

はじめに

Nios® II C-to-Hardware Acceleration (C2H) コンパイラはソフトウェア・ファンクションからハードウェア・アクセラレータを生成する強力なツールです。このアプリケーション・ノートでは、C2H コンパイラを使用して、scatter-gather DMA (ダイレクト・メモリ・アクセス) ファンクションのCコードを、等価なハードウェア・アクセラレータに変換する方法を示します。このハードウェア・アクセラレータはインターネットデータの検査に使用可能なチェックサム値を出力します。このアプリケーション・ノートに含まれる参照ファイルは scatter-gather DMA およびチェックサム関数を実現するCコードを提供します。チェックサム・コードは2バージョンがあります：1つはNios II プロセッサ上でソフトウェアとして動作します。もう1つはハードウェア・アクセラレータになり、CコードをFPGA内の同等なロジックに置き換えます。ハードウェア・バージョンのスピードアップは通常2桁以上です。正確なスピードアップはターゲットFPGAによって異なります。

このデザイン例では scatter-gather DMA コードをチェックサム・コードと結合して使用していますが、ネットワークング、オーディオ、ビデオ、または高速I/Oなど多様なアプリケーションのフロントエンドDMAアクセラレータとして応用可能です。このアプリケーション・ノートの最後のセクションではC2Hコンパイラによってシステム性能をさらに改善する方法について記載しています。

前提条件

このアプリケーション・ノートは次の項目に詳しいエンジニアを対象に書かれています。

- ANSI C の構文および使い方
- SPOC Builder による Nios II ハードウェア・システムの定義および生成
- Altera® Quartus® II 開発ソフトウェアによる Nios II ハードウェア・システムのコンパイル
- Nios II ソフトウェア・プロジェクトの作成、コンパイル、および実行
- Nios II C2H コンパイラ



C2H コンパイラの基本について詳しくは、「[Nios II C2H Compiler user Guide](#)」を参照してください。Nios II システムの定義、生成、およびコンパイルについては「[Nios II Hardware Development Tutorial](#)」を参照してください。Nios II ソフトウェア・プロジェクトについては、Nios II IDE ヘルプ・システムにおける「[Nios II Software Development Tutorial](#)」を参照してください。

ハードウェアおよびソフトウェア要件

このアプリケーション・ノート用のデザイン・ファイルを使用するためには、次のソフトウェアおよびハードウェアが必要です。

- Windows または Linux コンピュータにインストールされた、Quartus II 開発ソフトウェア v6.0 以降
- Nios II エンベデッド・デザイン・スイート (EDS) v6.0 以降
- Nios 開発ボード Cyclone II Edition など Altera が提供した開発ボード
- Altera USB-Blaster™ ケーブルなどターゲット・ハードウェアに互換性のある JTAG ダウンロード・ケーブル

背景：Scatter-Gather DMA

エンベデッド・システムの多くは DMA エンジンを用いてデータ・スループットを増加させ、プロセッサへのデータ転送負荷を軽減させる構成をとります。通常、アクセスされるメモリアドレスはアドレス空間全体に分散されています。

Scatter-gather DMA エンジンは、プロセッサが各メモリ領域毎の DMA 転送命令を発行することなしに、分散したメモリ領域に対する複数の DMA 転送を実行することができます。scatter-gather DMA は、転送エンジンが行うべき全ての転送内容を記述したディスクリプタ・テーブルと呼ばれるバッファ・アドレスとバッファサイズのパアを含むテーブルを作成します。この DMA エンジンは、ソフトウェアの介入が不要なため、転送領域を切り換える際のオーバーヘッドを非常に小さくできます。

ソフトウェアのみによる実装

例 1 は RFC 1071 での Braden、Borman、Patridge による実装に基づいたチェックサム・アルゴリズムのソフトウェアコードです。これはプロセッサで実行する場合の典型的な例です。これは C2H コンパイラに最適化されたものではなく、ハードウェアの知識を活用してより高い性能を得られるようコーディングされたものではありません。

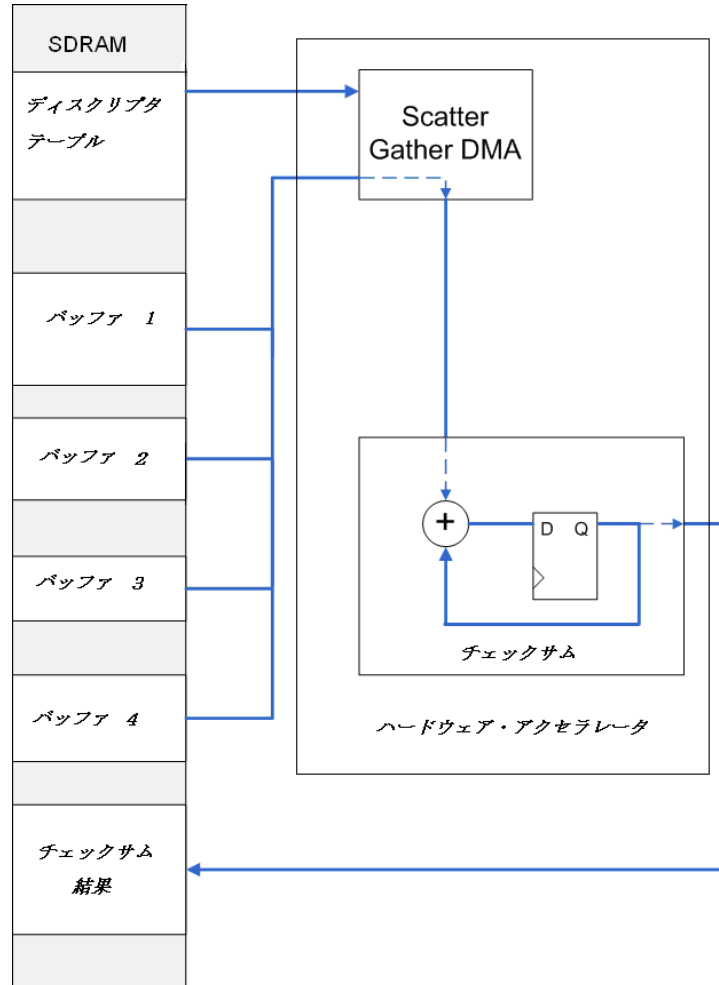
例 1. インターネット・チェックサムのソフトウェア実装

```
/* *****  
 * Portable C implementation of the Internet checksum, derived *  
 * from Braden, Borman, and Partridge's example implementation *  
 * in RFC 1071. *  
 * *  
 * Inputs:  unsigned short *:  base address of the buffer to be summed *  
 *          int:                length of the buffer to be summed *  
 * Outputs: unsigned short:    calculated 16 bit checksum *  
 * ***** */  
unsigned short sw_checksum(unsigned short * addr, int count)  
{  
    /* Compute Internet Checksum for "count" bytes  
     * beginning at location "addr".  
     */  
    register long sum = 0;  
  
    while( count > 1 ) {  
        /* This is the inner loop */  
        sum += *addr++;  
        count -= 2;  
    }  
  
    /* Add left-over byte, if any */  
    if( count > 0 )  
        sum += * (unsigned char *)addr;  
  
    /* Fold 32-bit sum to 16 bits */  
    while (sum>>16)  
        sum = (sum & 0xffff) + (sum >> 16);  
  
    return (~sum);  
}
```

高速化された実装

この項では、C2H コンパイラによって最良の結果が得られるよう、例 1 に示されたチェックサム実装を変更する方法について説明します。例 2 に示された C2H コンパイラの scatter-gather DMA ファンクションの実装は直接的です。これはソフトウェア開発者が DMA エンジンがない際に複数のバッファ位置をアクセスする手法に似ています。scatter-gather DMA のデザインにはイーサネット・パケットを検証するためのチェックサム・ファンクションが統合されています。プロセッサに大量なメモリ・アクセスを実行させるチェックサム検証は時間がかかります。チェックサムの演算にとって主要なボトルネックはメモリ・スループットのため、チェックサム・アルゴリズムと DMA 動作を 1 つのハードウェア・アクセラレータで実現することで、システム性能の最大化が可能となります。図 1 はハードウェア・アクセラレータおよびそのメモリへの接続に関するブロック図です。

図 1. 簡素化された Scatter-Gather DMA



例 2 は C2H コンパイラによるハードウェア化に最適化された C コードを示します。ポインタ `addr` はメイン・メモリにおける各バッファ内のデータのアクセスに使用されます。数値 `count` は高速化されたファンクションでの DMA ループ数のコントロールに使用されます。

例 2. チェックサムのハードウェア実装を統合した Scatter-gather DMA

```
void hw_checksum(unsigned long * table_address, unsigned long table_length,
                unsigned short *return_values)
{
    void * addr;
    unsigned long buffer_ctr;
    /*****
     *          Scatter-gather DMA loop          *
     *****/
    * Use "addr" to perform read operations on the memory and "count" *
    * to keep track of the length of the buffer.  These values are *
    * loaded from the table using the pointer "table_address." The *
    * number of entries in the table is "table_length." *
    *****/
    for(buffer_ctr = 0; buffer_ctr < table_length; buffer_ctr++)
    {
        /* read in the next base address and length of the buffer */
        addr = (void *)(*table_address++); /* buffer address */
        count = *table_address++; /* buffer length */

        /*****
         * Code that the C2H Compiler Maps to Hardware You can replace *
         * the checksum with your own application re-using the scatter *
         * gather DMA. *
         *****/
        sum = 0;
        /*****
         * Using the scatter-gather DMA access the memory 32 bits at a *
         * time.  Split the word into half words and add these to the *
         * accumulator "sum".  Count is expressed in bytes so it must *
         * decrease by 4 bytes per word access.  The pointer "addr" must *
         * advance by 4 bytes per word access *
         *****/
        while (count > 3) {
            temp_data = *(unsigned long *)addr;
            sum += (temp_data & 0xFFFF) + ((temp_data & 0xFFFF0000) >> 16);
            count -= 4;
            addr += 4;
        }
        /* Add left-over half word when applicable.  This is a half *
         * word access so the pointer "addr" must advance by 2 *
        sum += ((count == 2) || (count == 3)) ? *(unsigned short*)addr : 0;
        addr += ((count == 2) || (count == 3)) ? 2 : 0;

        /* Add left-over byte when applicable.  This is the last *
         * possible access so no need to move the pointer "addr" *
        sum += ((count == 1) || (count == 3)) ? *(unsigned char *)addr : 0;

        /* Fold 32-bit sum to 16 bits.  The first fold could result in *
         * a 17 bit sum so a second fold guarantees that the result *
         * fits within 16 bits */

        /* 1st fold */
        sum = (sum & 0xffff) + (sum >> 16);
        /* 2nd fold */
        return_values[buffer_ctr] = ~((sum & 0xffff) + (sum >> 16));

        /*****
         *          End of Code that the C2H Compiler Maps to Hardware          *
         *****/
    }
}
```

C2Hによって作成された scatter-gather DMA アクセラレータ・ファンクションは、任意のリード・ライト・アクセスの組み合わせを実行できます。提供されたリファレンス・デザインでは、各バッファ内にデータを転送するCコードがメイン・メモリ（SDRAM）に存在しているデータ・バッファを読み出すアクセラレータ・ファンクションに置き換えられます。C2H コンパイラは、addr アドレス内容の読み出し動作をSDRAMメモリに接続された Avalon マスター・ポートに置き換えます。DMA ディスクリプタ・テーブルにおける各エントリのベース・アドレスおよびバッファサイズを設定するコードはソフトウェアとして残ります。チェックサム演算を実行している残りのCコードはまとめて1つのアクセラレータ・ファンクションに変換されます。

表3はディスクリプタ・テーブルを示します。各行は個々のバッファのアドレスとバッファサイズの値を表します。Nios II プロセッサのアドレス・スペースにおける任意のバッファ位置を処理するために、バッファ・アドレスとバッファ・サイズの値は32ビット幅となります。アクセラレータが動作中は、ディスクリプタ・テーブルまたはバッファ内容が変更されないよう注意する必要があります。使用中のディスクリプタ・テーブルを変更すると、同期化およびデータ完全性の問題が起ることがあります。

表3. ディスクリプタ・テーブル

バッファ・アドレス (32 ビット)	バッファ・サイズ (32 ビット)
A0	L0
A1	L1
....
A(N-1) (I)	L(N-1) (I)

表1の注：

(1) NはDMAエンジンにアクセスされるバッファの数を表します。

チェックサム

インターネット・チェックサムはパケットが送信先に到着した際に、データの完全性を検証します。これは次の3つの主要項目で構成されます。

1. 16ビット数値を32ビットのアクキュムレータに加算します。
2. 32ビット・アクキュムレータを16ビットの数値に折り返します。
3. 折り返した数値をビット反転し、呼び出し側に返送します。

加算

16ビットのデータを使用するチェックサムアルゴリズムは32ビットのNios II プロセッサに効率的に実行できません。性能は32ビット・メモリ・アクセスの使用により改善されます。機能仕様がデータを16ビット数値としてアクキュムレータに加算される必要があるため、最適化されたコードはワードの上位16ビットと下位16ビットを平行で累算します。データ・バッファは32ビットの境界で終わることが保証されていないため、追加コードがバッファの終わりの条件をチェックします。

折り返し

加算が終わった後、16 ビットのみにより占められる結果を作成するために、累算値を折り返す必要があります。折り返し動作は 32 ビット・アキュムレータの値を 2 つの 16 ビット・セグメントに分けて、それらを加算します。第一回目の折り返し加算結果が 16 ビットの値をオーバーフローした場合でも、第二回目の折り返し加算により 16 ビットをオーバーフローしない結果を得ます。折り返しコードは並列化により、第二回目の折り返し加算の実行と最終チェックサム結果の保存を 1 つのハードウェア動作で可能になるよう工夫されています。

ビット反転

データ破壊テストを強化するために、呼び出し側に返送して期待値と比較する前に累算値はビット反転されます。インターネット・チェックサムについて詳しくは、業界標準の RFC 1071 を参照してください。

リファレンス・デザイン

この項では、リファレンス・デザインの概要とそれを作成するためのステップについて説明します。

デザイン例の概要

提供された例は、線形的に増加するデータ内容を含む SDRAM に 50 のバッファを作成します。各バッファは、ダイナミックに割り当てられて、64 と 1500 バイトの間のランダムな長さを持っています。まず、ソフトウェアのみで実装されたチェックサム・アルゴリズムを用い各バッファの結果を計算します。この計算は、時間測定の精度を向上するために 1000 回繰り返されます。そして、C2H コンパイラによって作成されたアクセラレータ・ファンクションを使用して同じテストを実行します。この例は、ハードウェアとソフトウェア実装の結果を比較して、Nios II IDE の Console ビューに結果を出力します。

このデザイン例では、プロセッサがデータ・キャッシュを持つ場合には、一回のキャッシュ・フラッシュが起こります。キャッシュ・フラッシュは一回で十分です：scatter-gather DMA は何回呼び出されても、転送されたデータはメモリ上に配置されることが保証されています。

マルチ・マスターシステムでは、キャッシュ・コヒーレンスは問題になるかもしれません。アクセラレータがいくつかの位置から呼び出される場合に、一回のデータ・キャッシュ・フラッシュは、キャッシュのコヒーレンスを保証するには十分でないかもしれません。C2H ビューでは、**Use hardware accelerator in place of the software implementation. Flush data cache before each call** を選択してください。このオプションは自動的にアクセラレータ・ラッパー・ファンクションにキャッシュ・フラッシュ・コードを加えるので、コードにキャッシュ・フラッシュを入れる必要はありません。



キャッシュについて詳しくは、「Nios II ソフトウェア開発ハンドブック」のセクション 3 の「**キャッシュ・メモリ**」章を参照してください。

デザイン・ファイルのダウンロード

このアプリケーション・ノートのためのソフトウェア・ファイルはアプリケーション・ノートの資料ページにあります。ソフトウェア・ファイルへのハイパーリンクは www.altera.co.jp/literature/lit-an.jsp にあるこのドキュメントの次に記載されています。二つのファイルがあります：

- `checksum_software.c` –メイン・プログラムおよび Nios II プロセッサに実行されるチェックサム計算の C コード
- `hw_checksum.c` –ハードウェア・アクセラレータとして実行するように最適化されたチェックサム計算の C コード

ハードドライブ上にこれらの C ファイルをコピーしてください。

システムの作成

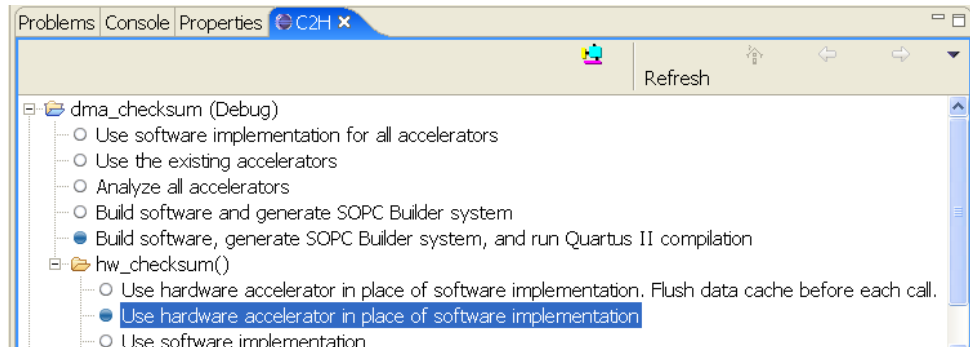
以下のステップでは Nios II EDS と共にインストールされた **standard** ハードウェアのデザイン例をベースとし、scatter-gather DMA チェックサム・デザインを作成していきます。提供されたファイル (`checksum_software.c` と `hw_checksum.c`) は、このシステムに必要とされる機能を **standard** デザインに追加実装します。

以下のステップは、どのように追加ソース・ファイルをシステムに追加し、ソフトウェアとハードウェアをコンパイルして、システムを動作させるかを説明します：

1. 使用しているボードの **standard** デザイン・プロジェクトのディレクトリを作業ディレクトリにコピーします。例えば、`dma_checksum` という作業ディレクトリに `Nios II EDS install path>\examples\hdl>\development board>\standard` をコピーします。
2. Nios II IDE を開いて、**Blank Project** ソフトウェア・テンプレートに基づく新しい C/C++ アプリケーション・プロジェクトを作成します。プロジェクト名は `dma_checksum` とします。ターゲット・ハードウェアには、`dma_checksum` ディレクトリの SOPC Builder システム・ファイル `std_<FPGA>.ptf` を使用します。
3. `checksum_software.c` と `hw_checksum.c` を Nios II IDE 上の `dma_checksum` プロジェクトにコピーします。
4. NiosII IDE エディタで `hw_checksum.c` を開きます。
5. ファンクション名の `hw_checksum()` をハイライトして、それを右クリックして、次に **Accelerate with the Nios IIC2H Compiler** をクリックします。

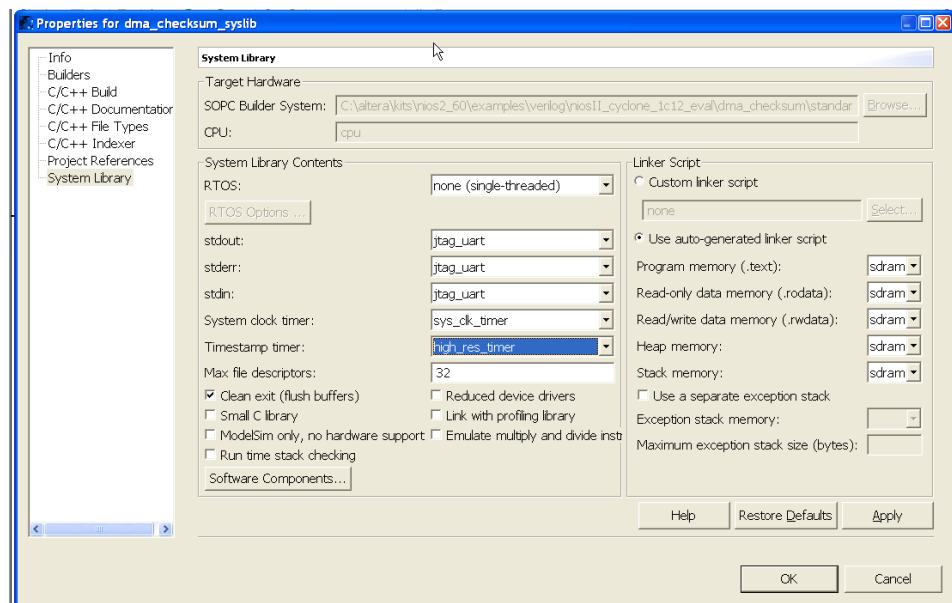
6. C2H ビューでは、**Build software, generate SOPC Builder system, and run Quartus II compilation** と **Use hardware accelerator in place of software implementation** を選択します。(図 2 を参照)

図 2. ハードウェア・アクセラレータの選択



7. `dma_checksum_syslib` プロジェクトの **System Library properties** ページ上で、**Timestamp timer** を `high_res_timer` に設定します。(図 3 を参照)

図 3. high res timer の設定



8. C/C++Projects ビューで、`dma_checksum` プロジェクトを右クリックして、**Build Project** をクリックし、プロジェクトを再作成します。C2H Compiler は、ハードウェア・アクセラレータを含む FPGA コンフィギュレーション・ファイル (`.sof`) を作成するために、関数 `hw_checksum()` のハードウェア・アクセラレータを作成して、SOPC Builder システムを再生成して、Quartus II プロジェクトを再コンパイルします。この過程はコンピュータによって違い、完成するまでは 30 分かかかるかもしれません。
9. Quartus II Programmer を使用して、新しい FPGA コンフィギュレーション・ファイルで FPGA をコンフィギュレーションします。

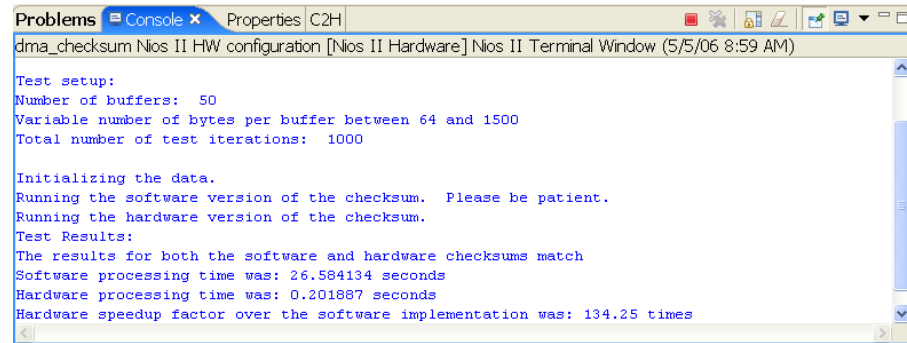
10. C/C++Projects ビューで、**dma_checksum** プロジェクトを右クリックし、**Run As** をポイントして、**Nios II Hardware** をクリックし、実行可能ファイル (.elf) をボードにダウンロードします。

11. Console ビューで結果を観測します。

期待値

ソフトウェアとハードウェアのバージョンの 1000 繰り返しデザインを実行した後、以下のテキストが Console ビューに表示されます。使用するターゲット・デバイスによって、結果は異なるかもしれません。

図 4. Console ビューの出力



```
dma_checksum Nios II HW configuration [Nios II Hardware] Nios II Terminal Window (5/5/06 8:59 AM)
Test setup:
Number of buffers: 50
Variable number of bytes per buffer between 64 and 1500
Total number of test iterations: 1000

Initializing the data.
Running the software version of the checksum. Please be patient.
Running the hardware version of the checksum.
Test Results:
The results for both the software and hardware checksums match
Software processing time was: 26.584134 seconds
Hardware processing time was: 0.201887 seconds
Hardware speedup factor over the software implementation was: 134.25 times
```

機能強化

この例で得た性能は十分なものですが、追加のデザイン変更によりシステム性能をさらに改善することができます。この項は性能をさらに強化するためのデザイン変更点について紹介します。

メモリ・サブシステム

システムにおけるメモリのアーキテクチャは効率および性能に影響を与えます。このデザインは単一のメモリ・デバイスで Nios II ソフトウェア、ディスクリプタ・テーブル、および多数のデータ・バッファを保存します。性能を向上させるためには、バッファおよびディスクリプタ・テーブルに低遅延 SRAM (オンチップまたはオフチップ) を使用します。同一メモリを共用する代わりに、Nios II ソフトウェア、ディスクリプタ・テーブル、およびバッファ用に独立したメモリを割り当てます。独立したメモリを使用すると、Avalon スイッチ・ファブリックにおけるアービトレーション・ロジックが最小限に抑えられ、より高い最大クロック周波数 (f_{MAX}) が得られます。独立したメモリの使用についてもう 1 つの主要な利点はメモリ競合の減少です。バッファを 2 つ以上のメモリに配置すると、複数のバッファを同時にアクセスするようソース・コードの変更が可能になり、向上したメモリ帯域幅を利用してデータ・スループットを増加することができます。

DMA デザイン

このデザインにおける DMA エンジンを実装する C コードはコードのコンパクト性より柔軟性を重視しています。バッファ・サイズを 32 ビットの倍数にするようにデザインを修正した場合には、転送の最後での 16 および 8 ビットのワードをアクセスするロジックを削除することができます。この最適化はアクセラレータがバッファの切り換えに要する時間を短縮します。得られたスピードアップはバッファのサイズに応じて異なります。大きなバッファに対してこの改善は小さいですが、小さなバッファに対してこの改善は著しいものとなります。

DMA エンジンにおける並列度を増加させるには、複数の for ループを作成し、それぞれがディスクリプタ・テーブルの内容を分担します。例えば、それぞれの `buffer_ctrs` がテーブルの四分の一をアクセスすれば、C2H コンパイラは 4 つの Avalon マスター・ポートを作成し、ディスクリプタ・テーブル内容を完了する時間を短縮します。このデザイン変更はメモリ・サブシステムが DMA における 400% のスピードアップに対応できる十分な帯域幅を有する場合のみ著しい性能向上が得られます。

他の応用

DMA エンジンを使用し、チェックサム演算を使用しないプロジェクトの場合でも、コードの変更で必要なハードウェア・アクセラレータを作成することが可能です。例えば、チェックサム・コードを FIFO の動作をエミュレートする単一アドレスにデータを書き込む for ループに置き換えます。もう 1 つの可能性は高速化された DMA ファンクションを使用して他のベンダーによって提供された IP ブロックにデータを転送することで、既存のシステムに対する最少の変更でシステム性能を向上させることが可能です。



101 Innovation Drive
San Jose, CA 95134
www.altera.com
Technical Support
www.altera.com/support

Copyright © 2009 Altera Corporation. All rights reserved. Altera, The Programmable Solutions Company, the stylized Altera logo, specific device designations, and all other words and logos that are identified as trademarks and/or service marks are, unless noted otherwise, the trademarks and service marks of Altera Corporation in the U.S. and other countries. All other product or service names are the property of their respective holders. Altera products are protected under numerous U.S. and foreign patents and pending applications, maskwork rights, and copyrights. Altera warrants performance of its semiconductor products to current specifications in accordance with Altera's standard warranty, but reserves the right to make changes to any products and services at any time without notice. Altera assumes no responsibility or liability arising out of the application or use of any information, product, or service described herein except as expressly agreed to in writing by Altera Corporation. Altera customers are advised to obtain the latest version of device specifications before relying on any published information and before placing orders for products or services.



I.S. EN ISO 9001