

この資料は英語版を翻訳したもので、内容に相違が生じる場合には原文を優先します。こちらの日本語版は参考用としてご利用ください。設計の際には、最新の英語版で内容をご確認ください。

NII51003-5.1.0

はじめに

この章では、アセンブリ言語レベルでのプロセッサ機能を中心に、Nios® II プログラミング・モデルについて説明します。以下の機能をプログラマ向けに解説しています。

- 汎用レジスタ、3-1 ページ
- コントロール・レジスタ、3-3 ページ
- スーパーバイザ・モードでの権限とユーザ・モードでの権限、3-5 ページ
- ハードウェア支援デバッグ処理、3-14 ページ
- 例外処理、3-8 ページ
- ハードウェア割り込み、3-9 ページ
- 未実装命令、3-11 ページ
- メモリおよびペリフェラルの構成、3-15 ページ
- キャッシュ・メモリ、3-16 ページ
- プロセッサのリセット状態、3-16 ページ
- 命令セットのカテゴリ、3-17 ページ
- カスタム命令、3-23 ページ

高水準ソフトウェア開発ツールについては、ここでは説明していません。ソフトウェアの開発については、「Nios II ソフトウェア開発ハンドブック」を参照してください。

汎用レジスタ

Nios II アーキテクチャは、r0 から r31 までの 32 個の 32 ビット汎用レジスタを提供します。3-2 ページの表 3-1 を参照してください。一部のレジスタにはアセンブラで認識される名前が付いています。zero レジスタ (r0) は常に値 0 を返すため、zero に書き込んでも無効です。ra レジスタ (r31) は、プロシージャ・コールで使用された戻りアドレスを保持しており、call 命令と ret 命令によって暗黙的にアクセスされます。C および C++ コンパイラは共通のプロシージャ・コール規則を使用して、レジスタ r1 から r23 および r26 から r28 に特定の目的を割り当てます。



詳細については、「Nios II プロセッサ・リファレンス・ハンドブック」の「アプリケーション・バイナリ・インタフェース」の章を参照してください。

et (r24)、bt (r25)、ea (r29)、および ba (r30) など、一部のレジスタへのアクセスは、特定の実行モードに限定されます。詳細については、3-5 ページの「動作モード」を参照してください。

| 表 3-1. Nios II レジスタ・ファイル | | | | | |
|--------------------------|------|-------------|------|----|--------------------|
| 汎用レジスタ | | | | | |
| レジスタ | 名称 | 機能 | レジスタ | 名称 | 機能 |
| r0 | zero | 0x00000000 | r16 | | |
| r1 | at | 一時的なアセンブラ | r17 | | |
| r2 | | 戻り値 | r18 | | |
| r3 | | 戻り値 | r19 | | |
| r4 | | レジスタ引数 | r20 | | |
| r5 | | レジスタ引数 | r21 | | |
| r6 | | レジスタ引数 | r22 | | |
| r7 | | レジスタ引数 | r23 | | |
| r8 | | 呼び出し元保存レジスタ | r24 | et | 一時的な例外 (1) |
| r9 | | 呼び出し元保存レジスタ | r25 | bt | 一時的なブレークポイント (2) |
| r10 | | 呼び出し元保存レジスタ | r26 | gp | グローバル・ポインタ |
| r11 | | 呼び出し元保存レジスタ | r27 | sp | スタック・ポインタ |
| r12 | | 呼び出し元保存レジスタ | r28 | fp | フレーム・ポインタ |
| r13 | | 呼び出し元保存レジスタ | r29 | ea | 例外戻りアドレス (1) |
| r14 | | 呼び出し元保存レジスタ | r30 | ba | ブレークポイント戻りアドレス (2) |
| r15 | | 呼び出し元保存レジスタ | r31 | ra | 戻りアドレス |

表 3-1 の注：

- (1) このレジスタはユーザ・モードでは使用できません。
- (2) このレジスタはユーザ・モードまたはスーパーバイザ・モードでは使用できません。JTAG デバッグ・モジュールで排他的に使用されます。

コントロール・レジスタ

ct10 から ct15 までの 6 個の 32 ビット・コントロール・レジスタがあります。すべてのコントロール・レジスタには、アセンブラで認識される名前が付いています。

コントロール・レジスタは、汎用レジスタとは異なる方法でアクセスされます。特殊命令の rdctl および wrctl によってのみ、コントロール・レジスタの読み出しと書き込みが可能です。コントロール・レジスタはスーパーバイザ・モードでのみアクセス可能で、ユーザ・モードではアクセスできません。詳細については、3-5 ページの「動作モード」を参照してください。

コントロール・レジスタの詳細は、表 3-2 に示します。コントロール・レジスタと例外処理の関係についての詳細は、3-10 ページの図 3-2 を参照してください。

| レジスタ | 名称 | 31...2 | 1 | 0 |
|------|----------|----------------|----|------|
| ct10 | status | 予約済み | U | PIE |
| ct11 | estatus | 予約済み | EU | EPIE |
| ct12 | bstatus | 予約済み | BU | BPIE |
| ct13 | ienable | 割り込みイネーブル・ビット | | |
| ct14 | ipending | 割り込みペンディング・ビット | | |
| ct15 | cpuid | 固有のプロセッサ識別子 | | |

status (ct10)

status レジスタの値によって、Nios II プロセッサの状態を制御します。プロセッサのリセット後に、すべてのステータス・ビットがクリアされます。3-16 ページの「プロセッサのリセット状態」を参照してください。表 3-3 に示すように、PIE と U の 2 ビットが定義されています。

| ビット | 説明 |
|---------|--|
| PIE ビット | PIE はプロセッサ割り込みイネーブル・ビットです。PIE が 0 の場合、外部割り込みは無視されます。PIE が 1 の場合、ienable レジスタの値に応じて、外部割り込みの受け入れが可能です。 |
| U ビット | U はユーザ・モード・ビットです。1 はユーザ・モード、0 はスーパーバイザ・モードを示します。 |

estatus (ctl1)

estatus レジスタは、例外処理中の status レジスタの保存されたコピーを保持します。EPIE と EU の 2 ビットが定義されています。これらは、表 3-3 で定義するとおり、PIE と U の値を保存したものです。

例外ハンドラで estatus を調べて、例外発生前のプロセッサのステータスを確認できます。プロセッサは、割り込みからの復帰時に eret 命令によって、estatus を status にコピーし、例外発生前の status の値を復元します。



詳細については、3-8 ページの「例外処理」を参照してください。

bstatus (ctl2)

bstatus レジスタは、デバッグ・ブレイク処理中の status レジスタの保存されたコピーを保持します。EPIE と BU の 2 ビットが定義されています。これらは、3-3 ページの表 3-3 で定義するとおり、PIE と U の値を保存したものです。

ブレイクが発生すると、status レジスタの値が bstatus にコピーされます。bstatus を使用すると、status レジスタをブレイク発生前の値に復元することができます。



詳細については、3-7 ページの「デバッグ・モード」を参照してください。

ienable (ctl3)

ienable レジスタは、外部ハードウェア割り込みの処理を制御します。ienable レジスタの各ビットは、irp0 から irp31 までの割り込み入力の 1 つに対応します。ビット値が 1 の場合は対応する割り込みがイネーブルされていることを意味し、ビット値が 0 の場合は、対応する割り込みがディセーブルされていることを意味します。



詳細については、3-8 ページの「例外処理」を参照してください。

ipending (ctl4)

ipending レジスタの値は、どの割り込みがペンディング中であるかを示します。ビット n の値が 1 の場合は対応する irqn 入力のアサートされ、対応する割り込みが ienable レジスタでイネーブルされていることを意味します。ipending レジスタに値を書き込んだ場合の影響は不定です。

cpuid (ctl5)

cpuid レジスタは、マルチプロセッサ・システムでプロセッサを一意に識別するスタティック値を保持します。cpuid の値はシステム生成時に決定されます。cpuid レジスタに値を書き込んでも無効となります。



詳細については、3-8 ページの「例外処理」を参照してください。

動作モード

Nios II プロセッサには、以下の 3 つの動作モードがあります。

- スーパーバイザ・モード
- ユーザ・モード
- デバッグ・モード

以下のセクションでは、モードおよびモード間の移行について説明します。この説明では、システム・コードとアプリケーション・コードを区別しています。

- システム・コードは、オペレーティング・システム (OS) や低水準ハードウェア・ドライバなど、システム・レベルの機能を実行するルーチンで構成されています。一般に、システム・コードはランタイム・ライブラリまたは OS カーネルの一部として提供されます。通常、システム・コードはスーパーバイザ・モードで実行されます。
- アプリケーション・コードは、システム・コードで提供されるサービス上で動作するルーチンで構成されています。アプリケーション・コードは、一般にターゲット・アプリケーションを作成するプログラマが記述します。

スーパーバイザ・モード

スーパーバイザ・モードでは、すべての定義済みのプロセッサ機能が利用でき制約はありません。一般に、システム・コードはスーパーバイザ・モードで実行されます。ただし、オペレーティング・システムを使用しないシンプルなプログラムは無限にスーパーバイザ・モードのままで、アプリケーション・コードはスーパーバイザ・モードのもとで正常に実行できます。

汎用レジスタ bt (r25) および ba (r30) は、スーパーバイザ・モードでは利用できません。プログラムがこれらのレジスタに値を格納しないようにすることはできませんが、値が格納された場合、その値はデバッグ・モードで変更できます。また、bstatus レジスタ (ctl12) もスーパーバイザ・モードでは利用できません。

プロセッサがスーパーバイザ・モードの場合、U ビットは 0 です。プロセッサは、リセット直後はスーパーバイザ・モードになります。

ユーザ・モード

ユーザ・モードは、スーパーバイザ・モード機能の限定されたサブセットを提供します。ユーザ・モードでは、複数のタスクを監視するオペレーティング・システムに対して高い信頼性が実現されます。システム・コードは、ユーザ・モードに切り替えてからアプリケーション・コードに制御を渡すことを選択できます。

ユーザ・モードでは、一部のプロセッサ機能にはアクセスできず、アクセスを試みると例外が発生します。コントロール・レジスタはユーザ・モードでは利用できません。さらに、汎用レジスタの `et` (r24)、`bt` (r25)、`ea` (r29)、および `ba` (r30) も利用できません。ユーザ・モードで実行しているプログラムがこれらのレジスタに値を格納しないようにすることはできませんが、格納された値はスーパーバイザ・モードの例外ルーチンまたはデバッグ・モードで変更できます。

プロセッサがユーザ・モードの場合、以下の命令のいずれかを発行すると例外が発生します。

- `rdctl`
- `wrctl`
- `bret`
- `eret`
- `initd`
- `initi`

プロセッサがユーザ・モードの場合、U ビットは 1 です。

プロセッサ実装とユーザ・モード・サポート

一部の Nios II プロセッサ実装では、ユーザ・モードはサポートされていません。これらのコアでは、すべてのコードがスーパーバイザ・モードで実行され、U ビットは常に 0 になります。したがって、正しく実行させるには、U ビットの特定の値に動作が左右されないようにアプリケーション・コードを記述する必要があります。

アプリケーション・コードは、ユーザ・モードとスーパーバイザ・モードのどちらでも正常に実行されます。ユーザ・モードをサポートしていない Nios II プロセッサ・コアでは、システム・コードからユーザ・モードまたは制限されたリソース保護のためのアクセス違反例外を利用することはできません。



ユーザ・モードをサポートするプロセッサについて詳しくは、「Nios II プロセッサ・リファレンス・ハンドブック」の「[Nios II コア実装の詳細](#)」の章を参照してください。

デバッグ・モード

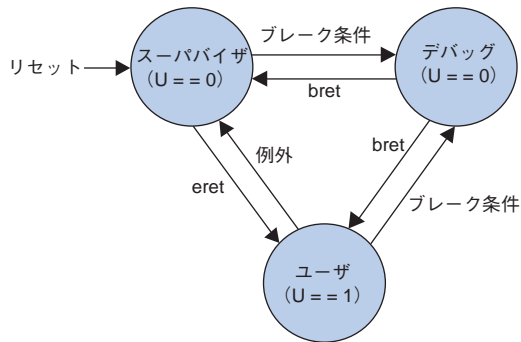
デバッグ・モードは、ブレークポイントやウォッチポイントなどの機能を実現するために、ソフトウェア・デバッグ・ツールで使用されています。システム・コードとアプリケーション・コードはデバッグ・モードでは実行できません。プロセッサは、break 命令後、または JTAG デバッグ・モジュールがハードウェアでブレークを強制した場合にのみ、デバッグ・モードに移行します。

デバッグ・モードでは、ソフトウェア・デバッグ・ツールからすべてのプロセッサ機能が無制限に利用できます。デバッグ・モードでは U ビットは 0 です。詳細については、3-14 ページの「[ブレーク処理](#)」を参照してください。

モードの変更

図 3-1 は、ユーザ・モード、スーパーバイザ・モード、およびデバッグ・モード間の移行を示します。

図 3-1. 動作モード間の遷移



プロセッサは、リセット後にスーパーバイザ・モードで起動します。

プログラムは、eret (例外復帰) 命令を使用してスーパーバイザ・モードからユーザ・モードに切り替えることができます。eret は、estatus レジスタ (ctl11) の値を status レジスタ (ctl10) にコピーし、ea (r29) レジスタのアドレスに制御を移します。プロセッサをリセットしてから最初にユーザ・モードに入るには、システム・コードは estatus レジスタと ea レジスタを設定し、eret 命令を実行する必要があります。

プロセッサは例外処理が発生するまでユーザ・モードで維持され、例外処理が発生した時点でスーパーバイザ・モードに再入します。すべての例外処理で、U ビットは 0 にクリアされ、status の内容は estatus に保存されます。例外処理ルーチンが estatus レジスタを変更しないと仮定すれば、eret を使用して例外処理から復帰すると、例外処理発生前のモードに復元されます。

プロセッサは、ソフトウェア・デバッグ・ツールで指示された場合のみ、デバッグ・モードに入ります。システム・コードとアプリケーション・コードは、プロセッサがデバッグ・モードに移行するタイミングを制御することはできません。プロセッサはデバッグ・モードを終了すると、必ず前の状態に戻ります。



詳細については、3-8 ページの「例外処理」と 3-14 ページの「ブレイク処理」を参照してください。

例外処理

例外は、プログラムの通常の実行フローから制御を移すことであり、これはプロセッサの内部または外部のイベントによって発生し、即時の対処が求められます。例外処理とは、例外に反応して動作し、その後で例外が発生する前の実行状態に戻ることをいいます。

例外が発生すると、プロセッサは自動的に以下のステップを実行します。プロセッサは、

1. status レジスタ (ct10) の内容を estatus レジスタ (ct11) にコピーして、例外発生前のプロセッサの状態を保存します。
2. status レジスタの U ビットをクリアして、プロセッサを強制的にスーパーバイザ・モードにします。
3. status レジスタの PIE ビットをクリアして、外部プロセッサ割り込みをディセーブルします。
4. 例外発生後の命令のアドレスを ea レジスタ (r29) に書き込みます。
5. 割り込みの原因を特定する例外ハンドラのアドレスに実行を移します。

例外ハンドラのアドレスは、システム生成時に指定されます。このアドレスは実行時に確定され、ソフトウェアでは変更できません。プログラマが例外ハンドラのアドレスに直接アクセスすることはなく、アドレスを意識しないでプログラムを記述できます。

例外ハンドラは各例外の原因を特定し、適切な例外ルーチンをディスパッチして割り込みに応答します。



例外と割り込み処理を利用するプログラムの記述に関する詳細な説明は、「Nios II ソフトウェア開発ハンドブック」の「例外処理」の章を参照してください。

例外タイプ

Nios II の例外は以下のカテゴリに分類されます。

- ハードウェア割り込み
- ソフトウェア・トラップ
- 未実装命令
- その他

各例外タイプについては、以下の項で詳細に説明します。

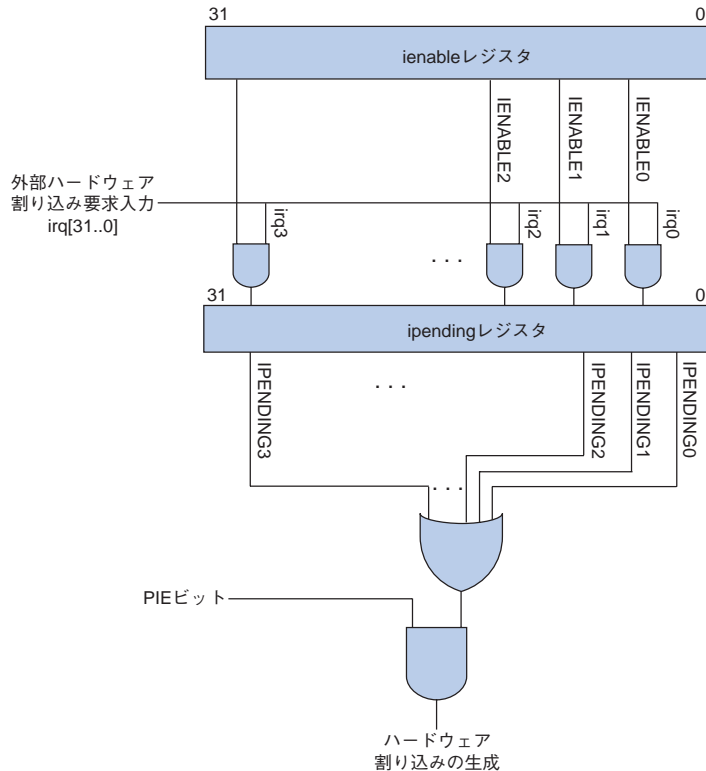
ハードウェア割り込み

ペリフェラル・デバイスなどの外部ソースは、プロセッサの 32 個の割り込み要求入力、`irq0` から `irq31` のいずれかをアサートすることによって、ハードウェア割り込みを要求できます。ハードウェア割り込みは、以下の 3 つの条件がすべて満たされた場合にのみ生成されます。

- `status` レジスタ (`ct10`) の `PIE` ビットが 1
- 割り込み要求入力 `irqn` がアサートされている
- `ienable` レジスタ (`ct13`) の対応するビット `n` が 1

ハードウェア割り込み時に `PIE` ビットは 0 に設定され、以降の割り込みをディセーブルします。`ipending` レジスタ (`ct14`) の値は、どの割り込み要求 (IRQ) がペンディングされているかを示します。ペリフェラル・デザインによって、プロセッサが明示的にペリフェラルに応答するまで、IRQ ビットは必ずアサートされたままです。図 3-2 に、`ipending`、`ienable`、`PIE`、および割り込みの生成の関係を示します。

図 3-2. ienable、ipending、PIE およびハードウェア割り込みの関係



ソフトウェア例外ルーチンは、ペンディング中の割り込みでどれが最も優先順位が高いかを判定し、適切な割り込みサービス・ルーチン (ISR) に制御を移します。ISR は、復帰前または PIE を再イネーブルする前に、割り込みを認識できないようにする (ソースでクリアするか、ienable を使用してマスクする) 必要があります。また ISR は、PIE を再イネーブルする前に、estatus (ct11) と ea (r29) を保存しなければなりません。

割り込みは、PIE ビットに 1 を書き込むと再イネーブルでき、それによって現在の ISR への割り込みを発生させることが可能です。通常、例外ルーチンは ienable を調整して、割り込みを再イネーブルする前に、優先順位が同じまたは低い IRQ がディセーブルされるようにします。



3-13 ページの「ネスト式例外」を参照してください。

ソフトウェア・トラップ

プログラムで trap 命令を発行すると、ソフトウェア・トラップ例外が生成されます。通常、プログラムはオペレーティング・システムによる処理を必要とするときに、ソフトウェア・トラップを発行します。オペレーティング・システム用の例外ハンドラは、トラップの原因を特定し、適切に応答します。

未実装命令

プロセッサがハードウェアで実装されていない有効な命令を発行すると、未実装命令例外が生成されます。例外ハンドラは例外を生成した命令を特定します。その命令がハードウェアで実装されていない場合は、ソフトウェアで動作をエミュレートする例外ルーチンに制御が渡されます。



詳細については、3-24 ページの「未実装の可能性のある命令」を参照してください。



「未実装命令」は「無効な命令」を意味するものではないことに注意してください。未定義つまり無効な命令ワードに対するプロセッサの動作は、Nios II コアによって異なります。大部分の Nios II コア実装では、無効な命令を実行したときの結果は不定です。詳細については、「Nios II プロセッサ・リファレンス・ハンドブック」の「Nios II コア実装の詳細」の章を参照してください。

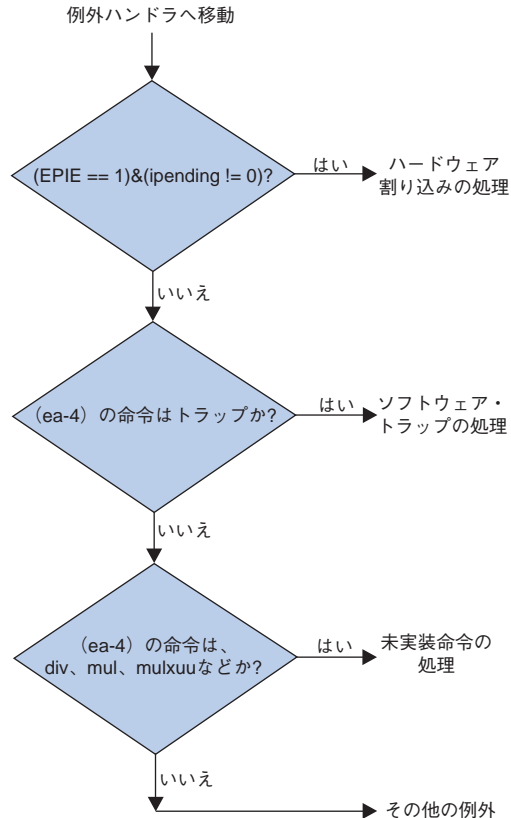
その他の例外

ここまでのセクションでは、本書の発行時点で Nios II アーキテクチャによって定義されているすべての例外タイプについて説明しています。ただし、一部のプロセッサ実装では、上記のカテゴリに分類されない例外が生成される可能性があります。例えば、今後の実装では、アクセス違反例外を生成するメモリ管理ユニット (MMU) が搭載される可能性があります。したがって、例外ハンドラの堅牢なものにするには、例外の原因を正確に特定できない場合は安全な応答 (警告を発行するなど) を提供することが必要です。

例外の原因の特定

例外ハンドラは、各例外の原因を特定し、適切な例外ルーチンに制御を移すことが必要です。図 3-3 に、例外のソースを特定するのに使用されるプロセスの例を示します。

図 3-3. 例外の原因を特定するプロセス



estatus レジスタ (ct11) の EPIE ビットが 1 で、ipending レジスタ (ct14) の値がゼロでない場合、例外は外部ハードウェア割り込みによって発生しています。そうでない場合、例外はソフトウェア・トラップまたは未実装命令によって発生した可能性があります。ソフトウェア・トラップと未実装命令を区別するには、アドレス ea-4 の命令を読み出します (このアドレスを読み出すには、Nios II データ・マスタがコード・メモリにアクセスできることが必要です)。命令が trap であれば、例外はソフトウェア・トラップです。アドレス ea-4 の命令が、ソフトウェアで実装された可能性のある命令の場合、例外は未実装命令によって発生しています。詳細については、[3-24 ページの「未実装の可能性のある命令」](#)を参照してください。上記のいずれの状態にも該当しない場合、例外タイプは認識されないため、例外ハンドラはその状態を報告する必要があります。

ネスト式例外

例外ルーチンでは、以下を実行する前に、特別な対策が必要です。

- trap 命令の発行
- 未実装命令の発行
- ハードウェア割り込みの再イネーブル

これらの動作を許可する前に、例外ルーチンは `estatus (ct11)` および `ea (r29)` を保存して、復帰前にこれらを正しく復元できるようにしておく必要があります。

例外からの復帰

`eret` 命令は、例外発生前のアドレスから実行を再開するのに使用されます。`et` レジスタ (`r24`) を除いて、例外処理中に変更されるレジスタは、例外処理から復帰する前に例外ルーチンで復元する必要があります。

`eret` 命令の実行時に、プロセッサは以下を行います。

1. `estatus (ct11)` の内容を `status (ct10)` にコピーする
2. プログラムの実行を `ea` レジスタ (`r29`) のアドレスに移す

戻りアドレス

戻りアドレスについては、例外処理ルーチンから復帰する場合に多少の考慮が必要です。例外が発生すると、`ea` には、例外が生成された位置より後の命令のアドレスが格納されます。

ソフトウェア・トラップおよび未実装命令例外から復帰するときには、ソフトウェア・トラップまたは未実装命令に続く命令から実行を再開する必要があります。したがって、`ea` には正しい戻りアドレスが格納されています。

それに対して、ハードウェア割り込み例外では、割り込みを受けた命令自体から実行を再開することが必要です。この場合、例外ハンドラは割り込みを受けた命令を指すように、`ea` から4を減算する必要があります。

ブレイク処理

ブレイクとは、break 命令または JTAG デバッグ・モジュールによって、プログラムの通常の実行フローから制御を移すことです。ソフトウェア・デバッグ・ツールでは、JTAG デバッグ・モジュールを介して Nios II プロセッサを制御することができます。デバッグ・モードで実行する場合は、デバッグ・ツールのみがプロセッサを制御し、アプリケーションとシステム・コードはこのモードでは実行できません。

ブレイク処理は、ソフトウェア・デバッグ・ツールがブレイクポイントやウォッチポイントなどのデバッグ機能や診断機能を実現する手段です。ブレイク処理は例外処理と類似していますが、ブレイク・メカニズムは例外処理から独立しています。ブレイクは例外処理中でも発生させることができますため、デバッグ・ツールで例外ハンドラのデバッグが可能です。

ブレイクの処理

プロセッサは、以下の条件でブレイク処理状態に入ります。

- プロセッサが break 命令を発行する
- JTAG デバッグ・モジュールがハードウェア・ブレイクをアサートする

ブレイクが発生すると、プロセッサは以下の処理を自動的に実行します。プロセッサは、

1. status レジスタ (ct10) の内容を bstatus (ct12) に格納します。
2. status レジスタの U ビットをクリアして、プロセッサを強制的にスーパーバイザ・モードにします。
3. status レジスタの PIE ビットをクリアして、外部プロセッサ割り込みをディセーブルします。
4. ブレイクに続く命令のアドレスを ba (r30) レジスタに書き込みます。
5. 実行をブレイク・ハンドラのアドレスに移します。ブレイク・ハンドラのアドレスは、システム生成時に指定されます。

ブレイクからの復帰

デバッグ・ツールは、ブレイク処理を実行した後、bret 命令を実行してプロセッサの制御を解放します。bret 命令は、status を復元し、ba 内のアドレスにプログラムの実行を戻します。

レジスタの使用

ブレイク・ハンドラは、bt (r25) を使用して、追加レジスタの保存に役立てることができます。bt を除くすべてのレジスタは、ブレイク処理ルーチンからの復帰後に必ずブレイク前の状態に戻ります。

メモリ・アクセスとペリフェラル・アクセス



Nios II のアドレスは 32 ビットであり、最大 4 ギガバイトのアドレス空間にアクセス可能です。しかし、多くの Nios II コア実装では、アドレスは 31 ビット以下に制限されています。

詳しくは、「Nios II プロセッサ・リファレンス・ハンドブック」の「[Nios II コア実装の詳細](#)」の章を参照してください。

ペリフェラル、データ・メモリ、およびプログラム・メモリは同じアドレス空間にマップされます。アドレス空間内のメモリとペリフェラルの位置は、システム生成時に決定されます。メモリまたはペリフェラルにマップされないアドレスに読み書きを行った場合、結果は不定です。

プロセッサのデータ・バスの幅は 32 ビットです。バイト、ハーフワード (16 ビット)、またはワード (32 ビット) データの読み出しと書き込みを実行する命令が用意されています。

Nios II アーキテクチャはリトル・エンディアンです。9 ビット以上の幅のデータがメモリ内に格納される場合、上位ビットが上位アドレスに配置されます。

アドレス指定モード

Nios II アーキテクチャは以下のアドレス指定モードをサポートしています。

- レジスタ・アドレス指定
- ディスプレースメント・アドレス指定
- 即値アドレス指定
- レジスタ間接アドレス指定
- 絶対アドレス指定

レジスタ・アドレス指定では、すべてのオペランドがレジスタとなり、結果はレジスタに再格納されます。ディスプレースメント・アドレス指定では、アドレスはレジスタと符号付き 16 ビット即値として計算されます。即値アドレス指定では、オペランドは命令自体に含まれる定数です。レジスタ間接アドレス指定は、ディスプレースメント・アドレス指定を使用しますが、ディスプレースメントは定数 0 です。限定範囲の絶対アドレス指定は、常に値が 0x00 のレジスタ `r0` によるディスプレースメント・アドレス指定を使用して実行されます。

キャッシュ・メモリ

Nios II のアーキテクチャと命令セットは、データ・キャッシュ・メモリと命令キャッシュ・メモリの有無に対応しています。キャッシュ管理は、キャッシュ管理命令を使用してソフトウェアで実行されます。キャッシュの初期化、必要に応じたキャッシュのフラッシュ、およびデータ・キャッシュのバイパスによるメモリ・マップド・ペリフェラルへの適切なアクセスを行う命令が用意されています。

一部の Nios II プロセッサ・コアは、アドレスの最上位ビットの値に応じてキャッシュをバイパスする、ビット 31 キャッシュ・バイパスと呼ばれるメカニズムをサポートしています。これらのプロセッサの実装アドレス空間は 2 G バイトで、アドレスの上位ビットはデータ・メモリ・アクセスのキャッシュ処理を制御します。



ビット 31 キャッシュ・バイパスをサポートするプロセッサ・コアについて詳しくは、「Nios II プロセッサ・リファレンス・ハンドブック」の「[Nios II コア実装の詳細](#)」の章を参照してください。

キャッシュ・メモリを搭載したプロセッサ・コア用に記述されたコードは、キャッシュ・メモリを搭載していないプロセッサ・コア上でも正常に動作します。ただし、その逆は成り立ちません。したがって、プログラムがすべての Nios II プロセッサ・コア実装で正常に動作するには、プログラムは命令キャッシュとデータ・キャッシュが存在するものとして動作する必要があります。キャッシュ・メモリを搭載しないシステムでは、キャッシュ管理命令は動作しないため影響はありません。キャッシュ管理の詳細な説明は、「Nios II ソフトウェア開発ハンドブック」を参照してください。

プロセッサ・リセット後のキャッシュ・コヒーレンシを保証するには、若干の考慮が必要です。詳しくは、[3-16 ページ](#)の「[プロセッサのリセット状態](#)」を参照してください。



キャッシュ・アーキテクチャおよびメモリ階層について詳しくは、「Nios II プロセッサ・リファレンス・ハンドブック」の「[プロセッサ・アーキテクチャ](#)」の章を参照してください。

プロセッサのリセット状態

リセット後に、Nios II プロセッサは以下を実行します。

1. status レジスタを 0x0 にクリアします。
2. リセット・アドレス、つまりリセット・ルーチンのアドレスに関連付けられた命令キャッシュ・ラインを無効化します。
3. リセット・アドレスから実行を開始します。

status (ct10) をクリアすると、プロセッサがスーパーバイザ・モードになり、ハードウェア割り込みをディセーブルします。リセット・キャッシュ・ラインを無効化すると、リセット・コードの命令フェッチは必ず非キャッシュ・メモリから行われます。リセット・アドレスは、システム生成時に指定されます。

リセット・アドレスに関連付けられた命令キャッシュ・ラインを除いて、キャッシュ・メモリの内容はリセット後も不定です。リセット後のキャッシュ・コヒーレンスを保証するには、リセット・ルーチンが命令キャッシュを即座に初期化する必要があります。次に、リセット・ルーチンまたは後続のルーチンのいずれかが、データ・キャッシュの初期化を実行することが必要です。

リセット状態は、以下を含む（ただし、これらに限定されない）その他のすべてのシステム・コンポーネントに対して不定です。

- 汎用レジスタ：ただし、永久にゼロである zero (r0) は除きます。
- コントローラ・レジスタ：ただし、0x0 にリセットされる status (ct10) は除きます。
- 命令およびデータ・メモリ
- キャッシュ・メモリ：ただし、リセット・アドレスに関連付けられた命令キャッシュ・ラインは除きます。
- ペリフェラル：リセット条件については、該当するペリフェラル・データ・シートまたは仕様を参照してください。
- カスタム命令ロジック：リセット条件については、カスタム命令の仕様を参照してください。

命令セットの カテゴリ

このセクションでは、実行される処理のタイプ別に、Nios II 命令について説明します。

データ転送命令

Nios II アーキテクチャは、ロード / ストア・アーキテクチャの一種です。レジスタ、メモリ、およびペリフェラル間のすべてのデータ移動は、ロード命令とストア命令によって処理されます。メモリとペリフェラルは共通アドレス空間を共有します。一部の Nios II プロセッサ・コアは、メモリ・キャッシングと書き込みバッファリングの両方または一方を使用して、メモリ帯域幅を向上させています。このアーキテクチャは、キャッシュ・アクセスと非キャッシュ・アクセスの両方に対する命令を提供しています。

表 3-4 で、ldw、stw、ldwio、および stwio 命令について説明します。

| 命令 | 説明 |
|----------------|--|
| ldw stw | ldw および stw 命令はそれぞれ、32 ビット・データ・ワードをメモリからロード / メモリに格納します。有効アドレスは、レジスタの内容と命令に含まれる符号付き即値との合計です。メモリ転送をキャッシュまたはバッファリングすると、プログラムの性能を向上させることができます。このキャッシングおよびバッファリングによってメモリ・サイクルの発生が乱れたり、キャッシングによって一部のサイクルが完全に抑制されることがあります。 I/O ペリフェラルのデータ転送では、ldwio および stwio を使用する必要があります。 |
| ldwio stwio | ldwio および stwio 命令はそれぞれ、キャッシングやバッファリングを行わずに、32 ビット・データ・ワードをペリフェラルからロード / ペリフェラルに格納します。ldwio および stwio 命令のアクセス・サイクルは、命令の順番どおり発生することが保証されており、抑制されることはありません。 |

表 3-5 のデータ転送命令は、バイト転送とハーフワード転送をサポートしています。

| 命令 | 説明 |
|---|--|
| ldb ldb stb ldh ldhu sth | ldb、ldb、ldh、および ldhu はバイトまたはハーフワードをメモリからレジスタにロードします。ldb および ldh は値を 32 ビットに符号拡張し、ldb および ldhu は値を 32 ビットにゼロ拡張します。 stb および sth はそれぞれ、バイト値およびハーフワード値を格納します。 メモリ・アクセスをキャッシュまたはバッファして、性能を向上させることができます。データを I/O ペリフェラルに転送するには、後述する命令の「io」バージョンを使用します。 |
| ldbio ldb stbio ldhio ldhuio sthio | これらの処理では、キャッシングまたはバッファリングを行わずに、バイトおよびハーフワード・データをペリフェラルからロード / ペリフェラルに格納します。 |

算術命令と論理命令

論理命令は and、or、xor、および nor 演算をサポートしています。算術命令は、加算、減算、乗算、および除算演算をサポートしています。表 3-6 を参照してください。

| 命令 | 説明 |
|----------------------------------|--|
| and or xor nor | これらは標準的な 32 ビット論理演算です。これらの演算は、2 つのレジスタ値を受け取り、ビット単位で結合して、第 3 のレジスタに対する結果を生成します。 |
| andi ori xori | これらの演算は、and、or、および xor 命令の即値バージョンです。16 ビットの即値が 32 ビットにゼロ拡張され、レジスタ値と結合されて結果を生成します。 |
| andhi orhi xorhi | and、or、および xor のこれらのバージョンでは、16 ビットの即値が論理的に 16 ビット左シフトされ、32 ビットオペランドを生成します。右からゼロがシフト・インされます。 |
| add sub mul div divu | これらは標準的な 32 ビット算術演算です。これらの演算では、2 つのレジスタを入力として受け取り、結果を第 3 のレジスタに格納します。 |
| addi subi muli | これらの命令は、add、sub、および mul 命令の即値バージョンです。命令ワードには符号付き 16 ビット値が含まれます。 |
| mulxss mulxuu | これらの命令は、32x32 乗算演算の上位 32 ビットへのアクセスを提供します。オペランドを符号付き値または符号なし値のどちらで扱うかに応じて、適切な命令を選択します。これらの命令の前に mul を付ける必要はありません。 |
| mulxsu | この命令は、64x64 符号付き乗算の 128 ビット結果を計算するのに使用されます。 |

移動命令

これらの命令は、レジスタの値または即値を別のレジスタにコピーする移動操作を実現します。表 3-7 を参照してください。

| 命令 | 説明 |
|--|--|
| mov movhi movi movui movia | mov はあるレジスタの値を別のレジスタにコピーします。movi は 16 ビットの符号付き即値をレジスタに移動し、値を 32 ビットに符号拡張します。movui および movhi はそれぞれ、レジスタの下位または上位 16 ビットに、16 ビットの即値を移動し、残りのビット位置にゼロを挿入します。レジスタにアドレスをロードするには、movia を使用します。 |

比較命令

Nios II アーキテクチャは、多数の比較命令をサポートしています。これらの比較命令はすべて、2 つのレジスタまたは 1 つのレジスタと即値を比較し、結果レジスタに 1 (真の場合) または 0 を書き込みます。これらの命令は C プログラミング言語の等号演算子および関係演算子をすべて実行します。表 3-8 を参照してください。

| 命令 | 説明 |
|--------|---------|
| cmpeq | == |
| cmpne | != |
| cmpge | 符号付き >= |
| cmpgeu | 符号なし >= |
| cmpgt | 符号付き > |
| cmpgtu | 符号なし > |
| cmple | 符号なし <= |
| cmpleu | 符号なし <= |
| cmplt | 符号付き < |

表 3-8. 比較命令 (2 / 2)

| 命令 | 説明 |
|--|---|
| cmpltu | 符号なし < |
| cmpeqi cmpnei cmpgei cmpgeui cmpgti cmpgtui cmplei cmpleui cmplti cmpltui | これらの命令は比較演算の即値バージョンです。レジスタの値と 16 ビットの即値を比較します。符号付き演算では、即値が 32 ビットに符号拡張されます。符号なし演算では上位ビットにゼロが入ります。 |

シフト命令と回転命令

シフトおよび回転操作は以下の命令で提供されます。回転またはシフトするビット数は、レジスタ内にまたは即値で指定できます。表 3-9 を参照してください。

表 3-9. シフト命令と回転命令

| 命令 | 説明 |
|---|--|
| rol ror roli | rol 命令と roli 命令は、左ビット回転を実行します。roli は即値を使用して、回転するビット数を指定します。ror 命令は右ビット回転を実行します。 roli を使用して同等の演算が実行できるため、ror の即値バージョンはありません。 |
| sll slli sra srl srai srli | これらのシフト命令は、C プログラミング言語の << 演算子および >> 演算子の動作を実現します。sll、slli、srl、srli 命令は、左および右論理ビット・シフト操作を実行し、ゼロを挿入します。sra および srai 命令は、算術右ビット・シフトを実行し、最上位ビットの符号ビットを複製します。slli、srli、および srai は、即値を使用して、シフトするビット数を指定します。 |

プログラム制御命令

Nios II アーキテクチャは、表 3-10 に示す無条件ジャンプと呼び出し命令をサポートしています。これらの命令には遅延スロットはありません。

| 命令 | 説明 |
|-------|---|
| call | この命令は、即値をサブルーチンの絶対アドレスとして使用してサブルーチンを呼び出し、レジスタ ra に戻りアドレスを格納します。 |
| callr | この命令は、レジスタ内の絶対アドレスに置かれているサブルーチンを呼び出して、戻りアドレスをレジスタ ra に格納します。この命令は C 関数ポインタを逆参照する役割を果たします。 |
| ret | ret 命令は、call または callr で呼び出されたサブルーチンから復帰するために使用されます。ret は、レジスタ ra のアドレスで指定される命令をロードして実行します。 |
| jmp | jmp 命令は、レジスタに格納された絶対アドレスへのジャンプを実行します。jmp は、C プログラミング言語の switch 文を実現するのに使用されます。 |
| br | 現在の命令に相対的な分岐。符号付き即値で、次に実行する命令のオフセットを設定します。 |

条件付き分岐命令は、レジスタ値を直接比較し、式が真の場合は分岐します。表 3-11 を参照してください。条件付き分岐は、C プログラミング言語の以下の等号比較および関係比較をサポートしています。

- == および !=
- < および <= (符号付きおよび符号なし)
- > および >= (符号付きおよび符号なし)

条件付き分岐命令には遅延スロットはありません。

| 命令 | 説明 |
|--|--|
| bge bgeu bgt bgtu ble bleu blt bltu beq bne | これらの命令は、2つのレジスタ値を比較する相対分岐を実現し、式が真の場合は分岐します。実装されている関係演算の説明は、3-20 ページの「比較命令」を参照してください。 |

その他の制御命令

表 3-12 に、その他の制御命令を示します。

| 命令 | 説明 |
|------------------------------------|---|
| trap eret | trap 命令と eret 命令は、例外を生成して、例外から戻ります。これらの命令は、call/ret のペアと類似していますが、例外に対して使用されます。trap は status レジスタの内容を estatus レジスタに、戻りアドレスを ea レジスタに保存してから例外ハンドラに実行を移します。eret は、estatus から status を復元し、ea のアドレスで指定された命令を実行して例外処理から復帰します。 |
| break bret | break 命令はブレイクを生成し、bret 命令はブレイクから復帰します。break と bret は、ソフトウェア・デバッグ・ツールによって排他的に使用されます。プログラマがアプリケーション・コードでこれらの命令を使用することはありません。 |
| rdctl wrctl | これらの命令は、status レジスタなどのコントロール・レジスタに読み出しと書き込みを行います。値は 1 つの汎用レジスタに読み書きされます。 |
| flushd flushi initd initi | これらの命令は、データ・キャッシュ・メモリと命令キャッシュ・メモリの管理に使用されます。 |
| flushp | この命令は、プリフェッチされたすべての命令をパイプラインからフラッシュします。この処理は、最近変更された命令メモリにジャンプする前に実行する必要があります。 |
| sync | この命令を実行すれば、すでに発行されたすべての操作が完了してから、後続のロードおよびストア操作が許可されるようになります。 |

カスタム命令

custom 命令は、カスタム命令ロジックへの低水準アクセスを提供します。カスタム命令の組み込みはシステム生成時に指定され、カスタム命令ロジックで実装される機能はデザインによって決まります。



詳しくは、「Nios II プロセッサ・リファレンス・ハンドブック」の「[プロセッサ・アーキテクチャ](#)」の章の「[カスタム命令](#)」セクションおよび「[Nios II Custom Instruction User Guide](#)」を参照してください。

マシン生成された C 関数とアセンブリ・マクロを利用すると、カスタム命令へのアクセスが可能になり、実装の詳細がユーザから隠蔽されます。したがって、ほとんどのソフトウェア開発者は、custom アセンブリ命令を直接使用することはありません。

無動作命令

Nios II アセンブラでは、無動作命令 `nop` が利用できます。

未実装の可能性がある命令

Nios II プロセッサ・コアには、ハードウェアですべての命令をサポートしていないものもあります。その場合、プロセッサは、未実装命令を発行した後で例外を生成します。未実装命令例外を生成できるのは、以下の命令のみです。

- `mul`
- `muli`
- `mulxss`
- `mulxsu`
- `mulxuu`
- `div`
- `divu`

その他のすべての命令は、未実装命令例外を生成しないことが保証されています。

例外ルーチンで上記の命令を使用する場合、それまでの例外が適切に処理される前に別の例外が発生する可能性があるため、注意が必要です。未実装命令の処理に関する詳細は、[3-11 ページの「未実装命令」](#)を参照してください。