Accelerating Functions with the C2H Compiler: Scatter-Gather DMA with Checksum

July 2006, Version 1.1 Application Note 417

Introduction

The Nios® II C-to-Hardware Acceleration (C2H) Compiler is a powerful tool that generates hardware accelerators for software functions. This application note uses the C2H Compiler to map C code for a scatter-gather direct memory access (DMA) function to equivalent hardware. The hardware accelerator provides data to an Internet checksum function. Reference files included with the application note provide C code to implement the scatter-gather DMA and the checksum. There are two versions of the checksum code: The first runs as software on the Nios II processor. The second becomes a hardware accelerator, replacing the C code with equivalent logic in the FPGA. The speedup for the hardware version is typically more than two orders of magnitude. The exact speedup depends on the target FPGA.

Although the scatter-gather DMA code in this example design is tightly integrated with the checksum code, it could be modified to become a front-end DMA accelerator for a number of different applications, such as networking, audio, video, or high speed I/O. The concluding sections of this application note mention additional architectural revisions you could make to the design to further improve system performance with the C2H Compiler.

Prerequisites

This application note assumes you are familiar with the following topics:

- ANSI C syntax and usage
- Defining and generating Nios II hardware systems with SOPC Builder
- Compiling Nios II hardware systems with the Altera® Quartus® II development software
- Creating, compiling, and running Nios II software projects
- Nios II C2H Compiler

To familiarize yourself with the basics of the C2H Compiler, refer to the *Nios II C2H Compiler User Guide*. To learn about defining, generating, and compiling Nios II systems, refer to the *Nios II Hardware Development Tutorial*. To learn about Nios II software projects, refer to the *Nios II Software Development Tutorial* available in the Nios II IDE help system.

Hardware & Software Requirements

To use the design files provided with this application note, you must have the following software and hardware:

- Quartus II development software version 6.0 or later, installed on a Windows or Linux computer
- Nios II Embedded Design Suite (EDS) version 6.0 or later
- Nios development board provided by Altera, such as the Nios Development Board, Cyclone® II Edition
- JTAG download cable compatible with your target hardware, such as a USB-Blaster™ cable

Background: Scatter-Gather DMA

Embedded systems frequently employ DMA engines to increase data throughput and offload memory copy operations from the processor. In many cases the memory locations being accessed are dispersed throughout the address space. A scatter-gather DMA engine can manage multiple DMA transfers to noncontiguous memory locations, instead of having the processor issue separate DMA transfers for each region of memory. A scatter-gather DMA creates a table of address and length pairs called a descriptor table, which prepares the engine for all of the transfers. The DMA engine incurs very little overhead when switching between buffer locations, because no software intervention is necessary.

Software-Only Implementation

Example 1 shows the software implementation of the checksum algorithm, which is based on the Braden, Borman and Partridge's implementation of RFC 1071. This implementation is typical for execution on a processor. It is not optimal for the C2H Compiler and does not exploit knowledge of the underlying hardware to create higher performance results.

Example 1: Software Implementation of Internet Checksum

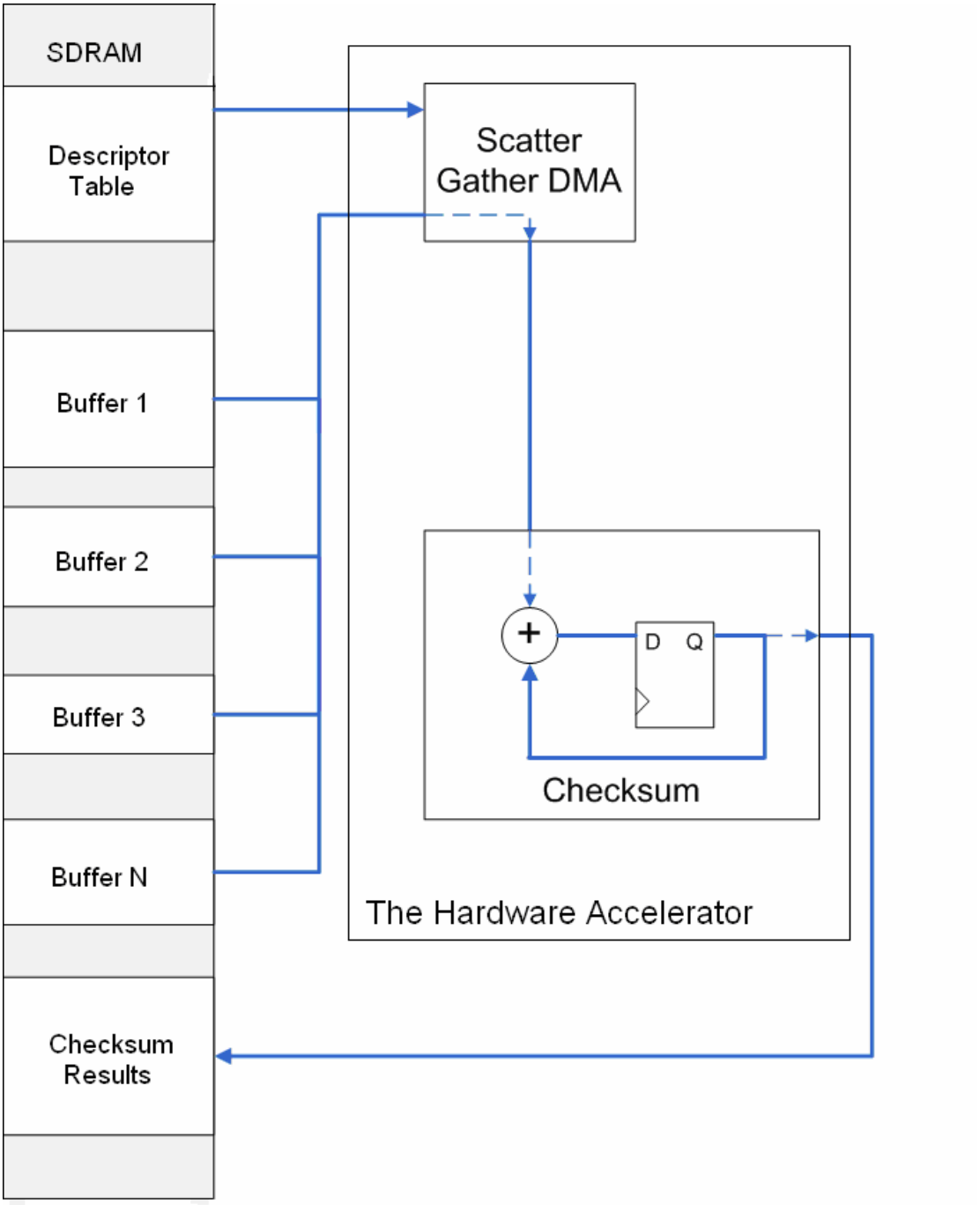
```
/************************************************************************
* Portable C implementation of the Internet checksum, derived           *
* from Braden, Borman, and Partridge's example implementation           *
* in RFC 1071.                                                          *
*                                                                       *
* 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);
}
```

Accelerated Implementation

This section discusses how you can refactor the checksum implementation shown in Example 1 for optimal results using the C2H Compiler. The implementation of the scatter-gather DMA function for the C2H Compiler, shown in Example 2, is straight-forward. It resembles the methodology a software developer would use to access multiple

buffer locations if no DMA engine were present. The scatter-gather DMA design is integrated with a checksum function to validate Ethernet packets. The checksum validation, which requires the processor to execute a large number of memory accesses, is time consuming. Because the primary bottleneck in the checksum computation is memory throughput, combining the checksum algorithm and DMA operation into a hardware accelerator maximizes system performance. Figure 1 shows a block diagram of the hardware accelerator and its connections to memory.

Figure 1: Simplified Scatter-Gather DMA



Example 2 shows the C code that has been optimized for hardware mapping by the C2H compiler. The pointer `addr` is used to access the data in each buffer located in main memory. The value `count` is used to control the number of iterations of the DMA loop contained in the accelerated function.

Example 2: Scatter-gather DMA Code Integrated with Hardware Implementation of Checksum

```
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          *
   /*****************************************************************/
  }
```

```
}
```

The scatter-gather DMA accelerated function created by the C2H Compiler can perform any combination of read and write accesses. In the reference design provided, the C code to transfer the data in each buffer is replaced by an accelerated function that reads from the data buffers residing in main memory (SDRAM). The C2H Compiler replaces the dereference operation for `addr` with an Avalon master port that connects to the SDRAM memory. The code to provide the base address and length of each entry in the descriptor table to the accelerated DMA function remains in software. The remainder of the C code to implement the checksum calculation is mapped to an accelerated function.

Table 1 shows the descriptor table. Each row represents an address and length value of an individual buffer. The address and length values are 32 bits wide in order to handle any buffer location within the Nios II processor's address space. No master port should modify the descriptor table or buffers while the accelerator is operating. Changing the descriptor table while it is in use can lead to synchronization and data integrity issues.

Table 1: Descriptor Table

Buffer Address (32 bit)	Buffer Length (32 bit)
A0	L0
A1	L1
....	....
A(*N*-1)	L(*N*-1)
Notes: (1) *N* represents the number of buffers to be accessed by the DMA engine.	

Checksum

The Internet checksum verifies the integrity of data when a packet reaches its destination. It is comprised of the following three main sections.

1. Summing 16-bit values into a 32-bit accumulator
2. Folding the 32-bit accumulator into a 16-bit value
3. Returning the inverted folded value to the caller

Summing

The checksum algorithm, which uses 16-bit data, runs inefficiently on the 32-bit Nios II processor. Performance is improved by employing 32-bit memory accesses. Because the functional specification requires the data to be added to the accumulator as 16-bit values, the optimized code accumulates the upper and lower half of the word in parallel. Additional code checks for the end-of-buffer condition because the data buffers are not guaranteed to end on a 32-bit boundary.

Folding

When the summation is complete, the accumulated value must be folded to create a result that only occupies 16 bits. The folding operation divides the 32-bit accumulator value into two, 16-bit segments and adds them together. If the result of the first fold overflows a 16-bit location, a second fold is performed to obtain a result that does not overflow the 16-bit memory location. The folding code takes advantage of the parallelization which is possible in a hardware implementation by performing the second fold and storing the final checksum result in one operation.

Inversion

To strengthen the data corruption test, the accumulated value is inverted before it is returned to the caller for comparison with the expected value. For more information about the Internet checksum refer to the industry standard RFC 1071.

The Reference Design

This section provides an overview of the reference design and leads you through the steps to create it.

Overview of the Example Design

The example provided creates 50 buffers in SDRAM containing linearly increasing data. Each buffer is dynamically allocated and has a random length between 64 and 1500 bytes. A software-only checksum algorithm calculates the results for each buffer. This calculation is repeated 1000 times to improve the accuracy of the time measurement. Then, the same test runs using the accelerated function created by the C2H compiler. The example compares the results of the hardware and software implementation and outputs the results to the Console view in the Nios II IDE.

In this example design, a single cache flush occurs if the processor is configured to have a data cache. A signal cache flush is optimal: although the scatter-gather DMA is called many times, the data to be transferred is guaranteed to reside in a memory.

In multiple master systems, cache coherency can be an issue. If the accelerator is called from several locations, a single data cache flush may not be sufficient to guarantee cache coherency. Select the "Use hardware accelerator in place of the software implementation. Flush data cache before each call." from the C2H view. This option automatically adds cache flushing code to the accelerator wrapper function so that you do not need to include a cache flush in your code.

For more information on caching, see the *Cache & Tightly-Coupled Memory* chapter in Section 3 of the *Nios II Software Developer's Handbook*.

Downloading the Design Files

The software files for this application note are available on the application notes literature page. A hyperlink to the software files appears next to this document at **www.altera.com/literature/lit-an.jsp**. There are two files:

- checksum_software.c - the main program and C code for the checksum calculation to be run on the Nios II processor
- hw_checksum.c - C code for the checksum calculation that has been optimized to run as a hardware accelerator

Copy these C files to a known location on your hard drive.

Creating the System

In the following steps you create a scatter-gather DMA checksum design starting with the **standard** hardware example design installed with the Nios II EDS. The supplied files, `checksum_software.c` and `hw_checksum.c`, implement the additional functionality required by the system.

The following steps describe how to add the additional source files to the system, compile the software and hardware, and run the system:

1. Copy the **standard** example design project directory for the board you are using to a working directory. For example, copy *<Nios II EDS install path>***examples**\ *<hdl>**<development board>***standard** to a working directory called **dma_checksum**.

2. Open the Nios II IDE and create a new C/C++ application project based on the **Blank Project** software template. Name this project **dma_checksum**. For the target hardware, use the **std_<*FPGA*>.ptf** SOPC Builder system file from the **dma_checksum** directory.

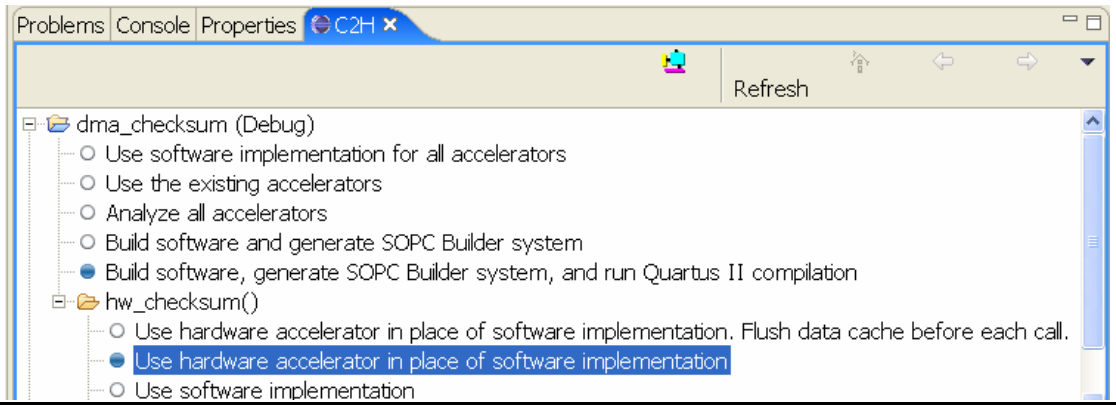
3. Copy **checksum_software.c** and **hw_checksum.c** into the **dma_checksum** project in the Nios II IDE.

4. Open **hw_checksum.c** in the Nios II IDE editor.

5. Highlight the function name `hw_checksum()`, right-click it, and then click **Accelerate with the Nios II C2H Compiler**.

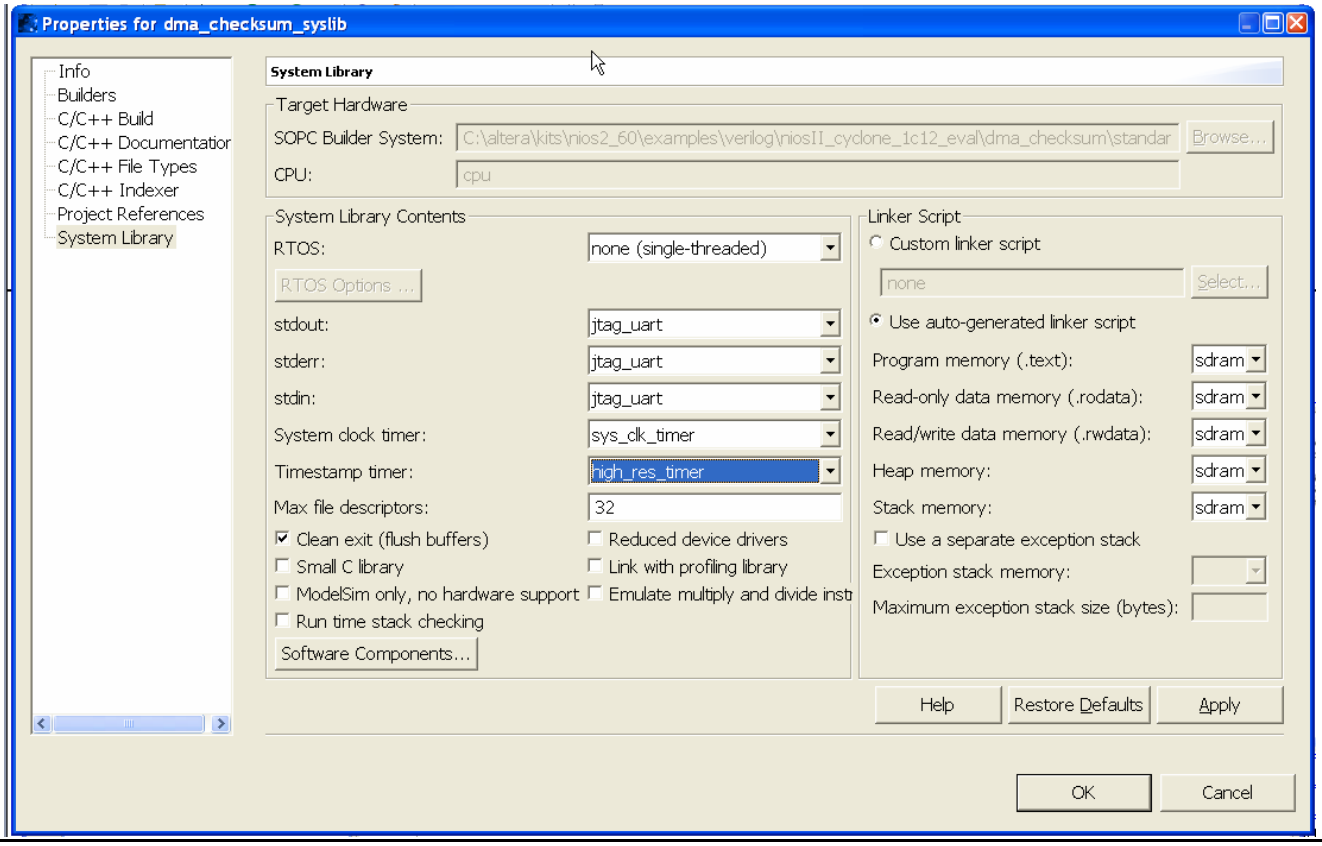
6. In the C2H view, select **Build software, generate SOPC Builder system, and run Quartus II compilation** and **Use hardware accelerator in place of software implementation**. See Figure 2.

Figure 2: Select the Hardware Accelerator



7. On the **System Library** properties page for the **dma_checksum_syslib** project, set the **Timestamp timer** to **high_res_timer.** See Figure 3.

Figure 3: Set the high_res_timer



8. Rebuild the project by right-clicking the project **dma_checksum i**n the C/C++ Projects view and clicking **Build Project**. The C2H Compiler builds the hardware accelerator for function `hw_checksum()`, regenerates the SOPC Builder system, and recompiles the Quartus II project to create an FPGA configuration file (**.sof**) that includes the hardware accelerator. This process can take 30 minutes to complete, depending on your computer.

9. Use the Quartus II Programmer to configure the FPGA with the new FPGA configuration file.

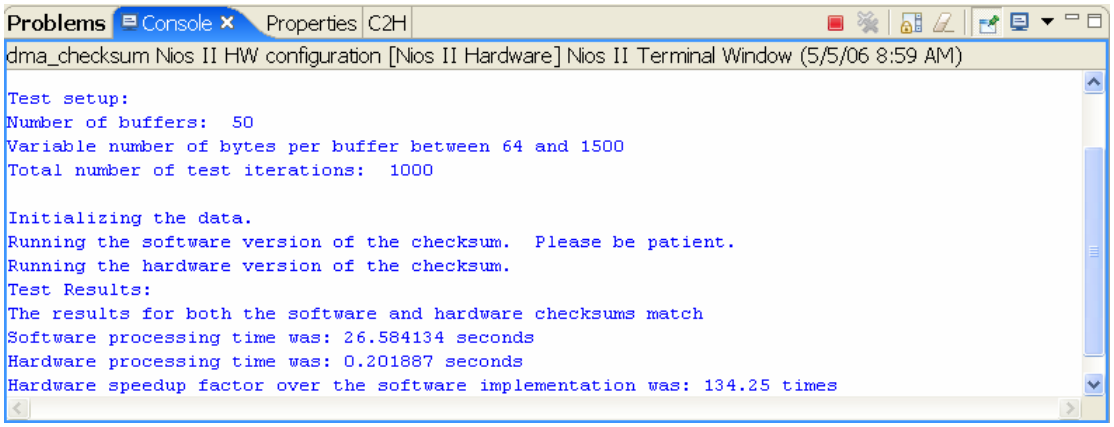
10. In the C/C++ Projects view, right-click the project **dma_checksum**, point to **Run As**, and then click **Nios II Hardware** to download the executable file (**.elf**) to the board and start execution.

11. Observe the results in the Console view.

Expected Results

After the design runs 1000 iterations of the software and hardware versions, the following text appears in the Console view. Depending upon the target device used, the results might vary.

Figure 4: Console View Output



Enhancements

Although the performance gains in this example are substantial, additional design changes can further improve system performance. This section introduces other enhancements you could perform to the design to further increase performance.

Memory Subsystem

The architecture of the memory in the system impacts efficiency and performance. This design uses a single memory device to store Nios II software, the descriptor table, and the numerous data buffers. To improve performance, use low-latency SRAM (on-chip or off-chip) for the buffer and descriptor table memory. Instead of sharing the same memory, specify separate memories for Nios II software, the descriptor table, and buffer locations. Using separate memories minimizes arbitration logic in the Avalon switch fabric and results in a higher maximum clock frequency ($\mathbf{f}_{MAX}$). Another key benefit of using separate memories is reduced memory contention. If you place the buffers in two or more physical memories, you can then modify the source code to access multiple buffers concurrently, taking advantage of the additional memory bandwidth to increase data throughput.

DMA Design

The C code to implement the DMA engine in this design favors flexibility over code compactness. If you modify the design to require the buffer sizes to be multiples of 32 bits, you can eliminate the logic that accesses 16- and 8-bit words at the end of the transfer. This optimization reduces the amount of time the accelerator needs to switch between buffers. The resulting speedup factor depends on the size of the buffers. For large buffers this improvement is minor. For small buffers this improvement is significant.

To increase parallelism in the DMA engine, create multiple `for` loops, each traversing a portion of the descriptor table. For example, if each of the `buffer_ctrs` accesses one-fourth of the table, the C2H Compiler will create four Avalon master ports, reducing the time to complete the entries in the descriptor table. This design change will only result in significant performance gains if the memory subsystem has sufficient bandwidth to accommodate the 400% speedup in the DMA.

Other Applications

For projects that employ the DMA engine but not the checksum operation, you can modify the code to create the required hardware accelerators. For example, replacing the checksum code with a `for` loop that writes data to a single address emulates the behavior of a FIFO. Another possibility is to use the accelerated DMA function to transfer data to IP blocks provided by other vendors, improving system performance with minimal changes to an existing system.2

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
www.altera.com
Applications Hotline:
(800) 800-EPLD
Literature Services:
literature@altera.com