Nios II システムのプロファイリング

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。

AN-391-3.0

このアプリケーション・ノートでは、GNU プロファイラ（**nios2-elf-gprof**）パフォーマンス・カウンタ・コンポーネント、およびタイム・スタンプ・インターバル・タイマ・コンポーネントとの Nios® II システムの性能を測定する方法について説明します。このアプリケーション・ノートでは、Altera® Nios II Software Build Tools （SBT）の開発フローにおけるパフォーマンスを測定するには、2 つのチュートリアルが含まれています。

必要条件

チュートリアルを使用するには、Quartus® II ソフトウェアと Qsys を含む Nios II システム用の Nios II SBT の開発フローを理解している必要があります。

ハードウェア・デザインの取得

このアプリケーション・ノートのチュートリアルは Nios II Ethernet Standard Design Example で動作します。

デザイン例を使用するには、システム上の作業ディレクトリに開発キットの **.zip** ファイルを解凍します。

☞ このアプリケーション・ノートでは、*<project_directory>* としてのソフトウェア例ではディレクトリと表記します。

ソフトウェア例の取得

このアプリケーション・ノートのソフトウェア例を入手するには、以下のステップに従います。

1. **profiler_software_examples.zip** をダウンロードします。
2. **profiler_software_examples.zip** を システム内の *<project_directory>* に解凍します。

☞ このアプリケーション・ノートでは、*<profiler_software_examples>* としてディレクトリを指します。

ツール

Nios II システムにハードウェアを変更せずに GNU プロファイラを使用することができます。このツールは、アプリケーション・コードにプロファイラのライブラリ関数の呼び出しを追加するようにコンパイラに指示します。

パフォーマンス・カウンタ・コンポーネントおよびタイム・スタンプ・コンポーネントは、Nios II システムのパフォーマンスを測定する最小限の侵入のハードウェア方法です。このアプリケーション・ノートでは、2 つのコンポーネントを説明し、比較します。これらの方法を使用するには、システムにハードウェア・コンポーネントを追加し、コンポーネントを起動および停止するためにソース・コードにマクロ呼び出しを追加します。ハードウェア・コンポーネントは測定を実行します。

コンパイラ速度最適化は、広く様々な程度に機能に影響します。コンパイラ・サイズの最適化はまた、さまざまな方法で機能に影響します。これらの違いは、キャッシュの使用やリソース競合に影響し、相対的な開始時刻を変更します。このため、機能の実行時間を増加されます。これらの理由では、最終的なフォームでアプリケーションを改善する方法について最も洞察力を得るために、-O3 コンパイラー・スイッチでコードを最適化し、コードのプロファイリングを実行する必要があります。

チュートリアルでは、以下の項で説明するように Nios II システムの性能を測定する 3 つのツールを使用します。

- GNU プロファイラ
- アルテラのパフォーマンス・カウンタ
- 高分解能のタイマ

さらに、プログラム・カウンタ・トレース収集ツールは、いくつかの Nios II プロセッサで使用できます。ただし、チュートリアルでは、このツールを使用しないでください。

最も CPU 時間を消費するコードの領域を識別するために GNU プロファイラを使用することができます。また、機能のボトルネックを分析するためにパフォーマンス・カウンタまたはタイマ・コンポーネントを使用することができます。

GNU プロファイラ

GNU プロファイラによる分析のために測定するようにソース・コードに最小限の変更が必要です。必要な変更を実装するには、以下のステップに従います。

1. Nios II SBT では、`hal.enable_gprof` と `hal.enable_exit` ボード・サポート・パッケージ（BSP）の設定をオンにして、プロジェクトで GNU プロファイラをイネーブルします。

   ☞ Eclipse 用の Nios II SBT を使用する場合、ソフトウェアはデフォルトで `hal.enable_exit` をイネーブルします。

2. `main()` ファンクションが返すことを確認します。

   ☞ 必要に応じて、`main()` が呼び出されるか、または `return()` が終了される場合、プロファイリングのために `exit()` を呼び出します。`exit()` ファンクションがは `BREAK 2` 命令を実行します。それによって、プロファイリング・データはホスト・コンピューター上の **gmon.out** を書き込ませます。

3. BSP とアプリケーション・プロジェクトを再作成します。

アルテラのパフォーマンス・カウンタ

パフォーマンス・カウンタは、選択したコード・セクションの実行時間を測定するハードウェアのカウンタ・ブロックです。パフォーマンス・カウンタ・コンポーネントは、7 コード・セクションまで追跡することができます。デフォルトで、コンポーネントは 3 つのコード・セクションを追跡します。カウンタのペアは各コード・セクションを追跡します。

- *Time*— コード・セクションの間にクロック・ティックの数をカウントし、64 ビット・タイム（クロック・ティック）カウンタが実行されます。
- *Occurrences*— コード・セクションの実行回数を測定する 32 ビットのイベント・カウンタです。

☞ Qsys 内のパフォーマンス・カウンタ・コンポーネントを編集することによって測定されたコード・セクションの最大数を変更することができます。

これらのカウンタは、C/C++ コードの指定のセクションの実行時間を測定することができます。マクロは、スタートとプログラム内のコード・セクションの終わりをマークすることができます。パフォーマンス・カウンタ・コンポーネントは、最大 7 ペアまでのカウンタを持っており、7 で測定されたセクションの C/C++ コードをサポートします。各測定セクションの開始時と終了時にコードにマクロを追加する必要があります。追加に、カウンタのビルド・インペアは個々コードセクションカウンタを集約し、これより各セクションはより大きなプログラムの小数として測定することが出来ます。

決定論と他のランタイムの問題を分析するためのパフォーマンス・カウンタを使用することができます。

☞ パフォーマンス・カウンタ・コンポーネントは、デバイス上のロジック・エレメント相当数の（LE）を占有し、パフォーマンス測定値を得るために、ソフトウェアの実装を必要とします。

高分解能のタイマ

高分解能のタイマでは、パフォーマンス・カウンタ・コンポーネントと対照的に、デバイス上の LE が多数に使用されません。そして、パフォーマンス測定を得るためにコード内のすべての関数呼び出しの重い実装を行う必要はありません。タイマは、測定するソース・コードのセクションで明示的なリードの呼び出しが必要とするので、その使用はプログラムでパフォーマンスの問題を特定するに適します。手動でソース・コードを実装する必要があります。しかし、この実装はあまり普及していないため、この実装は控えめになります。パフォーマンス・カウンタのマクロとは異なり、タイマは 2 つのファンクションの呼び出しを実行するために、より多くのプロセッサ・サイクルを必要とします：一つは測定したセクションの最初に時間を読み込み、もう一つは最後に時間を読み込みます。

プログラム・カウンタ・トレースの情報

Nios II プロセッサは、完全かつ正確なプログラム・カウンタのトレース情報を生成することができます。しかし、GNU プロファイラはこの情報を使用していません。この情報を生成するには、レベル 3 以上の JTAG デバッグ・モジュールで構成される Nios II プロセッサが必要です。レベル 3 の JTAG デバッグ・モジュールはオンチップ・トレース・データを作成します。オンチップ・トレース・バッファの約ダースほどの命令をキャプチャすることができます。

オフチップのトレース情報を生成するレベル 4 の JTAG デバッグ・モジュールを使用して Nios II コアを構成することによって、より大きなトレースを取得することができます。ただし、このオフチップ・トレース・データを収集するために、First Silicon Solutions, Inc. (FS2) または Lauterbach Datentechnik GmbH (Lauterbach) (www.lauterbach.com) ハードウェアを必要としています。

Lauterbach のハードウェアについて詳しくは、「*Embedded Design Handbook*」の「*Debugging Nios II Designs*」の章の「Debuggers」を参照してください。

コードの性能を測定するための GNU プロファイラ

以下の項では、パフォーマンス分析のための GNU プロファイラを使用することの利点と限界を説明します。このチュートリアルでは、パフォーマンス・データを収集と分析するための GNU プロファイラの使用方法を示します。

GNU プロファイラの利点

GNU プロファイラでパフォーマンスを測定する主な利点は、GNU プロファイラがシステム全体の概要を提供することです。GNU プロファイラはいくつかのオーバーヘッドが追加されますが、GNU プロファイラは、均等にシステム全体でこのオーバーヘッドを分配します。プロファイラを実装せずにフルスピードでアプリケーションを実行するときに、GNU プロファイラはほとんどのプロセッサ時間を消費してもほとんどのプロセッサ時間を消費する機能を識別します。

GNU プロファイラの制約

GNU プロファイラで使用するために、各関数呼び出しに命令を追加すると、次の方法でコードの動作に影響します。

- 各関数は、プロファイリング情報を収集するには追加の関数呼び出しのためにわずかに大きい。
- プロファイリング情報収集のために、各関数の入口と出口の時間。
- 命令キャッシュは、プロファイリング機能が命令キャッシュ・メモリであるため、ミスが高い高い。
- プロファイリング・データを記録するメモリは、データ・キャッシュの動作を変更することができる。

これらの効果は、プロファイリングによって明らかにしようとしている時間に微妙な問題を隠すことができます。

GNU プロファイラは、プログラム・カウンタの定期的なサンプリングに基づいて、インターポレーションによって各関数に費やされた時間のパーセンテージを決定します。GNU プロファイラは、システム・クロックのタイマ・ティックに定期的にサンプルを結び付けます。GNU プロファイラでは、割り込みをイネーブルした場合にだけのサンプルを取ることができます。したがって、割り込みルーチンで費やされたプロセッサ・サイクルを記録することはできません。

GNU プロファイラは個々の関数をプロファイルできません。システム全体をプロファイルするためにを使用することができます（使用しなくてもかまいません）。

プロファイリング・データは、システム・タイマ・ティックの解像度でプログラム・カウンタのサンプリングです。したがって、プロファイリング・データは、別の関数で費やされたプロセッサ時間の正確な表現ではなく、推定を提供します。システム・タイマ・ティックの周波数を増やすことによって、サンプリングの統計的有意性を向上させることができます。しかし、ティックの周波数を増加すると、サンプルを記録する時間を費やし増加します（それは、測定の完全性に影響する）。

☞ カスタム・ハードウェア・デザインで正常に GNU プロファイラを使用するには、デザインにシステム・クロック・タイマが含まれていることを確認する必要があります。GNU プロファイラは、このコンポーネントが適切な出力を生成する必要があります。

ソフトウェアの考慮事項

GNU プロファイラは、プロセッサの使用を追跡するための関数を使用してソース・コードを実装します。

プロファイラ力学

BSP を生成するスクリプトで `hal.enable_gprof` のスイッチをオンにして GNU プロファイラをイネーブルしてください。このスイッチをオンにすると、自動的に `-pg` のコンパイラ・スイッチをオンにして、BSP 付きの `altera_nios2` ソフトウェア・コンポーネントのプロファイリング・ライブラリのコードをリンクします。このコードは、それぞれのプロファイル関数への呼び出しの数をカウントします。

`-pg` のコンパイラー・オプションは、コンパイラがすべての関数呼び出しの開始時に `mcount()` 関数（ファイル **altera_nios2/HAL/src/alt_mcount.S** にある）への呼び出しを挿入するように強制します。`mcount()` への呼び出しは、コールグラフの構築をイネーブルするためにすべてのダイナミックな親と子の関数呼び出しの関係を追跡します。また、オプションは `nios2_pcsample()` 関数（**altera_nios2/HAL/src/**

alt_gmon.c ファイル内に位置する）をシステム・クロック割り込みでフォアグラウンド・プログラム・カウンタのサンプルとします。プログラムを実行すると、GNU プロファイラは、**gmon.out** というのホスト上のデータを収集します。**nios2-ELF-gprof** ユーティリティは、このファイルが読み込み可能であり、プログラムに関するプロファイル情報を表示できます。

プロファイリング・コードは、以下のステップを実行して、ターゲット上で動作します。

1. コンパイラは、関数コール・グラフを決定するためにコンパイラをイネーブルする mcount() に呼び出しで関数のプロローグを実装します。GNU プロファイラのマニュアルでは、このデータを関数呼び出しのアーク・データと表記します。
2. タイマ割り込みハンドラは、アラームがトリガされたときに実行するフォアグラウンド機能（ヒストグラム・データ）に関する情報をキャプチャするためにアラームをレジスタします。
3. ヒープは、プロファイリング・データを格納するためにターゲット・メモリを割り当てます。
4. BREAK 2 命令でコードが終了すると、**nios2-download** のユーティリティはターゲットからホストにプロファイル・データをコピーします。

nios2-elf-gprof のユーティリティは、正しく動作する関数呼び出しのアーク・データとヒストグラムデータを必要とします。

GNU プロファイラについて詳しくは、アルテラ・ウェブサイトの Nios II Embedded Design Suite Support ページでの Nios II GNU プロファイラのドキュメント（GCC ドキュメント付き）を参照してください。

プロファイラのオーバーヘッド

GNU プロファイラを使用すると、メモリとプロセッサ・サイクルが影響されます。

メモリ

.text セクションのサイズのプロファイリング情報の影響は、アプリケーション内の小さな関数の数に比例します。コード・オーバーヘッドの .text セクションのサイズは、nios2_pcsample() と mcount() 関数の追加のため、GNU プロファイラがプロファイリングをイネーブルしたときに増加します。GNU プロファイラは nios2_pcsample() への呼び出し付きのシステム・タイマを実装し、mcount() を呼び出してすべての機能を実装します。.text セクションには、追加の関数呼び出しによって、これら二つの関数の大きさによって増加します。

.text セクションへの影響を表示するには、**.objdump** の .text セクションのサイズを比較することができます。

GNU プロファイラは、プロファイリング中にヒープ上にデータを格納するバケットを使用します。各バケットのサイズは 2 バイトです。各バケットは .text セクションのコードの 32 バイトのサンプルを保持しています。ヒープから割り当てられたプロファイラ・バケットの合計数は 32 で .text セクションのサイズを分割するときに得られます。GNU プロファイラ・バケットが消費するヒープ・メモリは次のとおりです。

((.text セクションのサイズ) / 32) × 2 バイト

GNU プロファイラは GNU プロファイラがプロファイリング情報でコンパイルされるオブジェクト・コードのすべての機能を測定します。この機能セットはランタイム・ライブラリおよび BSP を含むライブラリ関数が含まれています。

プロセッサ・サイクル

GNU プロファイラは mcount() への呼び出し付きの個々の機能を追跡します。アプリケーション・コードは、多くの小さな関数が含まれる場合、プロセッサ時間の GNU プロファイラの影響は大きいです。しかし、プロファイリングされたデータの解像度が高くなっています。mcount() 付きのプロファイリングによって消費される追加のプロセッサ時間を計算するために、プロセッサが mcount() を実行する必要な時間を掛けて、アプリケーションで実行するランタイム関数呼び出しの数によって mcount() を実行します。

すべてのクロック・チックに、プロセッサは nios2_pcsample() 関数を呼び出します。nios2_pcsample() を使用してプロファイリングを実行する必要な追加のプロセッサ時間を計算するには、プロセッサがアプリケーションに必要なクロック・チック数でこの関数を実行するために必要な時間を掛けて、これは mcount() の呼び出しと実行に必要な時間も含まれています。nios2_pcsample() でプロファイリングを実行する必要な追加のプロセッサ時間を計算するには、プロセッサがアプリケーションに必要なクロック・チック数でこの関数を実行するために必要な時間を掛けます（mcount() の呼び出しと実行に要する時間を含む)。

プロファイリングに使用される追加のプロセッサ・サイクルの数を計算するには、nios2_pcsample() にすべての呼び出しを計算されたオーバーヘッドに mcount() にすべての呼び出しの計算されたオーバーヘッドを追加します。

ハードウェアの考慮事項

GNU プロファイラは、システム・タイマを必要とします。Nios II ハードウェア・デザインはシステム・タイマが含まれる場合、ユーザーのデザインを変更する必要はありません。

チュートリアル：GNU プロファイラを使用

デモンストレーション用に、このチュートリアルでは Nios II Embedded Evaluation Kit キットおよび Cyclone III Edition (NEEK) 開発キットの Nios II Ethernet Standard デザイン例を使用します。開発キットを対象とし、他の類似したデザイン例を使用することができます。

デザイン例を使用してデバイスを設定するには、以下のステップに従います。

1. Quartus II ソフトウェア・バージョン 11.0 以降を起動します。
2. ファイル・メニューの **Open Project** をクリックします。
3. **niosii_ethernet_standard_3c25.qpf** を開きます。
4. ツール・メニューの **Programmer** をクリックします。
5. デバイスに SRAM オブジェクト・ファイル（**.sof**）をダウンロードするには **Start** をクリックします。

☞ ソフトウェアが **Start** ボタンをディセーブルする場合、または **Hardware Setup** フィールドが USB-Blaster ケーブルがリストされていない場合、Programmer のツールについて詳しくは *「Introduction to the Quartus II Software」* のマニュアルを参照してください。

Nios II コマンド・ライン付きのプロファイラのサンプル

プロファイラのソフトウェアのサンプルを作成

Nios II コマンド・ライン・フローの profiler_gnu ソフトウェア・プロジェクトを作成するには、以下のステップに従います。

1. お使いの環境に応じて、以下のいずれかのステップを実行することにより、Nios II コマンド・シェルを開きます。
   - Windows オペレーティングシステムの Start メニューで、**Programs** > **Altera** > **Nios II EDS** *<version>* をポイントして、**Nios II** *<version>* **Command Shell** をクリックします。
   - Linux オペレーティング・システムのコマンドシェルでは、*<Nios II EDS install path>* にディレクトリを変更し、コマンドの `./sdk_shell` を入力します。
2. ディレクトリ *<profiler_software_examples>***/app/profiler_gnu** に変更します。
3. 以下のコマンドを入力して、**create-this-app** スクリプトを使用してアプリケーションを作成とビルドします。

```
./create-this-app ↵
```

create-this-app スクリプトは **create-this-bsp** を実行し、*<profiler_software_examples>***/bsp/hal_profiler_gnu** での **parameter_definition.tcl** から設定を読み込みます。この Tcl ファイルは、以下のラインが含まれています。

```
set_setting hal.enable_gprof true
set_setting hal.enable_exit true
```

最初の設定は GNU プロファイラをイネーブルし、第二の設定は exit()、そして main() を呼び出す alt_main() 関数をイネーブルします。

プロファイラ・ソフトウェアのサンプルを実行

アプリケーションを実行し、GNU プロファイラ・データを収集し、以下のステップに従います。

1. 第二 Nios II コマンド・シェルを開きます。
2. 第二のシェルで、次のコマンドを入力して **nios2-terminal** セッションを開きます。

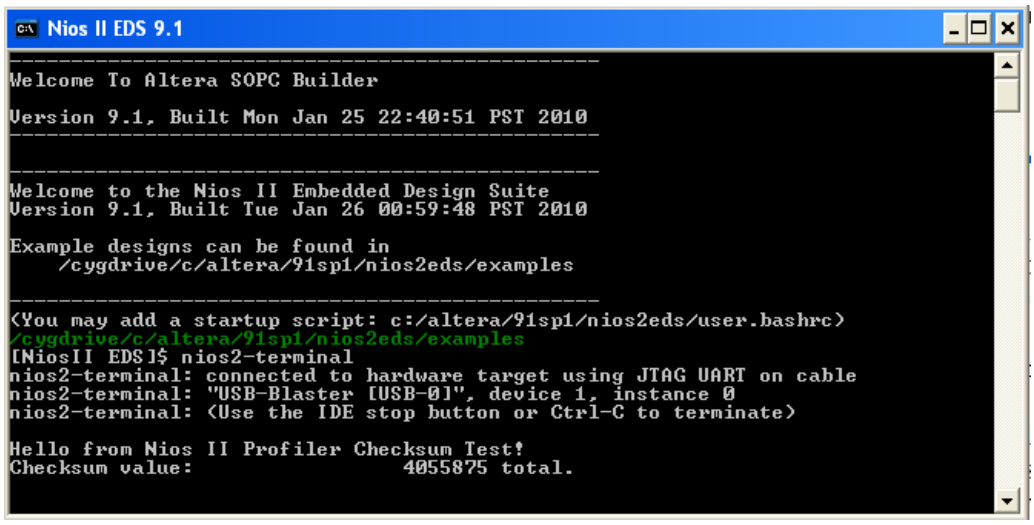
   ```
   nios2-terminal ↵
   ```

3. オリジナルの Nios II コマンド・シェルでは、次のコマンドを入力して、開発ボードに **.elf** をダウンロードし、デザインを実行します。そして、**gmon.out** に GNU プロファイラ・データを書き込みます。

   ```
   nios2-download -g --write-gmon gmon.out *.elf ↵
   ```

   GNU プロファイラは、アプリケーションの実行中にデータを収集し、アプリケーションが exit() 関数を呼び出すときに **gmon.out** というデータを書き込みます。図 1 に、Nios II コマンド・シェル内の GNU プロファイラの出力例を示します。
4. control-C を入力することによって、**nios2-terminal** を終了します。

図 1. Nios II コマンド・シェル上での GNU プロファイラ出力



GNU プロファイラ・レポートの作成

プロジェクトを実行すると、プロジェクトは、**gmon.out** を作成します。読み込み可能なフォーマットにこのファイルをフォーマットする必要があります。このファイルをフォーマットするには、以下のステップに従います。

1. オリジナルの Nios II コマンド・シェルで、*<profiler_software_examples>***/app/profiler_gnu** にディレクトリを変更します。
2. 次のコマンドを入力します。

```
nios2-elf-gprof profiler_gnu.elf gmon.out > report.txt ↵
```

このコマンドは、平坦なプロファイル・レポートおよびコール・グラフを生成します。それは、**report.txt** を表示できます。

3. **report.txt** を表示するには、任意のテキスト・エディタを使用しています。

GNU プロファイラ・レポートについて詳しくは、1-11 ページの「GNU プロファイラ・レポートの分析」を参照してください。

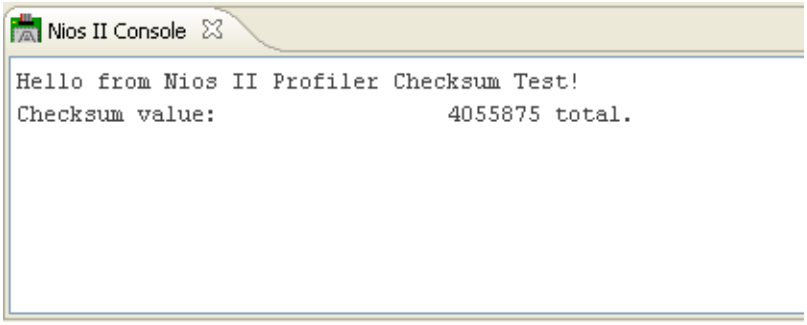
Eclipse 用の Nios II SBT とプロファイラの例

プロファイラ・ソフトウェアのサンプルの作成と実行

1. Eclipse 用の Nios II SBT を開始します。
2. File メニューの **New** をポイントし、**Nios II Application and BSP from template** をクリックします。
3. *<project_directory>* の SOPC Information File (**.sopcinfo**) を見つけるのに移動して、**SOPC Information File name** を設定します。
4. プロジェクトに名前を付けます。例えば、profiler_gnu。
5. **Templates** の **Blank Project** を選択します。
6. **Finish** をクリックして、新しいプロジェクトを作成します。
7. *<profiler_software_examples>***/eclipse_source_files/profiler_gnu** フォルダを位置して、このディレクトリ内のすべてのファイルをコピーします。Eclipse 用の Nios II SBT では、Project Explorer ビューの **profiler_gnu** を右クロックして、**Paste** を選択します。
8. 右クロックの Project Explorer ビューで、プロジェクトを右クリックし、Nios II にポイントし、**BSP Editor** をクリックします。
9. Nios II BSP Editor で、プロジェクトで GNU プロファイラをイネーブルする `hal.enable_gprof` をオンにします。
10. BSP プロジェクトを生成して終了します。
11. Project Explorer ビューでプロジェクトを右クロックして、**Build Project** をクリックします。
12. profiler_gnu ソフトウェアをダウンロードして実行するには、プロジェクトを右クロックて、**Run As** をポイントし、**Nios II Hardware** をクリックします。

図 2 に Nios II コンソールにソフトウェアの出力を示します。

図 2. profiler_gnu を実行した Nios II Console



GNU プロファイラ・レポートのビュー

ソフトウェアはプロジェクト・フォルダ内に **gmon.out** を作成します（Eclipse 用の Nios II SBT の Project Explorer ビューで表示可能）。**gmon.out** が表示されない場合は、右側のプロジェクトをクリックし、**Refresh** を選択します。**gmon.out** を開いたときに、Eclipse 用の Nios II SBT は、Profiling ビューに切り替えるレポートを表示することができます。GNU プロファイラ・レポートについて詳しくは、「GNU プロファイラ・レポートの分析」を参照してください。

GNU プロファイラ・レポートの分析

この項の情報は、コマンド・ラインまたは Eclipse 用の Nios II SBT が生成した GNU プロファイラ・レポートに適用されます。

GNU プロファイラ・レポートには、次のフォーマットの情報が含まれています。

- レポートの **flat profile** の領域は、処理時間を消費した順序で子関数を識別します。
- レポートの **call graph** の領域は、各関数とその子に費やされる時間の合計量によってソートされるプログラムのコール・ツリーを示します。このテーブルの各エントリは複数のラインで構成されます。左側のマージンにインデックス番号付きのラインは、現在の機能を示します。それは上記のラインは、この関数から呼び出される関数をリストし、その下のラインは、レポート自体と GNU プロファイラのドキュメントでさらに詳細な例外や条件との呼び出された関数を表示します。

詳細については、Nios II Embedded Design Suite Support ページでの GCC に関する資料付きの Nios II GNU プロファイラのマニュアルを参照してください。

例 1 に、前のチュートリアルから GNU プロファイラのレポートの抜粋を示します。例 1 では、フラット・プロファイルが `checksum_test_routine()` 関数が実行時に処理時間の 79.1979.19% を消費して呼び出すことを示します。

コール・グラフ・レポートの精度の記述は、レポートが 2.55 秒（2550 ミリ秒）を負担することを述べます。Nios II タイマ（sys_clk_timer）は 10 ミリ秒のタイマがあります。GNU プロファイラは完全なクロック周期が経過する前に、冒頭で一度、タイマ割り込みを呼び出し、その後に一度、10 ミリ秒ごとに呼び出します。したがって、正確なレポートは、GNU プロファイラがタイマ割り込みハンドラに 255 回呼び出すことを示します。インデックス 13 は GNU プロファイラが alt_avalon_timer_sc_irq() の（この測定方法の精度の範囲内である）256 回呼び出すことを示します。

☞ 得られた結果は例 1 と異なる場合がありますので注意してください。

例 1. フラット・プロファイルおよびコール・グラフの例

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 79.19      2.02     2.02        1     2.02     2.03  checksum_test_routine
 18.01      2.48     0.46        1     0.46     0.46  alt_busy_sleep

        .
        .
        .

            Call graph (explanation follows)


granularity: each sample hit covers 32 byte(s) for 0.39% of 2.55 seconds

index      % time    self    children    called     name
                0.00    0.00           273/273         alt_irq_entry 106
13    0.0    0.00    0.00             273         alt_irq_handler 13
                0.00    0.00           256/256         alt_avalon_timer_sc_irq 14
                0.00    0.00            17/17          altera_avalon_jtag_uart_irq 17

        .
        .
        .
```

パフォーマンス・カウンタおよびタイマ・コンポーネントの使用

GNU プロファイラは、ほとんどのプロセッサ時間を消費するコードの領域を識別した後、パフォーマンス・カウンタやタイマコンポーネントはさらにこれらの機能のボトルネックを分析することができます。

以下の項では、パフォーマンス分析のためのパフォーマンス・カウンタとタイマを使用することの利点と限界を説明します。このチュートリアルでは、パフォーマンスデータを収集し、分析するためのパフォーマンス・カウンタおよびタイマの使用方法を示します。

パフォーマンス・カウンタの利点

パフォーマンス・カウンタは少し侵入の測定を提供する Nios II 開発キットで使用可能な唯一のメカニズムです。各測定項の測定を開始および停止するための効率的なマクロを使用することができます。パフォーマンス・カウンタは、大きさの順番で、GNU プロファイラより早いです。測定データを収集する控えめな唯一つの方法は、特定のバス・アドレス上のトリガを使用してコンフィギュレーションされたロジック・アナライザなどの完全にハードウェア・ベースのソリューションとなります。

タイマの利点

パフォーマンス・カウンタとは異なり、同時にコードの 7 つのセクションのみ追跡することができ、タイマは制限がありません。タイマを 1,000 回読み込み可能であり、セクションの開始時間として 1,000 種類の変数にタイマを格納することができます。その後、1,000 終了タイマの測定値にタイマを比較することができます。唯一の実用的な制限要因は、メモリの消費量、プロセッサのオーバーヘッド、および複雑さです。

パフォーマンス・カウンタおよびタイマのハードウェア面での考慮事項

パフォーマンス・カウンタでパフォーマンスを測定する 1 つの欠点は、カウンタの大きさです。パフォーマンス・カウンタ・コンポーネントは、デバイス上の LE の多数を消費します。

3C120 デバイス上で、変更された標準のハードウェア・デザインで定義された 3 つのセクション・カウンタとの 1 つのパフォーマンス・カウンタ・コンポーネントが 671 ロジック・セル（LCS）および 420 LC レジスタを消費します。同じデザインでは、7 つのセクション・カウンタと定義される単一のパフォーマンス・カウンタは、1339 のロジック・セルおよび 808 LC レジスタを消費します。パフォーマンス・カウンタ・コンポーネントのリソース使用率は、すべてのデバイス上でほぼ同じです。

☞ リソースを節約するには、システムの最終バージョンからパフォーマンス・カウンタを削除します。

タイマは、パフォーマンス・カウンタよりも大幅に少ないですが、ハードウェア・リソースを消費します。タイマは、ハードウェア・リソースに大幅にパフォーマンス・カウンタ未満でだが、タイマーはを導入します。

☞ パフォーマンス・カウンタとタイマを追加すると f_{MAX} を低減することができます。

パフォーマンス・カウンタおよびタイマのソフトウェア面での考慮事項

パフォーマンス・カウンタおよびタイマの一般的な欠点は、状況認識の欠如であります。タイマ割り込みがコードのセクションの測定中に発生した場合、パフォーマンス・カウンタとタイマは、全測定時間にタイマ割り込みを処理するためのプロセッサにかかる時間を追加します。この効果は、マルチ・スレッド・システムでより多く発生しましたが、この効果は、単純な割り込みおよびマルチ・スレッド・コンテキストのスイッチングのために発生します。コードのセクションを測定しながら、多くのスレッドまたは割り込みサービス・ルーチンが実行される可能性があります。その結果、非常に大規模な、スキューされる測定が発生します。測定歪みの結果が予測不可能であり、測定しようとするコード・セクションの動作と相関性がありません。

コンテキスト・スイッチの影響を回避するには、ほとんどのマルチ・スレッドのオペレーティング・システムで一時的にスケジューラをロックするシステム・コールが必要です。また、セクションの測定中にコンテキスト・スイッチを回避するために割り込みをディセーブルすることができます。

☞ 割り込みをディセーブルすることまたはスケジューラをロックすると、システムの動作に影響しますので、最後の手段としてこれらのテクニックを使用する必要があります。

パフォーマンス・カウンタのソフトウェア面での考慮事項

各測定セクションの開始時刻と終了時刻を記録する PERF_BEGIN と PERF_END パフォーマンス・カウンタのマクロを使用する必要があります。

PERF_BEGIN と PERF_END は、パフォーマンス・カウンタ・コンポーネントへの書き込みはシングルです。これらのマクロは、2 つまたは 3 つのマシン命令が必要であり、非常に効率的です。

例 2 には、**altera_avalon_performance_counter.h** での PERF_BEGIN と PERF_END パフォーマンス・カウンタのマクロを示します。

例 2. altera_avalon_performance_counter.h での PERF_BEGIN および PERF_END のパフォーマンス・カウンタのマクロ

```
#define PERF_BEGIN(p,n) IOWR((p),(((n)*4)+1),0)

#define PERF_END(p,n) IOWR((p),(((n)*4) ),0)
```

グローバル・カウンタ

パフォーマンス・カウンタ・コンポーネントは、いくつかのカウンタが含まれています。Qsys 内の測定セクションの数を設定することができます。1-3 ページの「アルテラのパフォーマンス・カウンタ」で説明したように、各測定セクションのカウンタのいずれかのペアがあります。さらに、パフォーマンス・カウンタ・コンポーネントは、常にグローバルなカウンタがあります。

グローバル・カウンタは、測定の合計時間を測定します。グローバル・カウンタを停止すると、他のカウンタは実行されません。PERF_START_MEASURING と PERF_STOP_MEASURING マクロは、グローバル・カウンタを制御します。

他の方法でグローバル・カウンタを操作しようとしないでください。

パフォーマンス・カウンタについて詳しくは、*「Embedded Peripherals IP User Guide」*の *「Performance Counter Core」*の章を参照してください。

ハードウェアの考慮事項

パフォーマンス・カウンタとタイム・スタンプのインターバル・タイマは Qsys のコンポーネントです。既存のシステムに追加するときは、Qsys システムの再生成および Quartus II ソフトウェアの **.sof** のリコンパイルが必要です。タイマおよびパフォーマンス・カウンタは、ハードウェア・カウンタと同じく最終的にオーバーフローすることができます。

チュートリアル：パフォーマンス・カウンタとタイマの使用

このチュートリアルでは、ほとんどのプロセッサ時間を使用してコードのセクションを識別することによって、GNU プロファイラを使用することよりも正確な Nios II システムのパフォーマンスを測定するパフォーマンス・カウンタとタイム・スタンプのインターバル・タイマの使用方法を示します。

このチュートリアルでは、GNU プロファイラのチュートリアルと同じ NEEK デザインを使用します。このデザインでは、インターバル・タイマおよびパフォーマンス・カウンタがあります。タイマ・インターバルおよびパフォーマンス・カウンタが測定するセクションの数を変更することができます。

Nios II ハードウェア・デザインの変更

このチュートリアルでは、Nios II Ethernet Standard のデザイン例を変更する必要があります。Nios II Ethernet Standard のデザイン例を変更するには、以下のステップに従います。

1. Quartus II ソフトウェアで、Tools メニューの **Qsys** をクリックします。
2. *<project_directory>* で、**peripheral_system.Qsys** をクリックします。
3. **high_res_timer** モジュールを右クリックし、**Edit** をクリックします。
4. **Timeout period** で、インターバル期間を 1 に設定します。そして、単位を **us**（マイクロ秒）に設定します。
5. **Finish** をクリックします。
6. File メニューの **Save** をクリックします。
7. Nios II Ethernet Standar のデザイン例は階層ベースのデザインです。システムを生成するには、File メニューの **Open** をクリックし、**eth_std_main_system.Qsys** を選択します。
8. **Generation** タブをクリックします。
9. **Create HDL design files for synthesis and Create block symbol file (.bsf)** のオプションをオンにします。
10. Output Directory のパスが *<project_directory>***/eth_std_main_system** であることを確認します。
11. **Generate** をクリックします。ソフトウェアはプロンプトが表示された場合、システムを保存します。
12. 生成が完了したときに Qsys を終了します。
13. **.sof** を生成するには、Quartus II ソフトウェアでの Processing メニューの **Start Compilation** をクリックします。
14. 次のメッセージが表示されたら **OK** をクリックします。

```
Full Compilation was successful
```

デバイスへのハードウェア・デザインのプログラミング

変更したハードウェア・デザインをコンパイルした後、デバイスにハードウェア・デザインをプログラムすることができます。これを実行するには、以下のステップに従います。

1. Tools メニューの **Programmer** をクリックします。
2. デバイスに **.sof** をダウンロードするには **Start** をクリックします。

ソフトウェアが **Start** ボタンをディセーブルにすること、または **Hardware Setup** フィールドが USB-Blaster ケーブルがリストされていない場合、Programmer のツールのについて詳しくは、Quartus II ソフトウェアの「*Introduction to the Quartus II Software*」のマニュアルを参照してください。

Nios II コマンド・ラインを使用したパフォーマンス・カウンタの例

この項では、Nios II コマンド・ラインによるパフォーマンス・カウンタのソフトウェア例を作成して実行する方法について説明します。

パフォーマンス・カウンタのソフトウェアの作成例

Nios II ソフトウェア・ビルド・フローの **profiler_performance_counter** のソフトウェア・プロジェクトを作成するには、以下のステップに従います。

1. 1-8 ページの「プロファイラのソフトウェアのサンプルを作成」で説明した Nios II コマンド・シェルを開きます。
2. *<profiler_software_examples>*/**app/profiler_performance_counter** ディレクトリに変更します。
3. 次のコマンドを入力して、アプリケーションを作成とビルドします。

   ```
   ./create-this-app ↵
   ```

   create-this-app スクリプトは *<profiler_software_examples>*/**bsp/hal_profiler_performance_counter** で **parameter_definition.tcl** から設定を読み込み、**create-this-bsp** スクリプトを実行します。この Tcl ファイルは、以下の行が含まれています。

   ```
   set_setting hal.sys_clk_timer peripheral_subsystem_sys_clk_timer
   set_setting hal.timestamp_timer peripheral_subsystem_high_res_timer
   set_setting hal.enable_gprof true
   set_setting hal.enable_exit true
   ```

   最初の二行は Qsys システム内の対応するタイマにシステム・クロック・タイマとタイム・スタンプ・タイマを設定します。

   三行目は GNU プロファイラをイネーブルし、最後の行は、`alt_main()` 関数が `exit()`、そして `main()` の呼び出しをイネーブルします。

パフォーマンス・カウンタのソフトウェアの例の実行

アプリケーションを実行し、GNU プロファイラ・データを収集し、以下のステップに従います。

1. 第二の Nios II コマンド・シェルを開きます。
2. 第二のシェルで、以下のコマンドを入力して、**nios2-terminal** セッションを開きます。

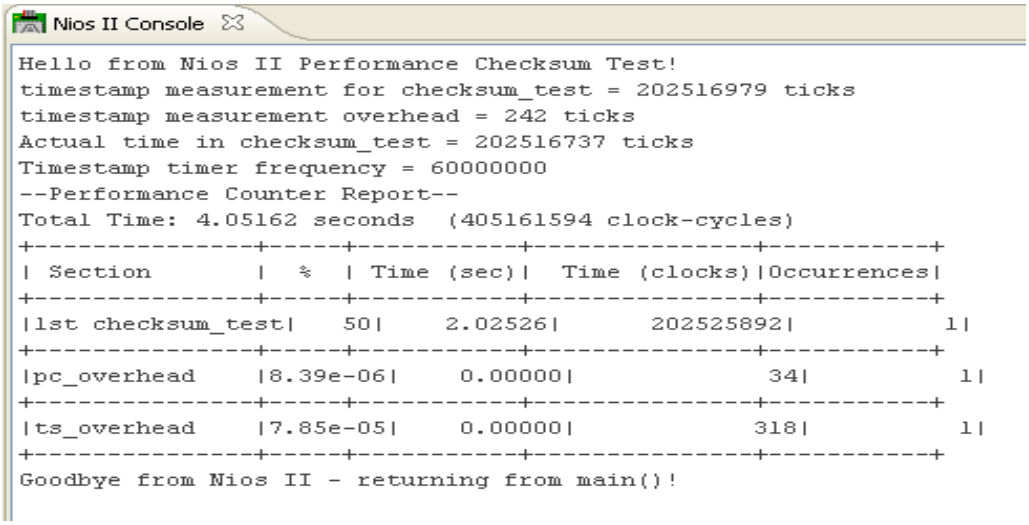
   ```
   nios2-terminal ↵
   ```

3. 元の Nios II コマンド・シェルで、次のコマンドを入力してプログラムを実行します。

   ```
   nios2-download -g *.elf ↵
   ```

図 3 に、Nios II コマンド・シェルに表示される出力の例を示します。出力は異なる場合があります。詳細については、「パフォーマンス・カウンタのレポートの分析」を参照してください。

図 3. Nios II コマンド・シェル上のパフォーマンス・カウンタのレポート



Eclipse 用の Nios II SBT とパフォーマンス・カウンタの例

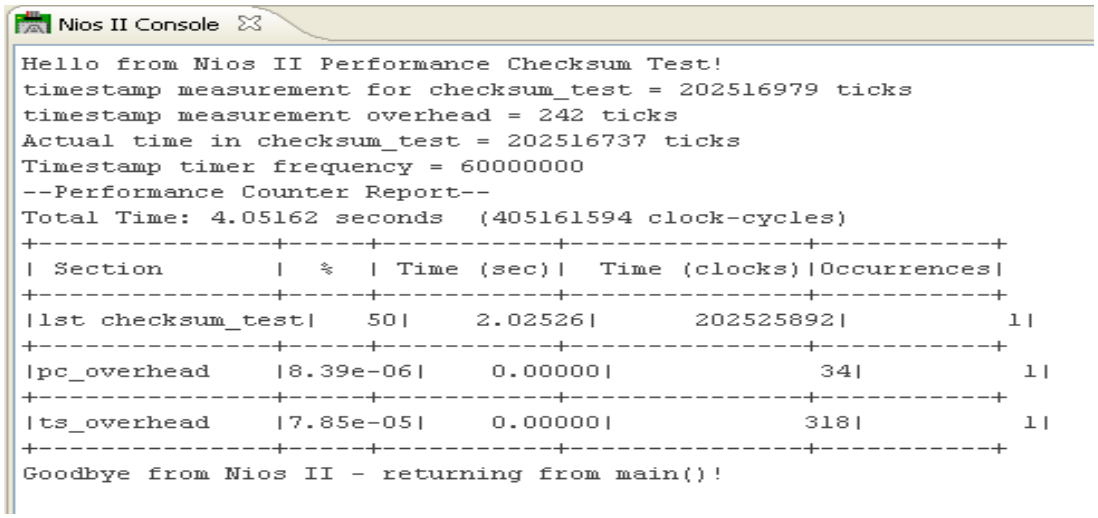
この項では、Eclipse 用の Nios II SBT と **profiler_performance_counter** ソフトウェアの例を作成して実行する方法について説明します。

1. Eclipse 用の Nios II SBT を開始します。
2. File メニューの **New** をポイントし、**Nios II Application and BSP from template** をクリックします。
3. *<project_directory>* ディレクトリを参照し、.**sopcinfo** を選択することにより、**SOPC Information File name** を設定します。
4. プロジェクトに名前を付けます（例：**profiler_performance_counter**）。
5. **Templates** の **Blank Project** を選択します。
6. 新しいプロジェクトを作成するには **Finish** をクリックします。
7. *<profiler_software_examples>***/eclipse_source_files/profiler_performance_counter** フォルダを見つけて、このディレクトリ内のすべてのファイルをコピーします。Eclipse 用の Nios II SBT では、プロジェクト・エクスプローラー・ビューで profiler_gnu を右クリックし、**Paste** を選択します。
8. Project Explorer ビューで、プロジェクトを右クリックして、**Nios II** にポイントして **BSP Editor** をクリックします。
9. Nios II BSP Editor で、プロジェクトの GNU プロファイラをイネーブルするには `hal.enable_gprof` をオンにします。
10. **peripheral_subsystem_sys_clk_timer** コンポーネントに `hal.sys_clk_timer` を設定します。
11. **peripheral_subsystem_high_res_timer** コンポーネントに `hal.timestamp_timer` を設定します。
12. BSP プロジェクトを生成して終了ます。
13. Project Explorer ビューを右クリックして、**Build Project** にポイントします。

14. **profiler_performance_counter** ソフトウェアを実行するには、アプリケーション・プロジェクトを右クリックして、**Run As** にポイントします。そして、**Nios II Hardware** をクリックします。

図 4 に、**profiler_performance_counter** を実行した後、Nios II Console 出力を示します。データは図 3 のコマンド・ラインの例に似ています。詳細については、「パフォーマンス・カウンタのレポートの分析」を参照してください。

図 4. Nios II Console でのパフォーマンス・カウンタのレポート



パフォーマンス・カウンタのレポートの分析

この項の情報は、コマンド・ラインまたは Eclipse 用の Nios II SBT が生成するパフォーマンス・カウンタのレポートに適用されます。

pc_overhead は PERF_BEGIN マクロへの単一の呼び出しに起因するパフォーマンス・カウンタ・コンポーネントのオーバーヘッドです。この数値は、PERF_BEGIN マクロの実行およびこの測定セクションの対応する PERF_END マクロドが含まれています。

ts_overhead は、タイム・スタンプ・オーバーヘッドであり、タイマを読み込むための単一の関数呼び出しのオーバーヘッドです。この数値は、測定を実施するためのパフォーマンス・カウンタのオーバーヘッドが含まれています。

結論

Nios II 開発環境は、プロジェクトのパフォーマンスを分析するためにいくつかのツールを提供します。ソフトウェアのみの GNU プロファイラのアプローチは、最小限のオーバーヘッドが追加されます。確定的なリアルタイムのパフォーマンスの問題を分析するには、ハードウェア・タイマまたはパフォーマンス・カウンタを使用することができます。タスクに最適なツールを選択するには、解決している問題を検討してください。

トラブルシューティング

以下の項では、発生し、問題をトラブルシューティングする方法を提案するいくつかの問題について説明します。

効果なしの nios2-elf-gprof –annotated-source スイッチ

プロファイラは、basic-block-count を追跡しないので、スイッチ（例えば、–annotated-source スイッチ）は動作しません。

存在しないセクション・カウンタのレジスタへの書き込み

例 3 でのパフォーマンス・カウンタのレポートでは、パフォーマンス・カウンタ・コンポーネントの存在しないセクション・カウンタを使用した後の発生することを示します。

例 3. 存在しないセクション・カウンタを使用した結果

```
--Performance Counter Report--
Total Time: 5.78751 seconds (289375582 clock-cycles)
+--------------------+--------+-------------+---------------+-----------+
|      Section       |   %    | Time (sec)  | Time (clocks) |Occurrences|
+--------------------+--------+-------------+---------------+-----------+
|sleep_tests         |    49.4|      2.86162|      143081026|          1|
+--------------------+--------+-------------+---------------+-----------+
|perf_begin_overhead | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|timestamp_overhead  | 7.6e-06|      0.00000|             22|          1|
+--------------------+--------+-------------+---------------+-----------+
|non_existent_counter|6.37e+12|368934881474.19104|        -1| 4294967295|
+--------------------+--------+-------------+---------------+-----------+
```

3 つのセクション・カウンタのみ（デフォルト値）を持つように、第四目のセクション・カウンタは Qsys で定義したパフォーマンス・カウンタ・コンポーネントを指定して仮定します。

例 3 では、テストは即座にパフォーマンス・カウンタ・コンポーネントのレジスタの後にマップされたレジスタで定義される他のコンポーネントのないハードウェア・デザイン上で実行されます。したがって、他のコンポーネントへの影響はありません。特定のハードウェア・デザインのために Qsys 内のコンポーネント・レジスタのベース・アドレスを設定する方法に応じて、予期しないシステム動作が発生する可能性があります。

main() の終わりに printf（）または perf_print_formatted_output() の呼び出しからの出力が途中で切り捨てられる可能性

Nios II プロセッサ・アプリケーションは main() から exit() または return() の間に、開発ワークステーションにプロファイリング・データを転送するための BREAK 命令を実行するときに、この問題が発生します。

対応策としては終了または返送する前に、usleep(500000) を呼び出します。この呼び出しでは、main を返送する前に（または exit() を呼び出す前に）JTAG UART への I/O を送信するために適切な遅延を作成します。出力がまだ部分的に切り捨てられた場合、usleep() に渡された遅延値を増加します。使用するには、# usleep() 関数のプロトタイプとしては #include <unistd.h> を使用します。

デバイスのリソースのほとんどを消費するハードウェア・デザインでのパフォーマンス・カウンタのフィッティング

開発時には、デプロイされたシステムでデバイスのサイズよりも大きなデバイスでシステムを測定することができます。

ほとんどのリソースを節約するために唯一つのセクション・カウンタだけのパフォーマンス・カウンタを設定します。

main() 関数を終了しても gmon.out ファイルのヒストグラムが表示さが含まれています。れない

システム用のシステム・タイマを定義していない場合、プロファイラは `nios2_pcsample()` 関数を呼び出すことはありません。また、**gmon.out** のためのヒストグラムを生成しません。使用するシステム用のシステム・タイマを定義してください。

参考文献

GNU プロファイラについて詳しくは、Nios II Embedded Design Suite Support で利用可能な GCC のドキュメントに含まれている Nios II GNU プロファイラのドキュメントを参照してください。

- アルテラは、`lib-gprof` のライブラリを書き換えるため、データ収集については、このアプリケーション・ノートの情報はアルテラの実装から逸脱されます。

パフォーマンス・カウンタについて詳しくは、*「Embedded Peripherals IP User Guide」*の *「Performance Counter Core」*の章を参照してください。高速タイマについて詳しくは、*「Embedded Peripherals IP User Guide」*の *「Timer Core」*の章を参照してください。

改訂履歴

表 1 に、このアプリケーション・ノートの改訂履歴を示します。

表 1. 改訂履歴

日付	バージョン	変更内容
2010 年 7 月	3.0	この改訂版には以下の変更が含まれています。 ■ Qsys で SOPC Builder を代替 ■ Updated 1-1 ページの「ハードウェア・デザインの取得」, 1-1 ページの「ソフトウェア例の取得」, 1-4 ページの「プログラム・カウンタ・トレースの情報」, 1-8 ページの「チュートリアル：GNU プロファイラを使用」, 1-8 ページの「プロファイラのソフトウェアのサンプルを作成」, 1-9 ページの「GNU プロファイラ・レポートの作成」, 1-10 ページの「プロファイラ・ソフトウェアのサンプルの作成と実行」1-11 ページの「GNU プロファイラ・レポートの分析」1-12 ページの「フラット・プロファイルおよびコール・グラフの例」, 1-16 ページの「Nios II ハードウェア・デザインの変更」, 1-17 ページの「パフォーマンス・カウンタのソフトウェアの作成例」, 1-17 ページの「パフォーマンス・カウンタのソフトウェアの例の実行」, および 1-18 ページの「Eclipse 用の Nios II SBT とパフォーマンス・カウンタの例」を更新。
2010 年 5 月	2.0	この改訂版には以下の変更が含まれています。 ■ Eclipse 用の Nios II SBT のためのドキュメント、ソフトウェア、およびスクリーン・ショットを更新。 ■ Eclipse フローの Nios II SBT を追加。 ■ NEEK 用の例を更新。
2008 年 7 月	1.3	この改訂版には以下の変更が含まれています。 ■ Quartus II ソフトウェアおよび Nios II EDS v8.0 用のドキュメントを更新。 ■ 命令付きの Nios II IDE で Nios II ソフトウェアのビルド・フローのリファランスを代替。 ■ Quartus II ソフトウェア v8.0 の更新。
2006 年 2 月	1.2	Quartus II ソフトウェアおよび Nios II EDS v5.1 SP1 の更新。
2005 年 11 月	1.1	Quartus II ソフトウェアおよび Nios II EDS v5.1 の更新。
2005 年 8 月	1.0	初版。