

この資料は、更新された最新の英語版が存在します。こちらの日本語版は参考用としてご利用下さい。設計の際は、必ず最新の英語版で内容をご確認下さい。

NII52007-1.0

はじめに

Nios® II プロセッサ・コアは、命令キャッシュとデータ・キャッシュを搭載することができます。この章では、Nios II プロセッサ上でプログラムの正常な動作を保証するために検討する必要があるキャッシュ関連問題について説明します。幸いなことに、HAL システム・ライブラリをベースとする大部分のソフトウェアは、キャッシュに対する特別な対策がなくても正常に動作します。ただし、一部のソフトウェアではキャッシュを直接管理する必要があります。キャッシュを直接制御する必要があるコードに対して、Nios II アーキテクチャでは以下の動作を実行する機能を提供しています。

- 命令キャッシュおよびデータ・キャッシュのラインの初期化
- 命令キャッシュおよびデータ・キャッシュのラインの消去
- 命令のロード中およびストア中のデータ・キャッシュのバイパス

この章では、キャッシュの管理が必要な以下の一般的な事例について説明します。

- リセット後のキャッシュの初期化
- デバイス・ドライバの記述
- プログラム・ローダまたは自己書き換えコードの記述
- マルチマスタ・システムまたはマルチプロセッサ・システムでのキャッシュの管理

Nios II のキャッシュ実装

Nios II コアの実装状態によって、Nios II プロセッサ・システムがデータ・キャッシュまたは命令キャッシュを搭載する場合としない場合があります。キャッシュ・メモリの有無に関わらず、ユーザはどの Nios II プロセッサ上でも正常に機能する汎用性の高いプログラムを記述できます。一方または両方のキャッシュを搭載していない Nios II コアの場合、キャッシュ管理操作を実行しても影響はなく、効果もありません。

現在のすべての Nios II コアで、ハードウェア・キャッシュ・コヒーレンスのメカニズムは搭載されていません。したがって、共有メモリにアクセスするマスタが複数存在する場合、ソフトウェアによって、すべてのマスタ間での整合性を明示的に維持することが必要です。



各 Nios II コア実装の機能に関する詳細は、「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。

特定の Nios II プロセッサ・システムの詳細は、system.h ファイルに定義されています。以下のコードは、system.h ファイルの一部を抜粋したものです。ここでは、キャッシュ・サイズや単一のキャッシュ・ラインのサイズなど、キャッシュのプロパティを定義しています。

例：キャッシュ構造を定義する system.h の一部

```
#define NIOS2_ICACHE_SIZE 4096
#define NIOS2_DCACHE_SIZE 0
#define NIOS2_ICACHE_LINE_SIZE 32
#define NIOS2_DCACHE_LINE_SIZE 0
```

このシステムは、32 バイト・ラインを持つ 4K バイト命令キャッシュを搭載し、データ・キャッシュは搭載していません。

キャッシュ管理用 HAL API 関数

HAL API は、キャッシュ・メモリを管理する以下の関数を提供しています。

- alt_dcachelush()
- alt_dcachelush_all()
- alt_icachelush()
- alt_icachelush_all()
- alt_uncached_malloc()
- alt_uncached_free()
- alt_remap_uncached()
- alt_remap_cached()



API 関数の詳細については、[10-1 ページの「HAL API リファレンス」](#)を参照してください。

その他の情報

この章では、Nios II プログラマに関するキャッシュ管理問題についてのみ扱います。キャッシュの基本的な動作については説明していません。「The Cache Memory Book (Jim Handy 著)」は、一般的なキャッシュ管理について解説している優れた参考書籍です。

リセット後の キャッシュの 初期化

リセットすれば、命令キャッシュおよびデータ・キャッシュの内容は読み込めなくなります。正しく動作させるには、キャッシュはソフトウェア・リセット・ハンドラの起動時に初期化する必要があります。

Nios II キャッシュは、常にイネーブルされており、ソフトウェアでディセーブルすることはできません。適切な動作を実行するために、1 回のプロセッサ・リセットによって命令キャッシュは、リセット・ハンドラ・アドレスに対応する 1 つの命令キャッシュ・ラインを無効化します。これによって、命令キャッシュはこのキャッシュ・ラインに対応する命令を強制的にメモリからフェッチします。リセット・ハンドラ・アドレスは、命令キャッシュ・ラインのサイズと一致する必要があります。

命令キャッシュの残りの部分の初期化は、リセット・ハンドラの最初の 8 つの命令で行われます。Nios II の `init_i` 命令は、1 つの命令キャッシュ・ラインを初期化するのに使用されます。将来の Nios II 実装で、`flush_i` 命令を使用して命令キャッシュを初期化すると、悪影響を及ぼす可能性があるため、この命令は使用しないでください。

各命令キャッシュ・ライン・アドレスに対して `init_i` を実行するループ内に、`init_i` 命令を配置します。以下のコードは、命令キャッシュを初期化するアセンブリ・コードの例を示します。

例：命令キャッシュを初期化するアセンブリ・コード

```

mov     r4, r0
movhi  r5, %hi(NIOS2_ICACHE_SIZE)
ori    r5, r5, %lo(NIOS2_ICACHE_SIZE)
icache_init_loop:
    init_i  r4
    addi   r4, r4, NIOS2_ICACHE_LINE_SIZE
    bltu  r4, r5, icache_init_loop

```

命令キャッシュを初期化した後に、さらにデータ・キャッシュを初期化する必要があります。Nios II の `init_d` 命令は、1 つのデータ・キャッシュ・ラインを初期化する場合に使用します。この目的で、`flush_d` 命令を使用しないでください。この命令はダーティ・ラインをメモリに書き戻すためです。リセット後は、キャッシュ・ライン・タグも含めて、データ・キャッシュは未定義になります。`flush_d` を使用すると、不用意にランダム・データがランダム・アドレスに書き込まれることがあります。`init_d` 命令によって、ダーティ・データが書き戻されることはありません。

各データ・キャッシュ・ライン・アドレスに対して `init_d` を実行するループ内に、`init_d` 命令を置きます。以下のコードは、データ・キャッシュを初期化するアセンブリ・コードの例を示します。

例：データ・キャッシュを初期化するアセンブリ・コード

```

mov     r4, r0
movhi  r5, %hi(NIOS2_DCACHE_SIZE)
ori    r5, r5, %lo(NIOS2_DCACHE_SIZE)
dcache_init_loop:
initd  0(r4)
addi   r4, r4, NIOS2_DCACHE_LINE_SIZE
bltu  r4, r5, dcache_init_loop

```

一方または両方のキャッシュを実装していない Nios II コアに対しても、命令キャッシュおよびデータ・キャッシュの初期化コードは正しく実行できます。対応する種類のキャッシュが存在しない場合、`initd` 命令と `initd` 命令は、単に `nop` 命令として扱われます。

HAL システム・ライブラリ・ユーザの場合

HAL をベースとするプログラムでは、キャッシュ・メモリの初期化を管理する必要はありません。HAL C ランタイム・コード (`crt0.S`) には、`alt_main()` または `main()` が呼び出される前にキャッシュの初期化を実行するデフォルトのリセット・ハンドラが用意されています。

デバイス・ ドライバの 記述

通常、デバイス・ドライバは、そのデバイスに関連付けられたコントロール・レジスタにアクセスします。これらのレジスタは、Nios II アドレス空間にマップされます。デバイス・レジスタにアクセスするときに、データ・キャッシュのためにアクセスが失われたり、延期されないようするために、データ・キャッシュをバイパスする必要があります。

デバイス・ドライバの場合、データ・キャッシュは `ldio/stio` 命令ファミリを使用してバイパスしなければなりません。データ・キャッシュのない Nios II コアでは、これらの命令は対応する `ld/st` 命令と同様に動作するため悪影響はありません。

C プログラムの場合、ポインタを `volatile` として宣言し、この `volatile` ポインタを使用してアクセスしても、データ・キャッシュをバイパスできないことに注意してください。`volatile` キーワードは単に、コンパイラがポインタを使用したアクセスを最適化しないようにするためのものです。



この `volatile` 動作は、第 1 世代の Nios プロセッサの手法とは異なります。

HAL システム・ライブラリ・ユーザの場合

HAL は C 言語マクロの IORD および IOWR を提供しており、これらはデータ・キャッシュをバイパスするための適切なアセンブリ命令に展開されます。IORD マクロは `ldwio` 命令に展開され、IOWR マクロは `stwio` 命令に展開されます。これらのマクロは、デバイス・レジスタにアクセスするために、HAL デバイス・ドライバから使用する必要があります。

表 7-1 に、利用可能なマクロを示します。これらのマクロはすべて実行すると、データ・キャッシュをバイパスします。一般に、ユーザのプログラムは、`system.h` で定義された値を `BASE` パラメータおよび `REGNUM` パラメータとして渡します。これらのマクロは、ファイル `<Nios II インストール・パス>/components/altera_nios2/HAL/inc/io.h` で定義されています。

マクロ	用途
IORD(BASE, REGNUM)	ベース・アドレス BASE を持つデバイス内部のオフセット REGNUM のレジスタ値を読み込みます。レジスタはバスのアドレス幅だけオフセットされていると仮定されます。
IOWR(BASE, REGNUM, DATA)	ベース・アドレス BASE を持つデバイス内部のオフセット REGNUM のレジスタに、DATA の値を書き込みます。レジスタはバスのアドレス幅だけオフセットされていると仮定されます。
IORD_32DIRECT(BASE, OFFSET)	アドレス BASE+OFFSET の位置で 32 ビットの読み込みアクセスを実行します。
IORD_16DIRECT(BASE, OFFSET)	アドレス BASE+OFFSET の位置で 16 ビットの読み込みアクセスを実行します。
IORD_8DIRECT(BASE, OFFSET)	アドレス BASE+OFFSET の位置で 8 ビットの読み込みアクセスを実行します。
IOWR_32DIRECT(BASE, OFFSET, DATA)	32 ビットの書き込みアクセスを実行して、アドレス BASE+OFFSET の位置で DATA の値を書き込みます。
IOWR_16DIRECT(BASE, OFFSET, DATA)	16 ビットの書き込みアクセスを実行して、アドレス BASE+OFFSET の位置で DATA の値を書き込みます。
IOWR_8DIRECT(BASE, OFFSET, DATA)	8 ビットの書き込みアクセスを実行して、アドレス BASE+OFFSET の位置で DATA の値を書き込みます。

プログラム・ローダまたは自己書き換えコードの記述

プログラム・ローダや自己書き換えコードなど、メモリに命令を書き込むソフトウェアでは、命令キャッシュおよび CPU パイプラインから古い命令を確実に消去することが必要です。この消去動作は、それぞれ `flushi` 命令および `flushp` 命令によって実行されます。さらに、データ・キャッシュをバイパスしないストア命令を使用して新しい命令がメモリに書き込まれた場合は、`flushd` 命令を使用して、データ・キャッシュからメモリ内へ新しい命令を消去する必要があります。

以下のコードは、新しい命令をメモリに書き込むアセンブリ・コードを示します。

例：新しい命令をメモリに書き込むアセンブリ・コード

```
/*
 * 新しい命令が r4 に存在し、
 * 命令アドレスが既に r5 に存在すると想定します。
 */
stw      r4, 0(r5)
flushd   0(r5)
flushi   r5
flushp
```

`stw` 命令は、`r4` 内の新しい命令を `r5` で指定される命令アドレスに書き込みます。データ・キャッシュが存在する場合、命令はそのデータ・キャッシュに書き込まれ、関連付けられたラインはダーティとしてマークされます。`flushd` 命令は、`r5` のアドレスに関連付けられたデータ・キャッシュ・ラインをメモリに書き込み、対応するデータ・キャッシュ・ラインを無効化します。`flushi` 命令は、`r5` 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。最後に、`flushp` 命令は、CPU パイプラインが、`r5` で指定されるアドレスの古い命令をプリフェッチしていないことを確認します。

上記のコード・シーケンスでは、`stwio` 命令の代わりに、`stw` と `flushd` をペアで使用していることに注意してください。`stwio` 命令を使用してもデータ・キャッシュは消去されないため、データ・キャッシュ内に古いデータが残る可能性があります。

このコード・シーケンスは、すべての Nios II 実装に対して有効です。Nios II コアが特定の種類のキャッシュを搭載していない場合、対応する消去命令 (`flushd` または `flushi`) は `nop` として実行されます。

HAL システム・ライブラリ・ユーザの場合

HAL API は、このキャッシュ管理に対応する機能を提供していません。

マルチ・マスタ / マルチ CPU システムの管理

Nios II アーキテクチャは、ハードウェア・キャッシュ・コヒーレンシを提供しません。代わりに、共有メモリを介して通信するときは、ソフトウェア・キャッシュ・コヒーレンシを提供する必要があります。すべてのマスタが最新の値を読み込み、新しいデータに古いデータを上書きしないようにするために、共有メモリにアクセスするすべてのプロセッサのデータ・キャッシュの内容は、ソフトウェアで管理しなければなりません。このような管理は、データ・キャッシュの消去およびバイパス機能を使用し、必要に応じて共有メモリとデータ・キャッシュ間でデータを移動することによって実行します。

`flushd` 命令を使用すると、データ・キャッシュとメモリには 1 つのラインに対して必ず同じ値が格納されます。ラインにダーティ・データが含まれる場合、そのデータはメモリに書き込まれます。次にそのラインはデータ・キャッシュ内で無効化されます。

最も重要なことは、常にデータ・キャッシュをバイパスすることです。プロセッサは、データ・キャッシュをバイパスするときに、アドレスがデータ・キャッシュ内に存在するかどうかチェックしません。ソフトウェアは、特定のアドレスがデータ・キャッシュ内に存在することを保証できない場合、データ・キャッシュをバイパスしてロードまたはストアを実行する前に、データ・キャッシュからアドレスを消去する必要があります。この処理によって、プロセッサはキャッシュ内の新しい (ダーティ) データをバイパスすることはなく、また誤ってメモリ内の古いデータにアクセスすることもなくなります。

ビット 31 キャッシュ・バイパス

`ldio/stio` 命令ファミリは、明示的にデータ・キャッシュをバイパスします。ビット 31 はデータ・キャッシュをバイパスするもう 1 つの手段です。ビット 31 キャッシュ・バイパスを使用すると、アドレスの最上位ビット (ビット 31) が 1 に設定されている場合に、標準的な `ld/st` 命令ファミリを使用してデータ・キャッシュをバイパスできます。ビット 31 の値は CPU 内部でのみ使用されます。つまり、アクセスされる実際のアドレスでは、ビット 31 は強制的にゼロになります。これにより、最大バイト・アドレス空間は 31 ビットに制限されます。

関連するアドレスのキャッシュ機能がアドレス内に含まれているため、ビット 31 を使用してデータ・キャッシュをバイパスするメカニズムはソフトウェアに好都合です。この使用方法により、標準的な `ld/st` 命令ファミリを使用するコードにアドレスを渡すことができ、同時にそのアドレスへのすべてのアクセスが常にデータ・キャッシュをバイパスすることを保証できます。

ビット 31 キャッシュ・バイパスは Nios II/f コアでのみ明示的に実現できます。その他の Nios II コアには使用しないでください。ビット 31 キャッシュ・バイパスをサポートしていないその他の Nios II コアは、他の実装へコードを容易に移植するために、最大バイト・アドレス空間を 31 ビットに制限しています。これらのコアは、実質的にビット 31 の値を無視するため、ビット 31 キャッシュ・バイパスを使用する Nios II/f コア用に記述されたコードは、その他の現行 Nios II 実装上でも正常に動作できます。一般に、この機能は Nios II コア実装状態に依存します。将来の Nios II コアでは、ビット 31 を他の目的に使用する可能性があります。



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 Nios II コア実装の詳細の章を参照してください。

HAL システム・ライブラリ・ユーザの場合

HAL は、`ldio` 命令ファミリに展開される C 言語の `IORD_*DIRECT` マクロと、`stio` 命令ファミリに展開される `IOWR_*DIRECT` マクロを提供します(表 7-1 を参照)。これらのマクロはキャッシュ不能なメモリ領域にアクセスするためのものです。

HAL は、キャッシュされないメモリ領域の割り当てと操作を実行する `alt_uncached_malloc()` ルーチン、`alt_uncached_free()` ルーチン、`alt_remap_uncached()` ルーチン、および `alt_remap_cached()` ルーチンを提供します。これらのルーチンは、データ・キャッシュ搭載または非搭載の Nios II コアで利用できます。つまり、データ・キャッシュを搭載した Nios II コア用に記述したコードは、データ・キャッシュ非搭載 Nios II コアと完全に互換性があります。

`alt_uncached_malloc()` ルーチンおよび `alt_remap_uncached()` ルーチンは、割り当てられたメモリ領域がデータ・キャッシュ内に存在しないこと、および以降の割り当て済みメモリ領域へのすべてのアクセスがデータ・キャッシュをバイパスすることを保証します。