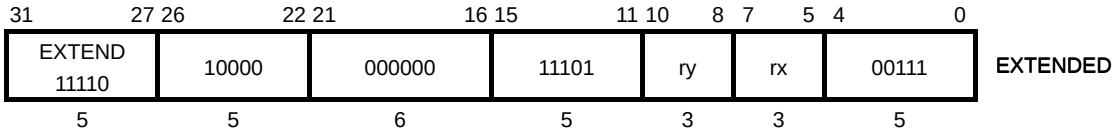
Bit Search One Forward

動作

if *rx* == 0　then　*ry* ⇐ 0 ;

　　else　*ry* ⇐ (bit position of *rx*[bit position] == 1) + 1 ;

コード



説明

汎用レジスタ *rx* の内容を下位側から上位側に向かって、1 をサーチしていき、その 1 が検出されたビット番号+1 が汎用レジスタ *ry* に格納されます。もし、*rx* の内容に 1 が検出されなかった場合は、0 が *ry* に格納されます。

例外

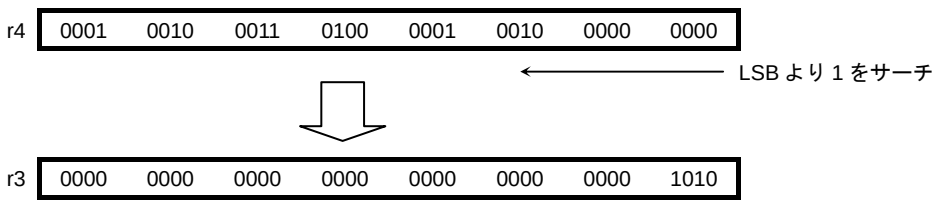
なし

使用例

汎用レジスタ r4 の値が、0x1234_1200 の場合、

```
BS1F  r3, r4
```

を実行すると、汎用レジスタ r3 は 0x0000_000A が格納されます。



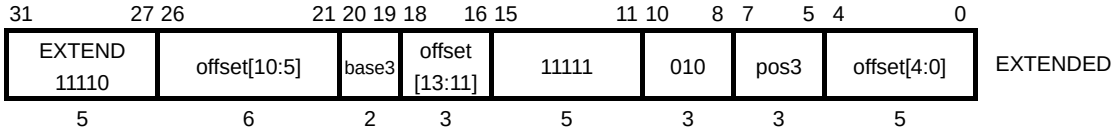
BSET *offset* (*base3*), *pos3*

Bit Set

動作

{zero-extend (*offset*) + (*base3*)} [*pos3*] ⇐ 1

コード



説明

14 ビット *offset* をゼロ拡張して、*base3* で指定された汎用レジスタの内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、1 にセットします。

base3	GPR
01	gp(r28)
10	sp(r29)
11	fp(r30)

offset は 14 ビットで、扱うことのできる数値の範囲は、0～ + 16383 です。

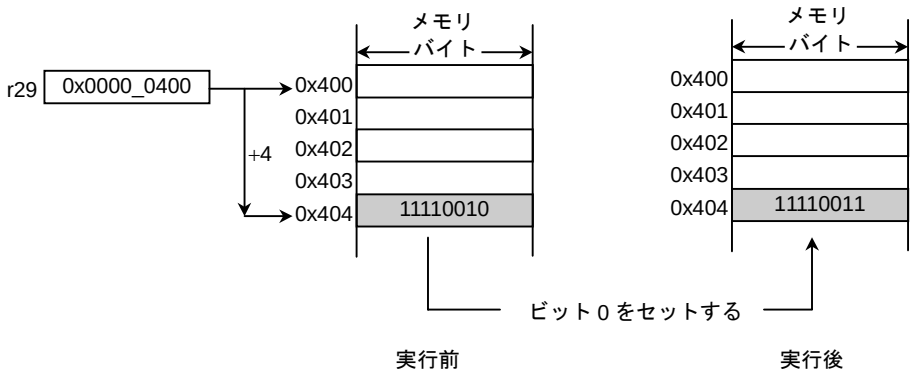
例外

アドレスエラー例外

使用例

sp レジスタ (r29) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、アドレス 0x0404 の内容が 0xF3 になります。

```
BSET 4(sp), 0
```


メモリ
バイト
r29
0x0000_0400
+4
0x400
0x401
0x402
0x403
0x404
11110010
メモリ
バイト
0x400
0x401
0x402
0x403
0x404
11110011
ビット 0 をセットする
実行前
実行後

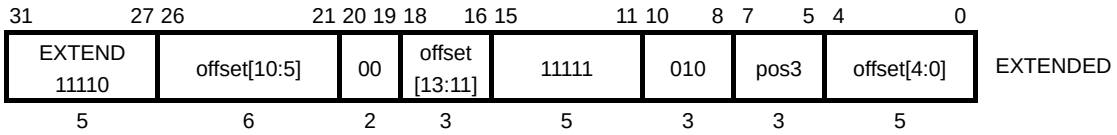
BSET *offset* (r0), *pos3*

Bit Set

動作

{sign-extend (*offset*)} [*pos3*] ⇐ 1

コード



説明

14 ビット *offset* を符号拡張して、汎用レジスタ r0 の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、1 にセットします。

offset は 14 ビットで、扱うことのできる数値の範囲は、−8192 ～ + 8191 です。

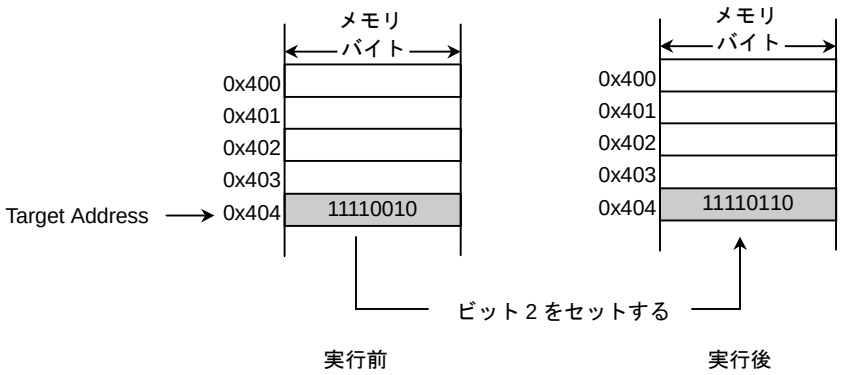
例外

アドレスエラー例外

使用例

アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、アドレス 0x0404 の内容が 0xF6 になります。

```
BSET 0x404(r0), 2
```



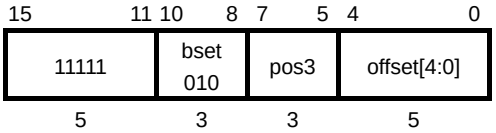
BSET *offset* (fp), *pos3*

Bit Set

動作

{zero-extend (*offset*) + (fp)} [*pos3*] ⇐ 1

コード



説明

5 ビット *offset* をゼロ拡張して、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を、1 にセットします。

offset は 5 ビットで、扱うことのできる数値の範囲は、0～ + 31 です。

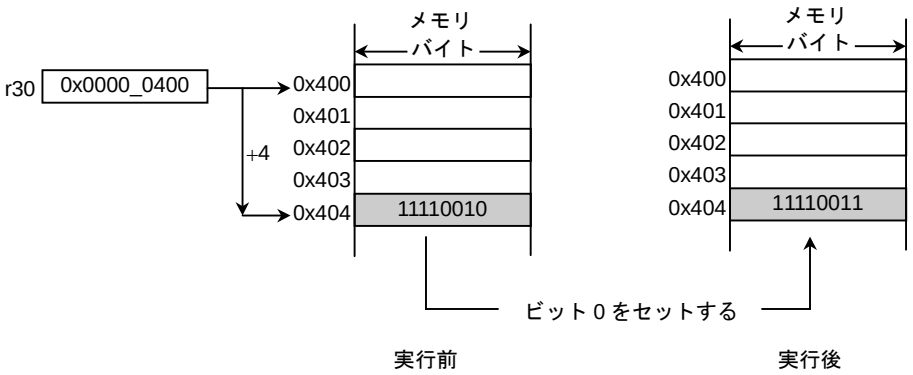
例外

アドレスエラー例外

使用例

fp レジスタ (r30)の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、アドレス 0x0404 の内容が 0xF3 になります。

```
BSET 4(fp), 0
```



BTEQZ *offset*

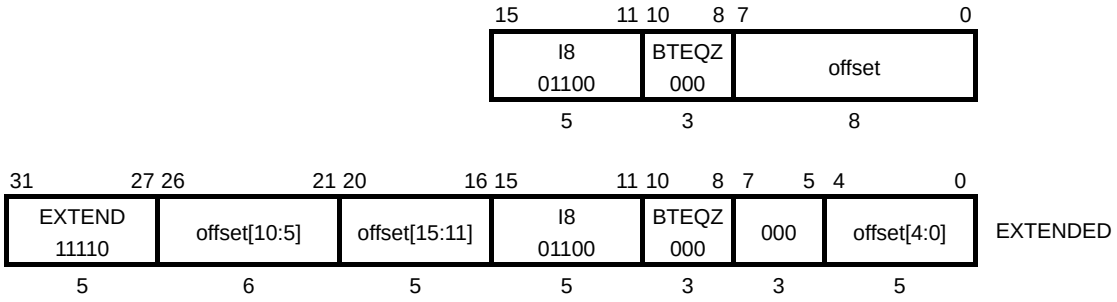
Branch On T8 Equal To Zero

動作

if t8 == 0 then pc ⇐ pc + 2 + sign-extend (*offset* || 0)

(EXTENDED)　　if t8 == 0 then pc ⇐ pc + 4 + sign-extend (*offset* || 0)

コード



説明

コンディションコードレジスタ t8 (r24) の内容が 0 の場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。パイプラインの遅延については、「5.3.4　分岐命令 (16 ビット ISA)」を参照してください。この命令には分岐遅延スロットは存在しません。分岐する場合に直後の命令は実行されません。ターゲットのアドレスは直後の命令のアドレスに対して相対的に計算されます。通常は PC + 2、EXTEND 命令では PC + 4 がベースになります。

offset は 8 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、–256 ~ +254 です。この範囲外の値を指定すると、BTEQZ 命令は EXTEND 命令により拡張され、符号付きの 17 ビットの即値 (–65536 ~ +65534) を扱えるようになります。この場合も、ターゲットアドレスは、拡張しない場合と同様に計算されます。

例外

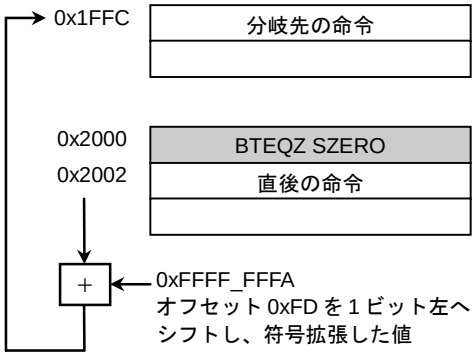
なし

使用例

```
BTEQZ  SZERO
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SZERO が 0x1FFC に絶対アドレス化される場合、以下に図示するように、SZERO はアセンブラ・リンカによりオフセット 0xFD に変換されます。したがって、命令コードは 0x60FD になります。

t8 の内容が 0 のとき、プログラムの処理はアドレス 0x1FFC に分岐します。t8 の内容が 0 以外の場合は、分岐せずに次の命令 (アドレス 0x2002) の実行に移ります。


0x1FFC
分岐先の命令
0x2000
BTEQZ SZERO
0x2002
直後の命令
+
0xFFFF_FFFA
オフセット 0xFD を 1 ビット左へ
シフトし、符号拡張した値

BTNEZ *offset*

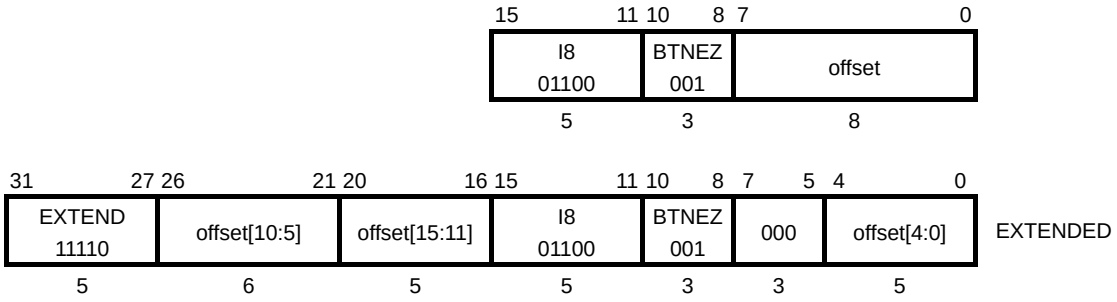
Branch On T8 Not Equal To Zero

動作

if t8 ≠ 0 then pc ⇐ pc + 2 + sign-extend (*offset* ‖ 0)

(EXTENDED)　if t8 ≠ 0 then pc ⇐ pc + 4 + sign-extend (*offset* ‖ 0)

コード



説明

コンディションコードレジスタ t8 (r24) の内容が 0 でない場合、1 命令 (2 命令サイクル) の遅延後、ターゲットアドレスに分岐します。パイプラインの遅延については、「5.3.4 分岐命令 (16 ビット ISA)」を参照してください。この命令には分岐遅延スロットは存在しません。分岐する場合に直後の命令は実行されません。ターゲットのアドレスは直後の命令のアドレスに対して相対的に計算されます。通常は PC + 2、EXTEND 命令では PC + 4 がベースになります。

offset は 8 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、–256 ~ +254 です。この範囲外の値を指定すると、BTNEZ 命令は EXTEND 命令により拡張され、符号付きの 17 ビットの即値 (–65536 ~ +65534) を扱えるようになります。この場合も、ターゲットアドレスは、拡張しない場合と同様に計算されます。

例外

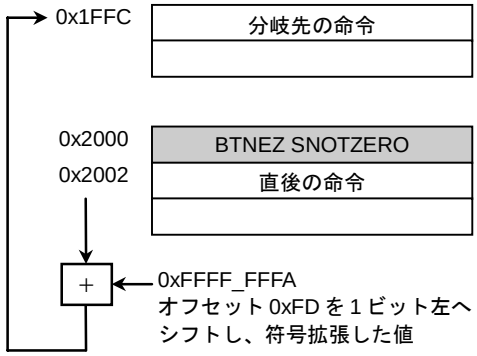
なし

使用例

```
BTNEZ SNOTZERO
```

上記の分岐命令がアドレス 0x2000 に置かれていて、ラベル SNOTZERO が 0x1FFC に絶対アドレス化される場合、以下に図示するように、SNOTZERO はアセンブラ・リンカによりオフセット 0xFD に変換されます。したがって、命令コードは 0x61FD になります。

t8 の内容が 0 のとき、プログラムの処理はアドレス 0x1FFC に分岐します。t8 の内容が 0 以外の場合は、分岐せずに次の命令 (アドレス 0x2002) の実行に移ります。


0x1FFC
分岐先の命令
0x2000
BTNEZ SNOTZERO
0x2002
直後の命令
+
0xFFFF_FFFA
オフセット 0xFD を 1 ビット左へ
シフトし、符号拡張した値

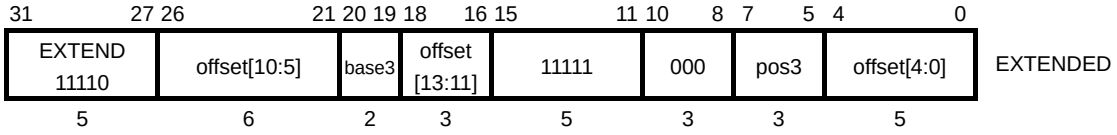
BTST *offset* (*base3*), *pos3*

Bit Test

動作

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 ‖ NOT ({zero-extend (*offset*) + (*base3*)} [*pos3*])

コード



説明

14 ビット *offset* をゼロ拡張して、*base3* で指定された汎用レジスタの内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を反転して、汎用レジスタ t8 (r24) の LSB ビットに格納し、残りのビットは 0 が格納されます。

base3	GPR
01	gp (r28)
10	sp (r29)
11	fp (r30)

offset は 14 ビットで、扱うことのできる数値の範囲は、0～ + 16383 です。

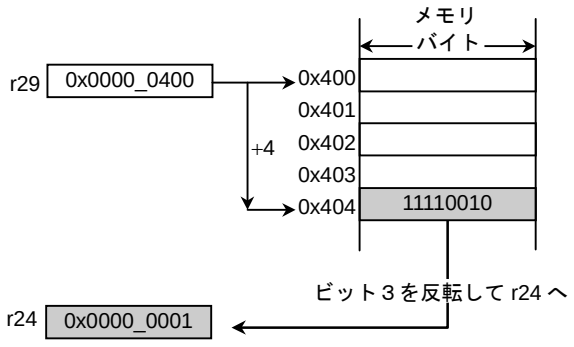
例外

アドレスエラー例外

使用例

sp レジスタ (r29) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r24 に 0x0000_0001 がロードされます。

```
BTST 4(sp), 3
```



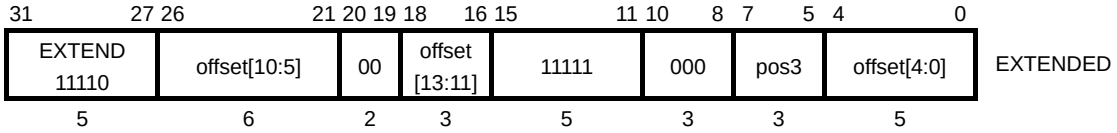
BTST *offset* (r0), *pos3*

Bit Test

動作

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 ‖ NOT ({sign-extend (*offset*)} [*pos3*])

コード



説明

14 ビット *offset* を符号拡張して、汎用レジスタ r0 の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を反転して、汎用レジスタ t8 (r24) の LSB ビットに格納し、残りのビットは 0 が格納されます。

offset は 14 ビットで、扱うことのできる数値の範囲は、−8192 ～ + 8191 です。

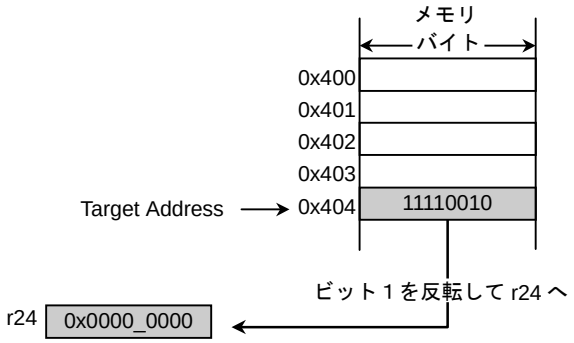
例外

アドレスエラー例外

使用例

アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r24 に 0x0000_0000 がロードされます。

```
BTST 0x404(r0), 1
```



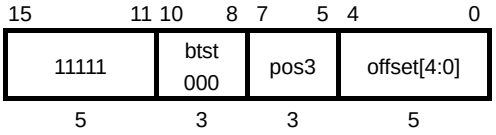
BTST *offset* (fp), *pos3*

Bit Test

動作

t8 ⇐ 31'b 000_0000_0000_0000_0000_0000_0000_0000 ‖ NOT ({zero-extend (*offset*) + (fp)} [*pos3*])

コード



説明

5 ビット *offset* をゼロ拡張して、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータから *pos3* で指定されたビット番号の内容を反転して、汎用レジスタ t8 (r24) の LSB ビットに格納し、残りのビットは 0 が格納されます。

offset は 5 ビットで、扱うことのできる数値の範囲は、0～ + 31 です。

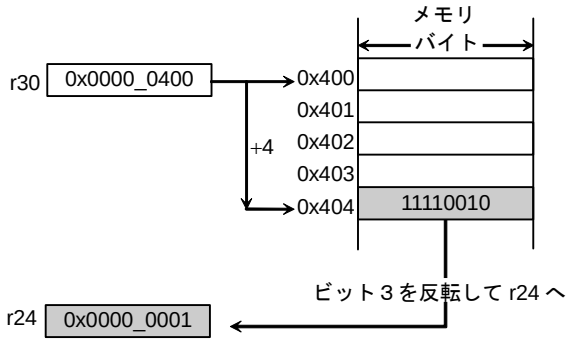
例外

アドレスエラー例外

使用例

fp レジスタ (r30) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r24 に 0x0000_0001 がロードされます。

```
BTST 4(fp), 3
```



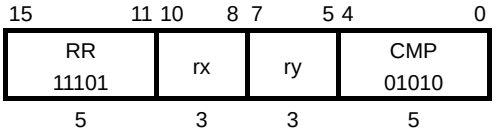
CMP *rx*, *ry*

Compare

動作

if *rx* == *ry* then t8 ⇐ 0; else t8 ⇐ non-zero value

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容の排他的論理和 (XOR) をとり、結果をコンディションコードレジスタ t8 (r24) に格納します。したがって、両者が等しい場合、t8 には 0 が入ります。

例外

なし

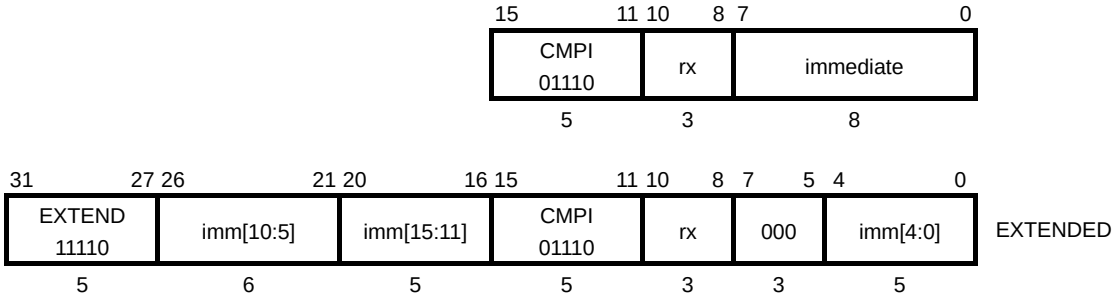
CMPI *rx*, *immediate*

Compare Immediate

動作

if *rx* == 0^{16} || (*immediate*$_{15..0}$) then t8 ⇐ 0; else t8 ⇐ non-zero value

コード



説明

8 ビット *immediate* をゼロ拡張した値と、汎用レジスタ *rx* の内容の排他的論理和 (XOR) をとり、結果をコンディションコードレジスタ t8 (r24) に格納します。したがって、両者が等しい場合、t8 には 0 が入ります。

immediate は 8 ビットで、扱うことのできる数値の範囲は、0~255 です。この範囲外の値を指定すると、CMPI 命令は EXTEND 命令により拡張され、符号なしの 16 ビットの即値 (0~65535) を扱えるようになります。

例外

なし

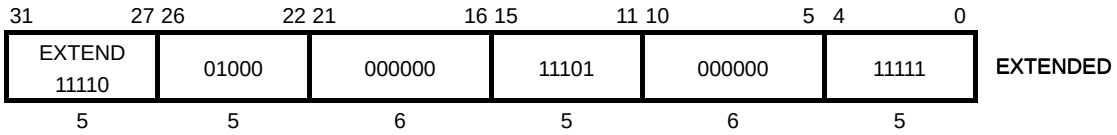
DERET

Debug Exception Return

動作

pc ⇐ DEPC, Debug[DM] ⇐ 0, Debug[IEXI] ⇐ 0

コード



説明

DERET 命令は、デバッグ例外処理から復帰するための命令です。DEPC レジスタの内容をプログラムカウンタ (PC) にロードすることにより実行されます。詳細については、「9.3.6 デバッグ例外からの復帰」を参照してください。

DERET 命令には、分岐遅延スロットがありません。1 命令 (2 命令サイクル) の遅延後、実行されます。

DERET 命令は、DEPC レジスタのビット 0 を PC の ISA モードビット (ビット 0) に復元し、デバッグ例外が実行される前の ISA モードになります。

DERET 命令を、ジャンプまたは分岐遅延スロットに置いてはいけません。

デバッグモードでないとき (Debug レジスタの DM ビットが 0 のとき) は、DERET 命令の動作は保証されません。

通常、デバッグ例外が発生すると、例外の原因となった命令のアドレスが自動的に DEPC レジスタに保存されます。MTC0 命令を使って DEPC レジスタに戻りアドレスを設定したい場合は、デバッグ例外ハンドラで少なくとも 2 つの命令を実行してから、DERET 命令を実行しなければなりません。

例外

なし

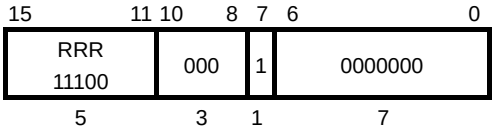
DI

Disable Interrupt

動作

Status[IE] ⇐ 0

コード



説明

DI 命令を実行すると、Status レジスタの IE ビットを 0 にクリアします。

例外

コプロセッサ使用不可例外

DIV *rx*, *ry*

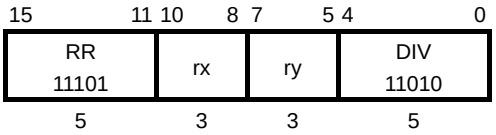
Divide

動作

LO ⇐ *rx* ÷ *ry*;

HI ⇐ *rx* MOD *ry*

コード



説明

汎用レジスタ *rx* の内容を汎用レジスタ *ry* の内容で除算します。両オペランドとも、符号付き整数として扱われます。商は LO レジスタに格納され、剰余は HI レジスタに格納されます。整数オーバフロー例外は発生しません。

除数が 0 の場合、DIV 命令の結果は確定しません。通常、DIV 命令の後に、ゼロ除算とオーバフローを検査する命令を置きます。

除算命令は、専用の除算ユニットで実行されるため、他の命令の実行を並行して継続できます。除算ユニットは、遅延サイクル、例外が起きたときでも、実行を継続します。

除算命令が完了する前に、MFHI、MFLO、MADD、 MADDU、MSUB、MSUBU 命令で除算結果を読もうとすると、パイプラインがストールします(「5.4　除算命令」を参照)。

例外

なし

DIVE *rx*, *ry*

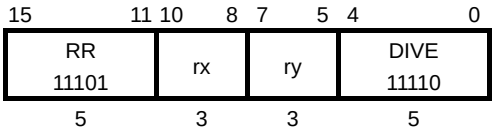
Divide Exception

動作

LO ⇐ *rx* ÷ *ry*

HI ⇐ *rx* MOD *ry*

コード



説明

汎用レジスタ *rx* の内容を汎用レジスタ *ry* の内容で除算します。両オペランドとも、符号付き整数として扱われます。商は LO レジスタに格納され、剰余は HI レジスタに格納されます。

除数が 0 の場合、あるいは演算結果がオーバフローした場合は、整数オーバフロー例外が発生します。

除算命令は、専用の除算ユニットで実行されるため、他の命令の実行を並行して継続できます。除算ユニットは、遅延サイクル、例外が起きたときでも実行を継続します。

除算命令が完了する前に、MFHI、MFLO、MADD、 MADDU、MSUB、MSUBU 命令で除算結果を読もうとすると、パイプラインがストールします (「5.4　除算命令」を参照)。

例外

整数オーバフロー例外

DIVEU *rx*, *ry*

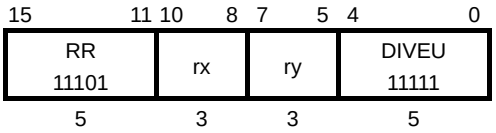
Divide Exception Unsigned

動作

LO ⇐ *rx* ÷ *ry*

HI ⇐ *rx* MOD *ry*

コード



説明

汎用レジスタ *rx* の内容を汎用レジスタ *ry* の内容で除算します。両オペランドとも、符号なし整数として扱われます。商は LO レジスタに格納され、剰余は HI レジスタに格納されます。

除数が 0 の場合は、整数オーバフロー例外が発生します。

除算命令は、専用の除算ユニットで実行されるため、他の命令の実行を並行して継続できます。除算ユニットは、遅延サイクル、例外が起きたときでも実行を継続します。

除算命令が完了する前に、MFHI、MFLO、MADD、 MADDU、MSUB、MSUBU 命令で除算結果を読もうとすると、パイプラインがストールします (「5.4　除算命令」を参照)。

例外

整数オーバフロー例外

DIVU *rx*, *ry*

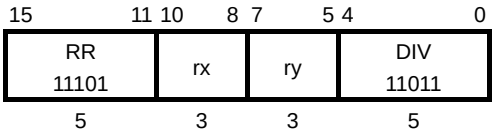
Divide Unsigned

動作

LO ⇐ *rx* ÷ *ry*;

HI ⇐ *rx* MOD *ry*

コード



説明

汎用レジスタ *rx* の内容を汎用レジスタ *ry* の内容で除算します。両オペランドとも、符号なし整数として扱われます。商は LO レジスタに格納され、剰余は HI レジスタに格納されます。整数オーバフロー例外を発生しません。DIV 命令との唯一の違いは、DIVU 命令ではオペランドが符号なし整数として扱われるという点だけです。

例外

なし

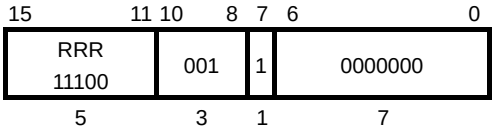
EI

Enable Interrupt

動作

Status[IE] ⇐ 1

コード



説明

EI 命令を実行すると、Status レジスタの IE ビットを 1 にセットします。

例外

コプロセッサ使用不可例外

ERET

Exception Return

動作

if Status[ERL] = 1 then pc ⇐ Error EPC

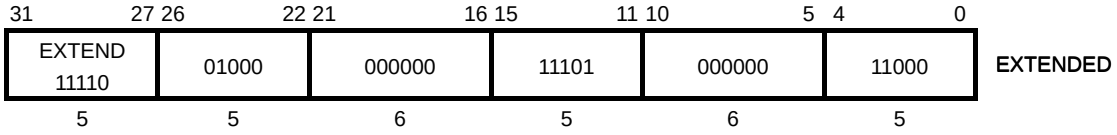
Status[ERL] ⇐ 0

else pc ⇐ EPC

Status[EXL] ⇐ 0

SSCR[CSS] ⇐ SSCR[PSS]

コード



説明

ERET 命令は、割り込み、例外、エラートラップから復帰するための命令です。

ERET 命令には分岐遅延スロットがありません。1 命令 (2 命令サイクル) の遅延後、分岐が実行されます。

ERET 命令は ErrorEPC レジスタのビット 0 を PC の ISA モードビット (ビット 0) に復元し、例外が発生する前の ISA モードになります。

ERET 命令は、ユーザーモードで Status レジスタの CU0 ビットを許可しないで実行すると、コプロセッサ使用不可例外が発生します。MTC0 命令を使って ErrorEPC レジスタあるいは EPC レジスタに戻りアドレスを設定したい場合および Status レジスタの値を書き換えた場合は、例外ハンドラで少なくとも 2 つの命令を実行してから、ERET 命令を実行しなければなりません。

本命令を実行した場合、Status レジスタの ERL ビット＝1 のときは、ErrorEPC レジスタから PC をロードし、同時に ERL ビットが 0 にクリアされます。そうでないときは、EPC レジスタから PC をロードし、同時に EXL ビットが 0 にクリアされます。

本命令を実行した場合、SSCR レジスタの PSS フィールドの値が CSS フィールドに書かれます。

本命令自体をジャンプ命令または分岐命令の遅延スロットに置いてはなりません。

例外

コプロセッサ使用不可例外

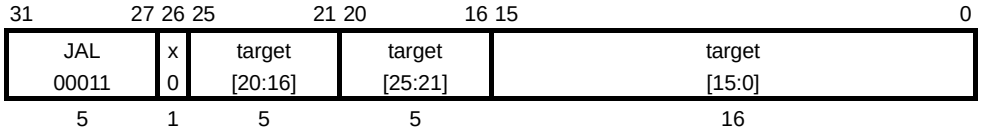
JAL *target*

Jump And Link

動作

ra ⇐ pc + 7; pc ⇐ pc[31:28] ‖ *target* ‖ 00

コード



説明

16 ビット ISA の JAL 命令は、32 ビット長で、1 命令 (2 命令サイクル) の遅延後、無条件にターゲットアドレスにジャンプします。「5.3.3　ジャンプ命令 (16 ビット ISA)」を参照してください。ターゲットアドレスは、ジャンプ遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。26 ビット *target* を 2 ビット左へシフトし、PC+4 の上位 4 ビットと連結した結果がターゲットアドレスになります。プログラムカウンタ (PC) の ISA モードビットは変化しません。

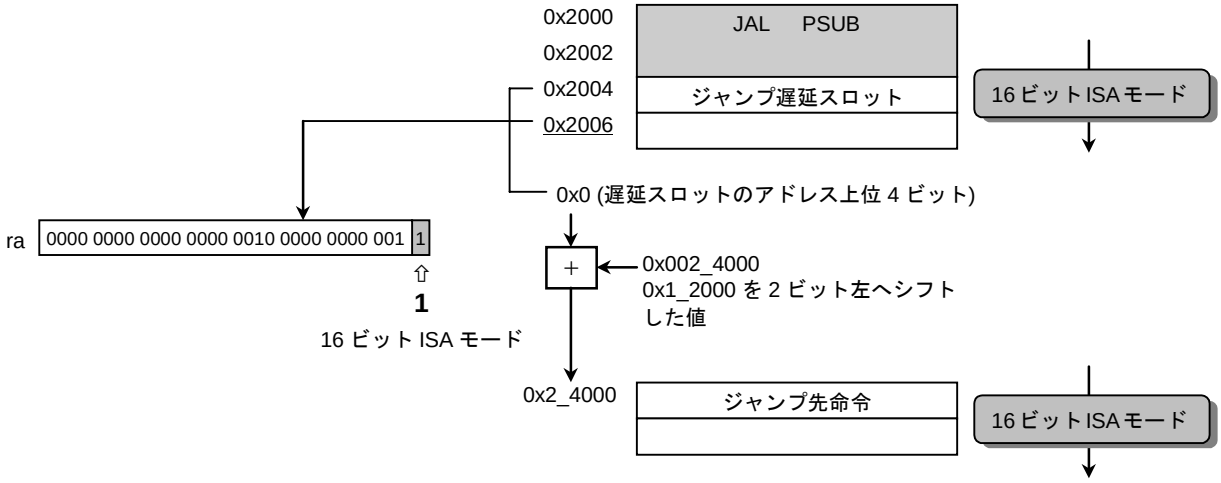
ジャンプ遅延スロットの次の命令のアドレスを、リンクレジスタ ra (r31) に格納します。また、ra の最下位ビットに、ISA モードビット (16 ビット ISA モード = 1) を格納します。

使用例

```
JAL  PSUB
```

上記のジャンプ命令がアドレス 0x2000 にあり、ラベル PSUB が 0x2_4000 に絶対アドレス化される場合、以下に図示するように、PSUB はアセンブラ・リンカにより 0x1_2000 に変換されます。

プログラムの処理は、無条件にアドレス 0x2_4000 にジャンプします。ジャンプ遅延スロット内の命令は、ジャンプの前に実行されます。また、ジャンプ遅延スロットの次の命令のアドレスが、ISA モードビットと共に ra に格納され、ra は 0x0000_2007 になります。


0x2000
0x2002
0x2004
0x2006
JAL　PSUB
ジャンプ遅延スロット
16 ビット ISA モード
0x0 (遅延スロットのアドレス上位 4 ビット)
ra
0000 0000 0000 0000 0010 0000 0000 001
1
1
16 ビット ISA モード
+
0x002_4000
0x1_2000 を 2 ビット左へシフト
した値
0x2_4000
ジャンプ先命令
16 ビット ISA モード

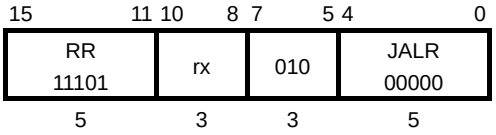
JALR ra, *rx*

Jump And Link Register

動作

ra ⇐ pc + 5; pc ⇐ *rx*

コード



説明

1 命令 (2 命令サイクル) の遅延後、汎用レジスタ ***rx*** の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。***rx*** の最下位ビットの値によって、ISA モードが切り換わります。また、ジャンプ遅延スロットの次の命令のアドレスを、ジャンプ前の ISA モードビットと共にリンクレジスタ ra (r31) に格納します。

32 ビット ISA では、命令はすべてワード境界で位置合わせされなければなりません。そのため、32 ビット ISA モードに移行する場合、ターゲットレジスタ (***rx***) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。

例外

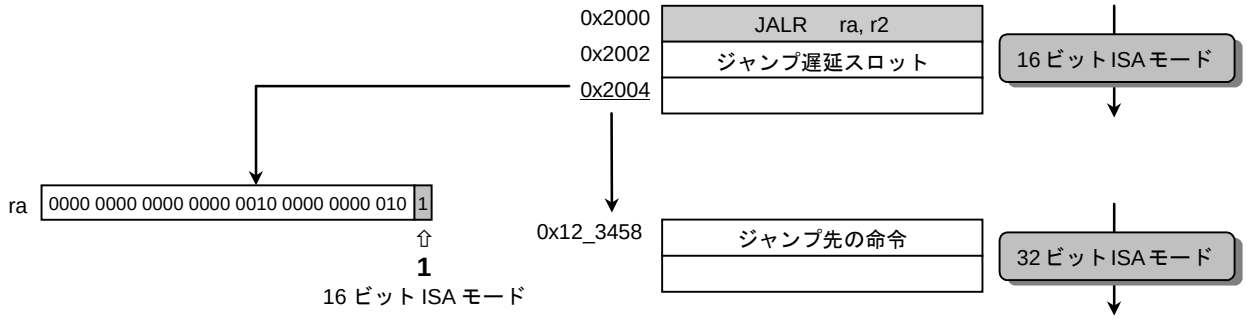
なし

使用例

レジスタ r2 の内容が 0x0012_3458 で、以下のジャンプ命令がアドレス 0x0000_2000 に置かれているとします。

```
JALR ra,r2
```

上記の命令を実行すると、プログラムの処理は、アドレス 0x0012_3458 にジャンプします。レジスタ r2 の最下位ビットは 0 なので、ジャンプ後の ISA モードビットは 0 に変化し、32 ビット ISA モードになります。また、ジャンプ遅延スロットの次の命令のアドレスが、ISA モードビットと共に ra に格納され、ra は 0x0000_2005 になります。


0x2000
JALR　ra, r2
0x2002
ジャンプ遅延スロット
0x2004
16 ビット ISA モード
ra
0000 0000 0000 0000 0010 0000 0000 010
1
1
16 ビット ISA モード
0x12_3458
ジャンプ先の命令
32 ビット ISA モード

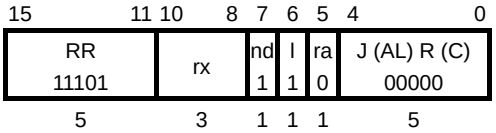
JALRC ra, *rx*

Jump And Link Register, Compact

動作

ra ⇐ pc + 3; pc ⇐ *rx*

コード



説明

1 命令 (2 命令サイクル) の遅延後、汎用レジスタ ***rx*** の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。***rx*** の最下位ビットの値によって、ISA モードが切り換わります。また、本命令はジャンプ遅延スロットを持たないので、本命令の次のアドレスを、ジャンプ前の ISA モードビットと共にリンクレジスタ ra (r31) に格納します。

32 ビット ISA では、命令はすべてワード境界で位置合わせされなければなりません。そのため、32 ビット ISA モードに移行する場合、ターゲットレジスタ (***rx***) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。

例外

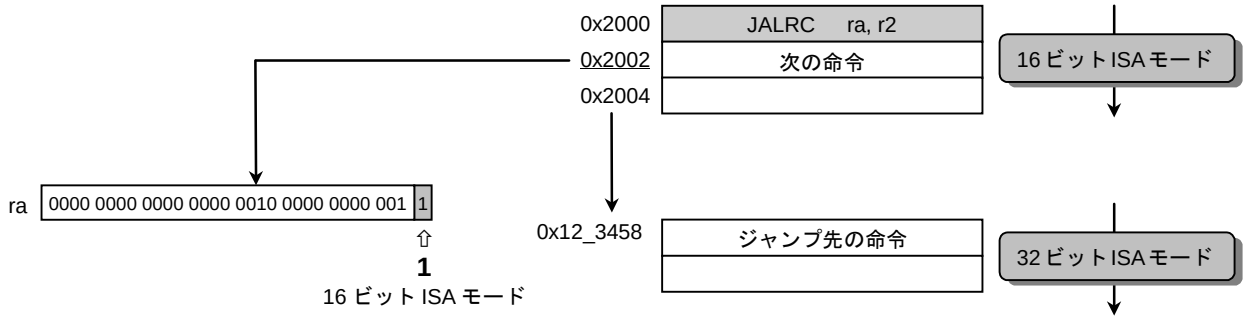
なし

使用例

レジスタ r2 の内容が 0x0012_3458 で、以下のジャンプ命令がアドレス 0x0000_2000 に置かれているとします。

```
JALRC ra,r2
```

上記の命令を実行すると、プログラムの処理は、アドレス 0x0012_3458 にジャンプします。レジスタ r2 の最下位ビットは 0 なので、ジャンプ後の ISA モードビットは 0 に変化し、32 ビット ISA モードになります。また、ジャンプ命令の次の命令のアドレスが、ISA モードビットと共に ra に格納され、ra は 0x0000_2003 になります。


0x2000
JALRC　ra, r2
0x2002
次の命令
0x2004
16 ビット ISA モード
ra
0000 0000 0000 0000 0010 0000 0000 001
1
1
16 ビット ISA モード
0x12_3458
ジャンプ先の命令
32 ビット ISA モード

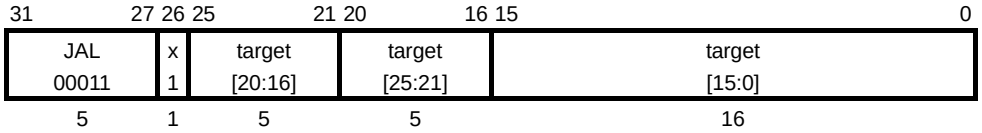
JALX *target*

Jump And Link eXchange

動作

ra ⇐ pc + 7; pc[31:1] ⇐ pc[31:28] ‖ *target* ‖ 00; pc[0] ⇐ NOT pc[0]

コード



説明

16 ビット ISA の JALX 命令は、32 ビット長で、1 命令 (2 命令サイクル) の遅延後、無条件にターゲットアドレスにジャンプします。「5.3.3　ジャンプ命令 (16 ビット ISA)」を参照してください。ターゲットアドレスは、ジャンプ遅延スロット内の命令のアドレス (PC+4) に対して相対的に計算されます。26 ビット *target* を 2 ビット左へシフトし、PC+4 の上位 4 ビットと連結した結果がターゲットアドレスになります。プログラムカウンタ (PC) の ISA モードビットが無条件に変化します。

ジャンプ遅延スロットの次の命令のアドレスを、リンクレジスタ ra (r31) に格納します。また、ra の最下位ビットに、ジャンプ前の ISA モードビットを格納します。

例外

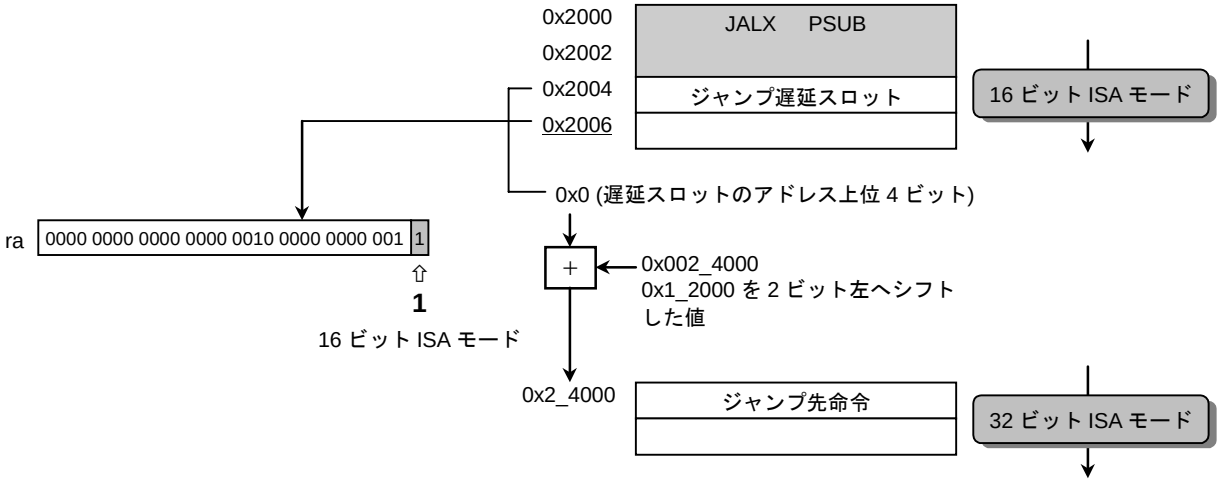
なし

使用例

```
JALX  PSUB
```

上記のジャンプ命令がアドレス 0x0000_2000 にあり、ラベル PSUB が 0x2_4000 に絶対アドレス化される場合、次のページに示すとおり、PSUB はアセンブラ・リンカにより 0x1_2000 に変換されます。

プログラムの処理は、無条件にアドレス 0x2_4000 にジャンプします。ジャンプ遅延スロット内の命令は、ジャンプの前に実行されます。ISA モードは無条件に変化し、32 ビット ISA モードに切り換わります。また、ジャンプ遅延スロットの次の命令のアドレスが、ISA モードビットと共に ra に格納され、ra は 0x0000_2007 になります。


0x2000
0x2002
0x2004
0x2006
JALX　PSUB
ジャンプ遅延スロット
16 ビット ISA モード
0x0 (遅延スロットのアドレス上位 4 ビット)
ra
0000 0000 0000 0000 0010 0000 0000 001
1
1
16 ビット ISA モード
+
0x002_4000
0x1_2000 を 2 ビット左へシフト
した値
0x2_4000
ジャンプ先命令
32 ビット ISA モード

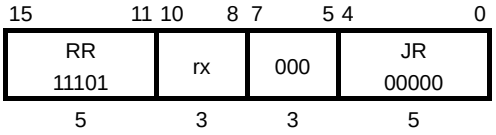
JR *rx*

Jump Register

動作

pc ⇐ *rx*

コード



説明

1 命令 (2 命令サイクル) の遅延後、汎用レジスタ ***rx*** の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。***rx*** の最下位ビットの値によって ISA モードが切り換わります。

32 ビット ISA では、命令はすべてワード境界に位置合わせされていなければなりません。そのため、32 ビット ISA モードにジャンプするとき、ターゲットレジスタ (***rx***) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、プロセッサがジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。

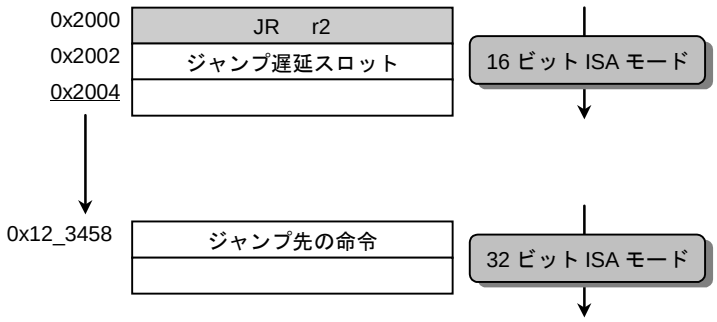
例外

なし

使用例

レジスタ r2 の値が 0x0012_3458 の場合、以下の命令を実行すると、図示したようにプログラムの処理がアドレス 0x0012_3458 へ移ります。r2 の最下位ビットはクリアされ、32 ビット ISA モードに切り換わります。ジャンプ遅延スロット内の命令はジャンプの前に実行されます。

```
JR r2
```



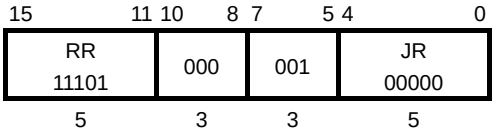
JR ra

Jump Register

動作

pc ⇐ ra

コード



説明

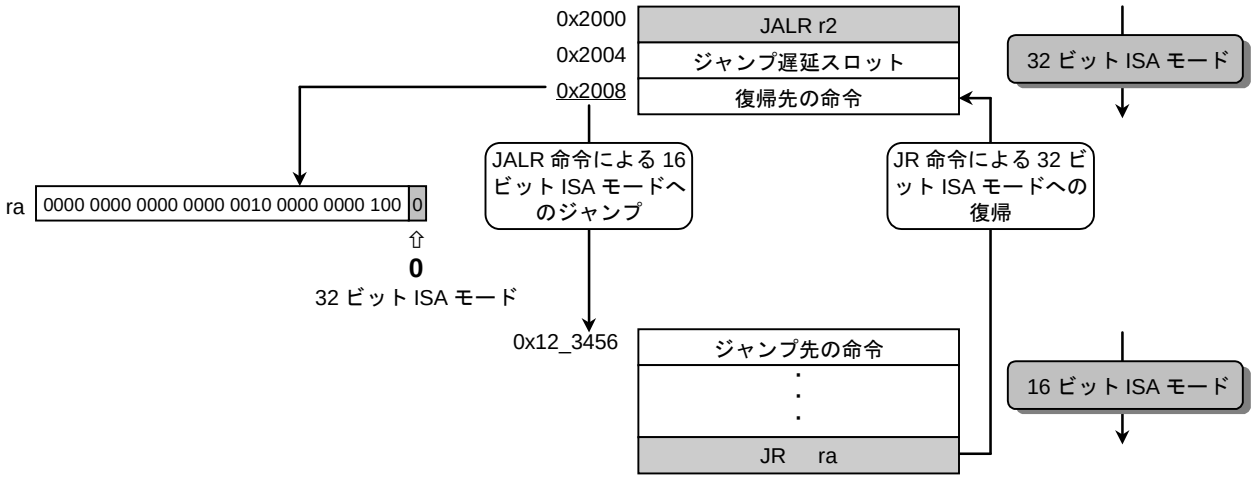
1 命令 (2 命令サイクル) の遅延後、リンクレジスタ ra (r31) の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。ra の最下位ビットの値によって ISA モードが切り換わります。

例外

なし

使用例

以下の例では、32 ビットのルーチンで JALR 命令により 16 ビット ISA モードに切り換えています。そして、16 ビットのルーチンの最後で、JR 命令により戻りアドレスをリンクレジスタ ra (r31) からプログラムカウンタ (PC) に復元しています。32 ビットの JALR 命令により、ISA モードが ra の最下位ビットに保存されているので、16 ビットルーチンの終わりで JR 命令を実行すると、32 ビット ISA モードに戻ります。



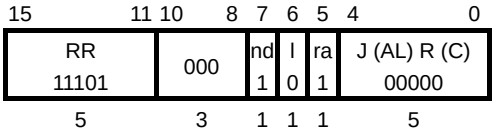
JRC ra

Jump Register ra, Compact

動作

pc ⇐ ra

コード



説明

1 命令 (2 命令サイクル) の遅延後、リンクレジスタ ra (r31) の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。ra の最下位ビットの値によって ISA モードが切り換わります。

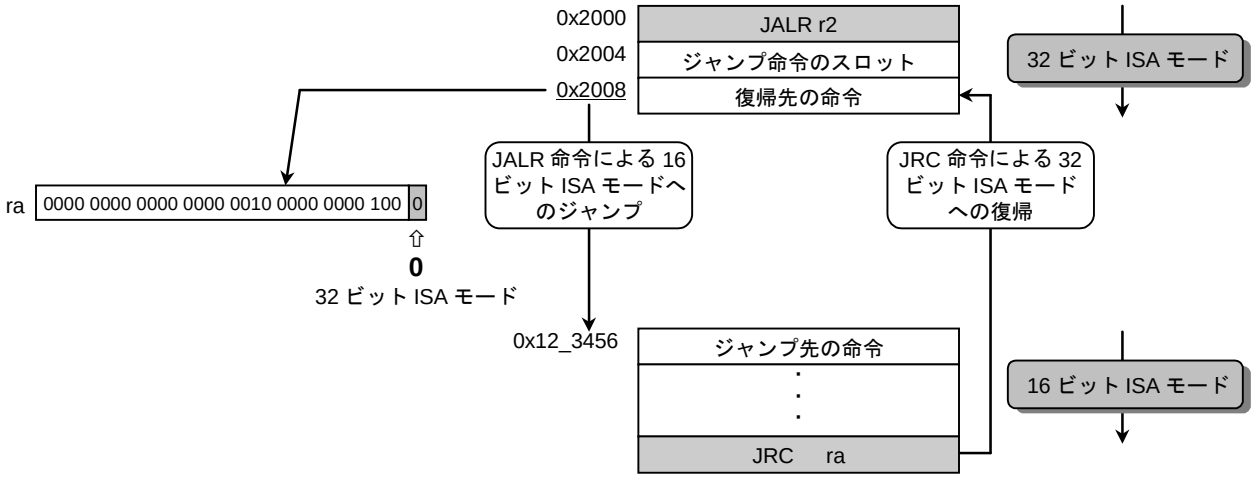
本命令は、ジャンプ遅延スロットを持っていません。

例外

なし

使用例

以下の例では、32 ビットのルーチンで JALR 命令により 16 ビット ISA モードに切り換えています。そして、16 ビットのルーチンの最後で、JRC 命令により戻りアドレスをリンクレジスタ ra (r31) からプログラムカウンタ (PC) に復元しています。32 ビットの JALR 命令により、ISA モードが ra の最下位ビットに保存されているので、16 ビットルーチンの終わりで JRC 命令を実行すると、32 ビット ISA モードに戻ります。



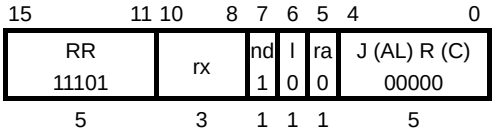
JRC *rx*

Jump Register, Compact

動作

pc ⇐ *rx*

コード



説明

1 命令 (2 命令サイクル) の遅延後、汎用レジスタ ***rx*** の最下位ビットを 0 にマスクしたアドレスに無条件にジャンプします。***rx*** の最下位ビットの値によって ISA モードが切り換わります。

本命令は、ジャンプ遅延スロットを持っていません。

32 ビット ISA では、命令はすべてワード境界に位置合わせされていなければなりません。そのため、32 ビット ISA モードにジャンプするとき、ターゲットレジスタ (***rx***) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、プロセッサがジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。

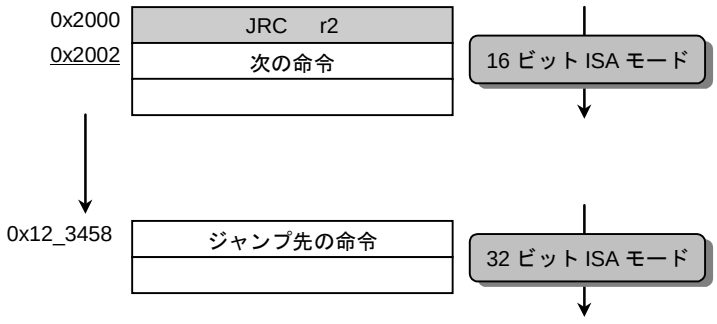
例外

なし

使用例

レジスタ r2 の値が 0x0012_3458 の場合、以下の命令を実行すると、図示したようにプログラムの処理がアドレス 0x0012_3458 へ移ります。r2 の最下位ビットはクリアされ、32 ビット ISA モードに切り換わります。JRC 命令の次の命令は実行されません。

```
JRC r2
```



LB *ry*, *offset* (*base*)

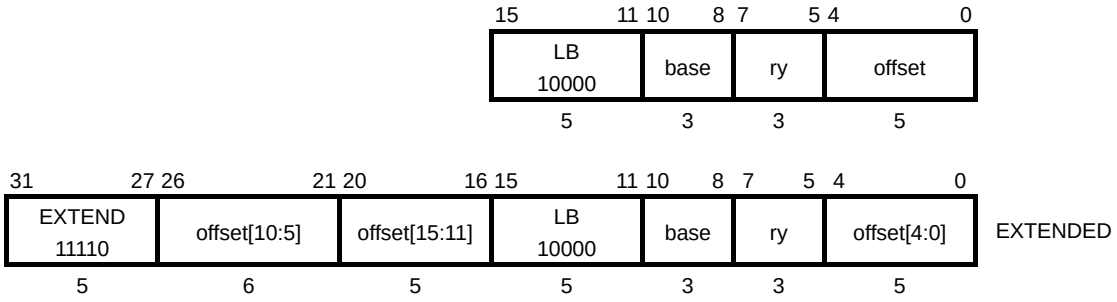
Load Byte

動作

ry = {zero-extend (*offset*) + (*base*)}

(EXTENDED)　*ry* = {sign-extend (*offset*) + (*base*)}

コード



説明

5 ビット *offset* をゼロ拡張して、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータを符号拡張して、汎用レジスタ *ry* にロードします。

offset は 5 ビットで、扱うことのできる数値の範囲は、0~31 です。この範囲外の値を指定すると、LB 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

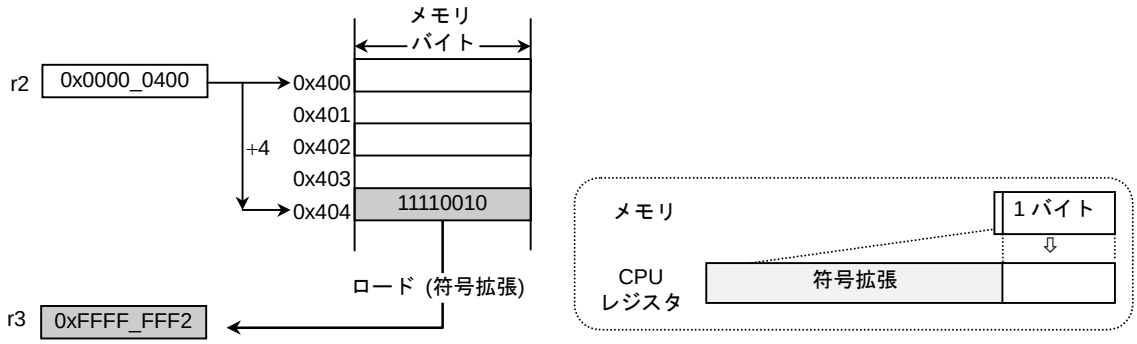
例外

アドレスエラー例外

使用例

レジスタ r2 の値が 0x0000_0400 で、アドレス 0x404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r3 に 0xFFFF_FFF2 がロードされます。

```
LB r3,4(r2)
```


メモリ
バイト
r2
0x0000_0400
0x400
0x401
+4
0x402
0x403
0x404
11110010
ロード (符号拡張)
r3
0xFFFF_FFF2
メモリ
1 バイト
CPU
レジスタ
符号拡張

LBU *ry*, *offset* (*base*)

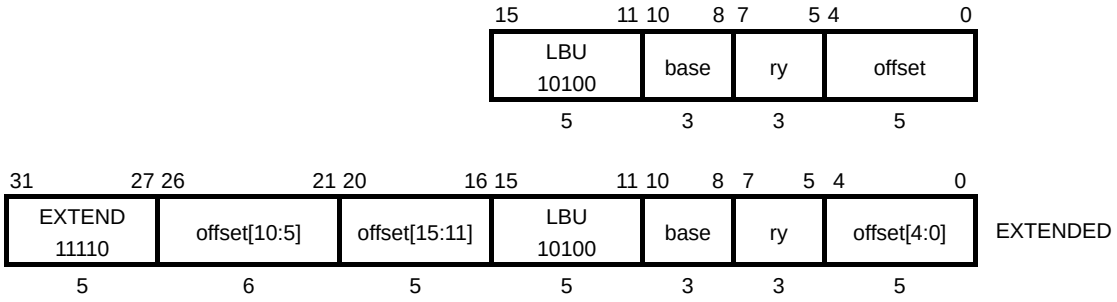
Load Byte Unsigned

動作

ry = {zero-extend (*offset*) + (*base*)}

(EXTENDED)　　*ry* = {sign-extend (*offset*) + (*base*)}

コード



説明

5 ビット *offset* をゼロ拡張して、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータをゼロ拡張して、汎用レジスタ *ry* にロードします。

offset は 5 ビットで、扱うことのできる数値の範囲は、0~31 です。この範囲外の値を指定すると、LBU 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

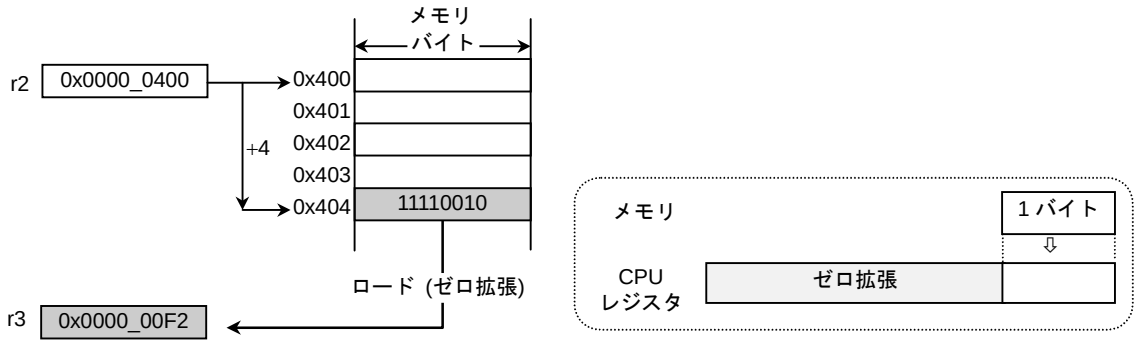
例外

アドレスエラー例外

使用例

レジスタ r2 の値が 0x0000_0400 で、アドレス 0x404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r3 に 0x0000_00F2 がロードされます。

```
LBU r3,4(r2)
```


メモリ
バイト
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
11110010
ロード (ゼロ拡張)
r3
0x0000_00F2
メモリ
1 バイト
CPU
レジスタ
ゼロ拡張

LBU *ry*, *offset* (fp)

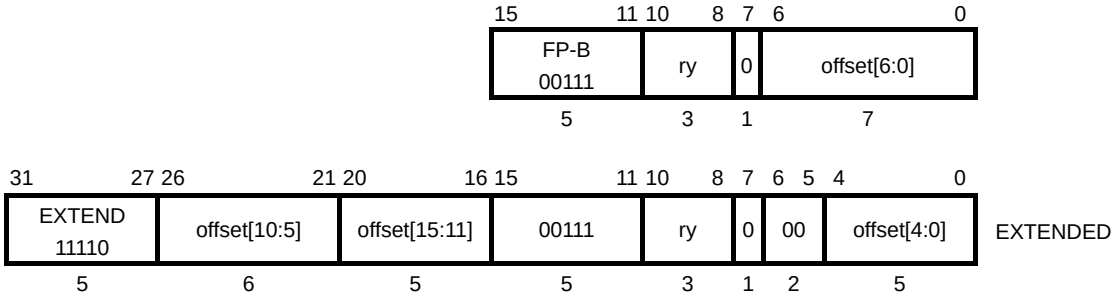
Load Byte Unsigned

動作

ry = {zero-extend (*offset*) + (fp)}

(EXTENDED)　*ry* = {sign-extend (*offset*) + (fp)}

コード



説明

7 ビット *offset* をゼロ拡張して、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータをゼロ拡張して、汎用レジスタ *ry* にロードします。

offset は 7 ビットで、扱うことのできる数値の範囲は、0~127 です。この範囲外の値を指定すると、LBU 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

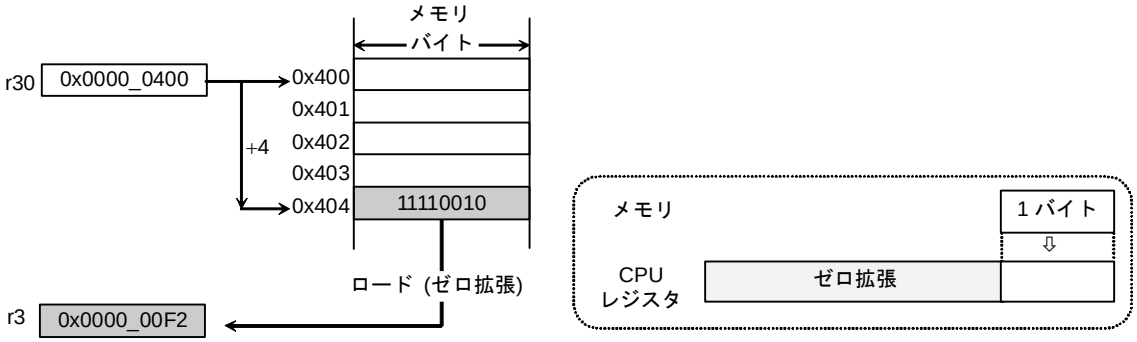
例外

アドレスエラー例外

使用例

fp レジスタ (r30) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r3 に 0x0000_00F2 がロードされます。

```
LBU r3,4(fp)
```


メモリ
バイト
r30
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
11110010
ロード (ゼロ拡張)
r3
0x0000_00F2
メモリ
1 バイト
CPU
レジスタ
ゼロ拡張

LBU *ry*, *offset* (sp)

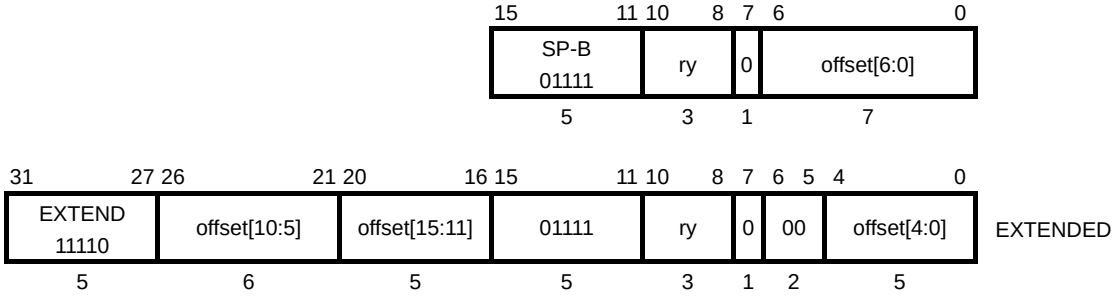
Load Byte Unsigned

動作

ry = {zero-extend (*offset*) + (sp)}

(EXTENDED)　*ry* = {sign-extend (*offset*) + (sp)}

コード



説明

7 ビット *offset* をゼロ拡張して、sp レジスタ (r29) の内容に加算することにより、実効アドレス (EA) を生成します。EA でアドレス指定されたメモリ中のバイトデータをゼロ拡張して、汎用レジスタ *ry* にロードします。

offset は 7 ビットで、扱うことのできる数値の範囲は、0~127 です。この範囲外の値を指定すると、LBU 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

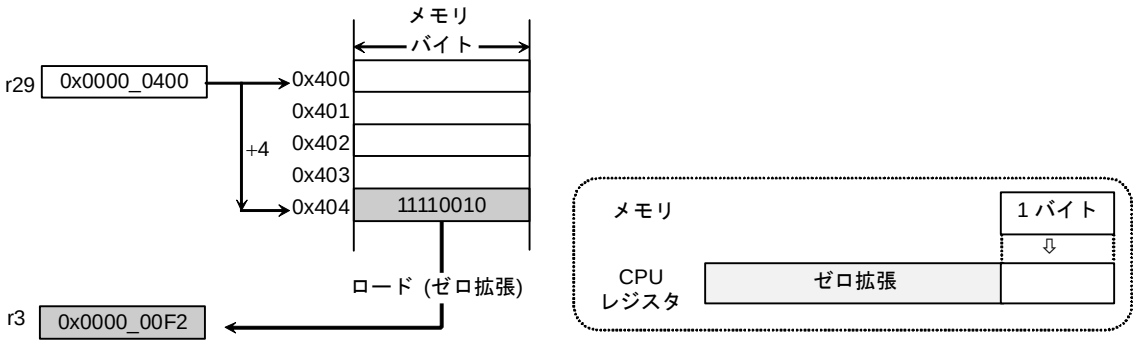
例外

アドレスエラー例外

使用例

sp レジスタ (r29) の値が 0x0000_0400 で、アドレス 0x0404 の内容が 0xF2 の場合、以下の命令を実行すると、レジスタ r3 に 0x0000_00F2 がロードされます。

```
LBU r3,4(sp)
```


メモリ
バイト
r29
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
11110010
ロード (ゼロ拡張)
r3
0x0000_00F2
メモリ
1 バイト
CPU
レジスタ
ゼロ拡張

LH *ry*, *offset* (*base*)

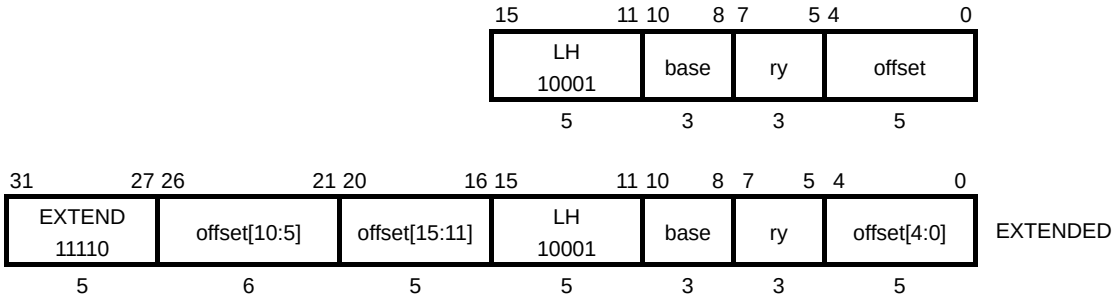
Load Halfword

動作

ry ⇐ {zero-extend (*offset* || 0) + (*base*)}

(EXTENDED)　　*ry* ⇐ {sign-extend (*offset*) + (*base*)}

コード



説明

5 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、汎用レジスタ *base* に加算することにより、実効アドレス (EA) を生成します。EA によってアドレス指定されたメモリ中のハーフワードデータを符号拡張し、汎用レジスタ *ry* にロードします。

offset は 5 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、2 刻みで 0~62 になります。この範囲外の値を指定すると、LH 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*offset* はシフトされません。

例外

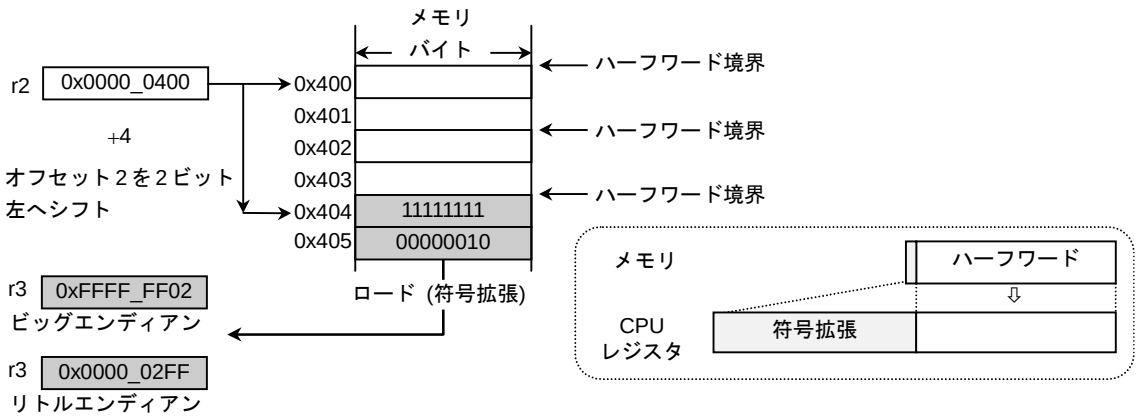
アドレスエラー例外

使用例

```
LH r3,4(r2)
```

レジスタ r2 の内容が 0x0000_0400 で、アドレス 0x404 と 0x405 の内容がそれぞれ 0xFF と 0x02 であるとします。オフセット値は 1 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、2 (2 進数 0010) に変換されます。したがって、命令コードは 0x8A62 になります。

上記の命令を実行すると、レジスタ r3 には、ビッグエンディアンのときは 0xFFFF_FF02 がロードされ、リトルエンディアンのときは 0x0000_02FF がロードされます。


メモリ
バイト
ハーフワード境界
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
11111111
00000010
+4
オフセット 2 を 2 ビット
左へシフト
ハーフワード境界
ハーフワード境界
ロード (符号拡張)
r3
0xFFFF_FF02
ビッグエンディアン
r3
0x0000_02FF
リトルエンディアン
メモリ
ハーフワード
CPU
レジスタ
符号拡張

LHU *ry*, *offset* (*base*)

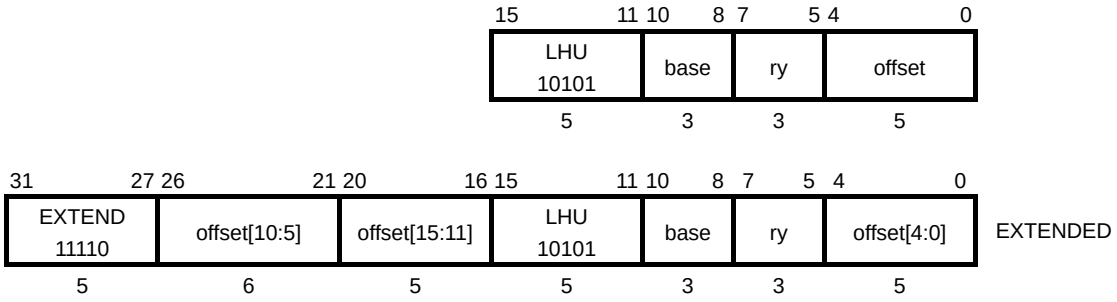
Load Halfword Unsigned

動作

ry ⇐ {zero-extend (*offset* || 0) + (*base*)}

(EXTENDED)　　*ry* ⇐ {sign-extend (*offset*) + (*base*)}

コード



説明

5 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。EA によってアドレス指定されたハーフワードデータをゼロ拡張し、汎用レジスタ *ry* にロードします。

offset は 5 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、2 刻みで 0~62 になります。この範囲外の値を指定すると、LHU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*offset* はシフトされません。

例外

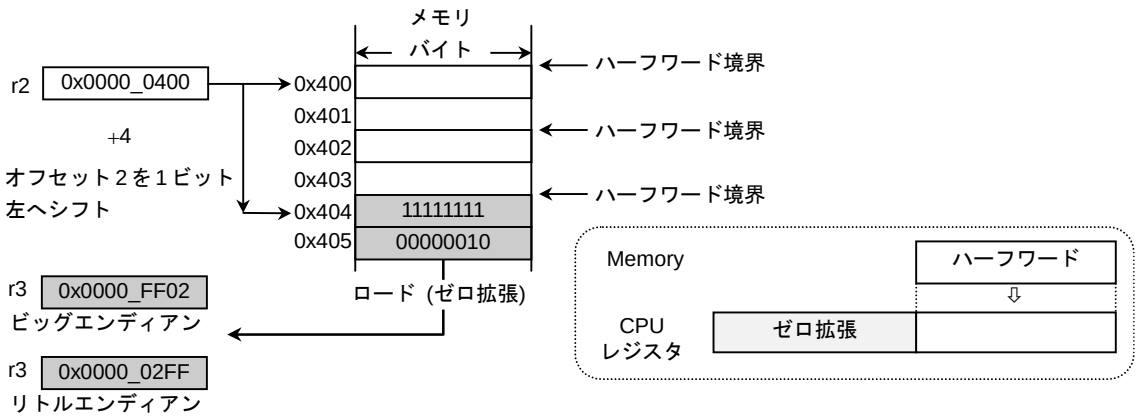
アドレスエラー例外

使用例

```
LHU r3,4(r2)
```

レジスタ r2 の内容が 0x0000_0400 で、アドレス 0x404 と 0x405 の内容がそれぞれ 0xFF と 0x02 であるとします。オフセット値は 1 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、2 (2 進数 0010) に変換されます。したがって、命令コードは 0xAA62 になります。

上記の命令を実行すると、レジスタ r3 には、ビッグエンディアンのときは 0x0000_FF02 がロードされ、リトルエンディアンのときは 0x0000_02FF がロードされます。


メモリ
バイト
ハーフワード境界
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
11111111
00000010
+4
オフセット 2 を 1 ビット
左へシフト
ロード (ゼロ拡張)
r3
0x0000_FF02
ビッグエンディアン
r3
0x0000_02FF
リトルエンディアン
Memory
ハーフワード
CPU
レジスタ
ゼロ拡張

LHU *ry*, *offset* (fp)

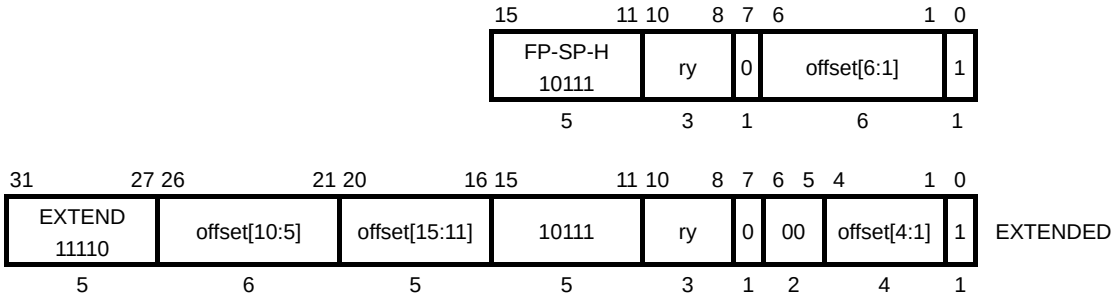
Load Halfword Unsigned

動作

ry ⇐ {zero-extend (*offset* || 0) + (fp)}

(EXTENDED)　*ry* ⇐ {sign-extend (*offset* || 0) + (fp)}

コード



説明

6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。EA によってアドレス指定されたハーフワードデータをゼロ拡張し、汎用レジスタ *ry* にロードします。

offset は 6 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、2 刻みで 0~126 になります。この範囲外の値を指定すると、LHU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32766) を扱えるようになります。この場合、*offset* は 1 ビットシフトされます。

例外

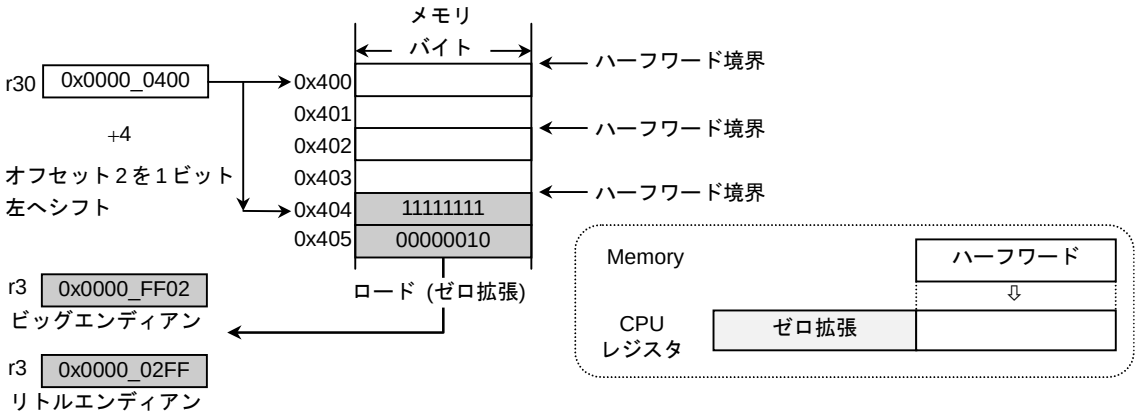
アドレスエラー例外

使用例

```
LHU r3,4(fp)
```

fp レジスタ (r30) の内容が 0x0000_0400 で、アドレス 0x0404 と 0x405 の内容がそれぞれ 0xFF と 0x02 であるとします。オフセット値は 1 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、2 (2 進数 0010) に変換されます。したがって、命令コードは 0xBB05 になります。

上記の命令を実行すると、レジスタ r3 には、ビッグエンディアンのときは 0x0000_FF02 がロードされ、リトルエンディアンのときは 0x0000_02FF がロードされます。


メモリ
バイト
ハーフワード境界
r30
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
11111111
00000010
+4
オフセット 2 を 1 ビット
左へシフト
ロード (ゼロ拡張)
r3
0x0000_FF02
ビッグエンディアン
r3
0x0000_02FF
リトルエンディアン
Memory
ハーフワード
CPU
レジスタ
ゼロ拡張

LHU *ry*, *offset* (sp)

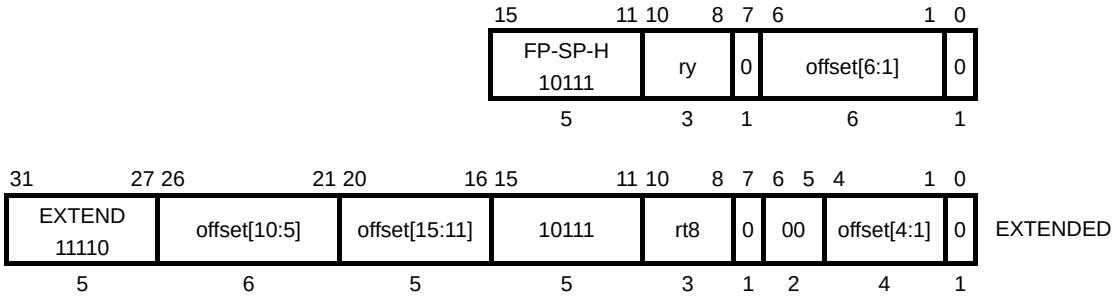
Load Halfword Unsigned

動作

ry ⇐ {zero-extend (*offset* || 0) + (sp)}

(EXTENDED)　*ry* ⇐ {sign-extend (*offset* || 0) + (sp)}

コード



説明

6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、sp レジスタ *(r29)* の内容に加算することにより、実効アドレス (EA) を生成します。EA によってアドレス指定されたハーフワードデータをゼロ拡張し、汎用レジスタ *ry* にロードします。

offset は 6 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、2 刻みで 0~126 になります。この範囲外の値を指定すると、LHU 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32766) を扱えるようになります。この場合、*offset* は 1 ビットシフトされます。

例外

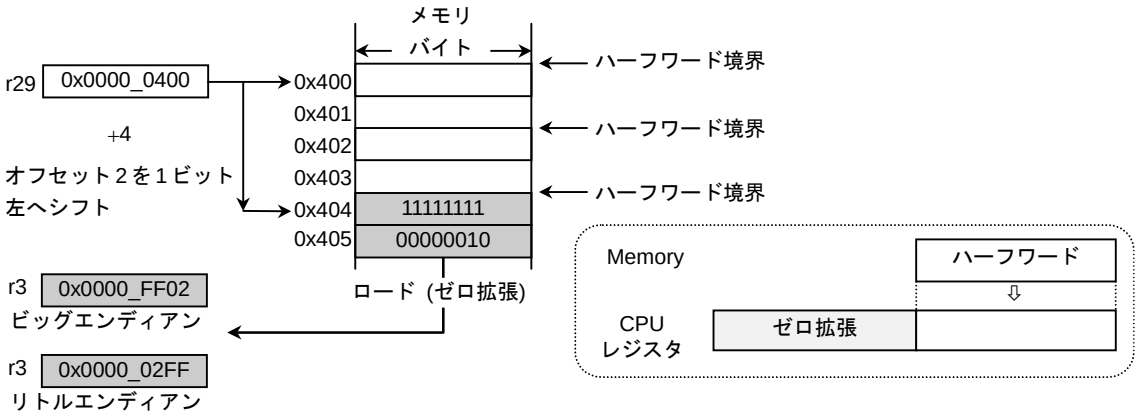
アドレスエラー例外

使用例

```
LHU r3,4(sp)
```

sp レジスタ (r29) の内容が 0x0000_0400 で、アドレス 0x0404 と 0x0405 の内容がそれぞれ 0xFF と 0x02 であるとします。オフセット値は 1 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、2 (2 進数 0010) に変換されます。したがって、命令コードは 0xBB04 になります。

上記の命令を実行すると、レジスタ r3 には、ビッグエンディアンのときは 0x0000_FF02 がロードされ、リトルエンディアンのときは 0x0000_02FF がロードされます。


メモリ
バイト
ハーフワード境界
r29
0x0000_0400
0x400
0x401
0x402
0x403
0x404
0x405
11111111
00000010
+4
オフセット 2 を 1 ビット
左へシフト
ハーフワード境界
ハーフワード境界
r3
0x0000_FF02
ビッグエンディアン
ロード (ゼロ拡張)
r3
0x0000_02FF
リトルエンディアン
Memory
ハーフワード
CPU
レジスタ
ゼロ拡張

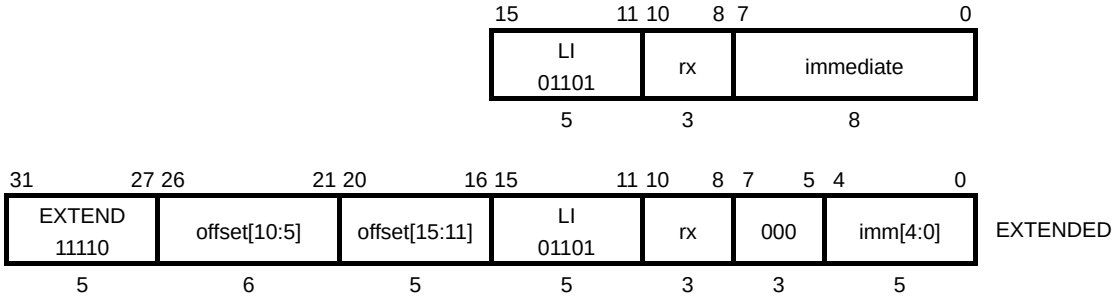
LI *rx*, *immediate*

Load Immediate

動作

$rx \Leftarrow 0^{16} \| (immediate_{15..0})$

コード



説明

8 ビット *immediate* をゼロ拡張し、汎用レジスタ *rx* に格納します。

immediate フィールドは 8 ビットで、扱うことのできる数値の範囲は、0~255 です。この範囲外の値を指定すると、EXTEND 命令により拡張され、符号なしの 16 ビットの即値 (0~65535) が扱えるようになります。

例外

なし

使用例

以下の命令を実行すると、レジスタ r3 に 0x0000_0012 がロードされます。

```
LI r3,0x12
```

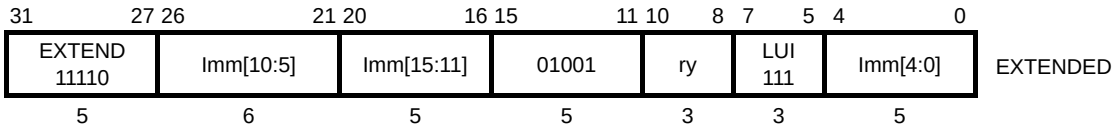
LUI *ry*, *immediate*

Load Upper Immediate

動作

ry ⇐ *immediate* ‖ 0x0000

コード



説明

16 ビット *immediate* を 16 ビット左へシフトし、下位 16 ビットのを 0 で埋めた値を汎用レジスタ *ry* にロードします。

例外

なし

使用例

以下の命令は、レジスタ r4 に 0x1234_0000 をロードします。

```
LUI r4,0x1234
```

LW *rx*, *offset* (pc)

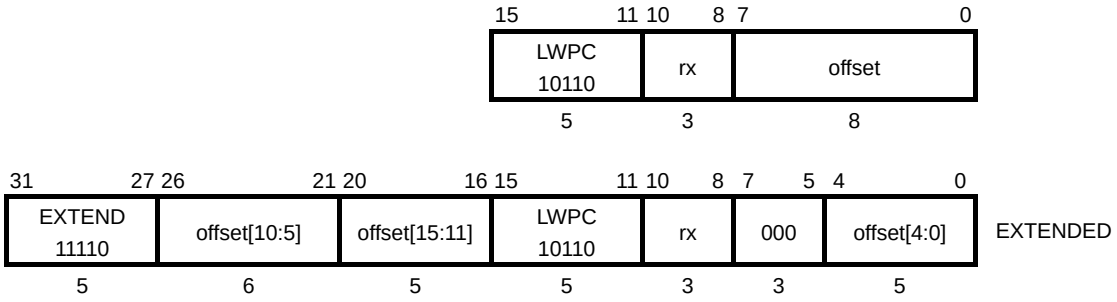
Load Word

動作

rx ⇐ {zero-extend (*offset* || 00) + (マスクベース PC)}

(EXTENDED)　　*rx* ⇐ {sign-extend (*offset*) + (マスクベース PC)}

コード



説明

8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、下位 2 ビットを 0 にマスクしたプログラムカウンタ (PC) 値に加算した結果が、実効アドレス (EA) になります。EA によってアドレス指定された 32 ビットの定数を、汎用レジスタ *rx* にロードします。

この命令により、32 ビットの定数をコード中に埋め込むことができます。この命令の近くに置かれた命令は、1 命令でこの定数を参照できます。

offset は 8 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~1020 になります。この範囲外の値を指定すると、LW 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。また、PC 相対アドレスを計算し、その結果を汎用レジスタに格納するための専用の命令 (ADDIUPC) も用意されています。

この命令では、PC 値をベース値として使用するので、その値をベース PC 値といいます。また、下位 2 ビットを 0 にマスクしたベース PC 値を、マスクベース PC 値といいます。命令が遅延スロット内に置かれているか、EXTEND 命令によって拡張されているかどうかによって、ベース PC 値は、以下のように異なります。

LW 命令の置かれている PC アドレス	ベース PC
JR・JALR 命令の遅延スロット内にある	JR・JALR 命令のアドレス
JAL・JALX 命令の遅延スロット内にある	JAL・JALX 命令の上位ハーフワードのアドレス
拡張されている	EXTEND コードのアドレス
拡張されていない (遅延スロット外)	LWPC 命令のアドレス

例外

アドレスエラー例外

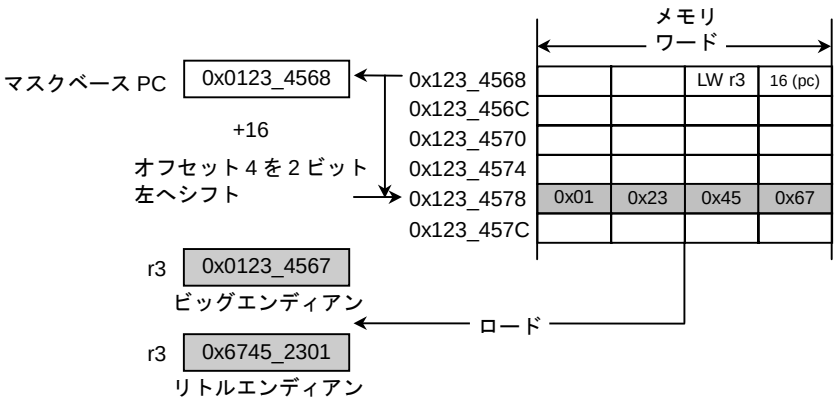
使用例

マスクベース PC がアドレス 0x0123_4568 を示し、アドレス 0x1234_5678~0x0123_457B の内容がそれぞれ 0x01、0x23、0x45、0x67 であるとします。

```
LW r3,16(pc)
```

以下に図示するように、オフセット値は 2 ビット左へシフトされるので、指定されたオフセット値 (16 = 2 進数 0001_0000) は、アセンブラ・リンカにより、コード 4 (2 進数 0000_0100) に変換されます。したがって、上記のロード命令の命令コードは 0xB304 になります。

上記の命令を実行すると、レジスタ r3 には、ビッグエンディアンのときは 0x123_4567 がロードされ、リトルエンディアンのときは 0x6745_2301 がロードされます。



LW *rx*, *offset* (sp)

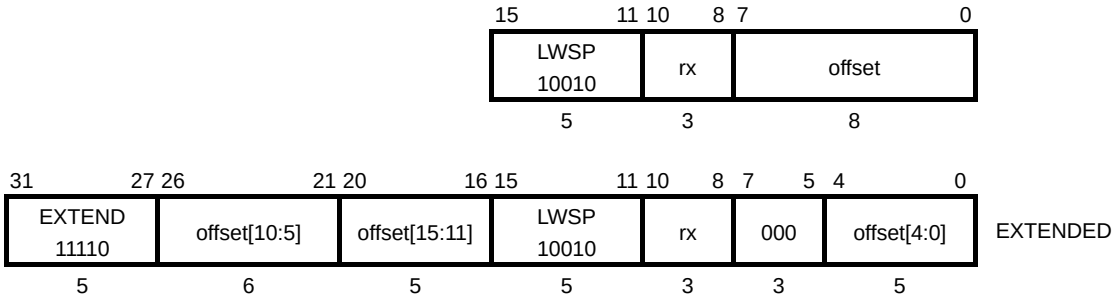
Load Word

動作

rx ⇐ {zero-extend (*offset* || 00) + (sp)}

(EXTENDED)　*rx* ⇐ {sign-extend (*offset*) + (sp)}

コード



説明

8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、スタックポインタレジスタ sp (r29) の内容に加算した結果が実効アドレス (EA) になります。EA によってアドレス指定されたワードデータを汎用レジスタ *rx* に格納します。

offset は 8 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~1020 になります。この範囲外の値を指定すると、LW 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。

例外

アドレスエラー例外

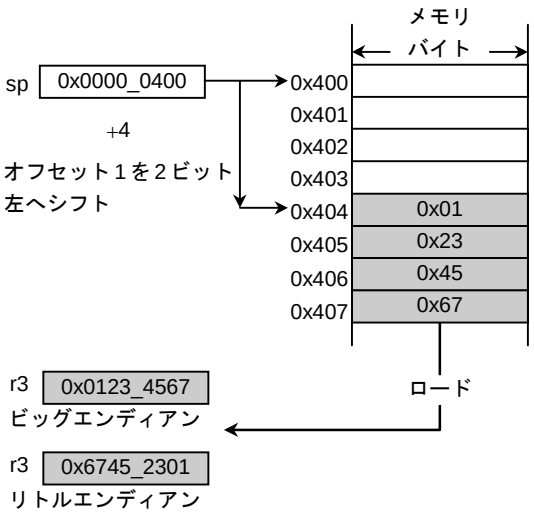
使用例

sp レジスタがアドレス 0x0000_0400 を示し、アドレス 0x0404~0x0407 の内容が、0x01、0x23、0x45、0x67 であるとします。

```
LW r3,4(sp)
```

以下に図示するように、オフセット値は 2 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、1 (2 進数 0001) に変換されます。したがって、上記の命令コードは 0x9301 になります。

上記の命令を実行するとレジスタ r3 には、ビッグエンディアンのときは 0x0123_4567 がロードされ、リトルエンディアンのときは 0x6745_2301 がロードされます。


メモリ
バイト
sp
0x0000_0400
+4
オフセット 1 を 2 ビット
左へシフト
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ロード
r3
0x0123_4567
ビッグエンディアン
r3
0x6745_2301
リトルエンディアン

LW *ry*, *offset* (*base*)

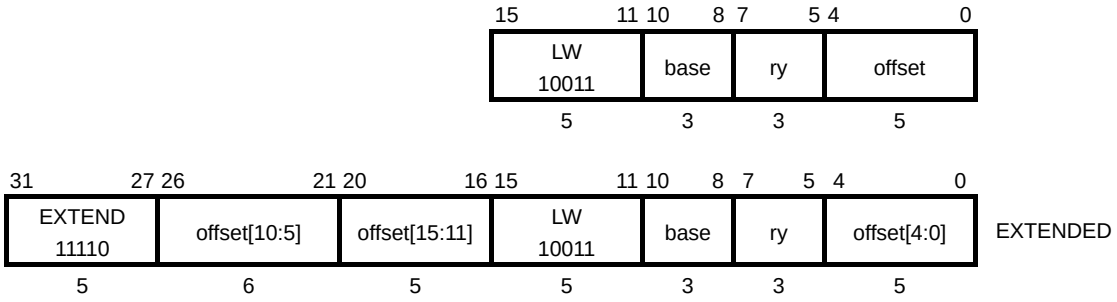
Load Word

動作

$ry \Leftarrow \{\text{zero-extend}\ (offset \parallel 00) + (base)\}$

(EXTENDED)　$ry \Leftarrow \{\text{sign-extend}\ (offset) + (base)\}$

コード



説明

5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成し、EA によってアドレス指定されたワードデータを汎用レジスタ *ry* にロードします。

offset は 5 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~124 になります。この範囲外の値を指定すると、LW 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*offset* はシフトされません。

例外

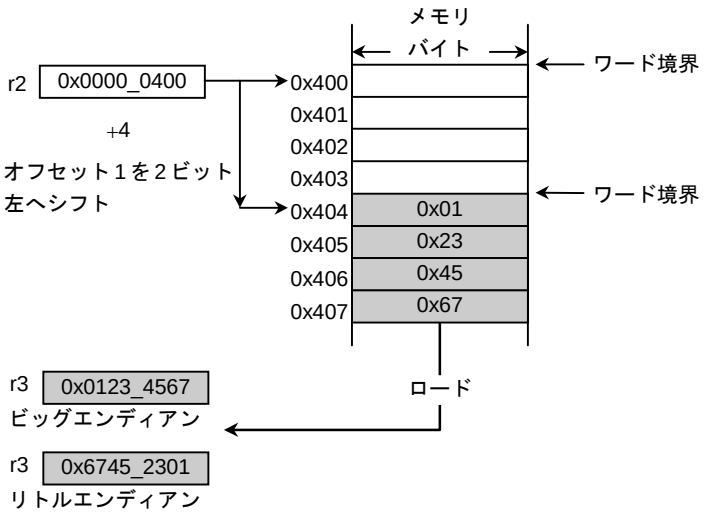
アドレスエラー例外

使用例

```
LW r3,4(r2)
```

レジスタ r2 の内容が 0x0000_0400 で、アドレス 0x404~0x407 の内容がそれぞれ 0x01、0x23、0x45、0x67 であるとします。オフセット値は 2 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、1 (2 進数 0001) に変換されます。したがって、命令コードは 0x9A61 になります。

上記の命令を実行すると、レジスタ r3 には、ビッグエンディアンのときは 0x0123_4567 がロードされ、リトルエンディアンのときは 0x6745_2301 がロードされます。


メモリ
バイト
ワード境界
r2
0x0000_0400
0x400
0x401
0x402
0x403
ワード境界
+4
オフセット 1 を 2 ビット
左へシフト
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ロード
r3
0x0123_4567
ビッグエンディアン
r3
0x6745_2301
リトルエンディアン

LW *ry*, *offset* (fp)

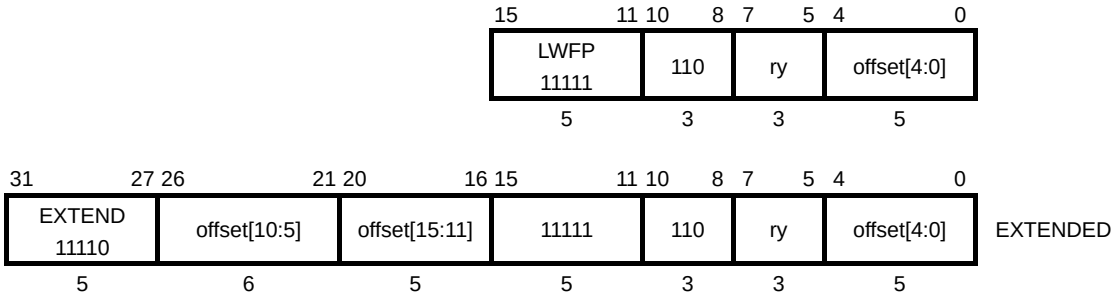
Load Word

動作

ry ⇐ {zero-extend (*offset* || 00) + (fp)}

(EXTENDED)　　*ry* ⇐ {sign-extend (*offset*) + (fp)}

コード



説明

5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、fp レジスタ (r30) の内容に加算した結果が実効アドレス (EA) になります。EA によってアドレス指定されたワードデータを汎用レジスタ *ry* に格納します。

offset は 5 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~124 になります。この範囲外の値を指定すると、LW 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。

例外

アドレスエラー例外

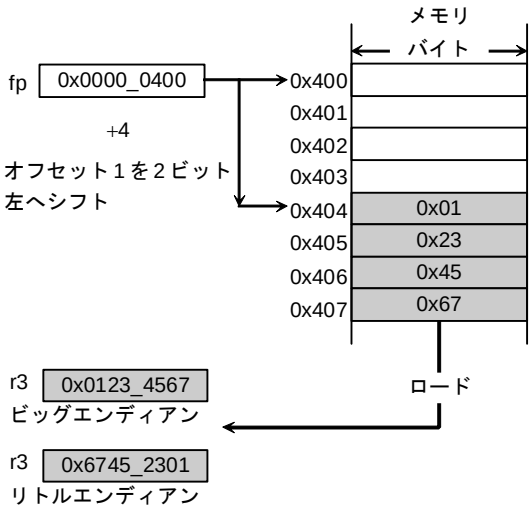
使用例

fp レジスタ (r30) がアドレス 0x0000_0400 を示し、アドレス 0x0404~0x0407 の内容が、0x01、0x23、0x45、 0x67 であるとします。

```
LW r3,4(fp)
```

以下に図示するように、オフセット値は 2 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、1 (2 進数 0001) に変換されます。したがって、上記の命令コードは 0xFE61 になります。

上記の命令を実行するとレジスタ r3 には、ビッグエンディアンのときは 0x0123_4567 がロードされ、リトルエンディアンのときは 0x6745_2301 がロードされます。


メモリ
バイト
fp
0x0000_0400
+4
オフセット 1 を 2 ビット
左へシフト
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ロード
r3
0x0123_4567
ビッグエンディアン
r3
0x6745_2301
リトルエンディアン

MADD *rx*, *ry*

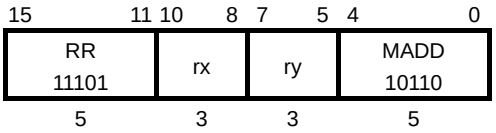
Multiply and Add

動作

HI ⇐ (HI ‖ LO) + (*rx* × *ry*) の上位ワード

LO ⇐ (HI ‖ LO) + (*rx* × *ry*) の下位ワード

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を乗算し、その積を HI・LO レジスタに格納されているダブルワードの値に加算します。*rx* と *ry* は符号付き整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

HI レジスタには、0x0000_0000 が、LO レジスタには 0xFFFF_FFFF が格納されています。汎用レジスタ r3 に 0x0123_4567 が、r4 に 0x89AB_CDEF が格納されているとします。

```
MADD r3,r4
```

このとき、上記の命令は、以下の演算を実行します。

0x0000_0000_FFFF_FFFF + (0x0123_4567 × 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0xFF79_5E36_C94E_4629
= 0xFF79_5E37_C94E_4628

結果の上位ワード 0xFF79_5E37 が HI レジスタに格納され、下位ワード 0xC94E_4628 が LO レジスタに格納されます。

MADDU *rx, ry*

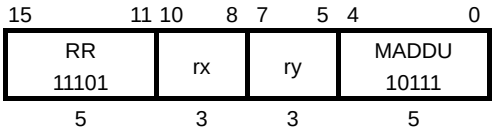
Multiply and Add Unsigned

動作

HI ⇐ (HI ‖ LO) + (*rx* × *ry*) の上位ワード

LO ⇐ (HI ‖ LO) + (*rx* × *ry*) の下位ワード

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を乗算し、その積を HI・LO レジスタに格納されているダブルワードの値に加算します。*rx* と *ry* は符号なし整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

HI レジスタには、0x0000_0000 が、LO レジスタには 0xFFFF_FFFF が格納されています。汎用レジスタ r3 に 0x0123_4567 が、r4 に 0x89AB_CDEF が格納されているとします。

```
MADDU r3,r4
```

このとき、上記の命令は、以下の演算を実行します。

0x0000_0000_FFFF_FFFF + (0x0123_4567 × 0x89AB_CDEF)
= 0x0000_0000_FFFF_FFFF + 0x009C_A39D_C94E_4629
= 0x009C_A39E_C94E_4628

結果の上位ワード 0x009C_A39E が HI レジスタに格納され、下位ワード 0xC94E_4628 が LO レジスタに格納されます。

MAX *rz, rx, ry*

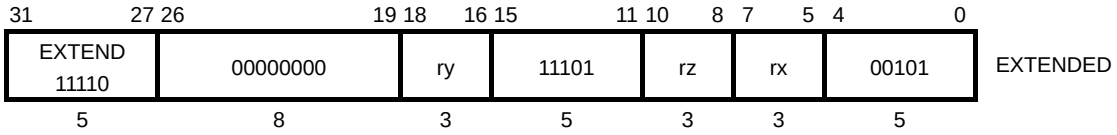
Maximum Signed

動作

if *rx* > *ry* then *rz* ⇐ *rx*;

else *rz* ⇐ *ry*;

コード



説明

汎用レジスタ rx と汎用レジスタ ry の内容を符号付値として比較をし、rx が大きい場合は汎用レジスタ rz に rx を格納し、それ以外の場合は rz に ry を格納します。

例外

なし

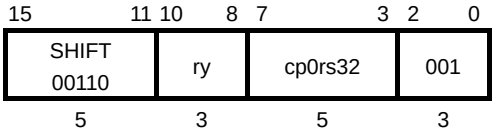
MFC0 *ry*, *cp0rs32*

Move from Coprocessor 0

動作

ry ⇐ CP0 のコプロセッサレジスタ *cp0rs32*

コード



説明

CP0 レジスタ *cp0rs32* の内容を汎用レジスタ *ry* にロードします。

Config1-3、IER レジスタには 16 ビット ISA モードではアクセスできません。

例外

コプロセッサ使用不可例外

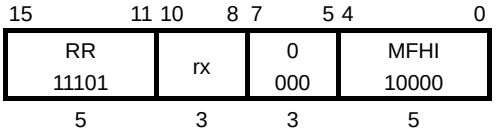
MFHI *rx*

Move From HI

動作

rx ⇐ HI

コード



説明

HI レジスタの内容を汎用レジスタ *rx* にロードします。

例外

なし

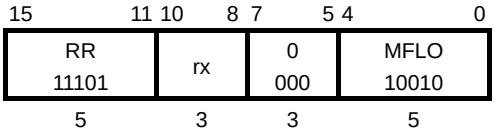
MFLO *rx*

Move From LO

動作

rx ⇐ LO

コード



説明

LO レジスタの内容を汎用レジスタ ***rx*** にロードします。

例外

なし

MIN *rz*, *rx*, *ry*

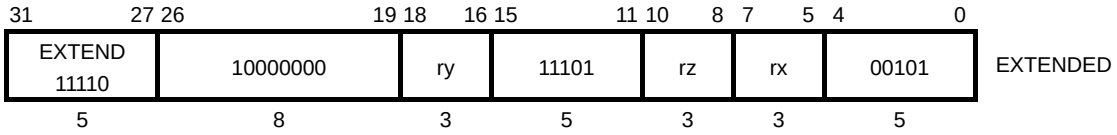
Minimum Signed

動作

if *rx* < *ry*　then *rz* ⇐ *rx*;

else　*rz* ⇐ *ry*;

コード



説明

汎用レジスタ rx と汎用レジスタ ry の内容を符号付値として比較をし、rx が小さい場合は汎用レジスタ rz に rx を格納し、それ以外の場合は rz に ry を格納します。

例外

なし

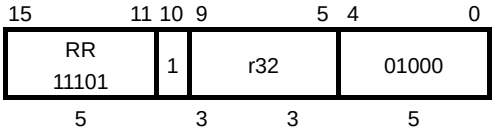
MOVE fp, *r32*

Move

動作

fp ⇐ *r32*

コード



説明

汎用レジスタ *r32*の内容を fp レジスタ *(r30)* に格納します。*r32*には 32 本の汎用レジスタ (r0~r31) のうちのどれでも指定できます。

16 ビット命令の *r32* フィールドのビット配置は下表の通りです。

r32 フィールド	レジスタ
00000	r0
00001	r1
00010	r2
00011	r3
00100	r4
00101	r5
00110	r6
00111	r7
01000	r8
01001	r9
01010	r10
01011	r11
01100	r12
01101	r13
01110	r14
01111	r15

r32 フィールド	レジスタ
10000	r16
10001	r17
10010	r18
10011	r19
10100	r20
10101	r21
10110	r22
10111	r23
11000	r24
11001	r25
11010	r26
11011	r27
11100	r28
11101	r29
11110	r30
11111	r31

例外

なし

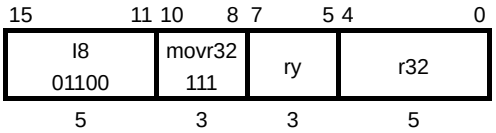
MOVE *ry*, *r32*

Move

動作

ry ⇐ *r32*

コード



説明

汎用レジスタ *r32* の内容を汎用レジスタ *ry* に格納します。*r32* には 32 本の汎用レジスタ (r0~r31) のうちのどれでも指定できます。*ry* には 16 ビット ISA でアクセスできる 8 本のレジスタのうちのいずれかを指定します。

16 ビット ISA では、32 本の汎用レジスタのうち r2～r7、r16、r17 の 8 本のレジスタしかアクセスできません。ただし、プロセッサには 32 ビット ISA モードで使用する 32 本のレジスタが存在するため、16 ビット ISA でアクセスできる 8 本のレジスタとその他の 24 本のレジスタ間で値を移送するための MOVE 命令が 16 ビット ISA に用意されています。MOVE 命令により、16 ビット ISA モードでも、32 本のすべての汎用レジスタを使用できるようになります。

16 ビット命令の *r32* フィールドのビット配置は下表の通りです。

r32 フィールド	レジスタ
00000	r0
00001	r1
00010	r2
00011	r3
00100	r4
00101	r5
00110	r6
00111	r7
01000	r8
01001	r9
01010	r10
01011	r11
01100	r12
01101	r13
01110	r14
01111	r15

r32 フィールド	レジスタ
10000	r16
10001	r17
10010	r18
10011	r19
10100	r20
10101	r21
10110	r22
10111	r23
11000	r24
11001	r25
11010	r26
11011	r27
11100	r28
11101	r29
11110	r30
11111	r31

例外

なし

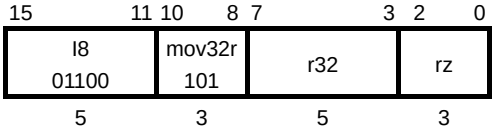
MOVE *r32*, *rz*

Move

動作

r32 ⇐ *rz*

コード



説明

汎用レジスタ *rz* の内容を汎用レジスタ *r32* に格納します。*rz* には 16 ビット ISA でアクセスできる 8 本のレジスタのうちのいずれかを指定します。*r32* には 32 本の汎用レジスタ (r0~r31) のうちのどれでも指定できます。

16 ビット ISA では、32 本の汎用レジスタのうち r2～r7、r16、r17 の 8 本のレジスタしかアクセスできません。ただし、プロセッサには 32 ビット ISA モードで使用する 32 本のレジスタが存在するため、16 ビット ISA でアクセスできる 8 本のレジスタとその他の 24 本のレジスタ間で値を移送するための MOVE 命令が 16 ビット ISA に用意されています。MOVE 命令により、16 ビット ISA モードでも、32 本のすべての汎用レジスタを使用できるようになります。

16 ビットコードの r32 フィールドのビット配置は、32 ビット ISA と異なります。r32 フィールドのビット配置は、[2:0] [4:3] のようになっています。コードとレジスタの対応を以下に示します。

コード	レジスタ
00000	r0
00001	r8
00010	r16
00011	r24
00100	r1
00101	r9
00110	r17
00111	r25
01000	r2
01001	r10
01010	r18
01011	r26
01100	r3
01101	r11
01110	r19
01111	r27

コード	レジスタ
10000	r4
10001	r12
10010	r20
10011	r28
10100	r5
10101	r13
10110	r21
10111	r29
11000	r6
11001	r14
11010	r22
11011	r30
11100	r7
11101	r15
11110	r23
11111	r31

例外

なし

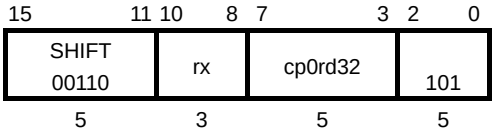
MTC0 *rx*, *cp0rd32*

Move to System Coprocessor 0 (CP0)

動作

CP0 レジスタの *cp0rd32* ⇐ *rx*

コード



説明

汎用レジスタ *rx* の内容を CP0 レジスタの *cp0rd32* にロードします。

Config1-3、IER レジスタには 16 ビット ISA モードではアクセスできません。

ERET 命令の直前または 2 命令前に、MTC0 命令により Status レジスタ、EPC レジスタ、ErrorEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、動作は不定となります。

同様に、DERET 命令の直前または 2 命令前に MTC0 命令により DEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、動作は不定になります。

MTC0 命令により、仮想アドレス変換システムの状態が変わることがあるので、直前や直後のロード・ストア命令の動作は確定しません。

例外

コプロセッサ使用不可例外

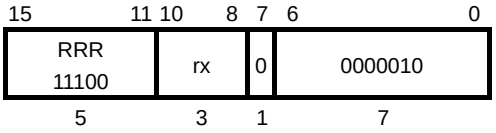
MTHI *rx*

Move To HI

動作

HI ⇐ *rx*

コード



説明

汎用レジスタ *rx* の内容を HI レジスタにロードします。

例外

なし

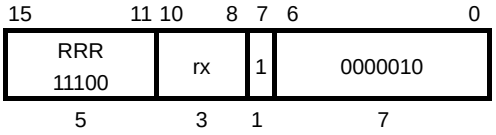
MTLO *rx*

Move To LO

動作

LO ⇐ *rx*

コード



説明

汎用レジスタ *rx* の内容を LO レジスタにロードします。

例外

なし

MULT *ry*, *rx*, *ry*

Multiply

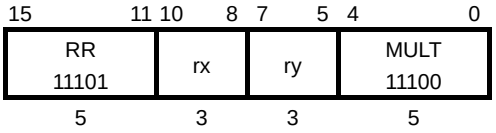
動作

HI ⇐ (*rx* × *ry*) の上位ワード;

LO ⇐ (*rx* × *ry*) の下位ワード;

ry ⇐ (*rx* × *ry*) の下位ワード;

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を乗算します。*rx* と *ry* は符号付き整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタと *ry* に格納します。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

汎用レジスタ r3 に 0x0123_4567 が、r4 に 0x89AB_CDEF が格納されているとします。

```
MULT r4,r3,r4
```

このとき、上記の命令は、以下の演算を実行します。

(0x0123_4567 × 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629

結果の上位ワード 0xFF79_5E36 が HI レジスタに格納され、下位ワード 0xC94E_4629 が LO レジスタと r4 に格納されます。

MULT *rx*, *ry*

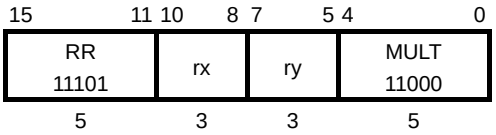
Multiply

動作

HI ⇐ (*rx* × *ry*) の上位ワード;

LO ⇐ (*rx* × *ry*) の下位ワード;

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を乗算します。*rx* と *ry* は符号付き整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

汎用レジスタ r3 に 0x0123_4567 が、r4 に 0x89AB_CDEF が格納されているとします。

```
MULT r3,r4
```

このとき、上記の命令は、以下の演算を実行します。

(0x0123_4567 × 0x89AB_CDEF)
= 0xFF79_5E36_C94E_4629

結果の上位ワード 0xFF79_5E36 が HI レジスタに格納され、下位ワード 0xC94E_4629 が LO レジスタに格納されます。

MULTU *rx*, *ry*

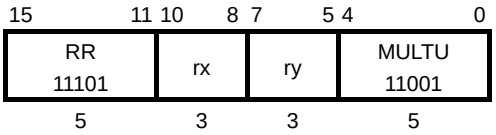
Multiply Unsigned

動作

HI ⇐ (*rx* × *ry*) の上位ワード;

LO ⇐ (*rx* × *ry*) の下位ワード;

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を乗算します。*rx* と *ry* は符号なし整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタに格納します。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

汎用レジスタ r3 に 0x0123_4567 が、r4 に 0x89AB_CDEF が格納されているとします。

```
MULTU r3,r4
```

このとき、上記の命令は、以下の演算を実行します。

(0x0123_4567 × 0x89AB_CDEF)
= 0x009C_A39D_C94E_4629

結果の上位ワード 0x009C_A39D が HI レジスタに格納され、下位ワード 0xC94E_4629 が LO レジスタに格納されます。

MULTU *ry*, *rx*, *ry*

Multiply

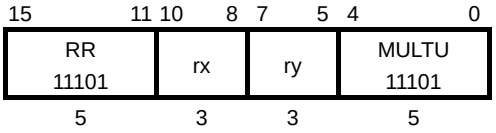
動作

HI ⇐ (rx × ry) の上位ワード;

LO ⇐ (rx × ry) の下位ワード;

ry ⇐ (rx × ry) の下位ワード;

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を乗算します。*rx* と *ry* は符号なし整数として扱います。結果の上位ワードを HI レジスタに格納し、下位ワードを LO レジスタと *ry* に格納します。

いかなる場合も、整数オーバフロー例外は発生しません。

例外

なし

使用例

汎用レジスタ r3 に 0x0123_4567 が、r4 に 0x89AB_CDEF が格納されているとします。

```
MULTU r4,r3,r4
```

このとき、上記の命令は、以下の演算を実行します。

(0x0123_4567 × 0x89AB_CDEF)
= 0x009C_A39D_C94E_4629

結果の上位ワード 0x009C_A39D が HI レジスタに格納され、下位ワード 0xC94E_4629 が LO レジスタと r4 に格納されます。

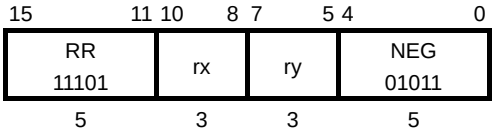
NEG *rx*, *ry*

Negate

動作

$rx = 0 - ry$

コード



説明

汎用レジスタ *ry* の内容の 2 の補数をとり、結果を汎用レジスタ *rx* に格納します。つまり、0 から *ry* の値を引いた結果が *rx* に格納されます。

例外

なし

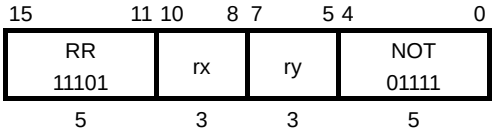
NOT *rx*, *ry*

NOT

動作

rx ⇐ *ry* NOR 0x0000_0000

コード



説明

汎用レジスタ *ry* の内容の 1 の補数をとり、結果を汎用レジスタ *rx* に格納します。*ry* の各ビットは反転されます。つまり、*rx* は *ry* と 0 の否定論理和 (NOR) をとった値となります。

例外

なし

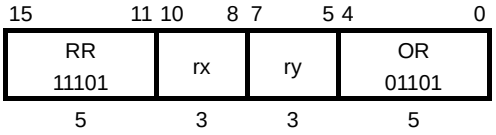
OR *rx*, *ry*

OR

動作

rx ⇐ *rx* OR *ry*

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容の論理和 (OR) をとり、結果を汎用レジスタ *rx* に格納します。

例外

なし

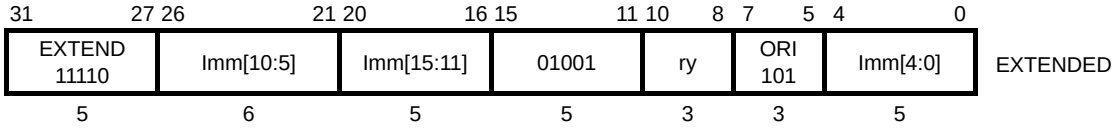
ORI *ry*, *immediate*

Logical OR Immediate

動作

$ry \Leftarrow ry$ OR $(0^{16} \| immediate_{15..0})$

コード



説明

16 ビット *immediate* をゼロ拡張した値と汎用レジスタ *ry* の内容との論理和 (OR) をとり、結果を汎用レジスタ *ry* に格納します。

immediate は 16 ビットです。これを超える値を扱いたい場合は、いったん汎用レジスタに格納してから、OR 命令を使います (「4.3.2　32 ビットの定数」を参照)。

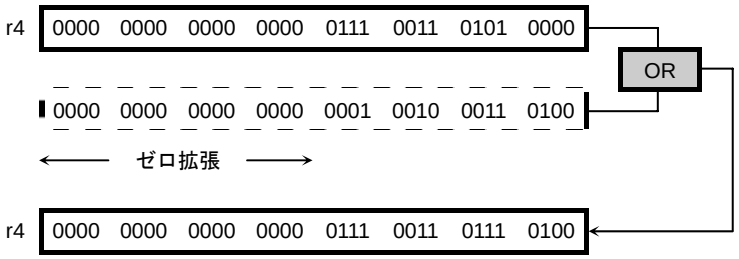
例外

なし

使用例

レジスタ r4 の値が 0x0000_7350 の場合、以下の命令を実行すると、図示したように、0x0000_7350 と 0x0000_1234 の論理和 0x0000_7374 がレジスタ r4 に格納されます。

```
ORI r4,0x1234
```



RESTORE *reg_list3*, *framesize4*

Restore Registers and Deallocate Stack Frame

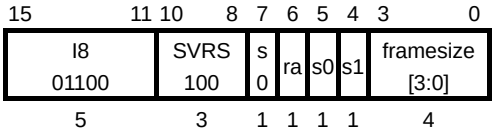
動作

ra(r31) ⇐ Stack　and/or　s1(r17) ⇐ Stack　and/or　s0(r16) ⇐ Stack ;

if *framesize4* == 0　then sp(r29) ⇐ sp + 128 ;

else sp(r29) ⇐ sp + (0 ‖ *framesize4* << 3) ;

コード



説明

命令コード中の ra, s0, s1 ビットがセットされているものに対し、スタックメモリからレジスタ r31(ra), レジスタ r16(s0), レジスタ r17(s1) にロードされ、*framsize4* の値によって SP レジスタの値は更新されます。レジスタは、大きい番号のレジスタに大きいスタックアドレスのデータがロードされます。

命令コードの ra, s0, s1 ビットは、*reg_list3* の内容が以下のようにエンコードされます。

reg_list3	ra	s0	s1
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

reg_list3 は、0 を指定することは禁止されています。もし 0 を指定した場合、動作は不定となります。

4 ビット *framesize4* は、3 ビット左にシフトし、ゼロ拡張されます。もし、*framesize4*=0000 と指定した場合は、*framesize4* の値として 128 が SP レジスタに加算されます。従って、*framsize4* の値は 8 刻みで+8～+128 までを取り扱うことができます。もし、範囲外の *framesize4* を指定した場合は、本命令は EXTEND 命令により拡張され、8 ビット framesize となり、0～+2040 の値を取り扱うことになります。この場合も、8 ビット framesize は、3 ビット左にシフトされます。

もし、スタックポインタの下位 2 ビットのどちらかの一方でも 0 でなかった場合、アドレスエラー例外が発生します。

動作の詳細

```
if framesize[3:0] = 0 then
   temp ⇐ sp (r29) + 128
else
   temp ⇐ sp (r29) + (0 || (framesize[3:0] << 3))
endif
temp2 ⇐ temp
if ra = 1 then
   temp ⇐ temp - 4
   r31   ⇐ Memory [temp]
endif
if s1 = 1 then
   temp ⇐ temp - 4
   r17   ⇐ Memory [temp]
endif
if s0 = 1 then
   temp ⇐ temp - 4
   r16   ⇐ Memory [temp]
endif
sp (r29) ⇐ temp2
```

例外

アドレスエラー例外

プログラミングノート

本命令は、ロードするデータ数、メモリアクセスタイムによって実行時間が異なります。本命令を実行中に検出された割込みから復帰した場合でも、再び最初から本命令を実行します。

RESTORE *reg_list3*, *xsregs*, *aregs*, *framesize8*

Restore Registers and Deallocate Stack Frame

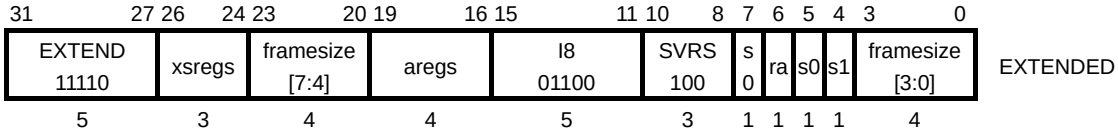
動作

ra(r31) ⇐ Stack and/or [r18-r23, r30] ⇐ Stack and/or

s1(r17) ⇐ Stack and/or s0(r16) ⇐ Stack and/or [r4-r7] ⇐ Stack ;

sp(r29) ⇐ sp + (0 ‖ *framesize8* << 3) ;

コード



説明

命令コードの ra ビットがセットされた場合は、スタックメモリからレジスタ r31(ra) にデータがロードされます。xsregs フィールドで示された番号によって、スタックメモリからレジスタ r30, r23-r18 にデータがロードされます。命令コードの s1, s0 ビットがセットされた場合は、スタックメモリからレジスタ r17, r16 にデータがロードされます。aregs フィールドで示された番号によって、スタックメモリからレジスタ r7-r4 にデータがロードされ、*framesize8* の値によって SP レジスタの値は更新されます。レジスタは、大きい番号のレジスタに大きいスタックアドレスのデータがロードされます。

xsregs フィールドによって指定された汎用レジスタのスタックについては、“動作の詳細” で、aregs フィールドによって指定された汎用レジスタのスタックについては、“Aregs フィールドの説明” でそれぞれ述べています。

命令コードの ra, s0, s1 ビットは、*reg_list3* の内容が以下のようにエンコードされます。

reg_list3	ra	s0	s1
0x0	0	0	0
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

少なくとも一つのレジスタは、指定しなければなりません。従って、*reg_list3*、*xsregs*、*aregs* で必ずロードするレジスタを指定しなければなりません。もし、どのレジスタもストアしない場合は、プロセッサの動作は不定です。

8 ビット framesize は、3 ビット左にシフトし、ゼロ拡張されます。したがって、*framesize8* の値は、8 刻みで 0～+2040 までを取り扱うことができます。

もし、スタックポインタの下位 2 ビットのどちらかの一方でも 0 でなかった場合、アドレスエラー例外が発生します。

Aregs フィールドの説明

標準 MIPS ABI (Application Binary Interface) では、レジスタ r4-r7 は関数へ引数を渡すために使用するレジスタ a0-a3 として扱います。これらのレジスタが、標準 MIPS ABI で使われるときには、受け取り側でなく、渡す側つまり呼び出し側の関数のスタックにも退避されます。ただし、標準 MIPS ABI では、レジスタ r4-r7 をスタックに退避しても、スタックからレジスタ r4-r7 への復元は行なう必要はありません。

しかし、標準 MIPS ABI 以外の関数呼び出し手順では、レジスタ r4-r7 は呼び出し側ではなく受け取り側のスタック領域に、静的レジスタ (関数の呼び出し前後で変化しないレジスタ) として退避し、受け取り側関数の処理の最後で復元することもあります。

拡張 RESTORE 命令の aregs フィールドへのエンコードは、拡張 SAVE 命令のエンコードと同じです。しかし、引数レジスタは RESTORE 命令の対象とせず、静的レジスタとして扱われるレジスタのみ、RESTORE 命令で復元します。

次の表は、RESTORE 命令の aregs フィールドのエンコードを示します。

aregs Encoding (binary)	静的レジスタの復元
0000	–
0001	r7
0010	r6, r7
0011	r5, r6, r7
1011	r4, r5, r6, r7
0100	–
0101	r7
0110	r6, r7
0111	r5, r6, r7
1000	–
1001	r7
1010	r6, r7
1100	–
1101	r7
1110	–
1111	予約

動作の詳細

```
temp   ⇐ sp (r29) + (0 || (framesize[7:0] << 3))
temp2 ⇐ temp
if ra = 1 then
   temp ⇐ temp - 4
   r31   ⇐ Memory [temp]
endif
if xsregs > 0 then
   if xsregs > 1 then
        if xsregs > 2 then
             if xsregs > 3 then
                  if xsregs > 4 then
                       if xsregs > 5 then
                            if xsregs > 6 then
                                 temp ⇐ temp - 4
                                 r30   ⇐ Memory [temp]
                            endif
                            temp⇐ temp - 4
                            r23   ⇐ Memory [temp]
                       endif
                       temp⇐ temp - 4
                       r22   ⇐ Memory [temp]
                  endif
                  temp ⇐ temp - 4
                  r21   ⇐ Memory [temp]
             endif
             temp ⇐ temp - 4
             r20   ⇐ Memory [temp]
        endif
        temp ⇐ temp - 4
        r19   ⇐ Memory [temp]
   endif
   temp ⇐ temp - 4
   r18   ⇐ Memory [temp]
endif
if s1 = 1 then
   temp ⇐ temp - 4
   r17   ⇐ Memory [temp]
endif
if s0 = 1 then
   temp ⇐ temp - 4
   r16   ⇐ Memory [temp]
endif
case aregs of (in binary)
   0000, 0100, 1000, 1100, 1110 : astatic ⇐ 0
   0001, 0101, 1001, 1101 : astatic ⇐ 1
   0010, 0110, 1010 : astatic ⇐ 2
   0011, 0111 : astatic ⇐ 3
```

```
    1011 : astatic ⇐ 4
    otherwise : UNPREDICTABLE
endcase
if astatic > 0 then
    temp ⇐ temp - 4
    r7     ⇐ Memory [temp]
    if astatic > 1 then
        temp ⇐ temp - 4
        r6     ⇐ Memory [temp]
        if astatic > 2 then
            temp ⇐ temp - 4
            r5     ⇐ Memory [temp]
            if astatic > 3 then
                temp ⇐ temp - 4
                r4     ⇐ Memory [temp]
            endif
        endif
    endif
endif
sp (r29) ⇐ temp2
```

例外

アドレスエラー例外

プログラミング ノート

本命令は、ロードするデータ数、メモリアクセスタイムによって実行時間が異なります。本命令を実行中に検出された割込みから復帰した場合でも、再び最初から本命令を実行します。

Aregs フィールドで予約コード (1111) を指定した場合、プロセッサの動作は不定です。

SADD *ry*, *rx*, *ry*

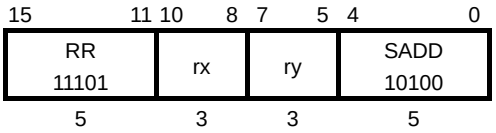
Saturated Add

動作

if overflow on *rx* + *ry* then　*ry* ⇐ 0x7FFF_FFFF (*rx* ≧ 0)　or 0x8000_0000 (*rx* < 0)

else　*ry* ⇐ *rx* + *ry*

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を加算し、オーバフローが発生した場合、*rx* の内容が 0 以上の場合は、正の数の最大値 (0x7FFF_FFFF) を、0 より小さい場合は負の数の最小値 (0x8000_0000) を *ry* へ格納する。オーバフローが発生しなかった場合は、加算した結果を *ry* に格納します。*rx* と *ry* の符号付き整数として扱います。

ただし、加算した結果がオーバフローしても、整数オーバフロー例外は発生しません。

例外

なし

SAVE *reg_list3, framesize4*

Save Registers and Set up Stack Frame

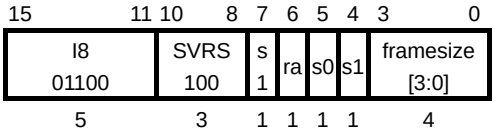
動作

Stack ⇐ ra(r31)　and/or　Stack ⇐ s1(r17)　and/or　Stack ⇐ s0(r16) ;

if *framesize4* == 0　then sp(r29) ⇐ sp - 128 ;

else sp(r29) ⇐ sp - (0 || *framesize4* << 3) ;

コード



説明

命令コード中の ra, s0, s1 ビットの状態によって、レジスタ r31(ra), レジスタ r16(so), レジスタ r17(s1) からスタックメモリにストアされ、*framsize4* の値によって SP レジスタの値は更新されます。レジスタは、大きい番号のレジスタが大きいスタックアドレスにストアされます。

命令コードの ra, s0, s1 ビットは、*reg_list3* の内容が以下のようにエンコードされます。

reg_list3	ra	s0	s1
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

reg_list3 は、0 を指定することは禁止されています。もし 0 を指定した場合、動作は不定となります。

4 ビット framesize は、3 ビット左にシフトし、ゼロ拡張されます。もし、*framesize4* を 0000 を指定した場合は、framesize の値として 128 が SP レジスタに加算されます。従って、*framsize4* の値は 8 刻みで+8～+128 までを取り扱うことができます。もし、範囲外の *framesize4* を指定した場合は、本命令は EXTEND 命令により拡張され、8 ビット framesize となり、0～+2040 の値を取り扱うことになります。この場合も、8 ビット framesize は、3 ビット左にシフトされます。

もし、スタックポインタの下位 2 ビットのどちらかの一方でも 0 でなかった場合、アドレスエラー例外が発生します。

動作の詳細

```
temp ⇐ sp (r29)
if ra = 1 then
   temp ⇐ temp - 4
   Memory [temp] ⇐ r31
endif
if s1 = 1 then
   temp ⇐ temp - 4
   Memory [temp] ⇐ r17
endif
if s0 = 1 then
   temp ⇐ temp - 4
   Memory [temp] ⇐ r16
endif
if framesize[3:0] = 0 then
   temp ⇐ sp (r29) - 128
else
   temp ⇐ sp (r29) - (0 || (framesize[3:0] << 3))
endif
sp (r29) ⇐ temp
```

例外

アドレスエラー例外

プログラミング ノート

本命令は、ロードするデータ数、メモリアクセスタイムによって実行時間が異なります。本命令を実行中に検出された割込みから復帰した場合でも、再び最初から本命令を実行します。

SAVE *reg_list3*, *xsregs*, *aregs*, *framesize8*

Save Registers and Set up Stack Frame

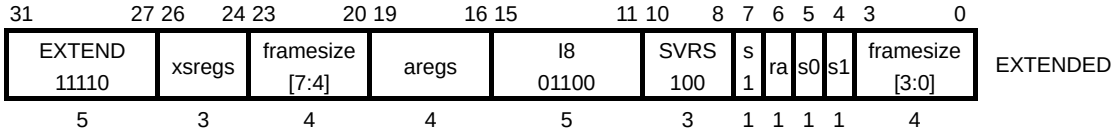
動作

Stack ⇐ ra(r31) and/or Stack ⇐ [r18-r23, r30] and/or

Stack ⇐ s1(r17) and/or Stack ⇐ s0(r16) and/or Stack ⇐ [r4-r7] ;

sp(r29) ⇐ sp - (0 ‖ *framesize8* << 3) ;

コード



説明

aregs フィールドで示された番号によって、Argument (関数の引数) としてレジスタ r4-r7 がスタックメモリにストアされます。命令コードの ra ビットがセットされた場合は、レジスタ r31(ra) がスタックメモリにストアされます。Xsregs フィールドで示された番号によって、レジスタ r18-r23, r30 がスタックメモリにストアされ、命令コードの s0, s1 ビットがセットされた場合は、それぞれに応じてレジスタ r16(s0), r17(s1) がスタックメモリにストアされます。aregs フィールドで示された番号によって、静的レジスタとしてレジスタ r4-r7 がスタックメモリにストアされ、*framesize8* の値によって SP レジスタの値は更新されます。レジスタは、大きい番号のレジスタが大きいスタックアドレスにストアされます。

xsregs フィールドによって指定された汎用レジスタのスタックについては、“動作の詳細” で、aregs フィールによって指定された汎用レジスタのスタックについては、“Aregs フィールドの説明” でそれぞれ述べています。

命令コードの ra, so, s1 ビットは、*reg_list3* の内容が以下のようにエンコードされます。

reg_list3	ra	s0	s1
0x0	0	0	0
0x1	0	0	1
0x2	0	1	0
0x3	0	1	1
0x4	1	0	0
0x5	1	0	1
0x6	1	1	0
0x7	1	1	1

少なくとも一つのレジスタは、指定しなければなりません。したがって、*reg_list3*、*xsregs*、*aregs* で必ずストアするレジスタを指定しなければなりません。もし、どのレジスタもストアしない場合は、プロセッサの動作は不定です。

8 ビット framesize は、3 ビット左にシフトし、ゼロ拡張されます。従って、*framesize8* の値は、8 刻みで 0～+2040 までを取り扱うことができます。

もし、スタックポインタの下位 2 ビットのどちらかの一方でも 0 でなかった場合、アドレスエラー例外が発生します。

Aregs フィールドの説明

標準 MIPS ABI (Application Binary Interface) では、レジスタ r4-r7 は関数へ引数を渡すために使用するレジスタ a0-a3 として扱います。これらのレジスタが、標準 MIPS ABI で使われるときには、受け取り側でなく、渡す側つまり呼び出し側の関数のスタックに退避されます。

しかし、標準 MIPS ABI 以外の関数呼び出し手順では、レジスタ r4-r7 は静的レジスタ (関数呼び出し前後で値が変わらないレジスタ) として、呼び出し側ではなく受け取り側の関数のスタック領域に退避することもあります。

Aregs フィールドは、0～4 個の引数レジスタと 0～4 個の静的レジスタ、またはそれらの混在をエンコードできます。レジスタは、昇順 (a0, a1, a2, a3) に引数に対応し、また降順 (r7, r6, r5, r4) に静的な値に対応します。

次の表は、aregs フィールドのエンコードを示します。

aregs Encoding (binary)	引数レジスタの退避	静的レジスタの退避
0000	–	–
0001	–	r7
0010	–	r6, r7
0011	–	r5, r6, r7
1011	–	r4, r5, r6, r7
0100	a0(r4)	–
0101	a0(r4)	r7
0110	a0(r4)	r6, r7
0111	a0(r4)	r5, r6, r7
1000	a0(r4), a1(r5)	–
1001	a0(r4), a1(r5)	r7
1010	a0(r4), a1(r5)	r6, r7
1100	a0(r4), a1(r5), a2(r6)	–
1101	a0(r4), a1(r5), a2(r6)	r7
1110	a0(r4), a1(r5), a2(r6), a3(r7)	–
1111	予約	予約

動作の詳細

```
temp ⇐ sp (r29)
case aregs of (in binary)
   0000, 0001, 0010, 0011, 1011 : args ⇐ 0
   0100, 0101, 0110, 0111 : args ⇐ 1
   1000, 1001, 1010 : args ⇐ 2
   1100, 1101 : args ⇐ 3
   1110 : args ⇐ 4
   otherwise : UNPREDICTABLE
endcase
if args > 0 then
   Memory [temp] ⇐ r4
   if args > 1 then
      Memory [temp + 4] ⇐ r5
      if args > 2 then
         Memory [temp + 8] ⇐ r6
         if args > 3 then
            Memory [temp + 12] ⇐ r7
         endif
      endif
   endif
endif
if ra = 1 then
   temp ⇐ temp - 4
   Memory [temp] ⇐ r31
endif
if xsregs > 0 then
   if xsregs > 1 then
      if xsregs > 2 then
         if xsregs > 3 then
            if xsregs > 4 then
               if xsregs > 5 then
                  if xsregs > 6 then
                     temp ⇐ temp - 4
                     Memory [temp] ⇐ r30
                  endif
                  temp ⇐ temp - 4
                  Memory [temp] ⇐ r23
               endif
               temp ⇐ temp - 4
               Memory [temp] ⇐ r22
            endif
            temp ⇐ temp - 4
            Memory [temp] ⇐ r21
         endif
         temp ⇐ temp - 4
         Memory [temp] ⇐ r20
      endif
      temp ⇐ temp - 4
      Memory [temp] ⇐ r19
```

```
    endif
    temp ⇐ temp - 4
    Memory [temp] ⇐ r18
endif
if s1 = 1 then
    temp ⇐ temp - 4
    Memory [temp] ⇐ r17
endif
if s0 = 1 then
    temp ⇐ temp - 4
    Memory [temp] ⇐ r16
endif
case aregs of (in binary)
    0000, 0100, 1000, 1100, 1110 : astatic ⇐ 0
    0001, 0101, 1001, 1101 : astatic ⇐ 1
    0010, 0110, 1010 : astatic ⇐ 2
    0011, 0111 : astatic ⇐ 3
    1011 : astatic ⇐ 4
    otherwise : UNPREDICTABLE
endcase

if astatic > 0 then
    temp ⇐ temp - 4
    Memory [temp] ⇐ r7
    if astatic > 1 then
        temp ⇐ temp - 4
        Memory [temp] ⇐ r6
        if astatic > 2 then
            temp ⇐ temp - 4
            Memory [temp] ⇐ r5
            if astatic > 3 then
                temp ⇐ temp - 4
                Memory [temp] ⇐ r4
            endif
        endif
    endif
endif
temp ⇐ sp (r29) - (0 || (framesize[7:0] << 3))
sp (r29) ⇐ temp
```

例外

アドレスエラー例外

プログラミング ノート

本命令は、ロードするデータ数、メモリアクセスタイムによって実行時間が異なります。本命令を実行中に検出された割込みから復帰した場合でも、再び最初から本命令を実行します。

Aregs フィールドで予約コード (1111) を指定した場合、プロセッサの動作は不定です。

SB *ry*, *offset* (*base*)

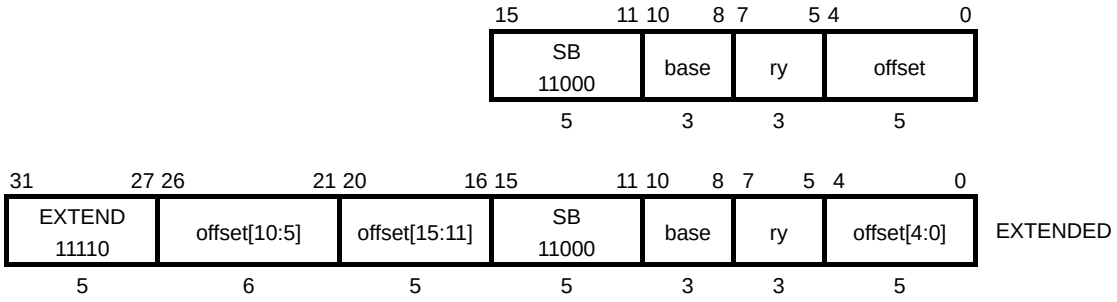
Store Byte

動作

ry = {zero-extend (*offset*) + (*base*)}

(EXTENDED)　*ry* = {sign-extend (*offset*) + (*base*)}

コード



説明

5 ビット *offset* をゼロ拡張した値と汎用レジスタ *base* の内容を加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *ry* の最下位バイトを、このアドレスにストアします。

ry の上位 3 バイトは無視されるため、符号付きと符号なしの区別はありません。

offset は 5 ビットで、扱うことのできる数値の範囲は、0~31 です。この範囲外の値を指定すると SB 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

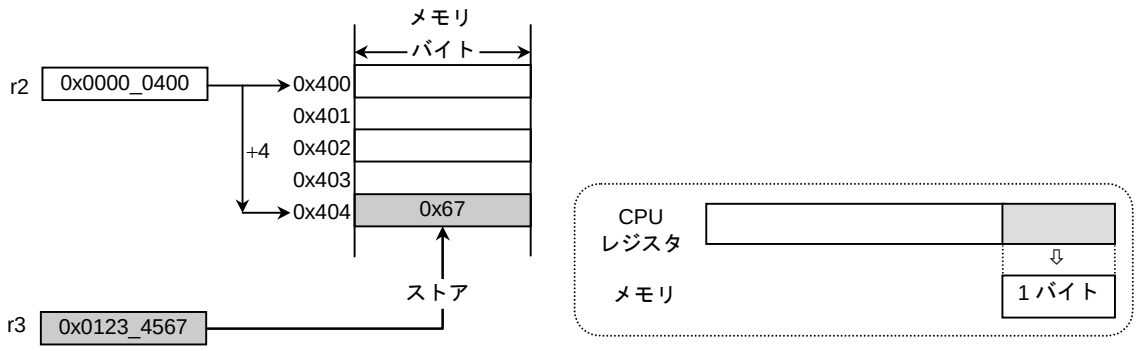
例外

アドレスエラー例外

使用例

レジスタ r2 の値が 0x0000_0400 で、r3 の値が 0x0123_4567 の場合、以下の命令を実行すると、0x67 がアドレス 0x404 に格納されます。

```
SB r3,4(r2)
```


メモリ
バイト
r2
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
0x67
ストア
r3
0x0123_4567
CPU
レジスタ
メモリ
1 バイト

SB *ry*, *offset* (fp)

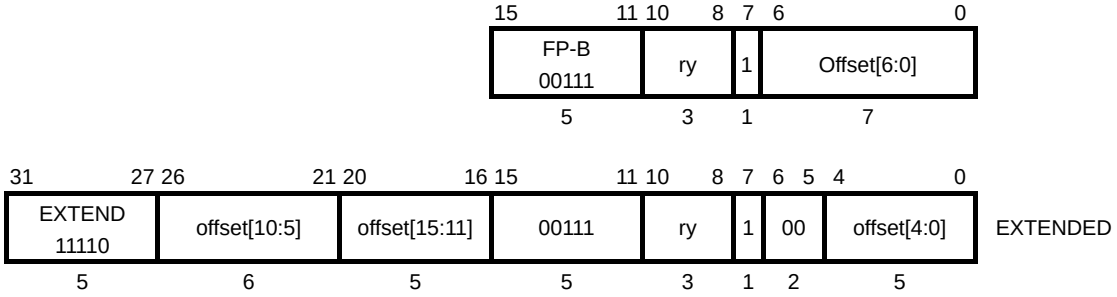
Store Byte

動作

ry = {zero-extend (*offset*) + (fp)}

(EXTENDED)　*ry* = {sign-extend (*offset*) + (fp)}

コード



説明

7 ビット *offset* をゼロ拡張した値と fp レジスタ (r30) の内容を加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *ry* の最下位バイトを、このアドレスにストアします。

ry の上位 3 バイトは無視されるため、符号付きと符号なしの区別はありません。

offset は 7 ビットで、扱うことのできる数値の範囲は、0~127 です。この範囲外の値を指定すると SB 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

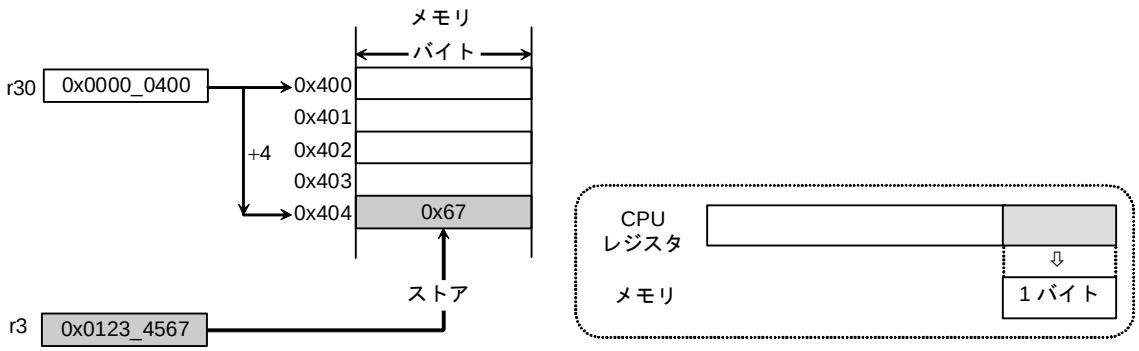
例外

アドレスエラー例外

使用例

fp レジスタ (r30) の値が 0x0000_0400 で、r3 の値が 0x0123_4567 の場合、以下の命令を実行すると、0x67 がアドレス 0x0404 に格納されます。

```
SB r3,4(fp)
```


メモリ
バイト
r30
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
0x67
ストア
r3
0x0123_4567
CPU
レジスタ
メモリ
1 バイト

SB *ry*, *offset* (sp)

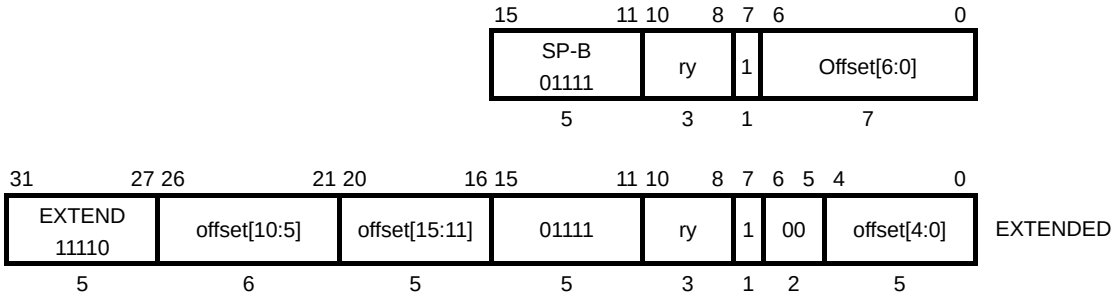
Store Byte

動作

ry = {zero-extend (*offset*) + (sp)}

(EXTENDED)　*ry* = {sign-extend (*offset*) + (sp)}

コード



説明

7 ビット *offset* をゼロ拡張した値と sp レジスタ (r29) の内容を加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *ry* の最下位バイトを、このアドレスにストアします。

ry の上位 3 バイトは無視されるため、符号付きと符号なしの区別はありません。

offset は 7 ビットで、扱うことのできる数値の範囲は、0~127 です。この範囲外の値を指定すると SB 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

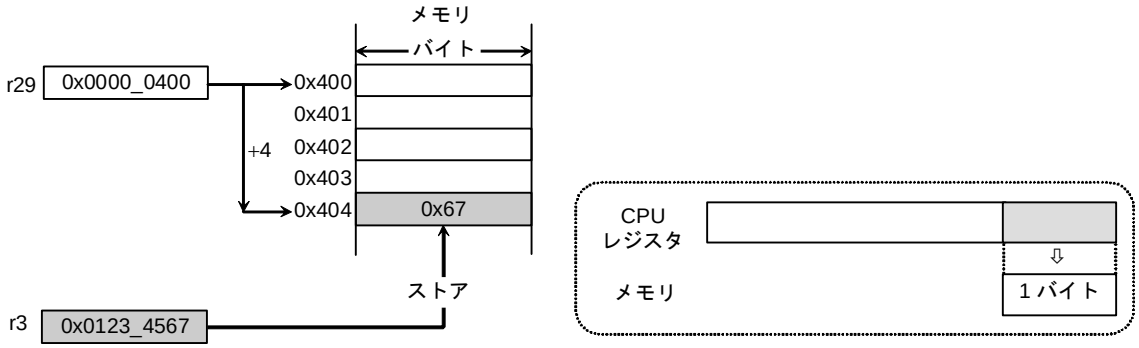
例外

アドレスエラー例外

使用例

sp レジスタ (r29) の値が 0x0000_0400 で、r3 の値が 0x0123_4567 の場合、以下の命令を実行すると、0x67 がアドレス 0x404 に格納されます。

```
SB r3,4(sp)
```


メモリ
バイト
r29
0x0000_0400
0x400
0x401
0x402
0x403
0x404
+4
0x67
ストア
r3
0x0123_4567
CPU
レジスタ
メモリ
1 バイト

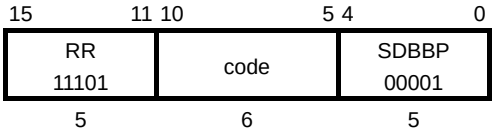
SDBBP *code*

Software Debug Breakpoint

動作

ソフトウエアデバッグブレークポイント例外

コード



説明

デバッグブレークポイントが発生し、無条件に制御を例外ハンドラに移します。

SDBBP 命令の *code* フィールドは、例外ハンドラに情報を渡すために使用できます。例外ハンドラが code フィールドを取り出すには、命令を含むメモリワードの内容をデータとしてロードする必要があります。詳細は「9.3　デバッグ例外」を参照してください。

SDBBP 命令は、開発ツールで使用しますので、ユーザープログラム中では記述しないでください。EJTAG を実装しない製品で実行すると予約命令例外が発生します。

例外

デバッグブレークポイント例外

予約命令例外

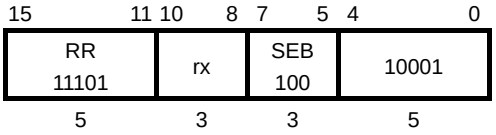
SEB *rx*

Sign-Extend Byte

動作

$rx \Leftarrow (rx[7])^{24} \parallel rx[7:0]$

コード



説明

汎用レジスタ *rx* の下位バイトを符号拡張して *rx* に格納する。

例外

なし

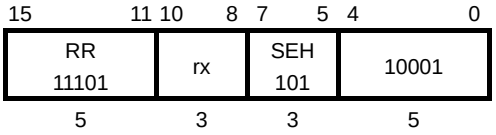
SEH *rx*

Sign-Extend Halfword

動作

$rx \Leftarrow (rx[15])^{16} \parallel rx[15:0];$

コード



説明

汎用レジスタ *rx* の下位ハーフワードを符号拡張して *rx* に格納する。

例外

なし

SH *ry*, *offset* (*base*)

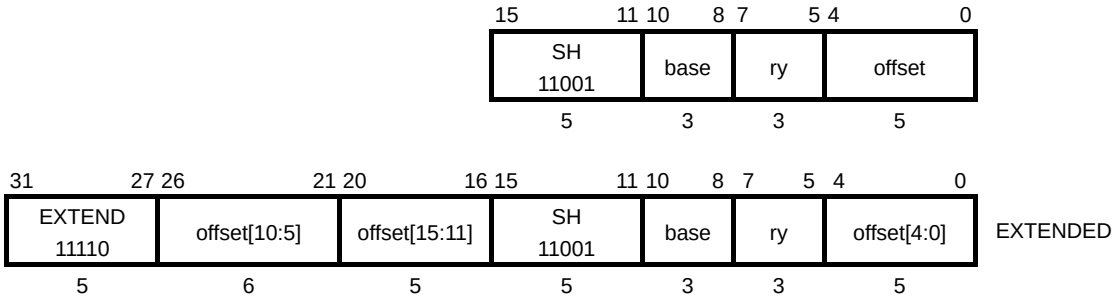
Store Halfword

動作

ry = {zero-extend (*offset* || 0) + (*base*)}

(EXTENDED)　　*ry* = {sign-extend (*offset*) + (*base*)}

コード



説明

5 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、汎用レジスタ *base* の内容に加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *ry* の下位ハーフワードをこのアドレスにストアします。

ry の上位ハーフワードは無視されるので、符号付き、符号なしの区別はありません。

offset は 5 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、2 刻みで 0~62 になります。この範囲外の値を指定すると、SH 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32767) を扱えるようになります。この場合、*offset* はシフトされません。

例外

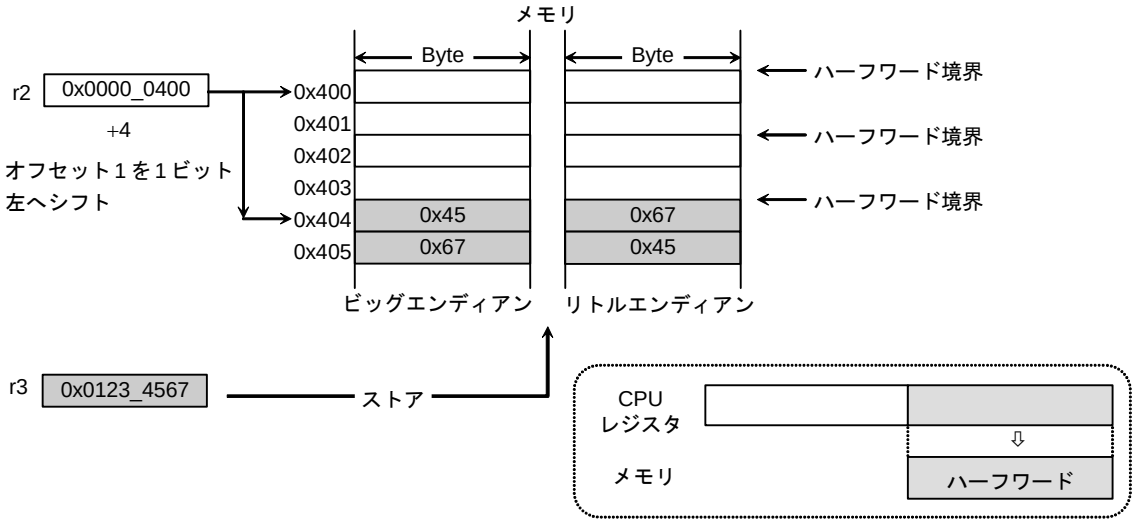
アドレスエラー例外

使用例

```
SH r3,4(r2)
```

レジスタ r2 の値が 0x0000_0400 で、r3 の値が 0x0123_4567 であるとします。オフセット値は 1 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アゼンブラ・リンカにより、2 (2 進数 0010) に変換されます。したがって、命令コードは 0xCA62 になります。

上記の命令を実行すると、ビッグエンディアンのときは、アドレス 0x404 に 0x45 が、アドレス 0x405 に 0x67 がストアされます。リトルエンディアンのときは、アドレス 0x404 に 0x67 が、アドレス 0x405 に 0x45 がストアされます。


メモリ
Byte
Byte
r2
0x0000_0400
+4
オフセット１を１ビット
左へシフト
0x400
0x401
0x402
0x403
0x404
0x405
0x45
0x67
0x67
0x45
ハーフワード境界
ハーフワード境界
ハーフワード境界
ビッグエンディアン
リトルエンディアン
r3
0x0123_4567
ストア
CPU
レジスタ
メモリ
ハーフワード

SH *ry*, *offset* (fp)

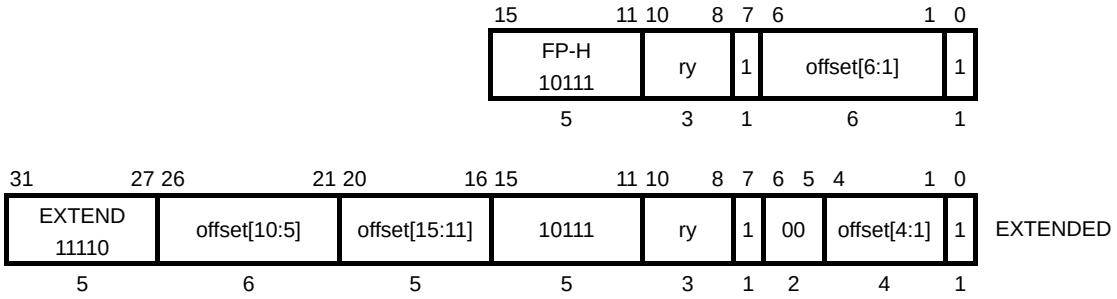
Store Halfword

動作

ry = {zero-extend (*offset* ‖ 0) + (fp)}

(EXTENDED)　*ry* = {sign-extend (*offset* ‖ 0) + (fp)}

コード



説明

6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、fp レジスタ (r30) の内容に加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *ry* の下位ハーフワードをこのアドレスにストアします。

ry の上位ハーフワードは無視されるので、符号付き、符号なしの区別はありません。

offset は 6 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、2 刻みで 0~126 になります。この範囲外の値を指定すると、SH 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32766) を扱えるようになります。この場合、*offset* は 1 ビットシフトされます。

例外

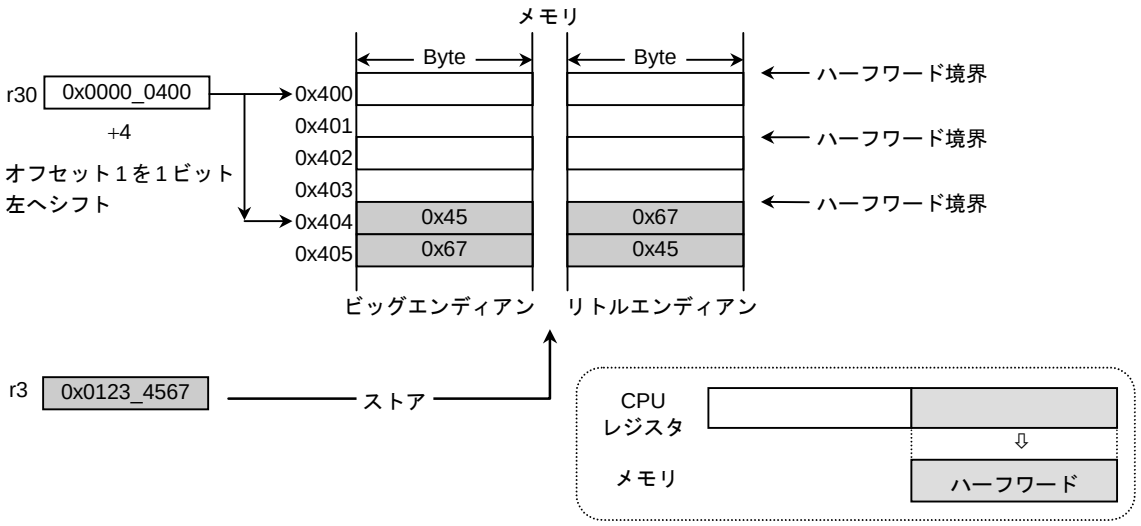
アドレスエラー例外

使用例

```
SH r3,4(fp)
```

fp レジスタ (r30) の値が 0x0000_0400 で、r3 の値が 0x0123_4567 であるとします。オフセット値は 1 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、2 (2 進数 0010) に変換されます。したがって、命令コードは 0xBB85 になります。

上記の命令を実行すると、ビッグエンディアンのときは、アドレス 0x0404 に 0x45 が、アドレス 0x0405 に 0x67 がストアされます。リトルエンディアンのときは、アドレス 0x0404 に 0x67 が、アドレス 0x0405 に 0x45 がストアされます。


メモリ
Byte
Byte
r30
0x0000_0400
+4
オフセット 1 を 1 ビット
左へシフト
0x400
0x401
0x402
0x403
0x404
0x405
0x45
0x67
0x67
0x45
ハーフワード境界
ハーフワード境界
ハーフワード境界
ビッグエンディアン
リトルエンディアン
r3
0x0123_4567
ストア
CPU
レジスタ
メモリ
ハーフワード

SH *ry*, *offset* (sp)

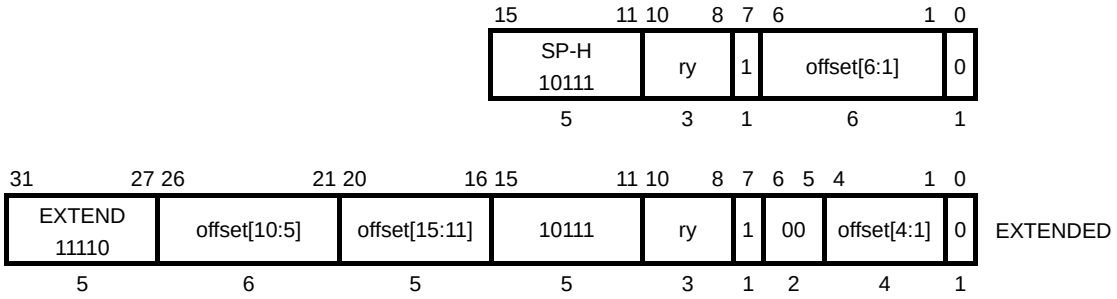
Store Halfword

動作

ry = {zero-extend (*offset* ‖ 0) + (sp)}

(EXTENDED)　*ry* = {sign-extend (*offset* ‖ 0) + (sp)}

コード



説明

6 ビット *offset* を 1 ビット左へシフトして、ゼロ拡張し、sp レジスタ (r29) の内容に加算することにより、実効アドレス (EA) を生成します。汎用レジスタ *ry* の下位ハーフワードをこのアドレスにストアします。

ry の上位ハーフワードは無視されるので、符号付き、符号なしの区別はありません。

offset は 6 ビットで、1 ビット左へシフトすることにより扱うことのできる数値の範囲は、2 刻みで 0~126 になります。この範囲外の値を指定すると、SH 命令は EXTEND 命令により拡張され、符号付きの 16 ビットの即値 (–32768 ~ +32766) を扱えるようになります。この場合、*offset* は 1 ビットシフトされます。

例外

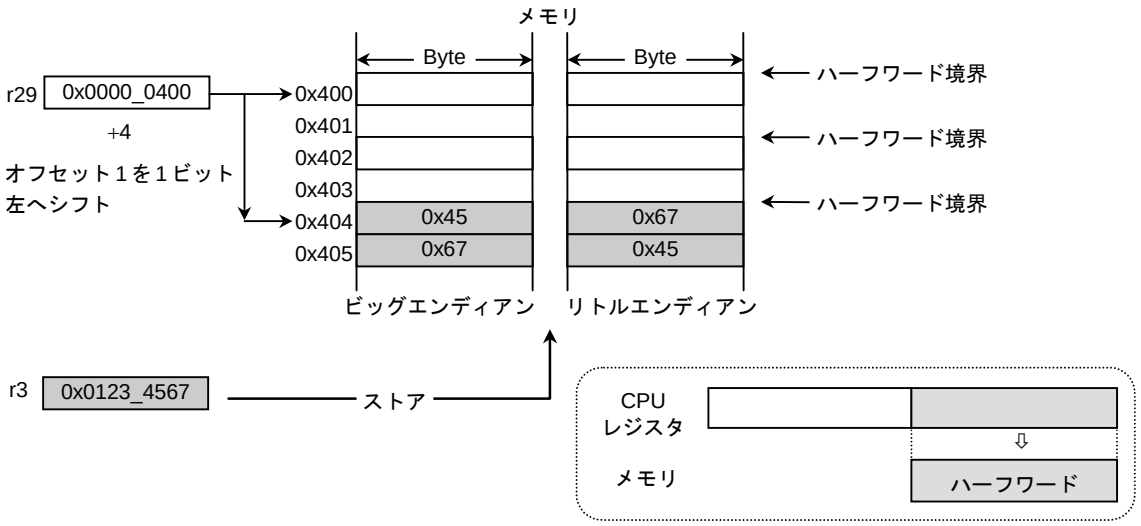
アドレスエラー例外

使用例

```
SH r3,4(sp)
```

sp レジスタ (r29) の値が 0x0000_0400 で、r3 の値が 0x0123_4567 であるとします。オフセット値は 1 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、2 (2 進数 0010) に変換されます。したがって、命令コードは 0xBB84 になります。

上記の命令を実行すると、ビッグエンディアンのときは、アドレス 0x0404 に 0x45 が、アドレス 0x0405 に 0x67 がストアされます。リトルエンディアンのときは、アドレス 0x0404 に 0x67 が、アドレス 0x0405 に 0x45 がストアされます。


メモリ
Byte
Byte
r29
0x0000_0400
+4
オフセット 1 を 1 ビット
左へシフト
0x400
0x401
0x402
0x403
0x404
0x405
0x45
0x67
0x67
0x45
ハーフワード境界
ハーフワード境界
ハーフワード境界
ビッグエンディアン
リトルエンディアン
r3
0x0123_4567
ストア
CPU
レジスタ
メモリ
ハーフワード

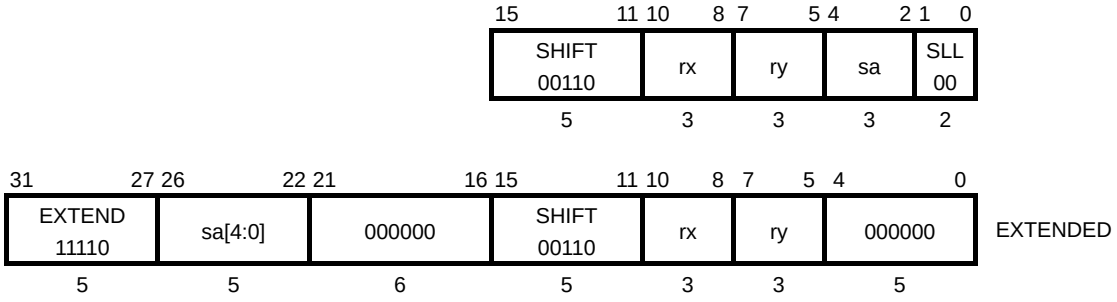
SLL *rx*, *ry*, *sa*

Shift Left Logical

動作

rx ⇐ *ry* << *sa*

コード



説明

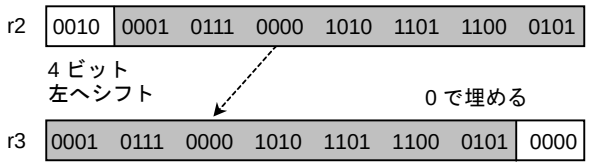
汎用レジスタ *ry* の 32 ビットの内容を *sa* ビット左へシフトし、右端の空いたビットを 0 で埋め、結果を汎用レジスタ *rx* に格納します。*sa* フィールドは 3 ビットです。したがって *sa* で指定できる数値の範囲は 1~8 です。000 は 8 ビットのシフトとして定義されています。

この範囲外のシフト量を指定すると、SLL 命令は EXTEND 命令により拡張され、sa フィールドは 5 ビットになり、0~31 のシフト量が扱えるようになります。

使用例

レジスタ r2 の内容が 0x2170_ADC5 の場合、以下の命令を実行すると、レジスタ r3 に 0x170A_DC50 が格納されます。

```
SLL r3,r2,4
```



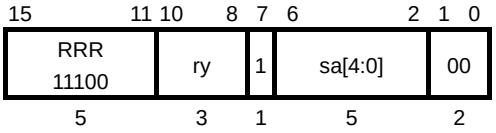
SLL *ry*, *sa5*

Shift Left Logical

動作

ry ⇐ *ry* << *sa5*

コード



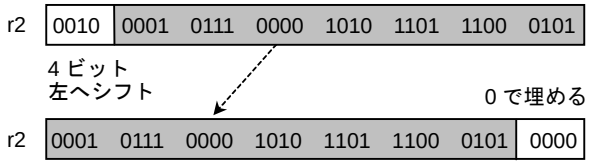
説明

汎用レジスタ *ry* の 32 ビットの内容を *sa* ビット左へシフトし、右端の空いたビットを 0 で埋め、結果を汎用レジスタ *ry* に格納します。*sa* フィールドは 5 ビットです。したがって、*sa* で指定できる数値の範囲は 1~31 です。*sa* フィールドを 00000 として定義することはできません。

使用例

レジスタ r2 の内容が 0x2170_ADC5 の場合、以下の命令を実行すると、レジスタ r2 に 0x170A_DC50 が格納されます。

```
SLL r2,4
```



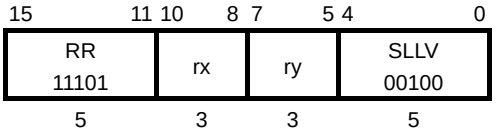
SLLV *ry*, *rx*

Shift Left Logical Variable

動作

ry << *rx* の下位 5 ビット

コード



説明

汎用レジスタ *ry* の 32 ビットの内容を、汎用レジスタ *rx* の下位 5 ビットで指定されたビット数、左へシフトし、右側の空いたビットを 0 で埋めます。結果を汎用レジスタ *ry* に格納します。

例外

なし

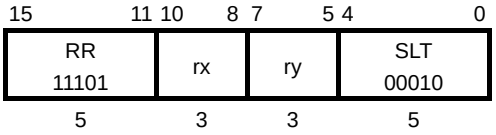
SLT *rx*, *ry*

Set On Less Than

動作

if *rx* < *ry* then t8 ⇐ 1; else t8 ⇐ 0

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を、符号付き整数として比較します。*rx* が *ry* より小さい場合は、コンディションコードレジスタ t8 (r24) に 1 を、そうでない場合は、t8 に 0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

例外

なし

SLTI *rx*, *immediate*

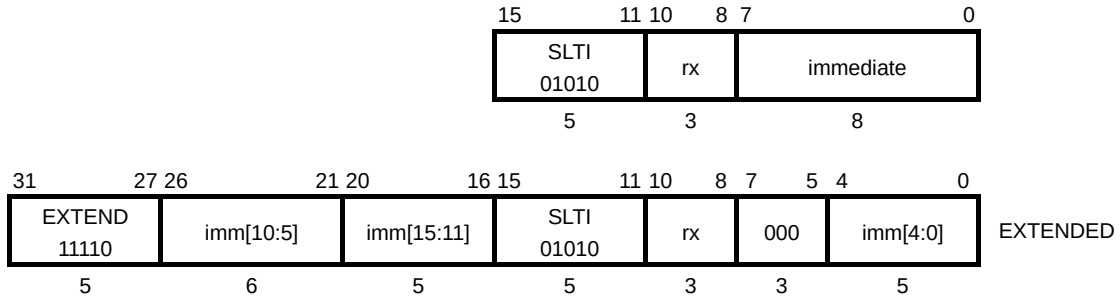
Set On Less Than Immediate

動作

if $rx < 0^{24} \parallel (immediate_{7..0})$ then t8 ⇐ 1; else t8 ⇐ 0

(EXTENDED)　if $rx < (immediate_{15})^{16} \parallel (immediate_{15..0})$ then t8 ⇐ 1; else t8 ⇐ 0

コード



説明

8 ビット *immediate* をゼロ拡張した値と汎用レジスタ *rx* の内容を符号付き整数として比較します。*rx* が *immediate* より小さい場合、コンディションコードレジスタ t8 (r24) に 1 を、そうでない場合は 0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

immediate は 8 ビットで、扱うことのできる数値の範囲は 0~255 です。この範囲外の値を指定すると、SLTI 命令は EXTEND 命令により拡張され、符号付き 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

例外

なし

SLTIU *rx*, *immediate*

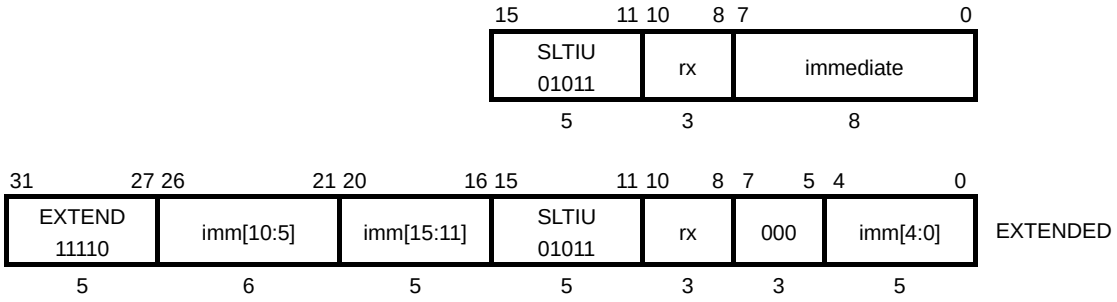
Set On Less Than Immediate Unsigned

動作

if (0 ‖ *rx*) < 0^{25} ‖ (*immediate*$_{7..0}$) then t8 ⇐ 1; else t8 ⇐ 0

(EXTENDED)　if (0 ‖ *rx*) < (*immediate*$_{15}$)^{17} ‖ (*immediate*$_{15..0}$) then t8 ⇐ 1; else t8 ⇐ 0

コード



説明

8 ビット *immediate* をゼロ拡張した値と汎用レジスタ *rx* の内容を、符号なし整数として比較します。*rx* が *immediate* より小さい場合は、コンディションコードレジスタ t8 (r24) に 1 を、そうでない場合は、0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

immediate は 8 ビットで、扱うことのできる数値の範囲は 0~255 です。この範囲外の値を指定すると、SLTIU 命令は EXTEND 命令により拡張され、符号付き 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

例外

なし

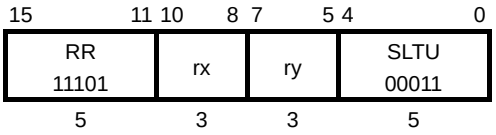
SLTU *rx*, *ry*

Set On Less Than Unsigned

動作

if (0 ‖ *rx*) < (0 ‖ *ry*) then t8 ⇐ 1; else t8 ⇐ 0

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容を、符号なし整数として比較します。*rx* が *ry* より小さい場合は、コンディションコードレジスタ t8 (r24) に 1 を、そうでない場合は、0 を格納します。

いかなる場合も、整数オーバフロー例外は発生しません。比較の際に行われる減算でオーバフローしても、比較は有効です。

例外

なし

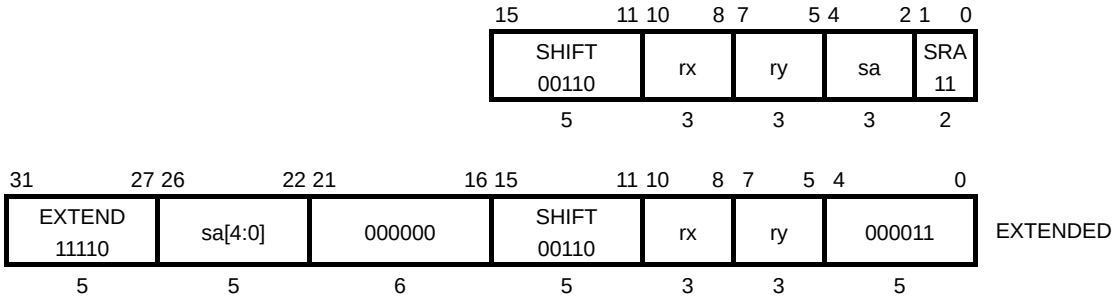
SRA *rx*, *ry*, *sa*

Shift Right Arithmetic

動作

rx ⇐ *ry* >> *sa*

コード



説明

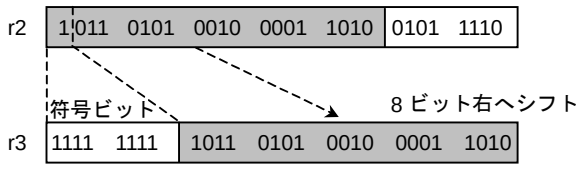
汎用レジスタ *ry* の 32 ビットの内容を *sa* ビット右へシフトし、左端の空いたビットを符号ビットで埋め、結果を汎用レジスタ *rx* に格納します。*sa* フィールドは 3 ビットです。したがって *sa* で指定できる数値の範囲は 1~8 です。000 は 8 ビットのシフトとして定義されています。

この範囲外のシフト量を指定すると、SRA 命令は EXTEND 命令により拡張され、sa フィールドは 5 ビットになり、0~31 のシフト量が扱えるようになります。

使用例

レジスタ r2 の内容が 0xB521_AE5E の場合、以下の命令を実行すると、レジスタ r3 に 0xFFB5_21AE が格納されます。

```
SRA r3,r2,8
```



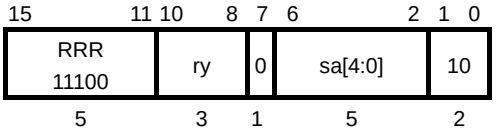
SRA *ry*, *sa5*

Shift Right Arithmetic

動作

ry ⇐ *ry* >> *sa5*

コード



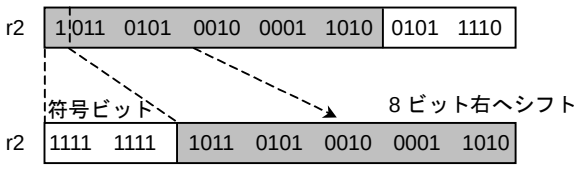
説明

汎用レジスタ *ry* の 32 ビットの内容を *sa* ビット右へシフトし、左端の空いたビットを符号ビットで埋め、結果を汎用レジスタ *ry* に格納します。*sa* フィールドは 5 ビットです。したがって、*sa* で指定できる数値の範囲は 1~31 です。*sa* フィールドを 00000 と定義することはできません。

使用例

レジスタ r2 の内容が 0xB521_AE5E の場合、以下の命令を実行すると、レジスタ r2 に 0xFFB5_21AE が格納されます。

```
SRA r2,8
```



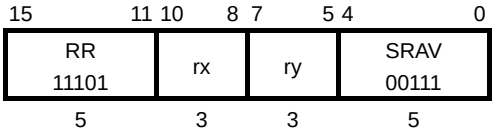
SRAV *ry*, *rx*

Shift Right Arithmetic Variable

動作

ry >> *rx* の下位 5 ビット

コード



説明

汎用レジスタ *ry* の 32 ビットの内容を、汎用レジスタ *rx* の下位 5 ビットで指定されたビット数、右にシフトし、左端の空いたビットを符号ビットで埋めます。結果を汎用レジスタ *ry* に格納します。

例外

なし

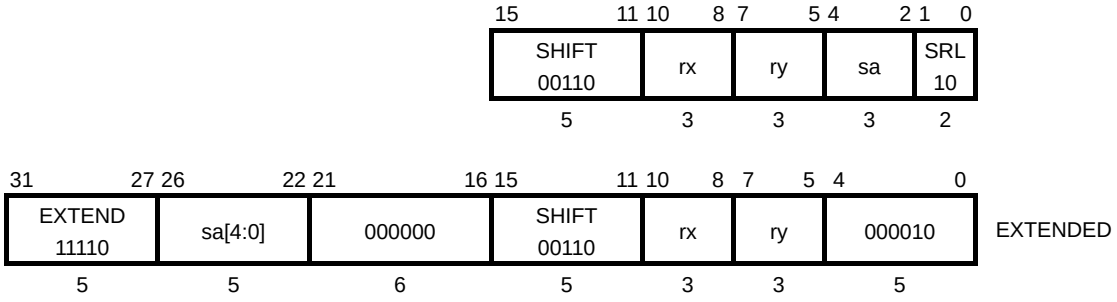
SRL *rx*, *ry*, *sa*

Shift Right Logical

動作

rx ⇐ *ry* >> *sa*

コード



説明

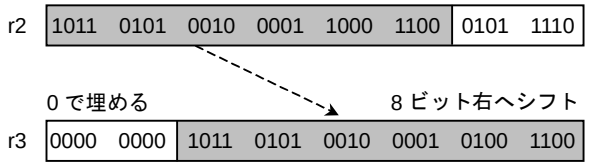
汎用レジスタ *ry* の 32 ビットの内容を *sa* ビット右へシフトし、左端の空いたビットを 0 で埋め、結果を汎用レジスタ *rx* に格納します。*sa* フィールドは 3 ビットです。したがって *sa* で指定できる数値の範囲は 1~8 です。000 は 8 ビットのシフトとして定義されています。

この範囲外のシフト量を指定すると、SRL 命令は EXTEND 命令により拡張され、sa フィールドは 5 ビットになり、0~31 のシフト量が扱えるようになります。

使用例

レジスタ r2 の内容が 0xB521_4C5E の場合、以下の命令を実行すると、レジスタ r3 に 0x00B5_214C が格納されます。

```
SRL r3,r2,8
```



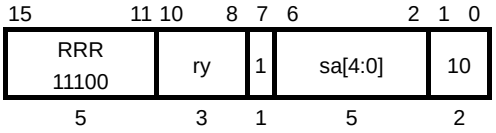
SRL *ry, sa5*

Shift Right Logical

動作

ry ⇐ *ry* >> *sa5*

コード



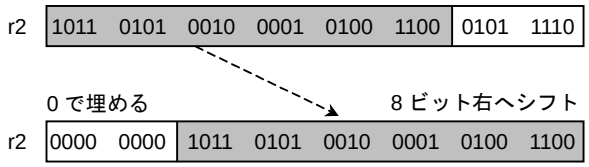
説明

汎用レジスタ *ry* の 32 ビットの内容を *sa* ビット右へシフトし、左端の空いたビットを 0 で埋め、結果を汎用レジスタ *ry* に格納します。*sa* フィールドは 5 ビットです。したがって、*sa* で指定できる数値の範囲は 1~31 です。*sa* フィールドを 00000 と定義することはできません。

使用例

レジスタ r2 の内容が 0xB521_4C5E の場合、以下の命令を実行すると、レジスタ r2 に 0x00B5_214C が格納されます。

```
SRL r2,8
```



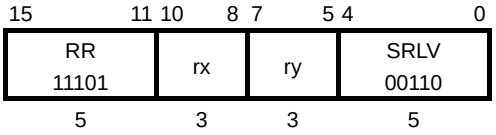
SRLV *ry*, *rx*

Shift Right Logical Variable

動作

ry >> *rx* の下位 5 ビット

コード



説明

汎用レジスタ *ry* の 32 ビットの内容を、汎用レジスタ *rx* の下位 5 ビットで指定されたビット数、右にシフトし、左端の空いたビットを 0 で埋めます。結果を汎用レジスタ *ry* に格納します。

使用例

なし

SSUB *ry, rx, ry*

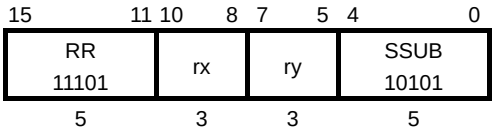
Saturated Subtract

動作

if overflow on *rx* − *ry* then *ry* ⇐ 0x7FFF_FFFF (*rx* ≥ 0) or 0x8000_0000 (*rx* < 0)

else *ry* ⇐ *rx* − *ry*

コード



説明

汎用レジスタ *rx* の内容から汎用レジスタ *ry* の内容を減算し、オーバフローが発生した場合、*rx* の内容が 0 以上の場合は、正の数の最大値 (0x7FFF_FFFF) を、0 より小さい場合は負の数の最小値 (0x8000_0000) を *ry* へ格納する。オーバフローが発生しなかった場合は、減算した結果を *ry* に格納します。*rx* と *ry* の符号付き整数として扱います。

但し、加算した結果がオーバフローしても、整数オーバフロー例外は発生しません。

例外

なし

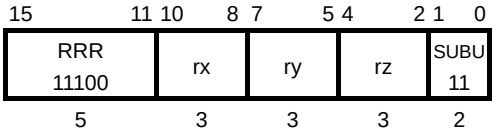
SUBU *rz*, *rx*, *ry*

Subtract Unsigned

動作

rz ⇐ *rx* − *ry*

コード



説明

汎用レジスタ *rx* の内容から汎用レジスタ *ry* の内容を減算し、結果を汎用ジレスタ *rz* に格納します。

整数オーバフロー例外は発生しません。

例外

なし

SW ra, *offset* (sp)

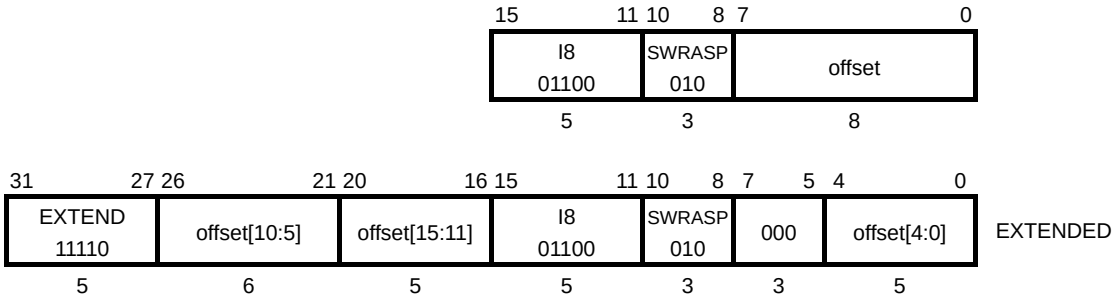
Store Word

動作

ra = {zero-extend (*offset* || 00) + (sp)}

(EXTENDED)　ra = {sign-extend (*offset*) + (sp)}

コード



説明

8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、スタックポインタレジスタ sp (r29) の内容に加算することにより実効アドレス (EA) を生成します。リンクレジスタ ra (r31) の内容をこのアドレスに格納します。

offset は 8 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~1020 になります。この範囲外の値を指定すると、SW 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

例外

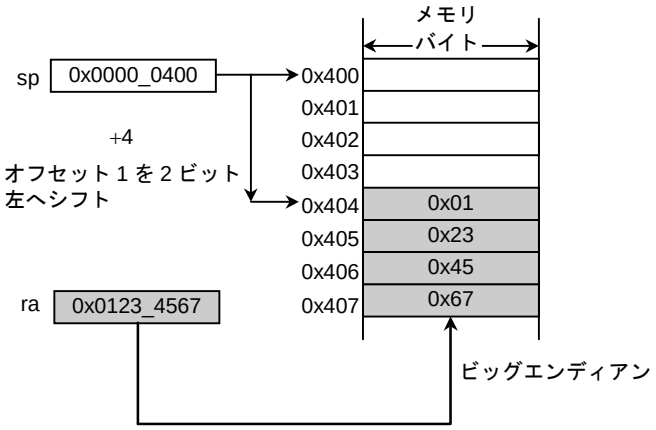
アドレスエラー例外

使用例

```
SW ra,4(sp)
```

レジスタ sp の内容が 0x0000_0400 で、レジスタ ra の内容が 0x0123_4567 であるとします。オフセット値は 2 ビット左へシフトされるので、オフセット (4 = 2 進数 0100) は、アセンブラ・リンカにより、1 (2 進数 0001) に変換されます。したがって、上記のストア命令の命令コードは 0x3101 になります。

ビッグエンディアンのときは、アドレス 0x404~0x407 に 0x0123_4567 がストアされます。


メモリ
バイト
sp
0x0000_0400
+4
オフセット 1 を 2 ビット
左へシフト
ra
0x0123_4567
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ビッグエンディアン

SW *ry*, *offset* (fp)

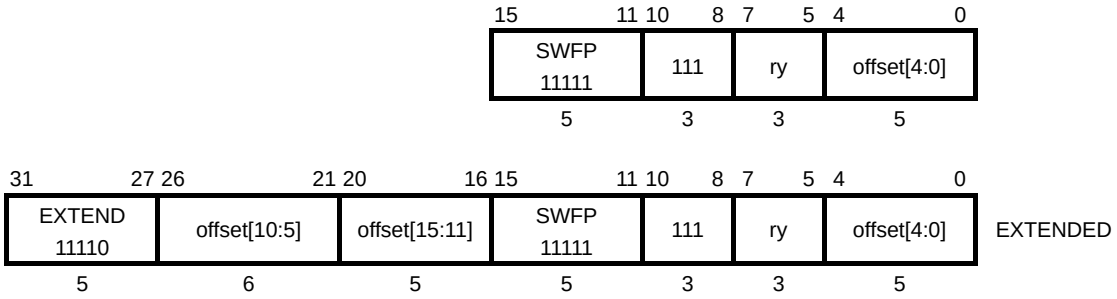
Store Word

動作

ry = {zero-extend (*offset* ‖ 00) + (fp)}

(EXTENDED)　　*ry* = {sign-extend (*offset*) + (fp)}

コード



説明

5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、fp レジスタ (r30) の内容に加算することにより実効アドレス (EA) を生成します。汎用レジスタ *ry* の内容をこのアドレスに格納します。

offset は 5 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~124 になります。この範囲外の値を指定すると、SW 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

例外

アドレスエラー例外

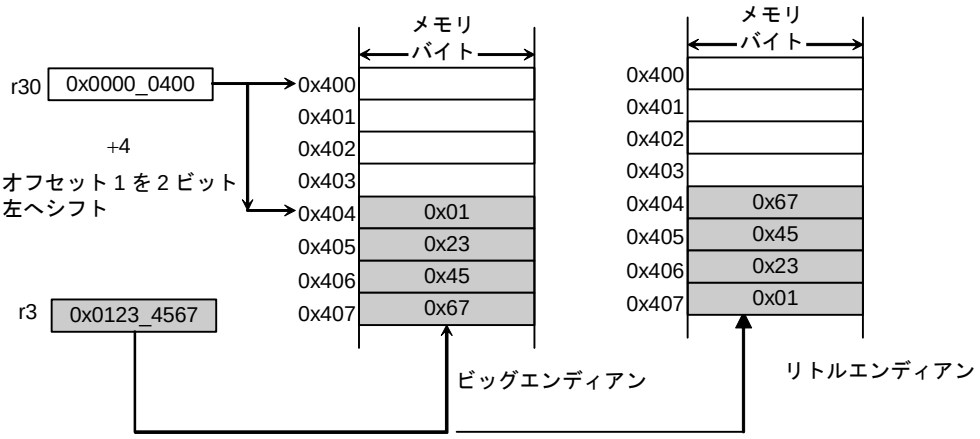
使用例

```
SW r3,4(fp)
```

レジスタ fp の内容が 0x0000_0400 で、レジスタ r3 の内容が 0x0123_4567 であるとします。オフセット値は 2 ビット左へシフトされるので、オフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、1 (2 進数 0001) に変換されます。したがって、上記のストア命令の命令コードは 0xFF61 になります。

ビッグエンディアンのときは、アドレス 0x0404~0x0407 に 0x0123_4567 がストアされます。

リトルエンディアンのときは、アドレス 0x0404~0x0407 に 0x6745_2301 がストアされます。


メモリ
バイト
r30
0x0000_0400
+4
オフセット 1 を 2 ビット
左へシフト
r3
0x0123_4567
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ビッグエンディアン
リトルエンディアン

SW *rx*, *offset* (sp)

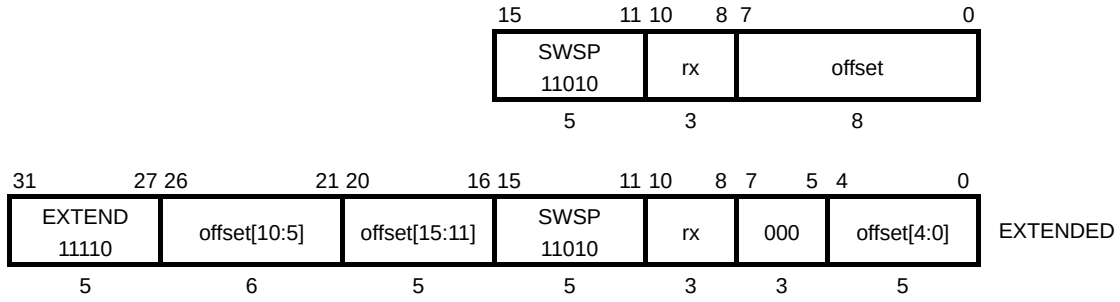
Store Word

動作

rx = {zero-extend (*offset* || 00) + (sp)}

(EXTENDED)　*rx* = {sign-extend (*offset*) + (sp)}

コード



説明

8 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、スタックポインタレジスタ sp (r29) の内容に加算することにより実効アドレス (EA) を生成します。汎用レジスタ *rx* の内容をこのアドレスに格納します。

offset は 8 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~1020 になります。この範囲外の値を指定すると、SW 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。

例外

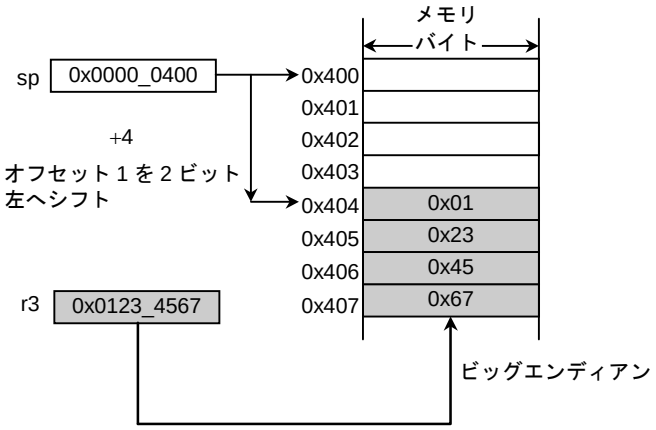
アドレスエラー例外

使用例

```
SW r3,4(sp)
```

レジスタ sp の内容が 0x0000_0400 で、レジスタ r3 の内容が 0x0123_4567 であるとします。オフセット値は 2 ビット左へシフトされるので、オフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、1 (2 進数 0001) に変換されます。したがって、上記のストア命令の命令コードは 0xD301 になります。

ビッグエンディアンのときは、アドレス 0x404~0x407 に 0x0123_4567 がストアされます。


メモリ
バイト
sp
0x0000_0400
+4
オフセット 1 を 2 ビット
左へシフト
r3
0x0123_4567
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ビッグエンディアン

SW *ry*, *offset* (*base*)

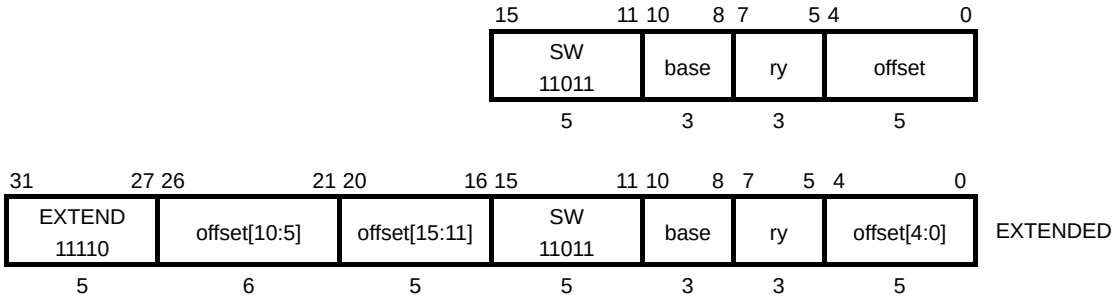
Store Word

動作

ry = {zero-extend (*offset* || 00) + (*base*)}

(EXTENDED)　　*ry* = {sign-extend (*offset*) + (*base*)}

コード



説明

5 ビット *offset* を 2 ビット左へシフトして、ゼロ拡張し、汎用レジスタ *base* の内容に加算することにより実効アドレス (EA) を生成します。汎用レジスタ *ry* の内容を、このアドレスに格納します。

offset は 5 ビットで、2 ビット左へシフトすることにより扱うことのできる数値の範囲は、4 刻みで 0~124 になります。この範囲外の値を指定すると、SW 命令は EXTEND 命令により拡張され、符号付きの 16 ビット即値 (–32768 ~ +32767) を扱えるようになります。この場合、*offset* はシフトされません。

例外

アドレスエラー例外

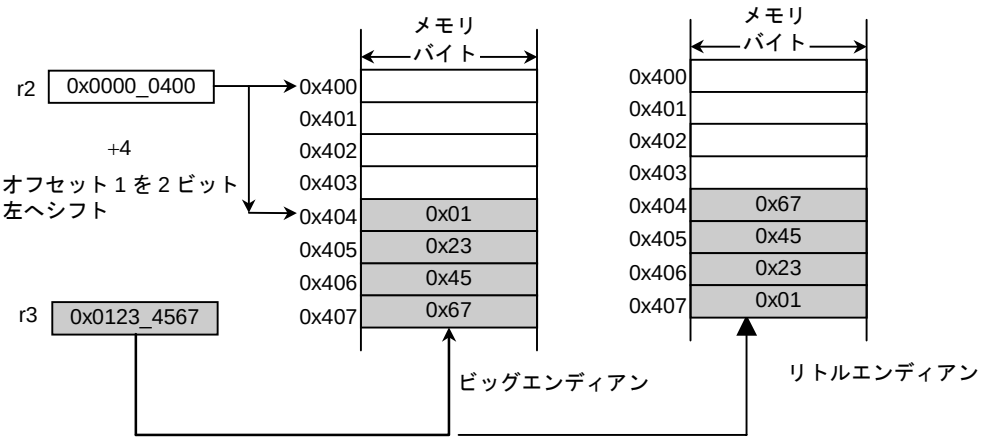
使用例

```
SW r3,4(r2)
```

レジスタ r2 の内容が 0x0000_0400 で、レジスタ r3 の内容が 0x0123_4567 であるとします。オフセット値は、2 ビット左へシフトされるので、指定されたオフセット値 (4 = 2 進数 0100) は、アセンブラ・リンカにより、1 (2 進数 0001) に変換されます。したがって、上記のストア命令の命令コードは 0xDAE1 になります。

ビッグエンディアンのときは、アドレス 0x0404~0x0407 に 0x0123_4567 がストアされます。

リトルエンディアンのときは、アドレス 0x0404~0x0407 に 0x6745_2301 がストアされます。


メモリ
バイト
r2
0x0000_0400
+4
オフセット 1 を 2 ビット
左へシフト
r3
0x0123_4567
0x400
0x401
0x402
0x403
0x404
0x405
0x406
0x407
0x01
0x23
0x45
0x67
ビッグエンディアン
リトルエンディアン

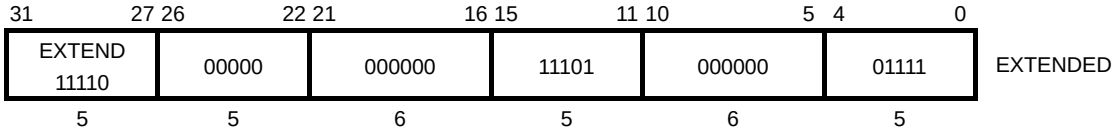
SYNC

Synchronize

動作

メモリ同期操作

コード



説明

SYNC 命令は、SYNC 命令の直前に実行したロード、ストアが完了するまで、命令パイプラインをインタロックし、後続の命令の実行を遅らせます。これにより、SYNC 命令の前の命令と後続の命令の実行順序を守ることができます。「5.2.4 SYNC 命令(32 ビット ISA/16 ビット ISA)」を参照してください。

ロード命令の後続命令が、そのロード結果を使用しない場合、パイプラインをストールせず実行が継続されます。この機能をノンブロッキングロードといいます。パイプラインの他の部分は、データと依存関係のない命令の実行を継続します。

例外

なし

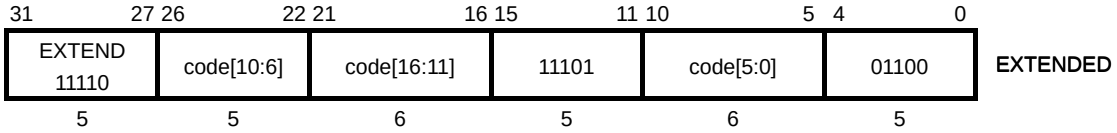
SYSCALL *code*

System Call

動作

システムコール例外

コード



説明

SYSCALL 命令を実行すると、システムコール例外が発生し、無条件に制御を例外ハンドラに渡します。

SYSCALL 命令の *code* フィールドを使用して、例外ハンドラにパラメータを送ることができます。これらのビットを調べるには、EPC レジスタが示す命令の内容をロードします。システムコール例外の詳細は、「9.1.10　システムコール例外」を参照してください。

例外

システムコール例外

WAIT

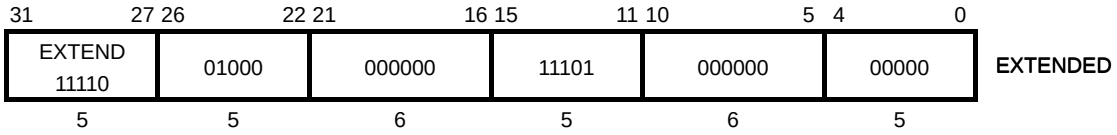
Enter Standby Mode

動作

if Status[RP] = 1　then　DOZE モード

else　HALT モード

コード



説明

WAIT 命令は、内部パイプラインを停止するために使用され、その結果 CPU の消費電力を低減させます。本命令を実行した場合に、Status レジスタの RP ビット＝1 のときは DOZE モードに、RP ビット＝0 の時は HALT モードに遷移します。「第 10 章　低消費電力モード」を参照してください。

WAIT 命令を分岐命令やジャンプ命令の遅延スロットに置くことは禁止されています。

また、WAIT 命令の直前または 2 命令前にて、MTC0 命令を使用し Status レジスタに書きこみを行うことは禁止されています。Status レジスタに書きこみを行った場合動作は不定となります。

例外

コプロセッサ使用不可例外

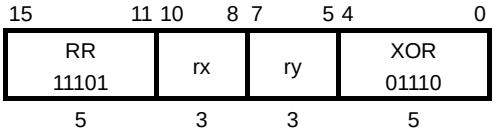
XOR *rx*, *ry*

Exclusive OR

動作

rx ⇐ *rx* XOR *ry*

コード



説明

汎用レジスタ *rx* の内容と汎用レジスタ *ry* の内容との排他的論理和 (XOR) をとり、結果を汎用レジスタ *rx* に格納します。

例外

なし

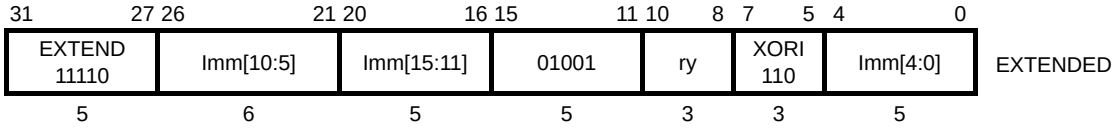
XORI *ry*, *immediate*

Exclusive OR Immediate

動作

$ry \Leftarrow ry \text{ XOR } (0^{16} \parallel immediate_{15..0})$

コード



説明

16 ビット *immediate* をゼロ拡張し、汎用レジスタ *ry* の内容との排他的論理和 (XOR) をとり、結果を汎用レジスタ *ry* に格納します。

immediate は 16 ビットです。これを超える値はいったん汎用レジスタに格納し、XOR 命令を使う必要があります (「4.3.2　32 ビットの定数」を参照)。

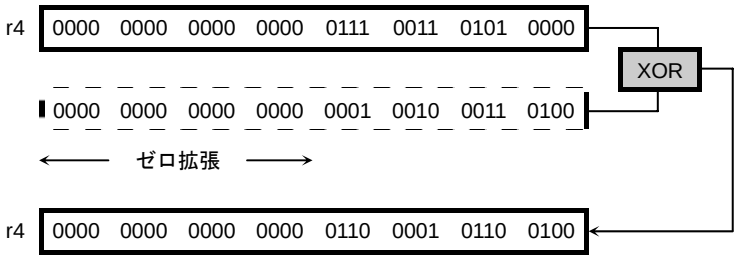
例外

なし

使用例

レジスタ r4 の値が 0x0000_7350 の場合、以下の命令を実行すると、結果の 0x0000_6164 がレジスタ r4 に格納されます。

```
XORI r4,0x1234
```



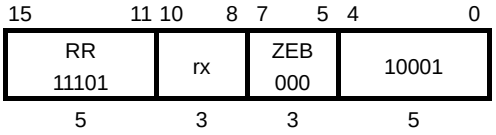
ZEB *rx*

Zero-Extend Byte

動作

$rx \Leftarrow 0^{24} \parallel rx[7:0]$

コード



説明

汎用レジスタ *rx* の下位バイトをゼロ拡張して *rx* に格納する。

例外

なし

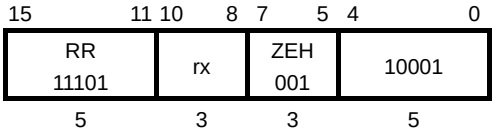
ZEH *rx*

Zero-Extend Halfword

動作

$rx \Leftarrow 0^{16} \parallel rx[15:0]$;

コード



説明

汎用レジスタ *rx* の下位ハーフワードをゼロ拡張して *rx* に格納する。

例外

なし

付録C　プログラミング制約

TX19A のようなパイプラインを備えた CPU では、パイプラインのスムーズな流れを乱す命令があります。この章では、アセンブリ言語プログラムで守らなければならない制約について説明します。

C.1　32 ビット ISA の制約

表 C-1　ロードストア命令

命令	制　約
LH　*rt*, *offset*(*base*) LHU　*rt*, *offset*(*base*) SH　*rt*, *offset*(*base*)	これらの命令で生成されるターゲットアドレスは、ハーフワード境界に位置合わせされていなければなりません。つまり、最下位ビットが 0 でなければなりません。0 でない場合、アドレスエラー例外が発生します。
LW　*rt*, *offset*(*base*) LWU　*rt*, *offset*(*base*) SW　*rt*, *offset*(*base*)	これらの命令で生成されるターゲットアドレスは、ワード境界に位置合わせされていなければなりません。つまり、下位 2 ビットが 0 でなければなりません。下位 2 ビットが 0 でない場合、アドレスエラー例外が発生します。

表 C-2　ジャンプ命令

命令	制　約
JALR　(*rd*,) *rs*	• *rd* として *rs* を指定できません。これは例外処理後、ジャンプ命令から実行を再開できなくなってしまうからです。 • 32 ビット ISA では、命令はすべてワード境界で位置合わせされなければなりません。したがって、32 ビット ISA のルーチンにジャンプするときは、ターゲットレジスタ (*rs*) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。
JR　*rs*	32 ビット ISA では、命令はすべてワード境界に位置合わせされていなければなりません。そのため、32 ビット ISA のルーチンにジャンプするときは、ターゲットレジスタ (*rs*) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。
すべてのジャンプ命令	ジャンプ命令はすべて、ジャンプ・分岐遅延スロットに置いてはいけません。ジャンプ・分岐遅延スロットに置いた場合、ジャンプ命令の動作は不定になります。

表 C-3　分岐・分岐ライクリ命令

命令	制　約
BGEZAL(L)　*rs*, *offset* BLTZAL(L)　*rs*, *offset*	レジスタ *rs* に r31 を指定できません。*rs* として r31 を指定すると、例外処理後、分岐命令から実行を再開できなくなってしまうからです。
すべての分岐命令	分岐命令はすべて、ジャンプ・分岐遅延スロットに置いてはいけません。ジャンプ・分岐遅延スロットに置いた場合、分岐命令の動作は不定になります。

表 C-4　システム制御コプロセッサ (CP0) 命令

命令	制　約
MTC0　*rt*, *rd* MFC0　*rt*, *rd* ERET WAIT	Status レジスタの CU0 ビットがクリアされている場合、ユーザーモードでこれらの命令を実行すると、コプロセッサ使用不可例外が発生します。カーネルモードでは、CU0 ビットの設定に関係なく、これらの命令を実行できます。
DERET	• DERET 命令の遅延スロットはありません。 • デバッグモードでないとき(Debug レジスタの DM ビットが 0 のとき) は、DERET 命令の動作は保証されません。 • MTC0 命令を使って DEPC レジスタに戻りアドレスを設定する場合は、デバッグ例外ハンドラで少なくとも 2 つの命令を実行してから、DERET 命令を実行しなければなりません。
MTC0　*rt*, *rd*	• ERET 命令の直前または 2 命令前に、MTC0 命令により Status、EPC、ErrorEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、Status, EPC, ErrorEPC レジスタの内容は不定になります。 • DERET 命令の直前または 2 命令前に、MTC0 命令により、DEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、DEPC レジスタの内容は不定になります。 • MTC0 命令によって SSCR レジスタを書き替えた場合、後続に 2 つの NOP 命令を必ず置いてください。
WAIT ERET DERET	• これらの命令は遅延スロットに置いてはいけません。
WAIT	• WAIT 命令の直前または 2 命令前にて、MTC0 命令を使用し Status レジスタに書きこみを行うことは禁止されています。Status レジスタに書きこみを行った場合動作は不定となります。

表 C-5　特殊命令

命令	制　約
SDBBP	SDBBP 命令は開発ツールで使用しますので、ユーザープログラムでは、この命令を記述しないでください。

C.2　16 ビット ISA の制約

表 C-6　ロード・ストア命令

命令	制　約
LH *ry*, *offset*(*base*) LHU *ry*, *offset*(*base*) LHU *ry*, *offset*(sp) LHU *ry*, *offset*(fp) SH *ry*, *offset*(*base*) SH *ry*, *offset*(sp) SH *ry*, *offset*(fp)	これらの命令で生成されるターゲットアドレスは、ハーフワード境界で位置合わせされていなければなりません。つまり、最下位ビットが 0 にクリアされていなければなりません。0 でない場合、アドエスエラー例外が発生します。

LW *ry*, *offset*(*base*)
LW *ry*, *offset*(pc)
LW *ry*, *offset*(sp)
LW *ry*, *offset*(fp)
SW *ry*, *offset*(*base*)
SW *ry*, *offset*(sp)
SW *ry*, *offset*(fp)
SW ra, *offset*(sp)
	これらの命令で生成されるターゲットアドレスは、ワード境界に位置合わせされていなければなりません。つまり、下位 2 ビットが 0 でなければなりません。下位 2 ビットが 0 でない場合、アドレスエラー例外が発生します。

表 C-7　ジャンプ命令

命令	制　約
JALR ra, *rx* JALRC ra, *rx*	• レジスタ *rx* に ra を指定できません。*rx* として ra を指定すると、例外処理後、ジャンプ命令から実行を再開できなくなってしまうからです。 • 32 ビット ISA では、命令はすべてワード境界に位置合わせされていなければなりません。そのため、32 ビット ISA のルーチンにジャンプするときは、ターゲットレジスタ (*rx*) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。
JR *rx* JRC *rx*	32 ビット ISA では、命令はすべてワード境界に位置合わせされていなければなりません。そのため、32 ビット ISA のルーチンにジャンプするときは、ターゲットレジスタ (*rx*) の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。
JR ra JRC ra	32 ビット ISA では、命令はすべてワード境界に位置合わせされていなければなりません。そのため、32 ビット ISA のルーチンにジャンプするときは、ra の下位 2 ビットは 0 でなければなりません。下位 2 ビットが 0 でない場合、ジャンプ先の命令をフェッチするときにアドレスエラー例外が発生します。
すべてのジャンプ命令	ジャンプ命令は、遅延スロットに置いてはいけません。

表 C-8　分岐命令

命令	制　約
すべての分岐命令	分岐命令は、ジャンプ遅延スロットに置いてはいけません。

表 C-9　特殊命令

命令	制　約
SDBBP	SDBBP 命令は開発ツールで使用しますので、ユーザープログラム中では、この命令を記述しないでください。

表 C-10　EXTEND 命令

命令	制　約
すべての EXTEND 命令	EXTEND 命令で拡張された命令は、ジャンプ遅延スロットに置いてはいけません。

表 C-11　システム制御コプロセッサ (CP0) 命令

命令	制　約
MTC0　*rt*, *rd* MFC0　*rt*, *rd* ERET WAIT	Status レジスタの CU0 ビットがクリアされている場合、ユーザーモードでこれらの命令を実行すると、コプロセッサ使用不可例外が発生します。カーネルモードでは、CU0 ビットの設定に関係なく、これらの命令を実行できます。
DERET	• DERET 命令の遅延スロットはありません。 • デバッグモードでないとき(Debug レジスタの DM ビットが 0 のとき) は、DERET 命令の動作は保証されません。 • MTC0 命令を使って DEPC レジスタに戻りアドレスを設定する場合は、デバッグ例外ハンドラで少なくとも 2 つの命令を実行してから、DERET 命令を実行しなければなりません。
MTC0　*rt*, *rd*	• ERET 命令の直前または 2 命令前に、MTC0 命令により Status、EPC、ErrorEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、Status, EPC, ErrorEPC レジスタの内容は不定になります。 • DERET 命令の直前または 2 命令前に、MTC0 命令により、DEPC レジスタに書き込むことは禁止されています。そのような書き込みを行うと、DEPC レジスタの内容は不定になります。
MFC0　*rt*, *rd* MTC0　*rt*, *rd*	Config1-3、IER レジスタはアクセスできません。
WAIT ERET DERET	• これらの命令は遅延スロットに置いてはいけません。
WAIT	• WAIT 命令の直前または 2 命令前にて､MTC0 命令を使用し Status レジスタに書きこみを行うことは禁止されています。Status レジスタに書きこみを行った場合動作は不定となります。

表 C-12　SAVE・RESTORE 命令

命令	制　約
すべての SAVE 命令 すべての RESTORE 命令	退避または復元する対象のレジスタを、少なくとも 1 つは設定しなければいけません。

付録D　TX19・TX19A・TX39 の相違点

表 D-1に当社TX19AとTX19 の相違点を示します。

表 D-1　TX19A と TX19 の比較

<table>
<tr><th>特　長</th><th>TX19A</th><th>TX19</th></tr>
<tr><td>アプリケーション</td><td colspan="2">低消費電力、高コード効率</td></tr>
<tr><td>命令セット</td><td>MIPS16e-TX</td><td>MIPS II + MIPS16 ASE</td></tr>
<tr><td>CPU レジスタ</td><td>• 汎用レジスタは、バンク構成 (8 バンク) です。
• それ以外は、TX19 と同じです。</td><td>• 32 本の汎用レジスタ
• プログラムカウンタ (PC)
• PC の最下位ビットが ISA モードを表します。
• 2 本の乗除算レジスタ (HI・LO)</td></tr>
<tr><td>CP0 レジスタ</td><td colspan="2">• TX19A, TX19 でコプロセッサのレジスタ番号が異なります。
• TX19A は、MIPS32 の CP0 に準拠しているためレジスタ番号、ビットアサインが TX19 とは全く異なります。
<table>
<tr><th>レジスタ番号</th><th>TX19A</th><th>TX19</th></tr>
<tr><td>3</td><td>—</td><td>Config</td></tr>
<tr><td>8</td><td>BadVAddr</td><td>BadVAddr</td></tr>
<tr><td>9</td><td>Count/IER</td><td>—</td></tr>
<tr><td>11</td><td>Compare</td><td>—</td></tr>
<tr><td>12</td><td>Status</td><td>Status</td></tr>
<tr><td>13</td><td>Cause</td><td>Cause</td></tr>
<tr><td>14</td><td>EPC</td><td>EPC</td></tr>
<tr><td>15</td><td>PRId</td><td>PRId</td></tr>
<tr><td>16</td><td>Config/Config 1,2,3</td><td>Debug</td></tr>
<tr><td>17</td><td>—</td><td>DEPC</td></tr>
<tr><td>22</td><td>SSCR</td><td>—</td></tr>
<tr><td>23</td><td>Debug</td><td>DESAVE</td></tr>
<tr><td>24</td><td>DEPC</td><td>—</td></tr>
<tr><td>30</td><td>ErrorEPC</td><td>—</td></tr>
</table></td></tr>
<tr><td>パイプライン</td><td colspan="2">5 段</td></tr>
<tr><td>WBU</td><td>32 ビット × 4 段</td><td>なし</td></tr>
<tr><td>乗算命令</td><td colspan="2">レイテンシー / 実行 = 2/ 1 サイクル</td></tr>
<tr><td>積和命令</td><td colspan="2">レイテンシー / 実行 = 2 / 1 サイクル</td></tr>
<tr><td>積差命令</td><td>レイテンシー / 実行 = 2 / 1 サイクル</td><td>なし</td></tr>
<tr><td>除算命令</td><td colspan="2">レイテンシー / 実行 = 35 / 24 サイクル
除算命令が終了する前に、MFHI 命令、MFLO 命令で除算結果を読もうとすると、パイプラインがストールします。</td></tr>
<tr><td>16ISA の拡張命令</td><td>1 サイクル</td><td>2 サイクル</td></tr>
<tr><td>マスカブル
割り込み</td><td>• 2 本のソフトウエア割り込み (IP[1:0])
• 3 本の割り込みコントローラからのハードウエア割り込み (IP[4:2])
• 応答したハードウエア割り込みのレベルに応じて、レジスタバンクが自動的に切り替えられます。</td><td>• 4 本のソフトウエア割り込み (Sw[3:0])
• 1 本の割り込みコントローラからのハードウエア割り込み (7 つの割り込みレベル)</td></tr>
<tr><td>例外ベクタ
アドレス</td><td colspan="2">
<table>
<tr><th colspan="2" rowspan="2">例外</th><th colspan="2">TX19A</th><th colspan="2">TX19</th></tr>
<tr><th>BEV = 1</th><th>BEV = 0</th><th>BEV = 1</th><th>BEV = 0</th></tr>
<tr><td colspan="2">Reset/NMI</td><td colspan="2">0xBFC0_0000</td><td colspan="2">0xBFC0_0000</td></tr>
<tr><td colspan="2" rowspan="2">Debug</td><td colspan="2">0xBFC0_0480</td><td colspan="2">0xBFC0_0200</td></tr>
<tr><td colspan="2">0xFF20_0200</td><td colspan="2">0xFF20_0200</td></tr>
<tr><td rowspan="5">割り込み</td><td>Swi3</td><td>–</td><td>–</td><td>0xBFC0_0240</td><td>0x8000_0140</td></tr>
<tr><td>Swi2</td><td>–</td><td>–</td><td>0xBFC0_0230</td><td>0x8000_0130</td></tr>
<tr><td>Swi1</td><td rowspan="3">0xBFC0_0380
or
0xBFC0_0400</td><td rowspan="3">0x8000_0180
or
0x8000_0200</td><td>0xBFC0_0220</td><td>0x8000_0120</td></tr>
<tr><td>Swi0</td><td>0xBFC0_0210</td><td>0x8000_0110</td></tr>
<tr><td>Hardware</td><td>0xBFC0_0260</td><td>0x8000_0160</td></tr>
<tr><td colspan="2">その他</td><td>0xBFC0_0380</td><td>0x8000_0180</td><td>0xBFC0_0380</td><td>0x8000_0080</td></tr>
</table></td></tr>
</table>

表 D-2にTX19、TX19A（32 ビットISA）、TX39、の命令セットの比較を示します。相違点は網かけで示してあります。

表 D-2　TX19/TX19A 32 ビット ISA と TX39 の命令セット

分類	命令	TX19A 32 ビット ISA		TX19 32 ビット ISA		TX39	
ロード・ストア命令	Load Byte	LB	rt, offset(base)	LB	rt, offset(base)	LB	rt, offset(base)
	Load Byte Unsigned	LBU	rt, offset(base)	LBU	rt, offset(base)	LBU	rt, offset(base)
	Load Halfword	LH	rt, offset(base)	LH	rt, offset(base)	LH	rt, offset(base)
	Load Halfword Unsigned	LHU	rt, offset(base)	LHU	rt, offset(base)	LHU	rt, offset(base)
	Load Word	LW	rt, offset(base)	LW	rt, offset(base)	LW	rt, offset(base)
	Load Word Left	LWL	rt, offset(base)	LWL	rt, offset(base)	LWL	rt, offset(base)
	Load Word Right	LWR	rt, offset(base)	LWR	rt, offset(base)	LWR	rt, offset(base)
	Store Byte	SB	rt, offset(base)	SB	rt, offset(base)	SB	rt, offset(base)
	Store Halfword	SH	rt, offset(base)	SH	rt, offset(base)	SH	rt, offset(base)
	Store Word	SW	rt, offset(base)	SW	rt, offset(base)	SW	rt, offset(base)
	Store Word Left	SWL	rt, offset(base)	SWL	rt, offset(base)	SWL	rt, offset(base)
	StoreWord Right	SWR	rt, offset(base)	SWR	rt, offset(base)	SWR	rt, offset(base)
	Synchronize	SYNC		SYNC		—	
ALU 即値命令	Add Immediate	ADDI	rt, rs, immediate	ADDI	rt, rs, immediate	ADDI	rt, rs, immediate
	Add Immediate Unsigned	ADDIU	rt, rs, immediate	ADDIU	rt, rs, immediate	ADDIU	rt, rs, immediate
	Set On Less Than Immediate	SLTI	rt, rs, immediate	SLTI	rt, rs, immediate	SLTI	rt, rs, immediate
	Set On Less Than Immediate Unsigned	SLTIU	rt, rs, immediate	SLTIU	rt, rs, immediate	SLTIU	rt, rs, immediate
	AND Immediate	ANDI	rt, rs, immediate	ANDI	rt, rs, immediate	ANDI	rt, rs, immediate
	OR Immediate	ORI	rt, rs, immediate	ORI	rt, rs, immediate	ORI	rt, rs, immediate
	Exclusive-OR Immediate	XORI	rt, rs, immediate	XORI	rt, rs, immediate	XORI	rt, rs, immediate
	Load Upper Immediate	LUI	rt, rs, immediate	LUI	rt, rs, immediate	LUI	rt, rs, immediate
2/3 オペランドレジスタタイプ命令	Add	ADD	rd, rs, rt	ADD	rd, rs, rt	ADD	rd, rs, rt
	Add Unsigned	ADDU	rd, rs, rt	ADDU	rd, rs, rt	ADDU	rd, rs, rt
	Subtract	SUB	rd, rs, rt	SUB	rd, rs, rt	SUB	rd, rs, rt
	Subtract Unsigned	SUBU	rd, rs, rt	SUBU	rd, rs, rt	SUBU	rd, rs, rt
	Set On Less Than	SLT	rd, rs, rt	SLT	rd, rs, rt	SLT	rd, rs, rt
	Set On Less Than Unsigned	SLTU	rd, rs, rt	SLTU	rd, rs, rt	SLTU	rd, rs, rt
	AND	AND	rd, rs, rt	AND	rd, rs, rt	AND	rd, rs, rt
	OR	OR	rd, rs, rt	OR	rd, rs, rt	OR	rd, rs, rt
	Exclusive-OR	XOR	rd, rs, rt	XOR	rd, rs, rt	XOR	rd, rs, rt
	NOR	NOR	rd, rs, rt	NOR	rd, rs, rt	NOR	rd, rs, rt
	Count Leading Ones in Word	CLO	rd, rs	—		—	
	Count Leading Zeros in Word	CLZ	rd, rs	—		—	
	Move Conditional on Not Zero	MOVN	rd, rs, rt	—		—	
	Move Conditional on Zero	MOVZ	rd, rs, rt	—		—	

分類	命令	TX19A 32 ビット ISA		TX19 32 ビット ISA		TX39	
シフト命令	Shift Left Logical	SLL	rd, rs, ra	SLL	rd, rs, ra	SLL	rd, rs, ra
	Shift Left Logical Variable	SLLV	rd, rs, rt	SLLV	rd, rs, rt	SLLV	rd, rs, rt
	Shift Right Logical	SRL	rd, rs, sa	SRL	rd, rs, sa	SRL	rd, rs, sa
	Shift Right Logical Variable	SRLV	rd, rs, rt	SRLV	rd, rs, rt	SRLV	rd, rs, rt
	Shift Right Arithmetic	SRA	rd, rs, sa	SRA	rd, rs, sa	SRA	rd, rs, sa
	Shift Right Arithmetic Variable	SRAV	rd, rs, rt	SRAV	rd, rs, rt	SRAV	rd, rs, rt
乗算・除算・積和・積差命令	Multiply	MUL	rd, rs, rt		—		—
		MULT	rs, rt	MULT	rs, rt	MULT	rs, rt
		MULT	rd, rs, rt	MULT	rd, rs, rt	MULT	rd, rs, rt
	Multiply Unsigned	MULTU	rs, rt	MULTU	rs, rt	MULTU	rs, rt
		MULTU	rd, rs, rt	MULTU	rd, rs, rt	MULTU	rd, rs, rt
	Divide	DIV	rs, rt	DIV	rs, rt	DIV	rs, rt
	Divide Unsigned	DIVU	rs, rt	DIVU	rs, rt	DIVU	rs, rt
	Move From HI	MFHI	rd	MFHI	rd	MFHI	rd
	Move From LO	MFLO	rd	MFLO	rd	MFLO	rd
	Move To HI	MTHI	rd	MTHI	rd	MTHI	rd
	Move To LO	MTLO	rd	MTLO	rd	MTLO	rd
	Multiply-and-Add	MADD	rs, rt	MADD	rs, rt	MADD	rs, rt
		MADD	rd, rs, rt	MADD	rd, rs, rt	MADD	rd, rs, rt
	Multiply-and-Add Unsigned	MADDU	rs, rt	MADDU	rs, rt	MADDU	rs, rt
		MADDU	rd, rs, rt	MADDU	rd, rs, rt	MADDU	rd, rs, rt
	Multiply and Subtract	MSUB	rs, rt		—		—
		MSUB	rd, rs, rt		—		—
	Multiply and Subtract Unsigned	MSUBU	rs, rt		—		—
		MSUBU	rd, rs, rt		—		—
ジャンプ命令	Jump	J	target	J	target	J	target
	Jump And Link	JAL	target	JAL	target	JAL	target
	Jump and Link eXchange	JALX	target	JALX	target		—
	Jump Register	JR	rs	JR	rs	JR	rs
	Jump And Link Register	JALR	(rd), rs	JALR	(rd), rs	JALR	(rd), rs
分岐命令	Unconditional Branch	B	offset		—		—
	Branch On Equal	BEQ	rs, rt, offset	BEQ	rs, rt, offset	BEQ	rs, rt, offset
	Branch On Not Equal	BNE	rs, rt, offset	BNE	rs, rt, offset	BNE	rs, rt, offset
	Branch On Greater Than Zero	BGTZ	rs, offset	BGTZ	rs, offset	BGTZ	rs, offset
	Branch On Greater Than Zero or Equal to Zero	BGEZ	rs, offset	BGEZ	rs, offset	BGEZ	rs, offset
	Branch On Less Than Zero	BLTZ	rs, offset	BLTZ	rs, offset	BLTZ	rs, offset
	Branch On Less Than Zero or Equal to Zero	BLEZ	rs, offset	BLEZ	rs, offset	BLEZ	rs, offset
	Branch On Less Than Zero And Link	BLTZAL	rs, offset	BLTZAL	rs, offset	BLTZAL	rs, offset
	Branch On Greater Than Zero And Link	BGEZAL	rs, offset	BGEZAL	rs, offset	BGEZAL	rs, offset

分類	命令	TX19A 32 ビット ISA	TX19 32 ビット ISA	TX39
分岐 Likely 命令	Branch And Link	BAL offset	—	—
	Branch On Equal Likely	BEQL rs, rt, offset	BEQL rs, rt, offset	BEQL rs, rt, offset
	Branch On Not Equal Likely	BNEL rs, rt, offset	BNEL rs, rt, offset	BNEL rs, rt, offset
	Branch On Greater Than Zero Likely	BGTZL rs, offset	BGTZL rs, offset	BGTZL rs, offset
	Branch On Greater Than Zero or Equal to Zero Likely	BGEZL rs, offset	BGEZL rs, offset	BGEZL rs, offset
	Branch On Less Than Zero Likely	BLTZL rs, offset	BLTZL rs, offset	BLTZL rs, offset
	Branch On Less Than Zero or Equal to Zero Likely	BLEZL rs, offset	BLEZL rs, offset	BLEZL rs, offset
	Branch On Less Than Zero And Link Likely	BLTZALL rs, offset	BLTZALL rs, offset	BLTZALL rs, offset
	Branch On Greater Than Zero And Link Likely	BGEZALL rs, offset	BGEZALL rs, offset	BGEZALL rs, offset
トラップ命令	Trap If Equal	TEQ rs, rt	—	—
	Trap If Equal Immediate	TEQI rs, Immediate	—	—
	Trap If Greater Than or Equal	TGE rs, rt	—	—
	Trap If Greater Than or Equal Immediate	TGEI rs, Immediate	—	—
	Trap If Greater Than or Equal Unsigned	TGEU rs, rt	—	—
	Trap If Greater Than or Equal Immediate Unsigned	TGEIU rs, Immediate	—	—
	Trap If Less Than	TLT rs, rt	—	—
	Trap If Less Than Immediate	TLTI rs, Immediate	—	—
	Trap If Less Than Unsigned	TLTU rs, rt	—	—
	Trap If Less Than Immediate Unsigned	TLTIU rs, Immediate	—	—
	Trap If Not Equal	TNE rs, rt	—	—
	Trap If Not Equal Immediate	TNEI rs, Immediate	—	—
コプロセッサ命令	Move To Coprocessor	—	MTCz rt, rd	MTCz rt, rd
	Move From Coprocessor	—	MFCz rt, rd	MFCz rt, rd
	Move Control To Coprocessor	—	CTCz rt, rd	CTCz rt, rd
	Move Control From Coprocessor	—	CFCz rt, rd	CFCz rt, rd
	Coprocessor Operation	—	COPz cofun	COPz cofun
	Branch On Coprocessor z True	—	BCzT offset	BCzT offset
	Branch On Coprocessor z True Likely	—	BCzTL offset	BCzTL offset
	Branch On Coprocessor z False	—	BCzF offset	BCzF offset
	Branch On Coprocessor z False Likely	—	BCzFL offset	BCzFL offset

分類	命令	TX19A 32 ビット ISA	TX19 32 ビット ISA	TX39
システム制御コプロセッサ命令	Move To CP0	MTC0　rt, rd	MTC0　rt, rd	MTC0　rt, rd
	Move From CP0	MFC0　rt, rd	MFC0　rt, rd	MFC0　rt, rd
	Restore From Exception	—	RFE	RFE
	Exception Return	ERET	—	—
	Debug Exception Return	DERET	DERET	DERET
	Cache	—	CACHE　op, offset(base)	CACHE　op, offset(base)
	Read Indexed TLB Entry (*1)	—	(TLBR)	(TLBR)
	Write Indexed TLB Entry (*1)	—	(TLBWI)	(TLBWI)
	Write Random TLB Entry (*1)	—	(TLBWR)	(TLBWR)
	Probe TLB for Matching Entry (*1)	—	(TLBP)	(TLBP)
特殊命令	System Call	SYSCALL　code	SYSCALL　code	SYSCALL　code
	Breakpoint	BREAK　code	BREAK　code	BREAK　code
	Software Debug Breakpoint Exception	SDBBP　code	SDBBP　code	SDBBP　code
	Enter Standby Mode	WAIT	—	—

(*1)：NOP 動作になります。

表 D-3に、TX19 とTX19Aの 16 ビットISAモードとMIPS16 ASEの命令セットの比較を示します。

表 D-3　TX19A/TX19 16 ビット ISA と MIPS16 ASE の命令セット

分類	命令	TX19A16 ビット ISA	TX19 16 ビット ISA	MIPS16 ASE
ロード・ストア命令	Load Byte	LB ry, offset(base)	LB ry, offset(base)	LB ry, offset(base)
	Load Byte Unsigned	LBU ry, offset(base)	LBU ry, offset(base)	LBU ry, offset(base)
		LBU ry, offset(sp)	—	—
		LBU ry, offset(fp)	—	—
	Load Halfword	LH ry, offset(base)	LH ry, offset(base)	LH ry, offset(base)
	Load Halfword Unsigned	LHU ry, offset(base)	LHU ry, offset(base)	LHU ry, offset(base)
		LHU ry, offset(sp)	—	—
		LHU ry, offset(fp)	—	—
	Load Word	LW ry, offset(base)	LW ry, offset(base)	LW ry, offset(base)
		LW ry, offset(pc)	LW ry, offset(pc)	LW ry, offset(pc)
		LW ry, offset(sp)	LW ry, offset(sp)	LW ry, offset(sp)
		LW ry, offset(fp)	—	—
	Load Word Unsigned	—	—	LWU ry, offset(sp)
	Load Doubleword	—	—	LD ry, offset(base)
		—	—	LD ry, offset(pc)
		—	—	LD ry, offset(sp)
	Store Byte	SB ry, offset(base)	SB ry, offset(base)	SB ry, offset(base)
		SB ry, offset(sp)	—	—
		SB ry, offset(fp)	—	—
	Store Halfword	SH ry, offset(base)	SH ry, offset(base)	SH ry, offset(base)
		SH ry, offset(sp)	—	—
		SH ry, offset(fp)	—	—
	Store Word	SW ry, offset(base)	SW ry, offset(base)	SW ry, offset(base)
		SW ry, offset(sp)	SW ry, offset(sp)	SW ry, offset(sp)
		SW ra, offset(sp)	SW ra, offset(sp)	SW ra, offset(sp)
		SW ry, offset(fp)	—	—
	Store Doubleword	—	—	SD ry, offset(base)
		—	—	SD ry, offset(pc)
		—	—	SD ry, offset(sp)
	Synchronize	SYNC	—	—

分類	命令	TX19A 16 ビット ISA	TX19 16 ビット ISA	MIPS16 ASE
ALU 即値命令	Add Immediate	ADDIU ry, rx, immediate	ADDIU ry, rx, immediate	ADDIU ry, rx, immediate
		ADDIU rx, immediate	ADDIU rx, immediate	ADDIU rx, immediate
		ADDIU sp, immediate	ADDIU sp, immediate	ADDIU sp, immediate
		ADDIU rx, pc, immediate	ADDIU rx, pc, immediate	ADDIU rx, pc, immediate
		ADDIU rx, sp, immediate	ADDIU rx, sp, immediate	ADDIU rx, sp, immediate
		ADDIU fp, immediate	—	—
	Doubleword Add Immediate	—	—	DADDIU ry, rx, immediate
		—	—	DADDIU ry, immediate
		—	—	DADDIU sp, immediate
		—	—	DADDIU ry, pc, immediate
		—	—	DADDIU ry, sp, immediate
	Set On Less Than Immediate	SLTI rx, immediate	SLTI rx, immediate	SLTI rx, immediate
	Set On Less Than Immediate Unsigned	SLTIU rx, immediate	SLTIU rx, immediate	SLTIU rx, immediate
	Compare Immediate	CMPI rx, immediate	CMPI rx, immediate	CMPI rx, immediate
	Load Immediate	LI rx, immediate	LI rx, immediate	LI rx, immediate
	Logical AND Immediate	ANDI rx, immediate	—	—
	Logical OR Immediate	ORI rx, immediate	—	—
	Logical Exclusive-OR Immediate	XORI rx, immediate	—	—
	Load Upper Immediate	LUI rx, immediate	—	—
2/3 オペランドレジスタタイプ命令	Add Unsigned	ADDU rz, rx, ry	ADDU rz, rx, ry	ADDU rz, rx, ry
	Doubleword Add Unsigned	—	—	DADDU rz, rx, ry
	Subtract Unsigned	SUBU rz, rx, ry	SUBU rz, rx, ry	SUBU rz, rx, ry
	Doubleword Subtract Unsigned	—	—	DSUBU rz, rx, ry
	Set On Less Than	SLT rx, ry	SLT rx, ry	SLT rx, ry
	Set On Less Than Unsigned	SLTU rx, ry	SLTU rx, ry	SLTU rx, ry
	Compare	CMP rx, ry	CMP rx, ry	CMP rx, ry
	Negate	NEG rx, ry	NEG rx, ry	NEG rx, ry
	AND	AND rx, ry	AND rx, ry	AND rx, ry
	OR	OR rx, ry	OR rx, ry	OR rx, ry
	Exclusive-OR	XOR rx, ry	XOR rx, ry	XOR rx, ry
	Not	NOT rx, ry	NOT rx, ry	NOT rx, ry
	Move	MOVE ry, r32	MOVE ry, r32	MOVE ry, r32
		MOVE r32, rz	MOVE r32, rz	MOVE r32, rz
		MOVE fp, r32	—	—
	Bit Search One Forward	BS1F ry, rx	—	—
	Bit Field Insert	BFINS ry, rx, bit2, bit1	—	—
	Maximum Signed	MAX rz, rx, ry	—	—
	Minimum Signed	MIN rz, rx, ry	—	—
	Sign-Extend Byte	SEB rx	—	—
	Sign-Extend Halfword	SEH rx	—	—
	Zero-Extend Byte	ZEB rx	—	—
	Zero-Extend Halfword	ZEH rx	—	—

分類	命令	TX19A 16 ビット ISA	TX19 16 ビット ISA	MIPS16 ASE
シフト命令	Shift Left Logical	SLL rx, ry, sa	SLL rx, ry, sa	SLL rx, ry, sa
		SLL ry, sa5	—	—
	Shift Left Logical Variable	SLLV ry, rx	SLLV ry, rx	SLLV ry, rx
	Shift Right Logical	SRL rx, ry, sa	SRL rx, ry, sa	SRL rx, ry, sa
		SRL ry, sa5	—	—
	Shift Right Logical Variable	SRLV ry, rx	SRLV ry, rx	SRLV ry, rx
	Shift Right Arithmetic	SRA rx, ry, sa	SRA rx, ry, sa	SRA rx, ry, sa
		SRA ry, sa5	—	—
	Shift Right Arithmetic Variable	SRAV ry, rx	SRAV ry, rx	SRAV ry, rx
	Doubleword Shift Left Logical	—	—	DSLL rx, ry, sa
	Doubleword Shift Left Logical Variable	—	—	DSLLV ry, rx
	Doubleword Shift Right Logical	—	—	DSRL rx, ry, sa
	Doubleword Shift Right Logical Variable	—	—	DSRLV ry, rx
	Doubleword Shift Right Arithmetic	—	—	DSRA rx, ry, sa
	Doubleword Shift Right Arithmetic Variable	—	—	DSRAV ry, rx
SAVE・RESTORE 命令	SAVE	SAVE reg_list3, framesize4	—	—
		SAVE reg_list3, xsregs, aregs, framesize8	—	—
	RESTORE	RESTORE reg_list3, framesize4	—	—
		RESTORE reg_list3, xsregs, aregs, framesize8	—	—
乗算・除算・積和命令	Mutiply	MULT rx, ry	MULT rx, ry	MULT rx, ry
		MULT ry, rx, ry	—	—
	Multiply Unsigned	MULTU rx, ry	MULTU rx, ry	MULTU rx, ry
		MULTU ry, rx, ry	—	—
	Doubleword Mutiply	—	—	DMULT rx, ry
	Doubleword Multiply Unsigned	—	—	DMULTU rx, ry
	Mutiply And Add	MADD rx, ry	—	—
	Mutiply And Add Unsigned	MADDU rx, ry	—	—
	Saturated Add	SADD ry, rx, ry	—	—
	Saturated Subtract	SSUB ry, rx, ry	—	—
	Divide	DIV rx, ry	DIV rx, ry	DIV rx, ry
	Divide Unsigned	DIVU rx, ry	DIVU rx, ry	DIVU rx, ry
	Doubleword Divide	—	—	DDIV rx, ry
	Doubleword Divide Unsigned	—	—	DDIVU rx, ry
	Divide Exception	DIVE rx, ry	—	—
	Divide Exception Unsigned	DIVEU rx, ry	—	—
	Move From HI	MFHI rx	MFHI rx	MFHI rx
	Move From LO	MFLO rx	MFLO rx	MFLO rx
	Move To HI	MTHI rx	—	—
	Move To LO	MTLO rx	—	—

分類	命令	TX19A 16 ビット ISA	TX19 16 ビット ISA	MIPS16 ASE
ビット操作命令	Bit Test	BTST offset(base3), pos3	—	—
		BTST offset(r0), pos3	—	—
		BTST offset(fp), pos3	—	—
	Bit Extract	BEXT offset(base3), pos3	—	—
		BEXT offset(r0), pos3	—	—
		BEXT offset(fp), pos3	—	—
	Bit Clear	BCLR offset(base3), pos3	—	—
		BCLR offset(r0), pos3	—	—
		BCLR offset(fp), pos3	—	—
	Bit Set	BSET offset(base3), pos3	—	—
		BSET offset(r0), pos3	—	—
		BSET offset(fp), pos3	—	—
	Bit Insert	BINS offset(base3), pos3	—	—
		BINS offset(r0), pos3	—	—
		BINS offset(fp), pos3	—	—
	Add Immediate to Memory Word	ADDMIU offset(base3), imm3	—	—
		ADDMIU offset(r0), imm3	—	—
コプロセッサ命令	Move To Coprocessor 0	MTC0 rx, cp0rd32	—	—
	Move From Coprocessor 0	MFC0 ry, cp0rs32	—	—
	Add Coprocessor 0 Immediate Unsigned	AC0IU cp0rt32, imm3	—	—
ジャンプ命令	Jump And Link	JAL target	JAL target	JAL target
	Jump And Link eXchange	JALX target	JALX target	JALX target
	Jump Register	JR rx	JR rx	JR rx
		JR ra	JR ra	JR ra
	Jump Register Compact	JRC rx	—	—
		JRC ra	—	—
	Jump And Link Register	JALR ra, rx	JALR ra, rx	JALR ra, rx
	Jump And Link Register Compact	JALRC ra, rx	—	—
分岐命令	Branch On Equal To Zero	BEQZ rx, offset	BEQZ rx, offset	BEQZ rx, offset
	Branch On Not Equal To Zero	BNEZ rx, offset	BNEZ rx, offset	BNEZ rx, offset
	Branch On T8 Equal To Zero	BTEQZ offset	BTEQZ offset	BTEQZ offset
	Branch On T8 Not Equal To Zero	BTNEZ offset	BTNEZ offset	BTNEZ offset
	Branch Unconditional	B offset	B offset	B offset
	Branch And Link	BAL offset	—	—
特殊命令	Breakpoint	BREAK code	BREAK code	BREAK code
	Software Debug Breakpoint Exception	SDBBP code	SDBBP code	SDBBP code
	Extend	EXTEND immediate	EXTEND immediate	EXTEND immediate
	Disable Interrupt	DI	—	—
	Enable Interrupt	EI	—	—
	System Call	SYSCALL code	—	—
	Exception Return	ERET	—	—
	Debug Exception Return	DERET	—	—
	Enter Standby Mode	WAIT	—	—

付録E　CPU 命令 (32 ビット ISA) のオペコードのビットエンコード

表 E-2～表 E-7にCPU命令セット (MIPS32 ISA) についてオペコードのビットエンコードを示します。

表 E-1　記号の意味

記号	意味
*	この記号のオペコードは､MIPS32 アーキテクチャの将来のバージョン用に確保されています。現行、これらの命令を実行すると予約命令例外が発生します。
β	この記号のオペコードは、予約命令例外が発生します。
θ	この記号のオペコードは、コプロセッサ使用不可例外あるいは、予約命令例外が発生します。
σ	この記号のオペコードは、EJTAG サポート命令として用意されています。EJTAG を実装していない場合は、これらの命令を実行すると予約命令例外が発生します。

表 E-2　MIPS32 Encoding of the Opcode Field

opcode		bits 28..26							
		0	1	2	3	4	5	6	7
bits 31..29		000	001	010	011	100	101	110	111
0	000	*SPECIAL*	*REGIMM*	J	JAL	BEQ	BNE	BLEZ	BGTZ
1	001	ADDI	ADDIU	SLTI	SLTIU	ANDI	ORI	XORI	LUI
2	010	*COP0*	(*COP1*) θ	(*COP2*) θ	(*COP3*) θ	BEQL	BNEL	BLEZL	BGTZL
3	011	β	β	β	β	*SPECIAL2*	JALX	β	*
4	100	LB	LH	LWL	LW	LBU	LHU	LWR	β
5	101	SB	SH	SWL	SW	β	β	SWR	(CACHE)
6	110	(LL)	(LWC1) β	(LWC2) β	(PREF)	β	(LDC1) β	(LDC2) β	β
7	111	(SC)	(SWC1) β	(SWC2) β	*	β	(SDC1) β	(SDC2) β	β

表 E-3　MIPS32 *SPECIAL* Opcode Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	SLL	β	SRL	SRA	SLLV	*	SRLV	SRAV
1	001	JR	JALR	MOVZ	MOVN	SYSCALL	BREAK	*	SYNC
2	010	MFHI	MTHI	MFLO	MTLO	β	*	β	β
3	011	MULT	MULTU	DIV	DIVU	β	β	β	β
4	100	ADD	ADDU	SUB	SUBU	AND	OR	XOR	NOR
5	101	*	*	SLT	SLTU	β	β	β	β
6	110	TGE	TGEU	TLT	TLTU	TEQ	*	TNE	*
7	111	β	*	β	β	β	*	β	β

表 E-4　MIPS32 *REGIMM* Encoding of rt Field

rt		bits 18..16							
		0	1	2	3	4	5	6	7
bits 20..19		000	001	010	011	100	101	110	111
0	00	BLTZ	BGEZ	BLTZL	BGEZL	*	*	*	*
1	01	TGEI	TGEIU	TLTI	TLTIU	TEQI	*	TNEI	*
2	10	BLTZAL	BGEZAL	BLTZALL	BGEZALL	*	*	*	*
3	11	*	*	*	*	*	*	*	*

表 E-5　MIPS32 *SPECIAL2* Encoding of Function Field

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	MADD	MADDU	MUL	θ	MSUB	MSUBU	θ	θ
1	001	θ	θ	θ	θ	θ	θ	θ	θ
2	010	θ	θ	θ	θ	θ	θ	θ	θ
3	011	θ	θ	θ	θ	θ	θ	θ	θ
4	100	CLZ	CLO	θ	θ	β	β	θ	θ
5	101	θ	θ	θ	θ	θ	θ	θ	θ
6	110	θ	θ	θ	θ	θ	θ	θ	θ
7	111	θ	θ	θ	θ	θ	θ	θ	SDBBPσ

表 E-6　MIPS32 *COP0* Encoding of rs Field

rs		bits 23..21							
		0	1	2	3	4	5	6	7
bits 25..24		000	001	010	011	100	101	110	111
0	00	MFC0	β	*	*	MTC0	β	*	*
1	01	*	*	*	*	*	*	*	*
2	10	*CO*							
3	11								

表 E-7　MIPS32 *COP0* Encoding of Function Field When rs = *CO*

function		bits 2..0							
		0	1	2	3	4	5	6	7
bits 5..3		000	001	010	011	100	101	110	111
0	000	*	(TLBR)	(TLBWI)	*	*	*	(TLBWR)	*
1	001	(TLBP)	*	*	*	*	*	*	*
2	010	*	*	*	*	*	*	*	*
3	011	ERET	*	*	*	*	*	*	DERETσ
4	100	WAIT	*	*	*	*	*	*	*
5	101	*	*	*	*	*	*	*	*
6	110	*	*	*	*	*	*	*	*
7	111	*	*	*	*	*	*	*	*

付録F　CPU 命令 (16 ビット ISA) のオペコードのビットエンコード

表 F-1 ~ 表 F-19にCPU命令セット (16 ビットISA) についてオペコードのビットエンコードを示します。

表中の「*」記号のオペコードは、予約命令例外が発生します。

表 F-1　Major Opcode Map

	Instruction Bits [13:11]							
[15:14]	000	001	010	011	100	101	110	111
00	addiusp	addiupc	b	***JAL(X)***	beqz	bnez	***SHIFT***	***FP-B***
01	***RRI-A***	***addiu8***	slti	sltiu	***I8***	li	cmpi	***SP-B***
10	lb	lh	lwsp	lw	lbu	lhu	lwpc	***FP-SP-H***
11	sb	sh	swsp	sw	***RRR***	***RR***	**extend**	***SPECIAL***

表 F-2　***JAL(X)*** Minor Opcode Map

Instruction Bit [26]	
0	1
jal	jalx

表 F-3　***FP-B*** and **extend** + ***FP-B*** Minor Opcode Map

Instruction Bit [7]	
0	1
lbfp	sbfp

表 F-4　***SP-B*** and **extend** + ***SP-B*** Minor Opcode Map

Instruction Bit [7]	
0	1
lbsp	sbsp

表 F-5　***FP-SP-H*** and **extend** + ***FP-SP-H*** Minor Opcode Map

	Instruction Bit [0]	
[7]	0	1
0	lhsp	lhfp
1	shsp	shfp

表 F-6　***SPECIAL*** and **extend** + ***SPECIAL*** Minor Opcode Map

Instruction Bits [10:8]							
000	001	010	011	100	101	110	111
btst	bclr	bset	bins	bal	bext	lwfp	swfp

表 F-7　**extend** + ***addiu8*** Minor Opcode Map

Instruction Bits [7:5]							
000	001	010	011	100	101	110	111
addiu8	*	addmiu	*	andi	ori	xori	lui

表 F-8　***RRI-A*** Minor Opcode Map

Instruction Bit [4]	
0	1
addiu	*

表 F-9　***I8*** and **extend** + ***I8*** Minor Opcode Map

Instruction Bits [10:8]							
000	001	010	011	100	101	110	111
bteqz	btnez	swrasp	adjsp	***SVRS***	mov32r	adjfp	movr32

表 F-10　***I8*** + ***SVRS*** and **extend** + ***I8*** + ***SVRS*** Minor Opcode Map

Instruction Bits [7]	
0	1
restore	save

表 F-11　***RRR*** Minor Opcode Map

	Instruction Bits [1:0]			
[7]	00	01	10	11
0	ac0iu	addu	sra/mthi	subu
1	sll/***INT***		srl/mtlo	

注)　mthi、mtlo、***INT*** は、Instruction Bits[6:2]が、“00000” でなければならない。

表 F-12　***RRR*** + ***INT*** Minor Opcode Map

Instruction Bits [10:8]							
000	001	010	011	100	101	110	111
di	ei	*	*	*	*	*	*

表 F-13　***SHIFT*** Minor Opcode Map

	Instruction Bits [1:0]			
[2]	00	01	10	11
0	sll	mfc0	srl	sra
1		mtc0		

表 F-14　***RR*** Minor Opcode Map

	Instruction Bits [2:0]							
[4:3]	000	001	010	011	100	101	110	111
00	***J(AL)R(C)***	sdbbp	slt	sltu	sllv	break	srlv	srav
01	movfp	*	cmp	neg	and	or	xor	not
10	mfhi	***CNVT***	mflo	*	sadd	ssub	madd	maddu
11	mult	multu	div	divu	mult	multu	dive	diveu

表 F-15　***RR*** + ***J(AL)R(C)*** Minor Opcode Map

Instruction Bits [7:5]							
000	001	010	011	100	101	110	111
jr rs	jr ra	jalr ra,rs	*	jrc rs	jrc ra	jalrc ra,rs	*

表 F-16　***RR*** + ***CNVT*** Minor Opcode Map

Instruction Bits [7:5]							
000	001	010	011	100	101	110	111
zeb	zeh	*	*	seb	seh	*	*

表 F-17　**extend** + ***RR*** Minor Opcode Map

	Instruction Bits [2:0]							
[4:3]	000	001	010	011	100	101	110	111
00	wait	(tlbr)	(tlbwi)	*	*	***MAX/MIN***	(tlbwr)	***BS1F/BFINS***
01	(tlbp)	*	*	*	syscall	*	*	sync
10	*	*	*	*	*	*	*	(cache)
11	eret	*	*	*	*	*	*	deret

表 F-18　**extend** + ***RR*** + ***MAX/MIX*** Minor Opcode Map

Instruction Bit [26]	
0	1
max	min

表 F-19　**extend** + ***RR*** + ***BS1F/BFINS*** Minor Opcode Map

Instruction Bit [26]	
0	1
bfins	bs1f