Nios II C2H Compiler

User Guide

nsai

I.S. EN ISO 9001

UG-N2C2HCMPLR-1.6

Contents

Chapter 1. Introduction to the C2H Compiler

Chapter 2. Getting Started Tutorial

Chapter 7. ANSI C Compliance and Restrictions

Additional Information

1. Introduction to the C2H Compiler

The Nios® II C-to-Hardware Acceleration (C2H) Compiler is a tool that allows you to create custom hardware accelerators directly from ANSI C source code. A hardware accelerator is a block of logic that implements a C function in hardware, which often improves the execution performance by an order of magnitude. Using the C2H Compiler, you can develop and debug an algorithm in C targeting an Altera® Nios II processor, and then quickly convert the C code to a hardware accelerator implemented in a field programmable gate array (FPGA).

The C2H Compiler improves the performance of Nios II programs by implementing specific C functions as hardware accelerators. The C2H Compiler is not designed to create arbitrary hardware systems from C code. Rather, the C2H Compiler is a tool for generating a hardware accelerator module, functionally identical to the original C function, that offloads and enhances the performance of the Nios II processor.

User Guide Overview

This user guide comprises the following chapters:

- Chapter 1, Introduction to the C2H Compiler provides a detailed background on the C2H Compiler and the concepts required to use it.
- Chapter 2, Getting Started Tutorial provides hands-on instructions that teach you the first steps to begin using the C2H Compiler.
- Chapter 3, C-to-Hardware Mapping Reference provides reference on how the C2H Compiler translates C constructs to hardware structures.
- Chapter 4, Understanding the C2H View helps you use the C2H view to get performance information and to control the compilation of accelerators.
- Chapter 5, Accelerating Code Using the Nios II Software Build Tools Command Line explains how to use the C2H Compiler with the Nios® II software build tools.
- Chapter 6, Pragma Reference summarizes usage of all C2H `#pragma` directives.
- Chapter 7, ANSI C Compliance and Restrictions documents all sections of the ANSI C specification that the C2H Compiler does not support.

Target Audience

This user guide assumes you have at least a basic understanding of hardware design for field programmable gate arrays (FPGAs). It also assumes you are fluent in the C language and you have experience with software design in C for microprocessors.

The C2H Compiler operates in conjunction with the following Altera tools:

■ Quartus II software for creating FPGA designs
■ SOPC Builder system integration tool for creating Nios II processor hardware systems
■ C programming environments for the Nios II processor:
   ● Nios II integrated development environment (IDE)
   ● Nios II software build tools

To benefit from this user guide, you do not need to be an expert in these tools, and you do not need an understanding of any particular Altera FPGA family. However, at least a basic understanding of each tool is required to use the C2H Compiler practically.

Introduction

This chapter introduces the Nios II C2H Compiler. The sections in this chapter discuss the features, background, and principles of the C2H Compiler, and describe the most appropriate types of C code for acceleration. After reading this chapter, you will understand all the concepts necessary to begin using the C2H Compiler.

Features

The C2H Compiler is founded on the following premises:

■ ANSI C syntax is sufficient to describe computationally intensive or memory access-intensive tasks.
■ A C-to-hardware tool must not disrupt existing software and hardware development flows.

Based on these premises, the C2H Compiler's design methodology provides the following features:

■ ANSI C compliance – The C2H Compiler operates on plain ANSI C code, and supports most C constructs, including pointers, arrays, structures, global and local variables, loops, and subfunction calls. The C2H Compiler does not require special syntax or library functions to specify the structure of the hardware. Unsupported ANSI C constructs are documented.

■ Straightforward C-to-hardware mapping – The C2H Compiler maps each element of C syntax to a defined hardware structure, giving you control over the structure of your hardware accelerator.

■ Integration with C language development environments for the Nios II processor, including the Nios II integrated development environment (IDE), and the Nios II software build tools. You control the C2H Compiler with the Nios II C development tools. You do not need to learn a new environment to use the C2H Compiler.

■ Based on SOPC Builder and Avalon system interconnect fabric – The C2H Compiler uses SOPC Builder as the infrastructure to connect hardware accelerators into Nios II systems. A C2H accelerator becomes a component within an existing Nios II system. SOPC Builder automatically generates system interconnect fabric to connect the accelerator to the system, saving you the time of manually integrating the hardware accelerator.

■ Reporting of generated results – The C2H Compiler produces a detailed report of hardware structure, resource usage, and throughput.

Hardware accelerators generated by the C2H Compiler have the following characteristics:

■ Parallel scheduling – The C2H Compiler recognizes events that can occur in parallel. Independent statements are performed simultaneously in hardware.

■ Direct memory access – Accelerators access the same memories that the Nios II processor does during execution.

■ Loop pipelining – The C2H Compiler pipelines the logic implemented for loops, based on memory access latency and the amount of code that operates in parallel.

■ Memory access pipelining – The C2H Compiler pipelines memory accesses to reduce the effects of memory latency.

Design Abstraction and the Rise of C for FPGAs

There is much interest in "C-to-gates" tools that promise a practical method to create hardware logic directly from C code. However, early attempts have had limited success gaining acceptance in the design community. This section discusses the historical background of the C2H Compiler, and looks at the questions "why is this methodology a good idea?" and "why now?"

C compilers and FPGA design tools have evolved along separate paths, but both are founded on the same premise: Higher levels of design abstraction enable engineers to create designs of greater size and complexity. Simultaneous with this evolution, Moore's law has delivered chips of increasing density and complexity, such as FPGAs capable of

implementing entire systems on a chip. As a result, the tools available to FPGA and software designers have undergone continual transformation of design-entry methods and behind-the-scenes optimization techniques. This transformation has enabled designers to create ever-bigger designs to fill ever-growing chip capacity.

Recent years have seen the broad acceptance of FPGA-based microprocessor cores, such as the Nios II processor, and system integration tools, such as SOPC Builder. These tools made it possible, for the first time, to implement C code easily in an FPGA-based system. Optimizing and evolving these tools is a natural next step for C-based design on FPGAs. This background sets the stage for practical advances in C-to-hardware technologies based on an established design methodology.

FPGA-based processors and system integration tools offer new ways to improve the performance of embedded systems. Traditional methods to increase performance of processor systems include:

■ Increasing clock speed
■ Upgrading to a processor with higher Dhrystone MIPS-per-megahertz performance
■ Coding critical sections of software in assembly language

FPGA-based processor systems enable additional optimization techniques capable of achieving much higher performance gains. These techniques include:

■ The ability to rapidly alter the FPGA design, allowing you to prototype a variety of architectures
■ The ability to divide and conquer processing tasks by instantiating multiple processor cores
■ The ability to augment a processor with custom hardware that off-loads processor-intensive operations into the FPGA fabric
■ The ability to adjust memory architecture for memory-intensive operations, such as using high-speed, point-to-point connections to fast memory buffers

The application of these techniques relies on real-world tools to implement them. Consequently, the acceptance of these techniques has grown as system integration tools, such as Altera's SOPC Builder, have matured and gained acceptance. It is a fortunate coincidence that these techniques also directly benefit C-to-gates methodologies. Flexibility of hardware architecture and ease of implementation are at the heart of the appeal of C-to-gates tools.

The Nios II C-to-Hardware Acceleration (C2H) Compiler represents Altera's next step in the evolution of embedded systems design. The C2H Compiler uses the infrastructure provided by SOPC Builder and the Nios II processor, and adds a higher level of abstraction: converting C functions directly to hardware.

What to Expect From the C2H Compiler

The C2H Compiler is not designed to build all types of FPGA systems. It is designed specifically to *augment* the performance of programs that run on the Nios II processor; it does not replace the processor. Two notable implications are:

■ The C2H Compiler assumes that your C code runs successfully on a Nios II processor system.
■ The result of using the C2H Compiler is a program that runs on a Nios II processor system.

The C2H Compiler works best on C code that adheres to certain structural rules. It works well for many types of programs, but not all. Through education and habit, programmers structure C programs with an existing compiler in mind. Experienced designers learn the particular structures that produce optimal compiled results. The C2H Compiler is also a C compiler. It takes ANSI C programs that execute normally on a processor. However, the program structure for producing optimal hardware results with the C2H Compiler often differs from code structured for execution on a processor. You achieve the best results if you have a reasonable understanding of how the C2H Compiler translates C structures to hardware. Refer to chapter Chapter 3, C-to-Hardware Mapping Reference for details.

The C2H Compiler is not a replacement for traditional HDL-based hardware design. Tasks such as connecting modules together and interfacing to bus protocols are not easily inferred from ANSI C code. In the hands of an experienced user, the C2H Compiler allows considerable control over circuit latency and parallelism. However, it does not provide the ability to define user logic with complex timing requirements. For example, the C2H Compiler does not allow you to create an arbitrary state machine that guarantees a particular operation on a specific clock cycle.

☞ The Nios II processor is little-endian. For Nios II compatibility, C2H accelerators expect to exchange little-endian data with the processor. If your accelerator must handle big-endian data, you can swap the byte order in the accelerated C code. Ensure that the data is in little-endian form when your accelerated function transfers it to any unaccelerated function.

C2H Support in Nios II Tool Flows

The Nios II IDE is the preferred tool flow for developing Nios II C2H programs. The Nios II IDE allows you to carry out the following important tasks:

■ Debug your function prior to accelerating
■ Generate the accelerator and incorporate it into your hardware
■ Test and profile your software and hardware with the C2H accelerator

Altera recommends creating new C2H systems with the Nios II IDE.

👣 For information about using the Nios II IDE, refer to the *Using the Nios II Integrated Development Environment* appendix to the *Nios II Software Developer's Handbook*, or to the Nios II IDE help system. For information about Nios II tool flows, refer to "Development Flows for Creating Nios II Programs" in the *Overview* chapter of the *Nios II Software Developer's Handbook*.

The Nios II Software Build Tools also provide command-line support for pre-existing command-line C2H projects.

👣 For information about using C2H on the command line, refer to Chapter 5, Accelerating Code Using the Nios II Software Build Tools Command Line.

☞ The Nios II Software Build Tools for Eclipse do not support the C2H Compiler.

C2H Compiler Concepts

This section describes fundamental concepts underpinning the C2H Compiler. These concepts help you better understand how the C2H Compiler works and how you can produce optimal results.

Simplicity and Ease of Use

The C2H Compiler minimizes interruptions to existing design flows. The flow to generate a hardware accelerator and link software for it uses the familiar Nios II and SOPC Builder design tools. When you create a Nios II software project, you specify which C function (or functions) compiles as a hardware accelerator rather than instructions on a processor . The

C2H Compiler calls other tools in the background to handle the hardware and software integration tasks. Specifically, the C2H Compiler automatically performs the following tasks in the background:

1. Calls SOPC Builder to specify how the accelerator connects to the system, and then generates the system hardware.

2. Calls the Quartus® II software to recompile the hardware design and generate an SRAM object file (**.sof**).

Rapid Design Iteration to Find Optimal Partitioning of Hardware and Software

The C2H Compiler allows you to move the dividing line between hardware and software easily in C code, without significant additional design effort. As a result, you have the freedom to design iteratively, and explore multiple architectures. By contrast, writing a hardware accelerator by hand in a hardware description language (HDL) would require a significant amount of time to create the logic design and integrate it into the system. Changing the functional or performance requirements of hand-written HDL blocks can significantly impact design time.

With the C2H Compiler, you can accelerate as many functions as necessary to achieve the desired performance. You can balance the trade-off between performance and resource utilization with simple edits to the C source.

With these tools available to you, the process of achieving desired system performance undergoes a profound change: The balance of design time shifts away from creating, interfacing, and debugging hardware in favor of perfecting the algorithm implementation and finding the optimal system architecture.

Accelerate Performance-Critical Sections of Code

The C2H Compiler converts only sections of code that you specify. A typical program contains a mix of performance-critical code and other code. Performance-critical sections are often iterative and simple, but consume the majority of a program's execution time on a processor. They might occupy the processor by either computing a value, moving data, or both. The best use of hardware resources is to accelerate only the performance-critical functions of a program, rather than converting an entire program to hardware.

The C2H Compiler Operates at the Function Level

Code you want to accelerate must be expressed as an individual C function. The C2H Compiler converts all code within and below the chosen function to a hardware accelerator block. If the function you are accelerating calls a subfunction, the C2H Compiler also converts the subfunction to a hardware accelerator. Therefore, you must be careful that subfunctions are also good candidates for C2H acceleration.

If the code you want to accelerate is not isolated in a separate function, a good practice is to partition the function to separate the critical section into its own function. The resulting hardware accelerator then replaces only processor-intensive tasks, rather than setup or control tasks which the processor can implement efficiently.

System Architecture

Figure 1–1 shows the architecture of a simple Nios II processor system that includes one hardware accelerator.

Figure 1–1. Example System Topology with Single Hardware Accelerator
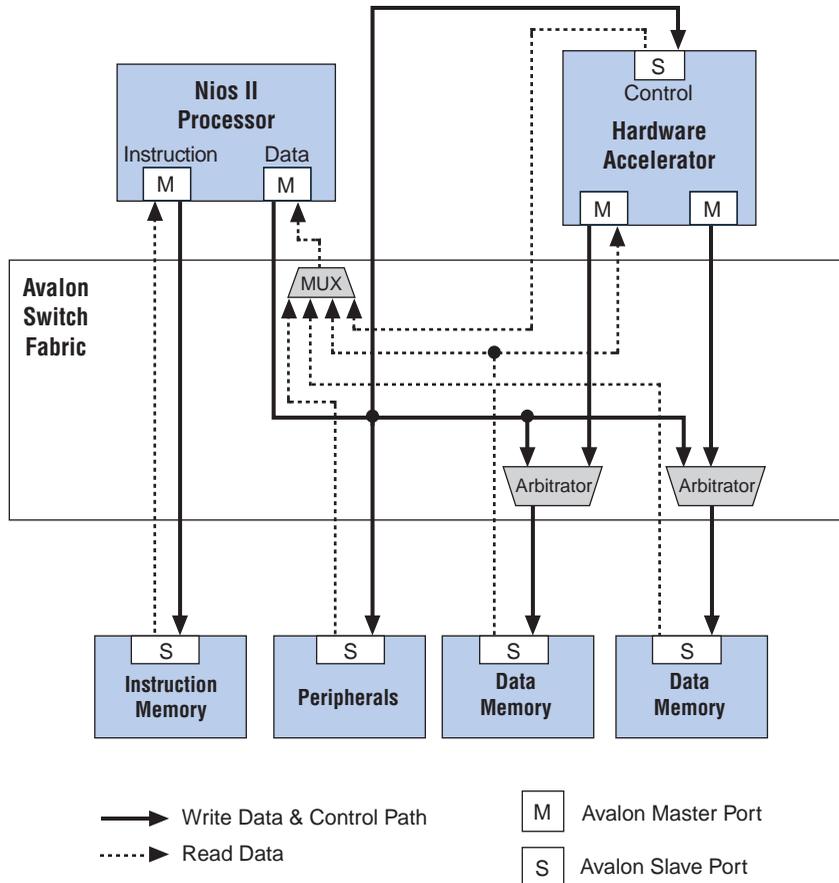
SOPC Builder automatically integrates the accelerator logic into the system as an SOPC Builder component. If there is more than one accelerator in the system, multiple accelerators appear in SOPC Builder. Accelerators are separate from the Nios II processor but can access the same memory devices that the Nios II processor can.

The accelerator's connections are managed by the C2H Compiler. You can manually customize the connections using pragma directives in the accelerated C code. Chapter 6, Pragma Reference, describes C2H Compiler pragma usage. You cannot edit the accelerator's connections in the SOPC Builder GUI.

Generation of a Hardware Accelerator

The C2H compilation flow shares commonalities with a conventional C compiler, but the scheduling of statements, optimization, and object generation is different. When generating a hardware accelerator, the C2H Compiler does the following:

1. Runs the GNU GCC preprocessor to evaluate macros, includes, and other preprocessing directives.

2. Parses code.

3. Creates a graph of data dependencies.

4. Performs some optimizations.

5. Determines the best sequence in which to perform each operation.

6. Generates an object file for the hardware accelerator. This object file is a synthesizable HDL file.

7. Generates a C wrapper function that isolates and hides the details of how the Nios II processor interacts with the hardware accelerator. The wrapper function is a C file that replaces the original C function at software link time.

The generated accelerator logic includes the following:

■ One or more state machines that manage the sequence of operations defined by the C function. On any clock cycle, an arbitrary number of computations and memory accesses can happen simultaneously, orchestrated by the state machines.
■ One or more Avalon Memory-Mapped (Avalon-MM) master ports, which fetch and store data as required by the state machines.
■ An Avalon-MM slave port and a set of memory-mapped registers that allow the processor to set up, start, and stop the accelerator.

The software wrapper, executing on the Nios II processor, controls the accelerator by reading and writing the register interface. From the perspective of the calling function, the result of calling the software wrapper is functionally the same as calling the original C function. The basic operation of the software wrapper is as follows:

1. Sets up parameters for the accelerator, similar to passing variables to the original, unaccelerated function.

2. Optionally flushes the processor's data cache to avoid cache coherency problems. Flushing the data cache might be necessary if the accelerator accesses the same memory that the processor does.

3. Starts the accelerator. Once an accelerator is running, it can return a value, terminate, or run continuously, depending on the design of the C source code.

4. Polls registers in the accelerator hardware to determine when the task completes.

5. If the function returns a result, reads the result value, and returns it to the calling function.

One-to-One Mapping From C Syntax to Hardware Structure

The C2H Compiler maps each element of C syntax to an equivalent hardware structure using straightforward translation rules that directly instantiate hardware resources based on the input C code. Once familiar with the C2H Compiler mappings, you can control the generated hardware structure with simple changes to your C source.

The following are examples of how the C2H Compiler translates C to hardware:

■ Mathematical operators (such as +, -, *, >>) become direct hardware equivalent circuits (such as add, subtract, multiply and shift circuits). These circuits might be shared between operations, depending on the degree of parallelism inherent in the C code.
■ Loops (such as for, while, do-while) become state machines that iterate over the operations inside the loop, until the loop condition is exhausted.
■ Pointer dereferences and array accesses (such as *p, array[i][j]) become Avalon-MM master ports that access the same memory that the processor does.
■ Statements not dependent on the result of a previous operation are scheduled as early as possible, allowing parallel execution to the extent possible.

■ Subfunctions called within an accelerated function are also converted to hardware using the same C-to-hardware mapping rules. The C2H Compiler creates only one hardware instance of the subfunction, regardless of how many times the subfunction is called within the top-level function. Isolating accelerated C code into a subfunction provides a method of creating a shared hardware resource within an accelerator.

The C2H Compiler performs certain optimizations when it can reduce logic utilization based on resource sharing.

Refer to Chapter 3, C-to-Hardware Mapping Reference for complete details of the C2H Compiler mappings.

Performance Depends on Memory Access Time

Applications that run on a processor are typically compute-bound, which means the performance bottleneck depends on the rate the processor executes instructions. Memory access time affects the execution time, but instruction and data caches minimize the time the processor waits for memory accesses.

With C2H hardware accelerators, the performance bottleneck undergoes a profound change: Applications typically become memory bound, which means the performance bottleneck depends on the memory latency and bandwidth. When multiple operations do not have data dependencies that require them to execute sequentially, the C2H Compiler schedules them in parallel. The resulting accelerator logic often must access memory to feed data to each parallel operation. If the hardware does not have fast access to memory, the hardware stalls waiting for data, reducing the performance and efficiency.

Achieving maximum performance from a hardware accelerator often involves examining your system's memory topology and data flow, and making modifications to reduce or eliminate memory bottlenecks. For example, if your C code randomly accesses a large buffer of data stored in slow SDRAM, performance suffers due to constant bank switching in SDRAM. You can alleviate this bottleneck by first copying blocks of data to an on-chip RAM, and allowing the accelerator to access this fast, low-latency RAM. Note that you can also accelerate the copy operation, which creates a direct memory access (DMA) hardware accelerator.

C Code Appropriate for Hardware Acceleration

This section describes guidelines for identifying code that is appropriate for the C2H Compiler.

Ideal Acceleration Candidates

Sections of C code that consume the most CPU time with the least amount of code are excellent candidates for acceleration. These tend to have the following characteristics:

- They contain a relatively small and simple loop or set of nested loops.
- They iterate over a set of data, performing one or more operations on the data per iteration, and then store the result.

Examples of such iterative tasks include memory copy-and-modify tasks, checksum calculations, data encryption, decryption, and filtering operations. In each of these cases, the C code iterates over a set of data many times, with either one or more memory reads or writes performed during each iteration.

Example 1–1 demonstrates a routine that performs a checksum calculation. This code excerpt is from a TCP/IP stack, and it calculates the checksum over ranges of data in a network protocol stack. Checksum calculations are typically a time-consuming part of an IP stack, because all data transmitted and received must be validated, which requires the processor to loop through all bytes.

Example 1–1. Checksum Calculation

```c
u16_t standard_chksum(void *dataptr, int len)
{
  u32_t acc;
  /* Checksum loop: iterate over all data in buffer */
  for(acc = 0; len > 1; len -= 2)
  {
    acc += *(u16_t *)dataptr;
    dataptr = (void *)((u16_t *)dataptr + 1);
  }
  /* Handle odd buffer lengths */
  if (len == 1)
  {
    acc += htons((u16_t)((*(u8_t *)dataptr)&0xff)<< 8);
  }
  /* Modify result for IP stack needs */
  acc = (acc >> 16) + (acc & 0xffffUL);
  if ((acc & 0xffff0000) != 0)
  {
    acc = (acc >> 16) + (acc & 0xffffUL);
  }
  return (u16_t)acc;
}
```

Accelerating this function could have a significant impact on execution time, especially the amount of time spent in the `for` loop. The remaining code executes once per call to format the result and check boundary cases. Accelerating the code outside the loop has little benefit, unless the entire `standard_chksum()` function is a called from another function that is also a good acceleration candidate. The most efficient hardware accelerator for this code would replace only the `for` loop. To accelerate the `for` loop only, you need to refactor the code to isolate the loop in a separate function.

Poor Acceleration Candidates

Accelerating some code can have negative performance impacts, or can unacceptably increase resource utilization, or both. Use the following guidelines to identify functions not to accelerate:

■ Code that contains many data or control dependencies must perform many sequential operations, and is a poor candidate for acceleration. A large number of dependencies makes it difficult for the C2H Compiler to fully optimize loops. Processors are designed to perform such operations efficiently.

■ If the code contains C syntax not supported by the C2H Compiler, it cannot be accelerated. Examples are floating point operations and recursive functions. Refer to Chapter 7, ANSI C Compliance and Restrictions.

■ Code that calls system and runtime library functions is a poor candidate for acceleration. For example, there is little point in accelerating printf() or malloc(). The underlying code contains a complex set of sequential operations and does not contain performance-critical loops.

■ Code that makes extensive use of global or external variables is a poor candidate for acceleration. Each time the C2H accelerator uses a global or external variable, it must access the Nios II processor's data memory, which is likely to cause a bottleneck.

There are exceptions to these guidelines. For example:

■ Experienced C coders often "unroll" iterative algorithms, representing them as a sequential set of operations to work better with an optimizing C compiler. If you can refactor the code and "roll up" the loop, you might be able to create an efficient hardware accelerator.

■ A critical inner loop might have a complex set of sequential operations which, if accelerated in hardware, consumes a lot of logic resources. This presents a trade-off: If the processor spends an unacceptable amount of time in this loop, it might be worth the hardware cost to accelerate the whole loop.

■ Some runtime library functions are iterative in nature. Examples include common data movement functions and buffer set functions, such as memcpy() or memset(). If your code calls one of these functions, you might consider writing a simple, custom implementation of the function, which you can then accelerate.

■ If your code uses global or external variables, it might be easy for you to refactor it to be suitable for acceleration. Refactor your code to copy the global or external variables to local storage, perform the calculation with the local variables, and then copy results back to global or external storage. The C2H Compiler implements local variables as fast, pipelined registers inside of the accelerator.

Understanding Code to Find Opportunities for Acceleration

The best way to obtain optimal results with the C2H Compiler is to understand your code, and know where the critical loops are. If you wrote the program from scratch, you probably understand where the critical sections of code are. If you are starting with an existing code base that you want to accelerate, the C2H Compiler can benefit you to the

extent that you analyze the code and understand it. In either case, the Nios II IDE profiling features can help you determine where the processor spends most of its time.

Examine the structure of the code for processor-specific or compiler-specific optimizations written into the structure of the code. These sections of code might result in poor performance with the C2H Compiler, and could benefit from refactoring for the C2H Compiler.

It can be difficult to identify the critical loop just by inspecting code, because programs often spend the majority of time iterating on just a few lines of code. The only way to know exactly where the processor spends the most time is to profile the application, and inspect the bottleneck functions.

Refer to *AN 391: Profiling Nios II Systems* for further information.

Next Steps

Now that you understand the underlying concepts of the Nios II C-to-Hardware Acceleration Compiler, you are ready for hands-on experience accelerating designs. Chapter 2, Getting Started Tutorialdescribes the C2H Compiler design flow, and gives step-by-step instructions to accelerate your first design. Altera also provides tutorials and application notes to deepen your understanding of the C2H Compiler.

Refer to the Nios II literature page for further C2H Compiler documentation: **www.altera.com/literature/lit-nio2.jsp**.

2. Getting Started Tutorial

Introduction

This chapter describes the design flow for the Nios® II C-to-Hardware Acceleration (C2H) Compiler. This chapter provides a design example and gives you a step-by-step tutorial to guide you through the process of creating your first hardware accelerator.

The example software design performs multiple iterations of a data-copy function. By accelerating the data-copy function, you achieve more than a 10-fold improvement in the execution performance. The resulting hardware accelerator resembles a hardware block with direct memory access (DMA) to copy data without processor intervention.

This tutorial assumes that you are familiar with the Nios II processor and the Nios II design flow.

For introductory information on designing with the Nios II processor, refer to the *Nios II Hardware Development Tutorial* available on the Altera Nios II literature page at **http://www.altera.com/literature/lit-nio2.jsp**, and to the *Nios II Software Development Tutorial* available in the Nios II integrated development environment (IDE) help system.

C2H Compiler Design Flow

This section discusses the design flow to create a hardware accelerator with the C2H Compiler.

Starting Point for the C2H Compiler Design Flow

The design flow for the C2H Compiler starts with one or more C files that compile successfully targeting the Nios II processor. Before you accelerate a function with the C2H Compiler, you must:

- Identify the functions that require acceleration.
- Debug the functions first targeting the Nios II processor. After accelerating a function, you can no longer debug individual C statements within the function.

You might have existing C code that you need to accelerate to improve performance. Alternatively, you might develop and debug a function in C with the explicit purpose of converting it to hardware. In either case, you achieve the best results if the C code is structured for the C2H Compiler. To start with, you can accelerate your code as-is, and determine if the results meet the design requirements.

Typical Design Flow

A typical design flow using the C2H Compiler to accelerate a function involves the following steps:

1. Develop and debug your application or algorithm in C targeting a Nios II processor system.

2. Profile the code to identify the areas that would benefit from hardware acceleration.

3. Isolate the code you want to accelerate into an individual C function.

4. Specify the function you want to accelerate in the Nios II IDE.

5. Rebuild the project in the Nios II IDE.

6. Profile the results in hardware, or observe estimates from the C2H report in the Nios II IDE.

7. If the results do not meet the design requirements, modify the C source code and system architecture (for example, the memory topology).

8. Return to Step 5, and iterate.

The typical C2H Compiler design flow is an iterative process of accelerating a function, comparing the performance to design requirements, and modifying C code to improve results. If you start with C code that is not optimized for the C2H Compiler, the first iteration of acceleration might not dramatically improve performance. Further iterations, modifying the C code for optimal hardware structure, often improve the final results significantly over the first pass results.

This tutorial does not describe techniques for optimizing hardware accelerator performance. For further information on optimizing C2H Compiler results, refer to the *Accelerating Nios II Systems with the C2H Compiler Tutorial*.

Software Requirements

The C2H Compiler in evaluation mode is installed as part of the Altera® Quartus® II Complete Design Suite. You can download the Quartus II Complete Design Suite free from the Altera website. Visit **www.altera.com** and click **Download**.

During the design process with the C2H Compiler, you use the following tools:

■ *Nios II Integrated Development Environment (IDE)* – You control acceleration options for individual functions in the Nios II IDE. The results of accelerating functions are reported in the Nios II IDE. The output is an executable linking file (**.elf**) targeting a Nios II CPU. The C2H Compiler also invokes SOPC Builder and optionally the Quartus II software in the background to regenerate the Nios II system and update the SRAM object file (**.sof**).

■ *SOPC Builder* – SOPC Builder manages the generation of C2H logic and Avalon-MM system interconnect fabric to connect hardware accelerators to the processor. During the software build process, the Nios II IDE can invoke SOPC Builder in the background to update the hardware accelerators when necessary and integrate them into the Nios II hardware design. The output is a set of hardware description language (HDL) files (**.v** or **.vhd**) and an SOPC Builder system file (**.sopcinfo**) defining your system: Nios II processor cores, peripherals, accelerators, on-chip memory, and interfaces to off-chip memory.

■ *Quartus II software* – The Quartus II software compiles and synthesizes HDL produced by the C2H Compiler and SOPC Builder tools, along with any other custom logic in your Quartus II project. During the software build process, the Nios II IDE can invoke the Quartus II software in the background to recompile the Quartus II project. The output is a **.sof** file that includes the updated Nios II system with accelerators.

OpenCore Plus Evaluation

Hardware accelerator blocks generated by the C2H Compiler support OpenCore®Plus evaluation. OpenCore Plus evaluation allows you to use the C2H Compiler and evaluate the performance of hardware accelerators in real systems before purchasing a license for the tool. With Altera's free OpenCore Plus evaluation feature, you can:

■ Verify the functionality of your design, as well as evaluate its size and speed easily
■ Generate time-limited device programming files for designs that include megafunctions
■ Program an FPGA and verify your design in hardware
■ Simulate the behavior of an accelerator in your system

OpenCore Plus hardware evaluation supports the tethered mode of operation for C2H. In tethered mode the accelerator runs indefinitely, as long as the target board remains connected to the host computer by an Altera download cable

You need to purchase a license for the Nios II C-to-Hardware Acceleration Compiler only when you are completely satisfied with the functionality and performance of your accelerated Nios II system, and want to take your design to production.

For more information on OpenCore Plus hardware evaluation, see *AN 320: OpenCore Plus Evaluation of Megafunctions*.

Tutorial

This section guides you through the steps to accelerate a function using the C2H Compiler. You create a new software project in the Nios II IDE using the provided example design files, accelerate a function, and observe the performance improvement.

This tutorial guides you through the steps to implement the example design. These steps start with a C source file and end with a running application that includes an accelerated function. The steps you perform are described in the following sections:

Tutorial Design

The hardware design for this tutorial is based on the **standard** hardware example design provided with the Nios II EDS. The software design is a C file named **dma_c2h_tutorial.c**, which is available for download from the Altera website. You can run the tutorial design on any Nios development board available from Altera.

You can download **dma_c2h_tutorial.c** from the Nios II literature page. The file is located next to this document (*Nios II C2H Compiler User Guide)* on the Altera Nios II literature page at **http://www.altera.com/literature/lit-nio2.jsp**.

The file **dma_c2h_tutorial.c** includes two functions:

■ do_dma() – This is the function you accelerate. It performs a block memory copy. do_dma() takes a source address pointer, a destination address pointer, and an integer number of bytes to copy. When implemented in hardware, do_dma() resembles DMA copy logic. The prototype for do_dma() is as follows:

```
int do_dma( int * __restrict__ dest_ptr,
        int * __restrict__ source_ptr, int length )
```

The __restrict__ qualifier informs the compiler that the pointers dest_ptr and source_ptr point to mutually exclusive address ranges. For further information about the __restrict__ qualifier, see "Pointer Aliasing" on page 3–32 of Chapter 3, C-to-Hardware Mapping Reference.

■ main() – main() calls do_dma() and measures the amount of time taken, so that you can compare the software implementation with the hardware accelerator.

main() performs the following actions:

1. Allocates two 1 MB buffers in main memory
2. Fills the source buffer with incrementing values
3. Fills destination buffer with all 0x0.
4. Calls the do_dma() function 100 times
5. Checks the copied data to ensure there were no errors
6. Frees the two allocated buffers

To measure the time it takes for the copy operations to complete, there are timer routines around the loop that calls the do_dma() function. After the application runs, the number of milliseconds that were spent performing the copy operations is displayed to the Console view in the Nios II IDE.

Set up the Hardware for the Project

To set up the hardware for the tutorial, perform the following steps:

1. Connect your Nios development board to power, and connect the board to your host computer with an Altera download cable.

2. Set up the hardware project directory

   a. Using a file management tool on the host computer, locate the **standard** hardware example design for your Nios development board. For example, on a Windows PC, use Windows Explorer to find the Verilog HDL design files for the Nios development board, Cyclone® II Edition at *<Nios II EDS install path>/* **examples/verilog/ niosII_cycloneII_2c35/standard**.

   b. Copy the **standard** directory and name the copied directory **c2h_tutorial_hw**. This new directory serves as the hardware design for the tutorial.

3. Start the Quartus II software.

4. Open the Quartus II project **standard.qpf** located in the **c2h_tutorial_hw** directory.

   The Quartus II software might give a warning "Do you want to overwrite the database ... created by Quartus II Version *<version>*... The database format is compatible..." if the project was created with an earlier version of the software. If so, click **Yes** to update the database.

5. Configure the FPGA on the Nios development board.

   a. On the Tools menu click **Programmer**. The Programmer appears, with the SRAM object file **standard.sof** automatically ready to download to the FPGA.

   b. Turn on the **Program/Configure** check box for **standard.sof**.

   c. Click **Start**. The programmer downloads the configuration data to the FPGA.

   ☞ If **Start** is not enabled, click **Hardware Setup** to configure your JTAG download cable.

Create the Software Project

To set up the software project for the tutorial, perform the following steps.

1. Start the Nios II IDE.

2. If the **Workspace Launcher** dialog box appears, click **OK** to accept the default workspace.

3. If the **Welcome to the Altera Nios II IDE** page displays, close it to view the workbench.

4. Create a new C/C++ Application project.

   a. On the File menu, point to **New** and click **C/C++ Application**. The **New Project** wizard appears.

   b. In the **Name** box, type c2h_tutorial_sw.

   c. In the **Select Project Template** list, select **Blank Project**.

   d. Use the **Select Target Hardware** settings to browse to and select the SOPC Builder system (**.ptf**) file in your **c2h_tutorial_hw** directory. After you specify the SOPC Builder system, the IDE automatically sets the **CPU** setting to **cpu**, which is the name of the only Nios II processor core available in this SOPC Builder system.

   e. Click **Finish**. The IDE generates a new project **c2h_tutorial_sw** and a new system library project **c2h_tutorial_sw_syslib**.

5. Download the software file **dma_c2h_tutorial.c** from the Nios II literature page and save it to a known location on your host computer. The file is located next to this document (*Nios II C2H Compiler User Guide)* on the Altera Nios II literature page at **http://www.altera.com/literature/lit-nio2.jsp**.

6. Import the C file **dma_c2h_tutorial.c** into the **c2h_tutorial_sw** project. The easiest way to do this is to use an external file management tool, such as Windows Explorer, and drag the file onto the **c2h_tutorial_sw** project folder in the C/C++ Projects view of the Nios II IDE.

Run the Project as Software Only

In this section, you build and run the project as a software-only implementation, and observe the time required to run the program. To run the program, perform the following steps:

1. In the C/C++ Projects view, right-click the **c2h_tutorial_sw** project, point to **Run As** and click **Nios II Hardware**. The Nios II IDE takes a few minutes to build and run the program.

2. Observe the execution time in the Console view. Example 2–1 shows results of approximately 86000 milliseconds. The results you see might be different, depending on the memory characteristics of the target board and the clock speed of the example design.

Example 2–1. Execution Results as Software-Only Implementation

```
This simple program copies 1048576 bytes of data from a source buffer to a
destination buffer.
The program performs 100 iterations of the copy operation, and calculates
the time spent.

Copy beginning
SUCCESS: Source and destination data match. Copy verified.
Total time: 86520 ms
```

Create and Configure a Hardware Accelerator

In this section, you create an accelerator for the do_dma() function. To create the hardware accelerator, perform the following steps:

1. Open the **dma_c2h_tutorial.c** source file in the Nios II IDE editor, if it is not already open.

2. In the source file, double-click the name of the do_dma() function to select it.

3. Right-click **do_dma** and click **Accelerate with the Nios II C2H Compiler**. The C2H view appears in the bottom pane of Nios II IDE.

☞ In this example, for simplicity, the do_dma() function exists in the same file as the rest of the application code. However, a good practice is to isolate functions for acceleration into a separate C file. The project makefile cannot determine specifically what part of a file has changed. As a result, if an accelerated function coexists in the same file with other unaccelerated code, the C2H Compiler is forced to rebuild the accelerator, even if you edit unrelated code.

4. Set the build options for the new accelerator, as shown in Figure 2–1.

   a. Click the **+** icon to expand **c2h_tutorial_sw** in the C2H view.

b. Turn on **Build software, generate SOPC Builder system, and run Quartus II compilation**. When you build the project in the Nios II IDE, this option causes the C2H Compiler to invoke SOPC Builder and the Quartus II software in the background to generate a new **.sof** file.

☞ Quartus II compilations can take a long time. You only need to turn on this option when you want to update the **.sof** file. You must regenerate the **.sof** file after you make changes that affect one or more hardware accelerators, and you want to run a program on the hardware system.

c. Expand **do_dma()** in the C2H view.

d. Under **do_dma()**, select **Use hardware accelerator in place of software implementation. Flush data cache before each call**. At run time, this option causes the program to activate the accelerator hardware for do_dma(). With this option, the C2H wrapper function flushes the processor data cache before activating the accelerator.

☞ The wrapper function needs to flush the data cache before activating the hardware accelerator if the processor has a data cache and if the processor writes to the same memory that the accelerated function operates on. Failing to flush the cache might result in cache coherency problems.

Figure 2–1. Setting the Build Options for the Accelerator
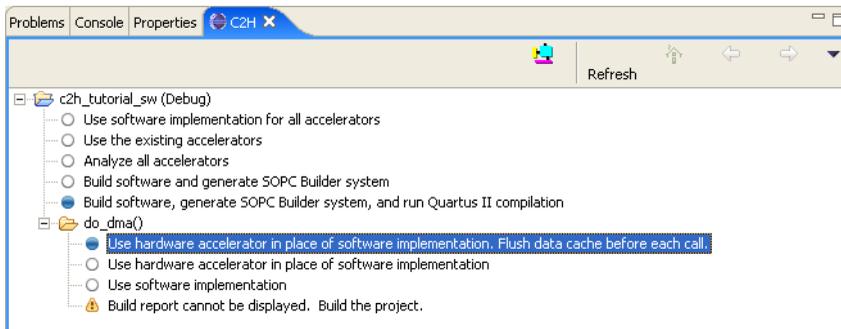
Rebuild the Project

To rebuild the project, perform the following step:

✓ In the C/C++ Projects view, right-click **c2h_tutorial_sw** and click
Build Project.

☞ The rebuild process can take over 20 minutes, depending on
your computer's performance and the target FPGA.

In the background, the Nios II IDE performs the following tasks:

1. Launches the C2H Compiler to analyze the do_dma() function,
   generates the hardware accelerator, and generates the C wrapper
   function.

2. Invokes SOPC Builder to connect the accelerator into the SOPC
   Builder system. The build process modifies the SOPC Builder
   system file (**.ptf**) in the Quartus II project directory to include the
   new accelerator as a component in the system.

3. Invokes the Quartus II software to compile the hardware project and
   regenerate the **.sof** file.

4. Rebuilds the C/C++ application project and links the accelerator
   wrapper function into the application.

Progress messages display in the Console view. The build process creates
the following files:

■ **accelerator_c2h_tutorial_sw_do_dma.v** (or **.vhd**) – This file is the
   HDL code for the accelerated function. It is stored in the Quartus II
   project directory, and the name follows the format **accelerator_<***IDE
   project name***>_<***function name***>**. This file is not visible in the Nios II
   IDE.
■ **alt_c2h_do_dma.c** – This file is the C2H accelerator driver file,
   containing the wrapper function for the accelerator. It is located in
   software project's **Debug** or **Release** directory, and the name follows
   the convention **alt_c2h_<***function name***>.c**. (If you use the Nios II
   software build tools, these files are located in your software
   application directory.)
■ **c2h_accelerator_base_addresses.h** — This is the C2H accelerator
   base addresses header file. It is located in the same directory as
   alt_c2h_<*function name***>.c**.

☞ If you copy or move a C2H project to a different directory, you must make sure you have the generated C source files and C2H makefile fragments in the new location. If you regenerate your accelerator in the new location, the C2H Compiler recreates these files for you. This is the simplest way, although not the fastest, to ensure that you have these files.

If you want to avoid regenerating, simply copy or move the two files when you copy or move your original C source files. Copy or move the files, **alt_c2h_**<*function_name*>**.c** and **c2h_accelerator_base_addresses.h,** to the subdirectory in the new project location corresponding to their original location.

Observe Results in the Report File

The C2H Compiler produces a detailed build report in the C2H view. The build report contains information about accelerator performance and resource utilization, which you can use to optimize your C code for the C2H Compiler. This section introduces the main features of the report file.

Inspect the report by performing the following steps:

1. Click the C2H view in the Nios II IDE. You can double click the **C2H** tab to view the report in full-screen mode.

2. In the C2H view, expand **c2h_tutorial_sw**, **do_dma()**, **Build report**.

☞ For designs with multiple accelerators, a build report appears under each function listed in the C2H view.

3. Expand the **Glossary** section. This section defines the terminology used in the rest of the report.

4. Expand the **Resources** section and all subsections, as shown in Figure 2–2.

Figure 2–2. Resource Section of the C2H Build Report
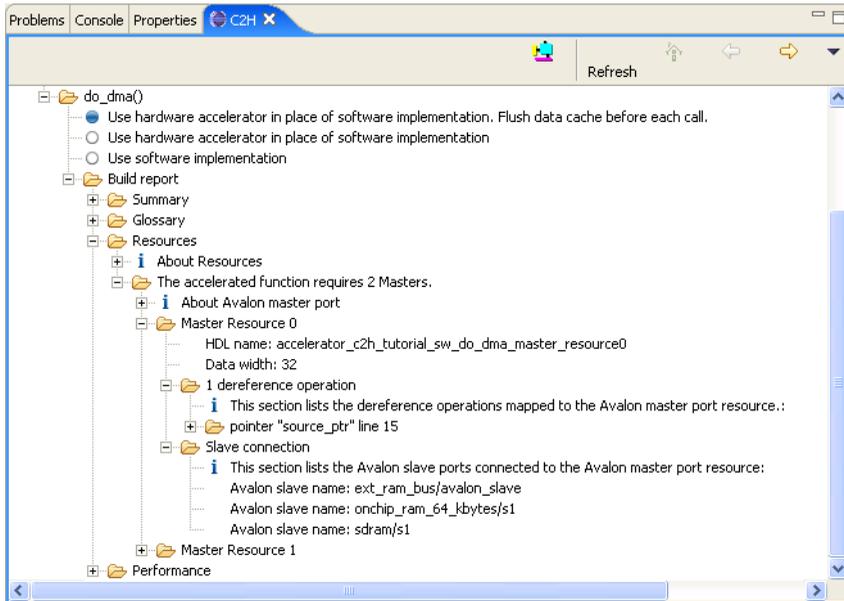
The **Resources** section lists all the master ports on the hardware accelerator. Each master port corresponds to a pointer dereference in the source code. In this example, there are two master ports: one for dereferencing the read pointer, *source_ptr, and one for dereferencing the write pointer, *dest_ptr.

5. Expand the **Performance** section and all subsections, as shown in Figure 2–3.

Figure 2–3. Performance Section of the C2H Build Report
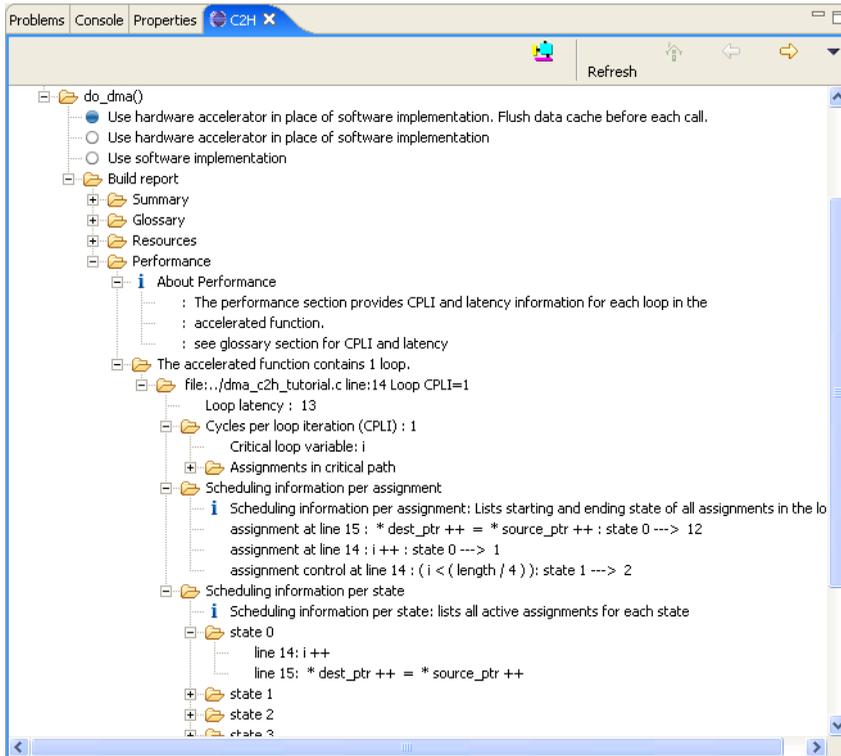
The **Performance** section shows the performance characteristics of each loop in the accelerated function. There are two metrics that determine a loop's performance: loop latency and cycles per loop-iteration (CPLI). Loop latency is the number of cycles required to fill the pipeline. CPLI is the number of cycles required to complete one iteration of the loop, assuming the pipeline is filled and no stalls occur. For example, consider the case of an accelerated function with one loop with loop latency of 13 and CPLI value of 1. (These values can differ, depending on the memory latency on your target board.) These numbers indicate that the pipeline takes 13 cycles to fill; once the pipeline is filled, the pipeline generates a new result every cycle.

☞ In general, the goal of optimizing an application for better accelerator performance is to reduce loop latency and CPLI.

For further information on optimizing C2H Compiler results, refer to the *Accelerating Nios II Systems with the C2H Compiler Tutorial*.

Observe the Accelerator in SOPC Builder

After the C2H Compiler adds the hardware accelerator to your SOPC Builder system, the accelerator appears in SOPC Builder.

To look at the newly-added accelerator in your SOPC Builder system, perform the following steps:

1. Return to the Quartus II window.

2. On the Tools menu click **SOPC Builder...** to open SOPC Builder.

3. On the **System Contents** tab in SOPC Builder, notice the new component **accelerator_c2h_tutorial_sw_do_dma**, located at the bottom of the table of active components.

4. Close SOPC Builder.

☞ You cannot modify the accelerator from within SOPC Builder. You must use the Nios II IDE interface to remove or alter it.

⚠️ CAUTION Close SOPC Builder while building projects in the Nios II IDE with the C2H Compiler. The C2H Compiler modifies the SOPC Builder system in the background. If SOPC Builder is open while you build a Nios II IDE project with C2H accelerators, the system displayed in the SOPC Builder window can become out-of-date.

If you inadvertently leave SOPC Builder open while building an accelerator with the C2H Compiler, be sure to close the SOPC Builder system file without saving it. If you save the out-of-date file, you overwrite your accelerator-enhanced system file.

Run the Project with the Accelerator

You are now ready to run the accelerated project. Perform the following steps:

1. Return to the Quartus II window, if it is not already open.

2. Configure the FPGA with the new **.sof** file that contains the accelerator.

     a.   If the Programmer window is not still open, on the Tools menu click **Programmer**. The Programmer window lists the file **standard.sof**.

     b.   Turn on the **Program/Configure** box for **standard.sof**.

☞   If you don't have a license for the C2H Compiler, the Quartus II software generates a time-limited **.sof** file with a different name. In this case, select **standard.sof** and click **Delete**, then click **Add** and open the time-limited **.sof** file.

     c.   Click **Start**. The programmer downloads the new configuration data to the FPGA.

3.   Return to the Nios II IDE window.

4.   In the C/C++ Projects view, right-click the **c2h_tutorial_sw** project, point to **Run As** and click **Nios II Hardware**. The Nios II IDE downloads the accelerated program to the board and runs it.

5.   Observe the execution time in the Console view. Example 2–2 shows timing results of approximately 8470 milliseconds. The results you see might be different, depending on your target board.

Example 2–2. Execution Results with Hardware Acceleration

```
This simple program copies 1048576 bytes of data from a source buffer to a
destination buffer.
The program performs 100 iterations of the copy operation, and calculates
the time spent.

Copy beginning
SUCCESS: Source and destination data match. Copy verified.
Total time: 8470 ms
```

Remove the Accelerator

You can remove an accelerator from a design by performing the following steps in the Nios II IDE.

1.   Right-click the function name in the C2H view and click **Remove C2H Accelerator**, as shown in Figure 2–4.

2. Rebuild the project in the Nios II IDE.

☞ You must rebuild the project to remove the hardware accelerator from the SOPC Builder system hardware.

Figure 2–4. Removing a C2H Accelerator
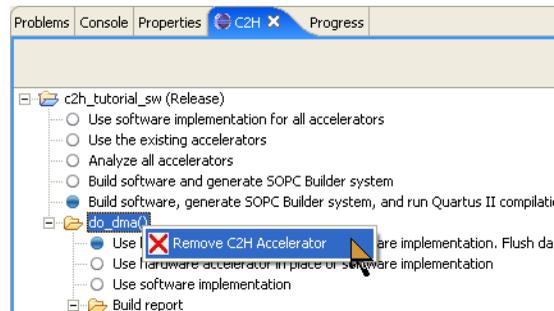
Removing the accelerator removes the hardware accelerator component from the SOPC Builder project, and replaces the C2H software wrapper with the original, unaccelerated function. The next time you build the project in the Nios II IDE, the C2H Compiler regenerates the SOPC Builder system and recompiles the Quartus II project to generate a **.sof** file without the accelerator hardware.

☞ To remove an accelerator from the hardware system, you must use the **Remove C2H Accelerator** command in the Nios II IDE. Do not use SOPC Builder to manually delete the component from the system. If you delete the component from the SOPC Builder system using the SOPC Builder GUI, the C2H Compiler produces undefined results the next time you build the software project.

Next Steps

Congratulations! You have successfully converted an ANSI C function to a hardware accelerator using the C2H Compiler and observed a significant performance increase.

After accelerating a function and running it for the first time, your next steps vary depending on your system requirements. If your starting goal is to off-load a routine from the processor to reduce CPU load, you might find that no additional action is required. If the hardware accelerator does not meet performance or resource requirements, you can perform one or

more iterations of optimization to produce better results. In either case, you can continue developing your system software and hardware, and the accelerator remains in place.

☞ It is common to be able to improve first-pass performance results significantly by optimizing the C code and system architecture.

If you modify the accelerated C code, the Nios II IDE automatically regenerates the accelerator hardware with the C2H Compiler the next time you build the C/C++ application project. Alternatively, you can disable an accelerator after it is built and relink the original software implementation, while leaving the hardware accelerator inactive in the hardware.

To get a better understanding of how the C2H Compiler translates C to hardware, read Chapter 3, C-to-Hardware Mapping Reference. After that, for further information on optimizing C2H Compiler results, refer to the *Accelerating Nios II Systems with the C2H Compiler Tutorial*.

3. C-to-Hardware Mapping Reference

This chapter describes how the Nios® II C-to-Hardware Acceleration (C2H) Compiler translates ANSI C constructs into functional blocks in a hardware accelerator. Understanding the C-to-hardware mappings enables you to write C functions optimized for the C2H Compiler to achieve higher performance and lower resource utilization.

One-to-One C-to-Hardware Mapping

The C2H Compiler translates each element of C syntax to an equivalent hardware structure using straightforward mapping rules. The mapping rules provide a one-to-one association between elements of C syntax and their equivalent hardware structures. By learning the C-to-hardware mappings, you can control the hardware structure of an accelerator, based on the structure of the C code.

The C2H Compiler can perform resource-sharing optimizations which reduce the resource utilization for an accelerator. In these cases, the result is a better than one-to-one mapping.

Arithmetic and Logical Operators

Every arithmetic and logical operator in the C code translates to a corresponding hardware block in the accelerator. Consider the function MAC() shown in Example 3–1.

Example 3–1. Function with Arithmetic and Logical Operators

```
long long MAC (int* a, int* b, int len)
{
  long long result = 0;
  while (len > 0)
  {
    result += *a++ * *b++;
    len--;
  }
  return result;
}
```

Table 3–1 lists the equivalent hardware structures resulting from this function.

Table 3–1. Hardware Structure for Arithmetic and Logical Operators

Line	C Element	Hardware Structure
`while (len > 0) {`	`while`	Finite state machine with nominal control logic. Refer to section "Iteration Statements" on page 3–5.
	`>`	32-bit comparator
`result += *a++ * *b++;`	`+=`	64-bit adder
	* (pointer)	Avalon-MM master port to read data (two total for `*a++` and `*b++`). Refer to section "Indirection Operator (Pointer Dereference)" on page 3–16.
	`++`	32-bit up-counter (two total for `*a++` and `*b++`)
	* (multiply)	32x32=64-bit multiplier
`len--;`	`--`	32-bit down-counter

Assignments

A C assignment operator stores the value of an expression to a variable. As a general rule, every assignment operator in the C code, such as =, translates to a registered signal in hardware. The value of an assignment's expression is calculated in one clock cycle. Figure 3–1 shows the hardware that results from the following statement:

```
int sum = x + y;
```

Figure 3–1. Hardware Resulting from Assignment
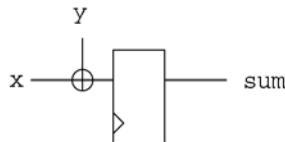
There are two types of exceptions to this rule:

■ Assignments that require zero logic elements in hardware
■ Assignments that use multiple registers to pipeline complex arithmetic operations

The following sections discuss these exceptions.

Unregistered Operations and Assignments

Certain logical and bit-wise operations involving constants are trivial and require no logic. In hardware, they are performed simply by manipulating wires. Table 3–2 lists the applicable operators and conditions. If an assignment consists solely of such operations, then its result is not registered.

Table 3–2. Operators That Can Result in Unregistered Assignments		
Operator	Description	Required Condition
>>	Right bit-wise shift	Right-hand side is constant
<<	Left bit-wise shift	Right-hand side is constant
&	bit-wise AND	Either operand is constant
|	bit-wise inclusive OR	Either operand is constant
^	bit-wise exclusive OR	Either operand is constant
~	bit-wise inversion	Right-hand side is unregistered
)	Type cast	Right-hand side is unregistered

The following assignment is an example of a zero logic-element operation.

```
int masked_data = data_in & 0x000fffff;
```

The C2H Compiler generates no register for the variable masked_data, because its value is represented simply by concatenating 12 bits of zeroes with the lower 20 bits of data_in.

Additional examples of unregistered assignments:

```
shift_by_constant     = data_in << 3;

or_with_constant      = data_in | 0xf0f0f0f0;

invert_shift_and_consts = (~data_in & 0xff) << 8;
```

Pipelined Operations and Assignments

The C2H Compiler always registers the results of the operators listed in Table 3–3. Some arithmetic operations, such as multiplication, use a large amount of logic, which creates a significant propagation delay through the circuit. Calculating these operations in series with other operations in a single clock cycle would incur unacceptable propagation delays and significantly reduce the maximum achievable clock frequency (f_{MAX}) for the system. The C2H Compiler pipelines these operations by giving each its own registered assignment. There are exceptions for cases in which an operation reduces to trivial logic, as listed in Table 3–3

Table 3–3. Complex Arithmetic Operations Pipelined by the C2H Compiler		
Operator	**Description**	**Exceptions**
*	Multiplication	Either operand is a constant power of 2, which reduces to left-shift operation
/	Division	Right-hand operand is a constant power of 2, which reduces to a right-shift operation
%	Modulus	Right-hand operand is a constant power of 2, which reduces to a masking operation
>>	Right bit-wise shift	Right-hand side is constant
<<	Left bit-wise shift	Right-hand side is constant

The general rule "one registered assignment for every = operator" can be amended to read, "one registered assignment for every = operator or complex arithmetic operator".

Figure 3–1 shows the hardware that results from the following statement:

```
int foo = a * b + x;
```

Figure 3–2. Pipelined Multiplication Operator
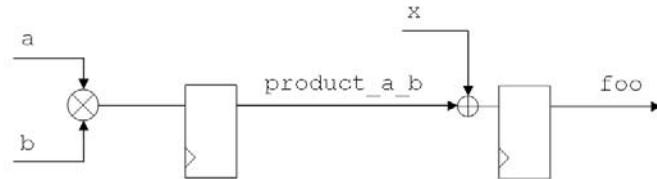
Figure 3–2 shows the hardware that results from the following statement:

```
int bar = a * b * c + x;
```

Figure 3–3. Two Stages of Pipelined Multiplication Operators
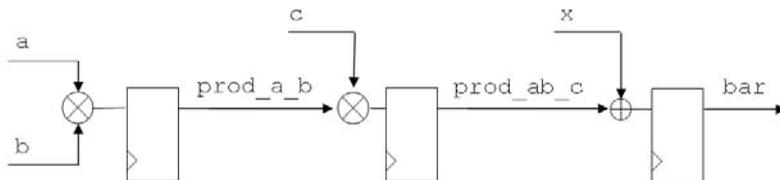
Iteration Statements

An iteration statement (`do`, `for`, or `while`), also known as a loop statement, translates to a finite state machine in hardware. The state machine controls execution of all the statements inside the loop block. A loop inhibits (stalls) its parent state machine. In other words, if a loop exists within an outer loop, the state machine for the outer loop stalls each iteration and waits for the inner loop state machine to complete.

The fundamental iteration statement for the C2H Compiler is the do loop, which evaluates its condition at the end of each iteration. See section "Loop Pipelining" on page 3–42 for information about loop state machines and scheduling.

Selection Statements

A selection statement (if-else, case, switch, and ?:) translates to a multiplexer in hardware. The structure of the hardware depends on the type of statement, as described in the following sections.

if Statement

An if-else statement translates to three elements in hardware:

■ Logic to perform all operations in the then block
■ Logic to perform all operations in the else block
■ Selection logic that determines which result to use

The results of each element are registered, and the registered signals feed a multiplexer.

If the if statement has both a then and an else block, the operations for both blocks execute in parallel. When all operations have completed, the multiplexer selects which value propagates to subsequent statements, based on the value of the control expression.

Figure 3–4 shows the circuit that results from the code in Example 3–2.

Example 3–2. If-else Logic

```
if (foo > bar)
   foo += bar;
else
   foo *= bar;
```

Figure 3–4. if-else Logic
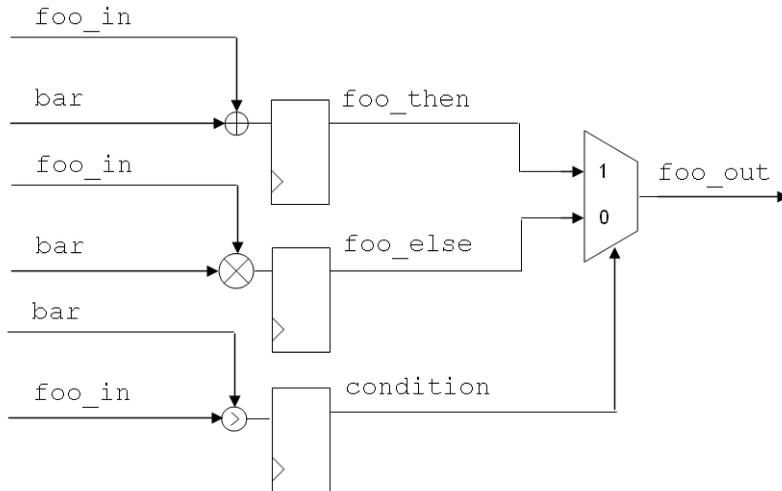
If the `if` statement has only a `then` or only an `else` block, the resulting logic is a simplification of the `if-else` case. The multiplexer selects whether to propagate the result of the `if` block or the initial values from before the if statement.

Figure 3–5 shows the circuit that results from the code in Example 3–3.

Example 3–3. if Logic Without else

```
if (foo > bar)
   foo += bar;
```

Figure 3–5. if Logic Without else
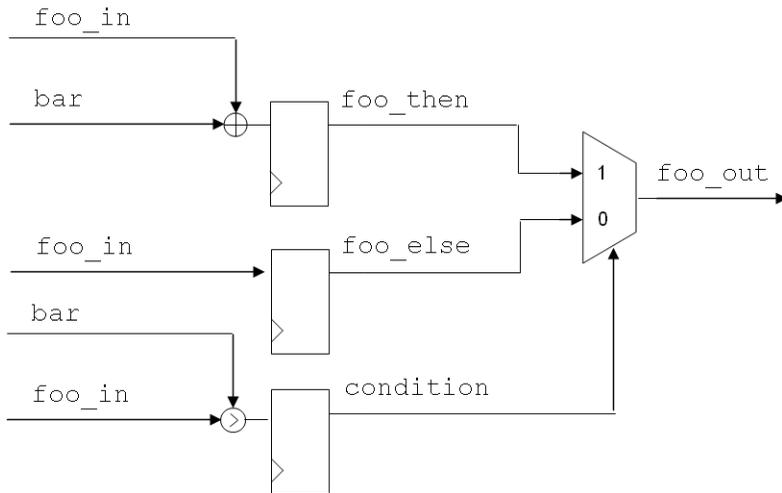
Conditional Operator ?:

The `?:` (conditional) operator is functionally equivalent to the `if-else` statement, but the placement of registers is different. The condition logic and selection logic compute in the same clock cycle, and the result is registered.

Figure 3–6 shows the circuit that results from the following code:

```
foo = (foo > bar) ? (foo + bar) : (foo * bar);
```

Figure 3–6. if-else Logic
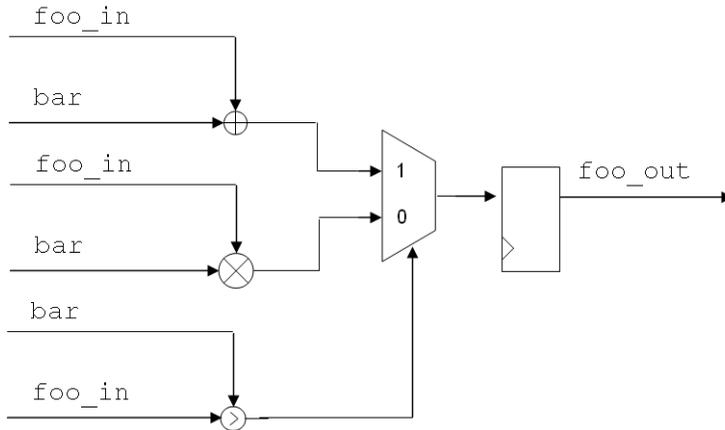
switch and case Statements

The C2H Compiler converts `switch` statements to functionally-equivalent nested if-else statements, and then translates the `if-else` statements to hardware, as described in section "if Statement" on page 3–6.

Table 3–4 shows an example of a switch statement converted to equivalent if-else statements.

Table 3–4. switch Statement Converted to if-else Statements	
switch Implementation	**if-else Implementation**

<pre>switch (byte_select)
{
  case 1:
    out = in & 0x0000ff00;
    break;
  case 2:
    out = in & 0x00ff0000;
    break;
  case 3:
    out = in & 0xff000000;
    break;
  default:
    out = in & 0x000000ff;
}</pre>
	<pre>if (byte_select == 1)
	  out = in & 0x0000ff00;
	else
	  if (byte_select == 2)
	    out = in & 0x00ff0000;
	  else
	    if (byte_select == 3)
	      out = in & 0xff000000;
	    else
	      out = in & 0x000000ff;</pre>

Figure 3–7 shows the logic that results of translating the if-else code from Table 3–4.

Figure 3–7. switch Logic
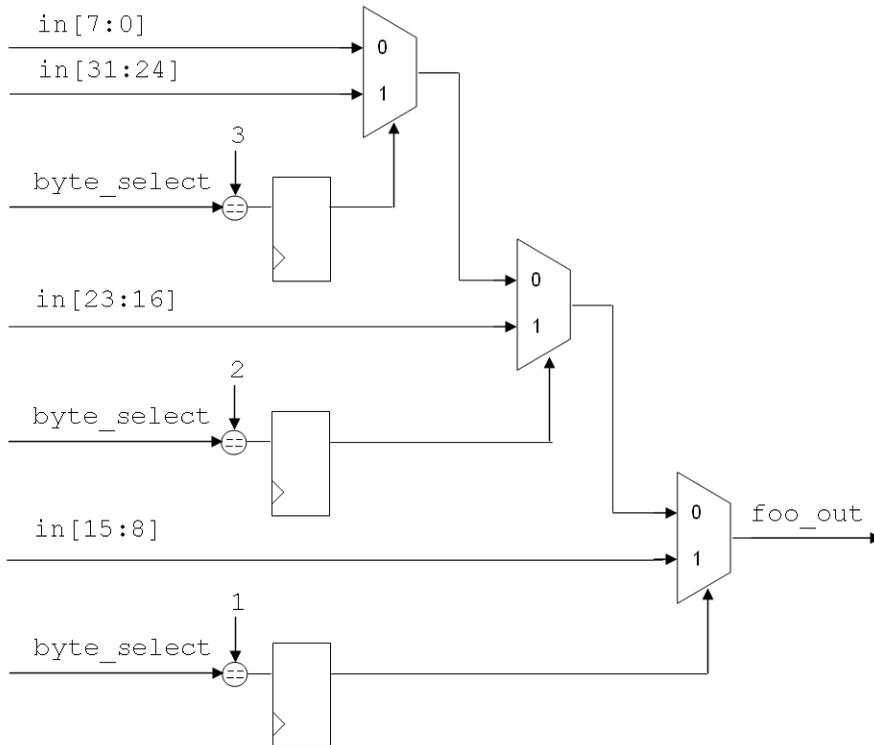
Subfunction Calls

A subfunction is a C function called from within an accelerated function. The C2H Compiler translates subfunctions to hardware using the same mapping rules as for the top-level function. The resulting HDL module for the accelerated subfunction becomes a submodule of the top-level function, as illustrated in Figure 3–8 on page 3–12.

The C2H Compiler translates the top-level function and all subfunctions to a single hardware accelerator. The C2H Compiler creates only one instance of the subfunction hardware logic, regardless of how many times the subfunction is called within the top-level function. If the calling function calls the subfunction multiple times, the subfunction logic becomes a shared resource. However, the subfunction is a private resource exclusive to the calling function. In other words, if multiple separate, accelerated functions call a common subfunction, the C2H Compiler creates separate instances of the subfunction logic.

Table 3–5 shows an example of a subfunction called by two different functions. In this case, functions `foo()` and `bar()` both call a subfunction `foobar_sub()`.

Table 3–5. Shared Subfunction foobar_sub()

Top-Level Accelerated Function: foo()	Top-Level Accelerated Function: bar()

<pre>void foo(int *data_in,
   int *data_out)
{
 	
   ...
   foobar_sub();
   ...
}</pre>
	<pre>void bar(int *data_in,
	   int *data_out)
	{
	
	   ...
	   foobar_sub();
	   ...
	   foobar_sub();
	   ...
	}</pre>

Figure 3–8 shows the hardware structure of the accelerators resulting from the code in Table 3–5. Logic for the function `foobar_sub()` exists within both accelerators.

Figure 3–8. Shared Subfunction Logic

The C2H Compiler does not support external subfunctions. You must locate the subfunction in the same source file as the accelerated function. This is because, unlike the `#include` construct, a C external function reference requires the presence of a linker. The C2H Compiler has no linker.

The Nios II C2H Compiler does not perform any type of inline substitution. It ignores the inline function specifier. You can achieve the effect of an inline function through the use of preprocessing macros. If you wish to call a function for which accelerated hardware is replicated for each call, then you must define a macro containing the logic for this function. Before parsing the accelerated code, the C2H Compiler calls the GNU GCC preprocessor, which evaluates the macro and replaces each macro call with your macro definition.

Macros and Preprocessing Directives

Any preprocessing directives in your code are processed before translation to hardware. In order to ensure identical interpretation between software and hardware, the C2H Compiler operates on the output of the **nios2-elf-gcc** preprocessor.

Variable Declarations

This section describes how the C2H Compiler translates variable declarations and other C elements that define data storage.

Local vs. Non-Local Variables

The C2H Compiler treats variables differently, depending on the scope of the variable. In general, local variables translate to memory elements inside the accelerator hardware; non-local variables translate to Avalon-MM master ports capable of accessing memory outside of the accelerator.

For the purposes of this document, any variable declared within the scope of an accelerated function is considered to be local. Example 3–4 illustrates locality of variables to a function mac(). In this example, only int my_global is not local, and all other variables are local.

Example 3–4. Local vs. Non-Local Variables

```
int my_global;  // my_global is not local to mac().
int mac(int *src, int *dst, int len)
       // src, dst, and len are local to mac().
{
  int res = 0;  // res is local to mac().
  while (len--)
  {
    int product = (*src++) * (*dst++);   // product is local to mac().
    res += product;
  }
  return res;
}
```

Scalar Variables

For local scalar variables, the C2H Compiler creates hardware registers inside the accelerator. For example, declaring a `char` creates an 8-bit register; declaring a `short int` creates a 16-bit register, and so forth. Declaring a pointer allocates only storage for the pointer itself. For example, declaring a `char*` creates a 32-bit register to store the address of a `char`. If you use a scalar variable within a pipelined loop, then its register is replicated for pipelining as needed.

The C2H Compiler considers a variable to be scalar if it is not an array, structure, or union. Example 3–5 demonstrates some examples of scalar variables.

Example 3–5. Scalar Variables
```
int i;
int *p;
char **c;
struct struct_type *pointer_to_struct;
int (*pointer_to_array)[8];
```

Example 3–6 demonstrates some examples of nonscalar variables.

Example 3–6. Nonscalar Variables
```
int data[1024];
struct struct_type tx_rec;
int *array_of_pointers[8];
```

Arrays, Structures, and Unions

For local arrays, structures, and unions, the C2H Compiler creates memory elements inside the accelerator. Depending on the size of the memory required and the target device family, the C2H Compiler can implement these memory elements either as logic elements or embedded memory blocks.

The memory elements are single-ported. As a result, a given array, union, or structure has a maximum bandwidth of one transaction per cycle, even if the C code could be structured to allow parallel scheduling of accesses.

☞ The following constructs are declared in the driver file (and therefore in the processor's data memory), and passed into the accelerator by reference:

- Arrays that are initialized
- Global variables
- Static variables

Avalon-MM master ports are generated even when local arrays (such as the ones discussed here) are referenced. These master ports only connect to internal slave ports inside the accelerator. However, the master ports do appear in the C2H Compiler build report.

Example 3–7 demonstrates the creation of a local 4 kbyte memory inside the accelerator to store array `data[1024]`. Because this memory buffer is large, it translates to an embedded memory block.

Example 3–7. Local Array That Uses 4 KBytes of On-Chip Memory Inside Accelerator

```
int my_func(int a_parameter)
{
  int data[1024]; // 1K 4-byte ints
  ... // Body of the function
  return data[0];
}
```

Global and Static Variables

Global and static variables must persist outside the scope of the accelerated function, and they have real addresses accessible by the processor. For this reason, global and static variables are stored in memory that the processor can access, outside of the accelerator hardware. In other words, the C2H Compiler does not affect the location of these variables; it creates logic in the accelerator capable of accessing the memory where the variables reside.

References to global and static variables translate to master ports in hardware, which enable access to a given variable's specific memory location. For further details, refer to section "Memory Accesses" on page 3–15.

Memory Accesses

Hardware accelerators generated by the C2H Compiler use Avalon-MM master ports to access memory, similar to the Nios II processor and other SOPC Builder components. The Altera® SOPC Builder system integration tool handles the task of physically connecting both accelerators and processors to memory, and creating arbitration logic. As a result, the

behavior of a C function accessing memory is the same, regardless of whether the function is implemented as hardware logic or software instructions.

For more information on SOPC Builder, Avalon interfaces, and how SOPC Builder generates system interconnect fabric, refer to the *Quartus II Handbook, volume 4: SOPC Builder* and the *Avalon Memory-Mapped Interface Specification*.

In order to maximize bandwidth, the C2H Compiler creates a master port on the accelerator for every C operator that accesses external memory. Multiple master ports allow the accelerator to read and write data to an unlimited number of locations simultaneously, thereby reducing the bandwidth limitations inherent in a CPU with a single data master port.

In some cases, the C2H Compiler can determine that master ports can be shared between several external memory operations without sacrificing performance. However, as a general rule, an Avalon-MM master port is created for each of the following:

■ Pointer dereference (* operator)
■ Index into an array ([operator)
■ Index into a struct or union (. or -> operator)
■ Usage of a global or static variable

Example 3–8 demonstrates various lines of code that generate a master port in hardware.

Example 3–8. C Statements that Generate Avalon-MM master ports

```
*my_ptr = 8;
data_in = *src;
dst[index] = data_out;
pixel = pixel_array[i][j];
buffer.input = 0x80000400;
current = s->next;
```

The following sections describe each case in detail.

Indirection Operator (Pointer Dereference)

The indirection operator (*) is the fundamental expression of dereferencing and indirection. This section describes how the C2H Compiler handles pointer dereferencing.

Because the array subscript operation and the member operation for structures and unions can be expressed in terms of an address computation and a pointer dereference, this section is fundamental to understanding how arrays, structures, and unions translate to hardware as well.

Creation of Avalon-MM master ports

In general, the C2H Compiler creates a master port on the accelerator for every instance of the indirection operator. The following are exceptions to this rule:

- The C2H Compiler identifies certain opportunities for optimization. In some cases it can collapse multiple master ports to a single master port without affecting performance, which reduces resource utilization.
- If the C2H Compiler determines that two pointers are exactly equivalent, it consolidates them to a single master port.
- There are considerations for multidimensional arrays. Refer to section "Array Subscript Operator" on page 3–26.

Consolidation of Equivalent Pointers

The C2H Compiler consolidates pointer dereferences when it determines that the address expressions are always equal and the referenced data cannot change between dereferences. In this case, the pointer dereferences share an Avalon-MM master port.

Example 3–9 shows two dereferences that are identical. The C2H Compiler consolidates them into a single master port.

Example 3–9. Equivalent Pointers

```
void equivalent_pointers(char *packed_data)
{
  char ms_nibble = *packed_data >> 4;
  char ls_nibble = *packed_data & 0x0f;
  ...
}
```

Example 3–10 shows two dereferences that are identical inside of a loop. The C2H Compiler consolidates them into a single master port.

Example 3–10. Equivalent Pointers in a Loop

```
void equivalent_pointers(char *packed_data, int len)
{
  int i = 0;
  while (i < len)
  {
    char ms_nibble = *(packed_data) >> 4;
    char ls_nibble = *(packed_data++) & 0x0f;
    ...
    i++;
  }
}
```

Example 3–11 demonstrates a case of non equivalent pointers. Example 3–11 is similar to Example 3–10, but `packed_data` increments between the two pointer dereferences. In this case the address expressions have different values, which translate to two separate master ports.

Example 3–11. Nonequivalent Pointers

```
void nonequivalent_pointers(char *packed_data, int len)
{
  int i = 0;
  while (i < len)
  {
    char ms_nibble = *(packed_data++) >> 4;
    char ls_nibble = *(packed_data) & 0x0f;
    ...
    i++;
  }
}
```

Example 3–12 demonstrates another case of non equivalent pointers. Example 3–12 is similar to Example 3–10, but a value is written to address `some_other_pointer` between the reads from address `(packed_data + i)`.

Example 3–12. Nonequivalent Pointers Due to Potential Aliasing

```
void nonequivalent_pointers(char *packed_data,
                            int *some_other_pointer,
                            int len)
{
  int i = 0;
  while (i < len)
  {
    char ms_nibble = *(packed_data + i) >> 4;
    char ls_nibble;
    *some_other_pointer = i;
    ls_nibble = *(packed_data + i) & 0x0f;
    ...
  }
}
```

In this code, the C2H Compiler cannot determine if `some_other_pointer` and `packed_data` overlap addresses (known as aliasing), which would affect the result of the second evaluation of `*(packed_data + i)`. Therefore, the C2H Compiler creates a separate master port for each dereference, creating a total of three master ports. For details on how to inform the C2H Compiler that two pointers do not alias, see section "Pointer Aliasing" on page 3–32.

Volatile Type Qualifier

The C2H Compiler does not consolidate or optimize pointers for dereferenced types that use the `volatile` type qualifier. The `volatile` type qualifier forces the variable to be evaluated strictly according to the rules of the language. `volatile` is normally used to access non-memory peripherals, such as timers and communication devices.

☞   The `volatile` type qualifier overrides the `__restrict__` pointer qualifier. For further information, see "Pointer Aliasing" on page 3–32.

Example 3–13 demonstrates the use of `volatile` to guarantee multiple, distinct reads from a constant address.

Example 3–13. volatile Type Qualifier
```
volatile char *DataFIFO = FIFO_BASE;
char Byte0 = *DataFIFO;
char Byte1 = *DataFIFO;
char Byte2 = *DataFIFO;
char Byte3 = *DataFIFO;
```

By comparison, Example 3–14 demonstrates two sections of code that are equivalent, due to the consolidation of equivalent pointers. In this case, the type of `*DataFIFO` is not declared `volatile`.

Example 3–14. Equivalent Pointers
```
char *DataFIFO = FIFO_BASE;
char Byte0 = *DataFIFO;
char Byte1 = *DataFIFO;
char Byte2 = *DataFIFO;
char Byte3 = *DataFIFO;

// The code above is equivalent to the following:
char *DataFIFO = FIFO_BASE;
char dereferenced_DataFIFO = *DataFIFO;
char Byte0 = dereferenced_DataFIFO;
char Byte1 = dereferenced_DataFIFO;
char Byte2 = dereferenced_DataFIFO;
char Byte3 = dereferenced_DataFIFO;
```

Avalon-MM Master Port Signal Generation

A dereference operation, such as `*(ptr_to_int + i)`, translates to an Avalon-MM master port on the accelerator. This section describes how hardware accelerator logic generates the signals that drive the master port.

Avalon-MM master ports created by the C2H Compiler comprise the following fundamental elements:

■ Logic to compute the address signal
■ For write transfers only, logic to compute the write-data signal
■ Logic to control the read-enable or write-enable signal

Each of these signals is registered at the master port interface of the hardware accelerator. Logic within the accelerator synchronizes these signals to produce coherent Avalon-MM master transfers at the master port.

Address Computation

Consider the pointer dereference in the following code which performs a read operation:

```
int j = *(ptr_to_int + i);
```

The C2H Compiler generates logic of the following form to compute the address signal:

```
ptr_to_int_i_addr = ptr_to_int + i * sizeof(int);
```

Figure 3–9 shows an example of the logic created for this pointer dereference for a read operation.

Figure 3–9. Address Generation for a Read Operation
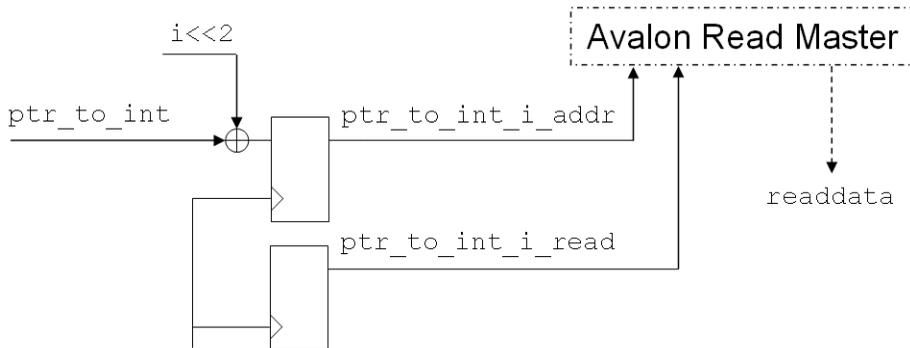
In Figure 3–9, first, the address expression is evaluated. Assuming `sizeof(int)` is 4, `i` must be multiplied by four, which is equivalent to left-shifting by 2 bits. Bitwise shift operators require no logic elements to compute, and the result is not registered. (See section "Unregistered Operations and Assignments" on page 3–3.) The signal `ptr_to_int_i_address` feeds registers that drive the address signals on the Avalon-MM master port. As soon as the address signal `ptr_to_int_i_address` is valid, read-enable control logic asserts the signal `ptr_to_int_i_read`, which initiates a transfer on the master

port. After some number of clock cycles determined by the slave memory latency and arbitration delay, valid `readdata` returns to the master port. (See section "Read Operations with Latency" on page 3–37.)

Data Computation

For write operations to dereferenced pointers, data-computation logic in the accelerator computes the value of the expression to write to memory. This value is the write-data for an Avalon-MM master transfer to memory. Data-computation logic operates in parallel with the address-computation logic.

Consider the pointer dereference in the following code which performs a write operation:

*(ptr_to_int + i) = x + y;

Figure 3–10 shows an example of the logic created for this pointer dereference for a write operation.

Figure 3–10. Data Generation for a Write Operation
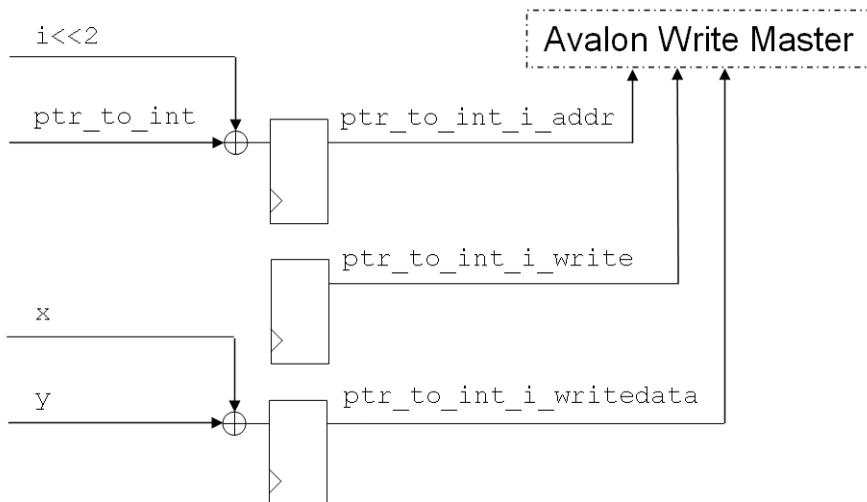
The write-data signal for the Avalon-MM master port is computed and registered in parallel with the address assignment. As soon as `ptr_to_int_i_addr` and `ptr_to_int_i_writedata` are valid, write-enable control logic asserts the signal `ptr_to_int_i_write`, which initiates a transfer on the master port.

Figure 3–11 shows the logic created for the following write operation to a dereferenced pointer. Translation of the data-computation logic follows the rules described in section "Assignments" on page 3–2.

```
*(ptr_to_int + i) = a*x + y;
```

Figure 3–11. Complex Write Operation
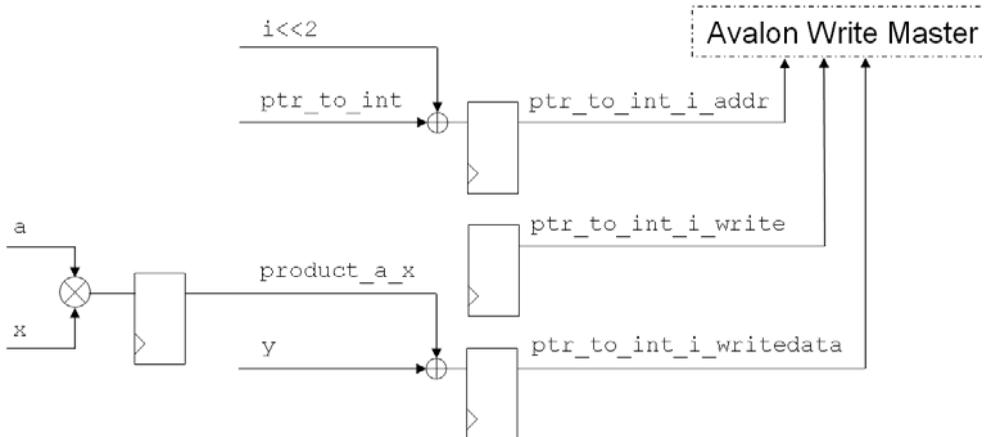
Master-Slave Connections

The C2H Compiler uses pragmas that allow user control of master-slave connections and arbitration shares. This section describes the pragmas to control master-slave connections.

The C language specification dictates that when a compiler implementation encounters a `pragma` directive it does not recognize, the compiler ignores the pragma. By using pragmas, you can write directives to optimize the C2H Compiler results, without making the C code incompatible with other compilers.

Specifying Master-Slave Connections

By default, the C2H Compiler connects all master ports of an accelerator to all the memory slave ports that the CPU data master port connects to. These default connections guarantee that the accelerator can access any memory addresses that the processor can access. However, master-slave connections have an associated hardware resource cost. The extra multiplexing and arbitration logic associated with a master-slave connection often reduces the maximum achievable frequency (f_{MAX}) of the system. If an accelerated function, by design, does not ever access certain memories, you can eliminate the connection to the slave memory to save resources and improve f_{MAX}.

The C2H Compiler provides a connection pragma that associates a pointer variable with an Avalon-MM slave port, which is typically a memory. A pointer variable can translate to one or more master ports, depending on how many times it is dereferenced in the C code. The connection pragma directs the C2H Compiler to connect all master ports generated for a particular variable to a specific slave port in the SOPC Builder system.

The connection pragma is defined as follows:

```
#pragma altera_accelerate connect_variable \
  <function name>/<variable name> to <module>[/<slave name>]
```

The connection pragma must be placed outside the function to accelerate in the same file. *<function name>* and *<variable name>* are the exact names of the accelerated function and the pointer variable. *<module>* is the exact name of the component instance, as specified in SOPC Builder. *<slave name>* is optional. If *<slave name>* is provided, it is the exact name of a specific slave port on *<module>*. If the module only contains one slave, you can omit *<slave name>*. However, if you omit *<slave name>* when the module contains multiple slaves, the compiler issues an error.

To connect a variable's master ports to multiple slave ports, you can use multiple pragmas. If you use the connection pragma for a specific variable, the C2H Compiler connects only the slave ports specified in pragma statements.

In addition to reducing arbitration logic, the connection pragma helps the C2H Compiler determine if two pointers overlap. If the memory connections for two separate variables are mutually exclusive, the compiler concludes that the pointers are never dependent on each other. For more information, refer to section "Pointer Aliasing" on page 3–32.

Example 3–15 illustrates usage of the connection pragma to connect two master ports for the variable my_ptr to the memory module named onchip_buffer.

Example 3–15. Pragma Connecting a Master Port to a Slave Port

```
#pragma altera_accelerate connect_variable foo/my_ptr to onchip_buffer

int foo(int *my_ptr)
{
  int x = *my_ptr;
  my_ptr[8] = 23;
}
```

Example 3–16 illustrates using multiple pragmas to connect a pointer variable's master ports to multiple slave ports.

Example 3–16. Pragma Connecting a Master Port to Multiple Slave Ports

```
#pragma altera_accelerate connect_variable foo/my_ptr to onchip_buffer_0
#pragma altera_accelerate connect_variable foo/my_ptr to ext_ram_bridge
#pragma altera_accelerate connect_variable foo/my_ptr to sdram
#pragma altera_accelerate connect_variable foo/my_ptr to \
                                            onchip_buffer_1/s2

int foo(int *my_ptr)
{
  int x = *my_ptr;
  my_ptr[8] = 23;
}
```

Specifying Arbitration Shares

Arbitration shares benefit memories that have higher efficiency when accessed sequentially, such as SDRAM. You can use arbitration shares to reduce interruptions to sequences of transfers with a specific slave. For example, if a master-slave connection has an arbitration share value of ten, then the arbitrator grants at least ten consecutive transfers to the master port when it begins a sequence of transfer requests.

The connection pragma with additional terms for arbitration share is defined as follows, where *<shares>* is a positive integer from 1 to 100:

```
#pragma altera_accelerate connect_variable \
  <function name>/<variable name> to
  <module>[/<slave name>] arbitration_share <shares>
```

Example 3–17 connects the variable x in function `myfunc` to the memory module named `sdram` with an arbitration share of 16.

Example 3–17. Pragma Specifying Arbitration Share
```
#pragma altera_accelerate connect_variable myfunc/x to sdram \
    arbitration_share 16
```

Specifying Flow Control
Avalon-MM transfers with flow control force a master port to obey flow control signals controlled by a slave port. For example, a slave FIFO might assert flow control signals to prevent write transfers when the FIFO memory is full. The C2H Compiler provides a flow control pragma which enables flow control for all master ports related to a specific pointer variable.

The flow control pragma is defined as follows:

```
#pragma altera_accelerate \
    enable_flow_control_for_pointer \
    <function name>/<variable name>
```

The flow control pragma must be placed in the same file as the function to be accelerated, but outside the function. *<function name>* and *<variable name>* are the exact names of the accelerated function and the pointer variable.

For details about Avalon-MM flow control, refer to the *Avalon Memory-Mapped Interface Specification*.

Array Subscript Operator

The C2H Compiler converts array subscript operations to equivalent pointer dereferences, and then translates the pointer dereferences to hardware. An array is a pointer with abstractions of length and dimension. The value of an array type is defined to be a pointer to its first element. The C language specification defines the array subscript operator as follows:

The definition of the subscript operator `[ ]` is that `E1[E2]` is identical to `(*((E1)+(E2)))`.

By this definition, any array subscripting (indexing) operation can be expressed in terms of an indirection (pointer dereference) operation.

Although an array is considered to be a pointer type, dereferencing an array variable does not always mean the same thing as dereferencing a pointer. For example, dereferencing or indexing once into a multidimensional array returns a pointer to the first element in another array. For an *N*-dimensional array, a dereference or index into any of the first *(N-1)* dimensions does not read a value from the array memory; it computes an offset from the array's base address to determine the address of a subsection of the array.

Example 3–18 demonstrates that indexing to the first level of a two-dimensional array does not result in a memory access.

Example 3–18. Indexing a Multidimensional Array without Causing a Memory Access

```
char a[LENGTH][WIDTH]; // Here's a two-dimensional array
// The following assignments are all equivalent.
char *subscripting   =          a[3];
char *dereferencing  =        *(a + 3);
char *offset         = (char *) (a + 3);
char *ptr_arithmetic = (char *) ((void *)a + 3*WIDTH);
```

Indexing into any of the first *(N-1)* dimensions of an *N*-dimensional array requires a multiplication operation, as demonstrated by the evaluation of `ptr_arithmetic` in Example 3–18. If the size of the resultant array is an integer power of two, then the multiplication operation is reduced to a constant-shift operation, which does not require a hardware multiplier. (Refer to section "Unregistered Operations and Assignments" on page 3–3.)

A series of subscript operations that index into all *N* dimensions of an *N*-dimensional array is equivalent to an indirection operation, which creates an Avalon-MM master port. Example 3–19 illustrates several cases that generate an Avalon-MM master port to dereference an array variable.

Example 3–19. Indexing an Array and Causing a Memory Access

```
char a[LENGTH][WIDTH]; /* a is a two-dimensional array */
// The following assignments are equivalent.
char *subscripting_a = a[3][2];
char *dereferencing_a = *(*(a + 3) + 2);

char b[LENGTH];        /* b is a one-dimensional array */
// The following assignments are equivalent.
char *subscripting_b = b[1];
char *dereferencing_b = *(b + 1);
```

Structure and Union Operators

The structure and union constructs in the C language provide an abstraction for creating objects that consist of multiple types. The . (member) and -> (structure pointer) operators abstract accesses to structures and unions. These operators can be converted to an equivalent address expression and indirection operation.

The C2H Compiler keeps storage for global structure and union variables in the Nios II processor's data memory. For global structures and unions, the C2H Compiler converts structure and union operations to equivalent pointer dereferences, and then translates the pointer dereference to hardware. The pointer dereference creates an Avalon-MM master port on the hardware accelerator, as described in section "Indirection Operator (Pointer Dereference)" on page 3–16.

For storage of local structure and union variables, the C2H Compiler uses on-chip memory resources inside the accelerator. For these local variables, the C2H Compiler generates master ports internal to the accelerator, which connect only to the internal memory. Refer to section "Arrays, Structures, and Unions" on page 3–14.

Member Operator .

The member operator (.) accesses a single member of a structure or union.

The struct member operation (mystruct.a) is equivalent to Example 3–20.

Example 3–20. Converted struct Member Operation
```
*((pointer_to_type_of_a) ((void *)&mystruct + offset_of_a))
```

Similarly, the union member operation (myunion.a) is equivalent to Example 3–21.

Example 3–21. Converted union Member Operation
```
*((pointer_to_type_of_a)(&myunion))
```

Consider the simple `struct` declaration shown in Example 3–22.

Example 3–22. Structure Declaration
```
struct s
{
  int element_a;
  int element_b;
  int element_c;
} my_struct;
```

In this example, the expression (`my_struct.c`) translates to the following:

```
*((int *)((void *)&my_struct + 2*sizeof(int)))
```

Structure Pointer Operator ->

The structure pointer operator is used to access a single member of a structure or union via a pointer to the structure or union. The structure pointer operator behaves similarly to the member operator, except that the left-hand side of the operator must be a pointer.

The structure pointer operation on a `struct` (e.g. `mystruct->a`) is equivalent to Example 3–23.

Example 3–23. Converted struct Pointer Operation
```
*((pointer_to_type_of_a) ((void *)mystruct + offset_of_a))
```

The structure pointer operation on a `union` (e.g. `myunion->a`) is equivalent to Example 3–24.

Example 3–24. Converted union Pointer Operation
```
*((pointer_to_type_of_a)myunion)
```

Consider the simple struct declaration shown in Example 3–25.

Example 3–25. Structure Pointer Declaration
```
struct s_ptr
{
  int element_a;
  int element_b;
  int element_c;
} * my_struct;
```

In this example, the expression (`my_struct->element_c`) translates to the following:

```
*((int *)((void *)mystruct + 2*sizeof(int)))
```

Scheduling

This section describes how the C2H Compiler schedules operations. The C2H Compiler is similar to a traditional C compiler in many respects: It parses code, creates a graph of the dependencies, performs some optimizations, schedules the sequence to execute each operation, and outputs an object file in the form of a hardware accelerator. However, fundamental differences exist between scheduling for a microprocessor and scheduling for a hardware accelerator.

Scheduling Concepts for Hardware Accelerators

A microprocessor has limited computational resources, defined by its arithmetic logic unit (ALU), and limited I/O resources, defined by its data bus architecture. In contrast, a hardware accelerator can have arbitrary computational and I/O resources, limited only by the practical bounds of resource utilization and the ability to achieve frequency performance. These resources can operate in parallel, stalling only to wait for data dependencies to resolve.

The C2H Compiler uses the following fundamental rule for scheduling: Perform computation operations and I/O operations as soon as data dependencies are resolved.

State Machines

Sections "One-to-One C-to-Hardware Mapping" on page 3–1, "Variable Declarations" on page 3–13, and "Memory Accesses" on page 3–15 described how the C2H Compiler translates individual operations, assignments, and memory accesses to atomic functional units in

hardware. After the C2H Compiler creates the functional units, it generates a hierarchy of state machines to control the operation and interaction of these units.

The C2H Compiler generates a distinct state machine for each of the following:

■ Accelerated function (top level)
■ Loop
■ Subfunction

The states comprise a sequence of stages that compute the results of the C function. The C2H Compiler assigns each operation to a state of the state machine. An arbitrary number of operations can execute during one state, allowing multiple operations to execute in parallel. Generally, the time for one state to execute equates to one clock cycle, although certain conditions cause stalls in the state machine's progression through states.

Data Dependencies

Scheduling of assignments within an accelerator is based on the data dependencies between the assignments. If assignment *B* depends on a value calculated in assignment *A*, then *B* cannot execute until *A* has completed. If two or more assignments are not dependent on each other, they can be scheduled in parallel.

One way to illustrate data dependencies is through a dependency graph. For each expression, arrows in a dependency graph represent where the inputs come from, and where the output is used. These arrows illustrate the flow of data through this function.

Figure 3–12 shows the dependency graph for Example 3–26.

Example 3–26. Data Dependency

```
int foo(int a, int b, int c)
{
  int x = a * b;
  int y = b * c;
  int z = x + y;
  return z;
}
```

Figure 3–12. Dependency Graph
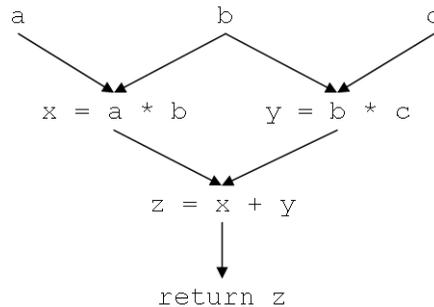
The C2H Compiler uses the dependency graph to assign each assignment to a state in the state machine. Figure 3–13 shows how the C2H Compiler assigns each assignment to a state.

Figure 3–13. Scheduling Assignments in a Dependency Graph
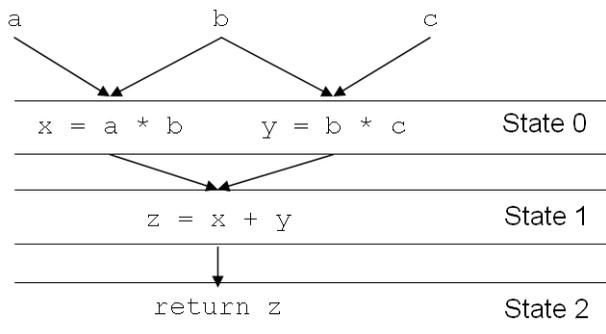
Pointer Aliasing

Aliasing is a situation where it is possible for a change to one variable or reference to affect another. In the C language, aliasing can occur due to the indirection introduced by pointers. If the address ranges referenced by two pointers overlap, the pointers alias. Pointer aliasing is another form of data dependency that the C2H Compiler must consider. Any read or write operation with a pointer is dependent on all pointer write-

operations that come before it. Because arrays and structures are equivalent to pointer operations, the same considerations apply when indexing into an array or structures.

This section describes the implications of aliasing on the C2H Compiler and outlines methods to prevent unnecessary dependencies.

Figure 3–14 shows the dependency graph for Example 3–27.

Example 3–27. Pointer Aliasing

```
void foo(int *ptr_a, int *ptr_b)
{
  int a, b;
  a     = *ptr_a;
  *ptr_a = a + 7;
  b     = *ptr_b;
  *ptr_b = b + 8;
}
```

Figure 3–14. Pointer-Related Data Dependency
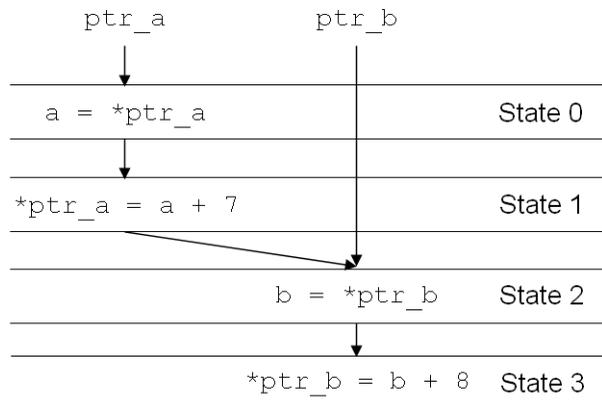
In this example, the C2H Compiler cannot determine whether or not `ptr_a` and `ptr_b` ever point to the same address. Therefore, it schedules conservatively, under the assumption that they do. The dependency graph shows that the read operation from `ptr_b` depends on the write operation to `ptr_a`. This is not a dependency on the variable `ptr_a`, but rather a dependency on a location in memory that is unknown at

compile-time due to the possibility of aliasing. This dependency causes the read operation from `ptr_b` to be scheduled at State 2, rather than at State 0.

__restrict__ Pointer Type Qualifier to Break Dependencies

If you know that a pointer never overlaps with another, you can inform the compiler by declaring the pointer to be a restricted pointer. The `restrict` type qualifier is introduced in the ISO C 99 specification. The compiler ignores any or all aliasing implications of a pointer qualified by `__restrict__`. The C99 specification states that if a pointer "p" is declared with restrict, and another pointer "q" accesses any location also accessed by "p," the behavior is undefined. In other words, a restricted pointer promises to never alias another pointer.

Example 3–28 demonstrates several pointers declared using the `__restrict__` type qualifier.

☞ The qualifier comes after the `*`; `__restrict__` qualifies the pointer type, not the type that the pointer points to.

Example 3–28. Pointer Declarations with _ _ restrict_ _
```
int * __restrict__ my_restricted_pointer_to_integer;
const int * __restrict__ my_restricted_pointer_to_constant_integer;
int * const __restrict__ my_constant_restricted_pointer_to_integer;
```

Figure 3–15 shows the dependency graph for Example 3–29, which uses the `__restrict__` type qualifier to inform the C2H Compiler that `ptr_a` and `ptr_b` do not alias:

Example 3–29. Using _ _restrict_ _
```
void foo(int * __restrict__ ptr_a,
         int * __restrict__ ptr_b)
{
  int a, b;
  a      = *ptr_a;
  *ptr_a = a + 7;
  b      = *ptr_b;
  *ptr_b = b + 8;
}
```

Figure 3–15. __restrict__ Pointer Type Breaks Dependencies
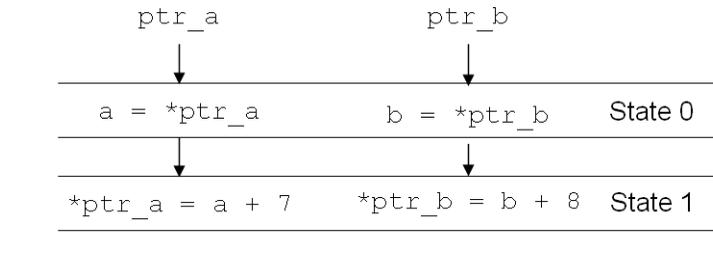
Although a pointer qualified with `__restrict__` creates no dependencies with other pointers, it can create dependencies with itself. Figure 3–16 shows the dependency graph for Example 3–30.

Example 3–30. Pointers Always Depend on Themselves

```
void foo(int * __restrict__ my_ptr,
         int offset_a,
         int offset_b)
{
  int a, b;
  a              = my_ptr[offset_a];
  my_ptr[offset_a] = a + 7;
  b              = my_ptr[offset_b];
  my_ptrb[offset_b] = b + 8;
}
```

Figure 3–16. Pointers Always Depend on Themselves
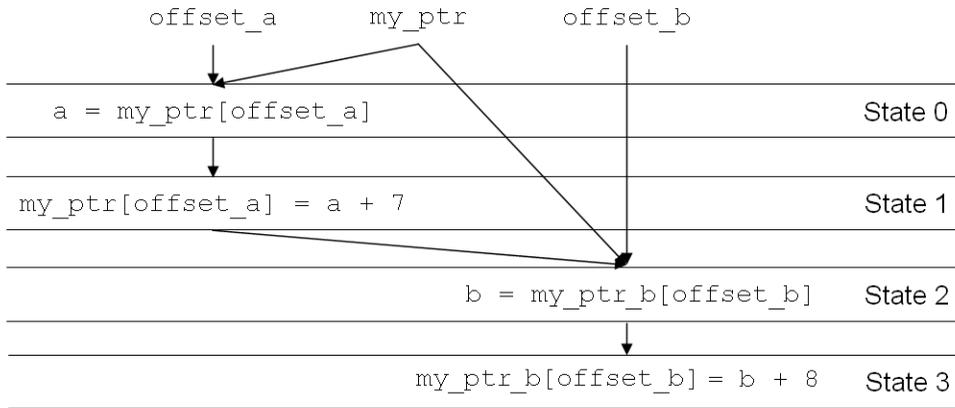
In this example, the C2H Compiler cannot schedule the two read operations in parallel, because it assumes that the two address expressions of my_ptr could overlap. Assuming that offset_a never equals offset_b, to make these operations execute in parallel, you need to declare another restricted pointer.

Figure 3–17 shows the dependency graph for Example 3–31, which introduces a new restricted pointer, my_ptr_b, to prevent the data dependency present in Figure 3–16.

Example 3–31. Using Another Pointer to Avoid Self-Dependence

```
void foo(int * __restrict__ my_ptr,
         int offset_a,
         int offset_b)
{
  int a, b;
  int * __restrict__ my_ptr_b = my_ptr;
  a                = my_ptr[offset_a];
  my_ptr[offset_a] = a + 7;
  b                = my_ptr_b[offset_b];
  my_ptr_b[offset_b] = b + 8;
}
```

Figure 3–17. Using Another Pointer to Avoid Self-Dependence

```
        offset_a        my_ptr        offset_b

    a = my_ptr[offset_a]      b = my_ptr_b[offset_b]      State 0

  my_ptr[offset_a] = a + 7 my_ptr_b[offset_b] = b + 8     State 1
```
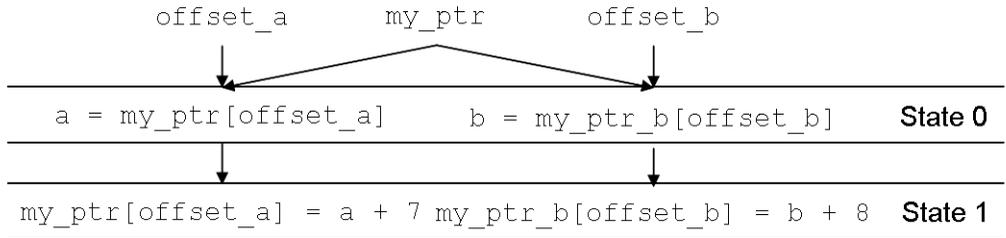
☞   If a data structure is referenced by two pointers and one or more
        of them is restrict-qualified, the ISO C 99 standard specifies that
        the behavior is undefined. Therefore, make sure that you fully
        understand the range of values that a pointer can take on during
        the execution of your application before applying the
        `__restrict__` qualifier. Improper application can result in
        undesirable functional changes to the code than cannot be
        debugged in software, due to the limitations of restrict-based
        optimizations in conventional compilers.

☞   The ISO C 99 standard specifies that the `volatile` type
        qualifier overrides the `__restrict__` pointer type. This
        means that `__restrict__` has no effect on `volatile`
        pointers. To break pointer dependencies between `volatile`
        pointers, use separate interrupt-enabled accelerators instead of
        multiple loops in the same accelerator. For details about
        interrupt-enabled accelerators, see "Interrupt Pragma" on
        page 6–4.

Using Physically Separate Slave Ports to Break Dependencies

If the master ports associated with two pointers do not access any shared
slave ports in the SOPC Builder system, then the C2H Compiler assumes
that the pointers do not alias. Section "Master-Slave Connections" on
page 3–23 describes how to limit the master-slave connections, which can
be an effective method to prevent aliasing.

Read Operations with Latency

Memory latency and other access delays affect how the C2H Compiler
schedules operations. Inherently, an operation cannot proceed until the
data for the operation arrives, which depends on memory latency. The
C2H Compiler generates logic within hardware accelerators to manage

and mitigate the effects of memory latency. Through close integration with SOPC Builder, the C2H Compiler can determine the latency characteristics of the slave ports connected to the accelerator. The C2H Compiler generates logic to maximize bandwidth for the specific memories in the system.

Avalon-MM pipelined read transfers increase the bandwidth for synchronous slave ports that require several cycles of latency to return data for the first access, but can return data every cycle thereafter. Using pipelined read transfers, a slave port can begin a new transfer before data from the previous transfer returns. There are only pipelined read transfers; Avalon-MM write transfers do not benefit from pipelined functionality.

The C2H Compiler takes memory latency into account when scheduling operations, allowing an accelerator to perform nondependent operations while waiting for data to return from a memory with latency. The master ports associated with a pointer might connect to multiple slave ports with different latency properties. In this case, the C2H Compiler uses the maximum latency of all slave ports.

Figure 3–18 shows the dependency graph for function `foo()`, shown in Example 3–32. This example uses the connection pragma to exclusively connect a pointer named `ptr_in` to a memory with two cycles of read latency. (Refer to section "Master-Slave Connections" on page 3–23.)

Example 3–32. Early Scheduling of Read Operation with Latency

```
#pragma altera_accelerate connect_variable \
    foo/ptr_in to \
    my_memory_with_two_cycles_read_latency

int foo(int *ptr_in, int x, int y, int z)
{
  int xy        = x * y;
  int xy_plus_z = xy + z;
  int ptr_data  = *ptr_in;
  int prod      = ptr_data * xy_plus_z;
  return prod;
}
```

Figure 3–18. Early Scheduling of Read Operation with Latency
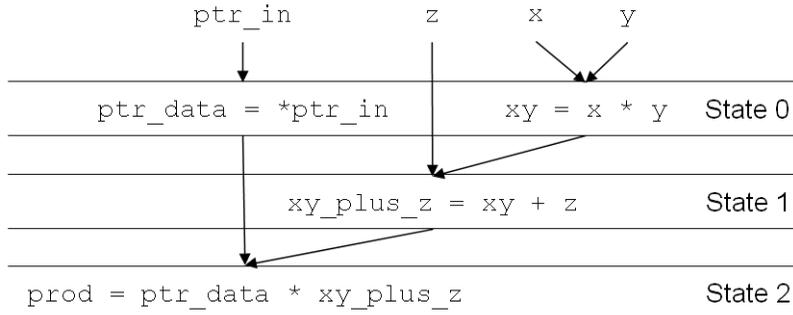
The C2H Compiler optimizes the dependency graph for this function by moving the read operation for `ptr_in` up to state 0. This optimization allows the calculation of `xy` and `xy_plus_z` to occur during the two cycles of latency required to fetch data for `ptr_in`.

Stalling

A state machine stalls when data needed for an operation is not available. A state machine might stall while waiting for one or more of the following actions to complete:

- Inner loop
- Subfunction call
- Memory transfer

The state machine does not proceed until all reasons for stalling are resolved.

Inner Loops

Each loop is implemented as a state machine, and an inner loop translates to a particular state within the state machine for its containing function or outer loop. In other words, an inner loop translates to a state machine within a state machine. As the state machine for an inner loop executes, the outer state machine stalls until the inner loop has completed.

For the purposes of scheduling, the C2H Compiler treats a loop and its dependencies as a unit. No lines of code past the loop block execute until the whole loop completes. Figure 3–19 shows the dependency graph for the function transform_and_hash_matrix(), shown in Example 3–33.

Example 3–33. Dependency Graph for a Function Containing a Loop

```
int transform_and_hash_matrix(int *matrix,
                      int length, int width)
{
  int n_words = length * width;
  int hash = 1;
  int i;
  for (i=0; i<n_words; i++)
  {
    ...perform some transform...
    hash = ...some hash calculation...
  }
  return hash;
}
```

Figure 3–19. Dependency Graph for a Function Containing a Loop

`n_words = length * width`	State 0
`for(i=0;i<n_words;i++){...}`	State 1
`return hash`	State 2
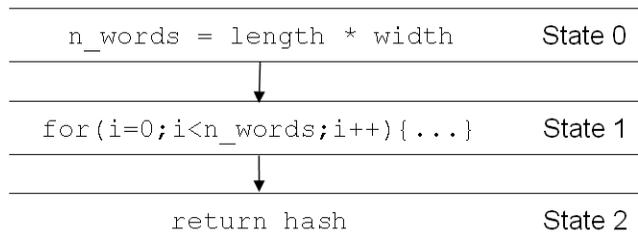
As shown in Figure 3–19, some part of the for loop depends on n_words, and so the C2H Compiler does not schedule the loop until after the assignment to n_words completes. The return statement outside the loop depends on hash, which is assigned inside the loop. As a result, the C2H Compiler does not schedule the return statement until the loop completes.

In this case, the state machine for transform_and_hash_matrix() has three states. However, the state machine does not complete in three clock cycles, because State 1 consists of a sub-state-machine, which requires multiple clock cycles to complete.

If multiple loops have no interdependencies, the C2H Compiler schedules the loops on the same state, allowing the loops to execute in parallel. The code in Example 3–34 has two `while` loops with no dependencies on each other. The C2H Compiler schedules these loops on the same state.

Example 3–34. Loops Without Interdependencies Scheduled in Parallel

```
void double_mac (int* __restrict__ a, int* __restrict__ b,
                 int* __restrict__ c, int* __restrict__ d,
                 long long* __restrict__ res_ab,
                 long long* __restrict__ res_cd,
                 int len)
{
  int len_cd = len;  // duplicate the length index
  // Compute the MAC for a & b
  long long mac_ab = 0;
  while (len_ab > 0)
  {
    mac_ab += *a++ * *b++;
    len_ab--;
  }
  // Compute the MAC for c & d
  long long mac_cd = 0;
  while (len_cd > 0)
  {
    mac_cd += *c++ * *d++;
    len_cd--;
  }
  *res_ab = mac_ab;
  *res_cd = mac_cd;
  return;
}
```

Subfunction Calls

A subfunction call can stall the state machine in the same way that an inner loop does. When a subfunction contains a looping structure or shares a data dependency with its caller, the subfunction is not pipelined. If this is the case, when the outer state machine reaches its state for the subfunction, the outer state machine stalls until the subfunction has completed.

If the subfunction does not contain loops or shared data dependencies the C2H Compiler can pipeline the subfunction. For details about pipelined subfunctions, see "Subfunction Pipelining" on page 3–49.

Memory Transfers

Avalon-MM system interconnect fabric manages arbitration between multiple Avalon-MM master ports that access a single slave port. A master port might have to wait several clock cycles before beginning a transfer due to arbitration. If a master port on an accelerator is being forced to wait, the state machine for the accelerator stalls until the transfer can proceed.

Loop Pipelining

The C2H Compiler structures the state machine for a loop so that iterations of the loop are pipelined. In other words, consecutive iterations of the loop can begin before prior iterations have completed.

Pipelining Loop Iterations

Figure 3–20 shows the dependency graph for the loop block in Example 3–35.

Example 3–35. Loop Block

```
int mac(int *data_array, int *coef_array, int len)
{
  int sum = 0;
  do
  {
    int x = *data_array++;
    int c = *coef_array++;
    int prod = c * x;
    sum += prod;
  } while (i++ < len);
  return sum;
}
```

Figure 3–20. Dependency Graph for a Loop Block
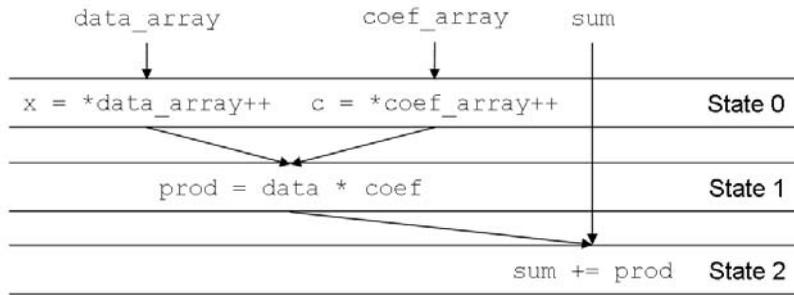
Figure 3–21 illustrates how the C2H Compiler schedules successive iterations of the loop shown in Figure 3–20. The C2H Compiler is able to start a new iteration of the loop immediately after the prior iteration completes State 0.

This is an example of an ideally-pipelined loop. Although the C2H Compiler can pipeline many loops ideally, it is sometimes not possible due to the lack of inherent parallelism in the code, as shown in "Loop-Carried Dependencies".

Figure 3–21. Pipelined Loop Iterations

Time	Iteration 0	Iteration 1	Iteration 2

0	(State 0)
	`x = *data_array++`
	`c = *coef_array++`		

1	(State 1)
	`prod = c * x`
		(State 0)
		`x = *data_array++`
		`c = *coef_array++`	

2	(State 2)
	`sum += prod`
		(State 1)
		`prod = c * x`
			(State 0)
			`x = *data_array++`
			`c = *coef_array++`

3		(State 2) `sum += prod`	(State 1) `prod = c * x`
4			(State 2) `sum += prod`

Loop-Carried Dependencies

Loop-carried dependencies are data dependencies that manifest when pipelining successive iterations of a loop. If the result of one calculation in an iteration of the loop is used in a later iteration of the loop, then a loop-carried dependency exists between the two operations.

Figure 3–22 shows the dependency graph for the `do` loop in Example 3–36, which has loop-carried dependencies.

Example 3–36. Loop-Carried Dependency

```
int simple_hash(int *data, int len)
{
  int hash = 0;
  do
  {
    int data_word = *data++;
    hash = hash + data_word;
    hash = hash ^ data_word;
  } while (len--);
  return hash;
}
```

Figure 3–22. Loop-Carried Dependency
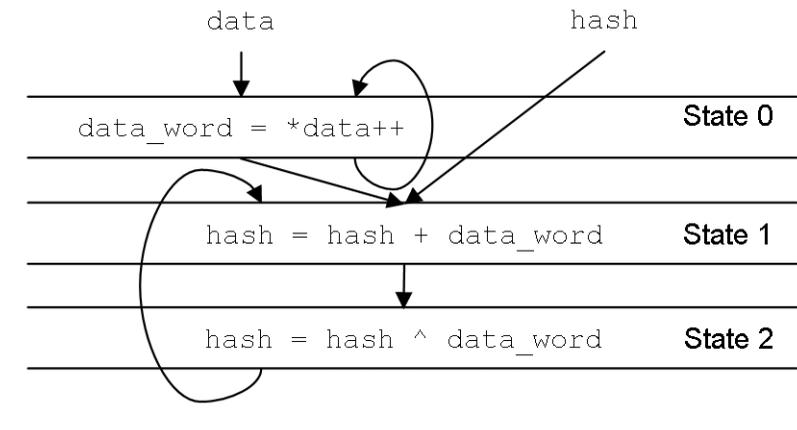
Variables `data` and `hash` have loop-carried dependencies, illustrated by the cyclic arrows in Figure 3–22. The arrow for `hash` indicates that the calculation on State 1 in iteration *N* is dependent on the result of the calculation from State 2 in iteration (*N*-1). The arrow for `data` in State 0 illustrates the ideal case in which a state depends only on its own output. The ideal case does not restrict the scheduling of successive iterations.

Figure 3–23 illustrates how the C2H Compiler schedules successive iterations of the loop shown in Figure 3–22, based on the restrictions imposed by hash. State 1 cannot execute until the previous iteration has completed State 2. The C2H Compiler schedules the states as shown in Figure 3–23 to satisfy the loop-carried dependency.

Figure 3–23. Pipelined Loop Iterations with a Loop-Carried Dependency

Time	Iteration 0	Iteration 1	Iteration 2
0	(State 0) data_word = *data++		
1	(State 1) hash += data_word		
2	(State 2) hash ^= data_word	(State 0) data_word = *data++	
3		(State 1) hash += data_word	
4		(State 2) hash ^= data_word	(State 0) data_word = *data++
5			(State 1) hash += data_word
6			(State 2) hash ^= data_word

In Figure 3–23, the cyclic arrow for hash in Figure 3–22 translates to straight arrows between iterations.

Pipelining Avalon-MM Read Transfers from Multiple Iterations

As discussed in section "Read Operations with Latency" on page 3–37, the C2H Compiler is aware of read latency in slave memories. Master ports on C2H accelerators can use Avalon-MM pipelined read transfers, which allow multiple read transfers to be pending at a given time. As a result, for a loop that reads from memory with latency, the next iteration of the loop can begin fetching data before data from the previous iteration has returned. For such a loop, the C2H Compiler creates a master port

that performs Avalon-MM pipelined read transfers. Inside the accelerator, the master port connects to a FIFO, which guarantees the accelerator can receive data for all pending read transfers, regardless of whether the state machine stalls.

Figure 3–24 shows the dependency graph for Example 3–37, which demonstrates a loop that pipelines memory accesses with latency. This example uses the connection pragma to connect the master port for variable `list` to a slave memory named `my_mem_with_two_cycles_read_latency`, as described in section "Master-Slave Connections" on page 3–23.

Example 3–37. Accessing Memory with Latency

```
#pragma altera_accelerate connect_variable \
    sum_elements/list to \
    my_mem_with_two_cycles_read_latency

int sum_elements (int *list, int len)
{
  int i;
  int sum = 0;
  for (i=0; i<len; i++)
    sum += *list++;
}
```

Figure 3–24. Accessing Memory with Latency
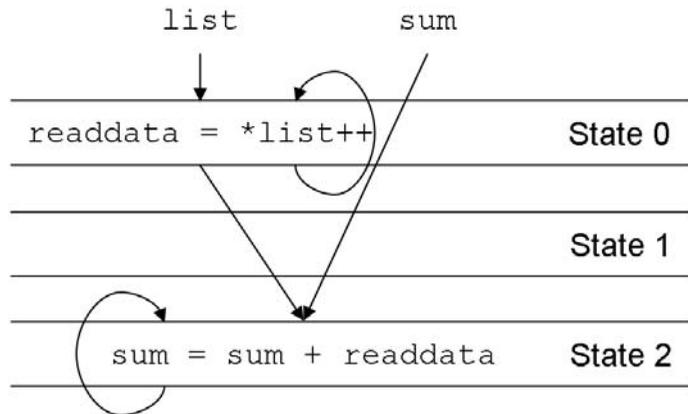
State 1 in Figure 3–24 remains empty because `list` has two cycles of read latency. The loop-carried dependencies on variables `sum` and `list` are ideal cases, which do not impose restrictions on the pipeline scheduling. Figure 3–25 illustrates how the C2H Compiler schedules successive iterations of the loop.

Figure 3–25. Pipelined Loop Iterations Reading Memory with Latency

Time	Iteration 0	Iteration 1	...	Iteration N-1	Iteration N
0	(State 0) `readdata = *list++`		...		
1	(State 1) (empty latency state)	(State 0) `readdata = *list++`	...		
2	(State 2) `sum += readdata`	(State 1) (empty latency state)	...		
3		(State 2) `sum += readdata`	...		
...	...	...	...	...	...
N-1			...	(State 0) `readdata = *list++`	
N			...	(State 1) (empty latency state)	(State 0) `readdata = *list++`
N+1			...	(State 2) `sum += readdata`	(State 1) (empty latency state)
N+2			...		(State 2) `sum += readdata`

As shown in Figure 3–25, the C2H Compiler is able to start a new iteration of the loop immediately after the prior iteration completes State 0. At Time 1, Iteration 1 starts a new read access from `list`, even though data from `list` hasn't returned for Iteration 0. Due to the two cycles of read latency, at any given time, there can be a maximum of two pending read operations.

Over successive iterations of a loop, the C2H Compiler hides the memory latency by pipelining the read transfers. Although multiple cycles of latency are required to fill the pipeline, successive iterations can complete at a rate of one per clock cycle, assuming no stalling occurs (see section "Stalling" on page 3–39).

Loop Latency and Cycles per Loop Iteration (CPLI)

There are two metrics that determine the efficiency of an accelerated loop: loop latency and cycles per loop iteration (CPLI).

Loop latency is the amount of time required for the first iteration of a loop to complete, assuming one clock cycle per state of the loop's state machine. An iteration completes when it passes from the first state through the last state, and so the loop latency is equal to the number of states in the loop's state machine.

CPLI is the minimum possible period for issuing successive loop iterations. A new iteration of the loop state machine is issued every *CPLI* clock cycles, assuming one clock per state. After initial loop latency is overcome, a loop's state machine produces results for each successive iteration every *CPLI* clock cycles. CPLI is determined by the loop-carried dependencies present in a loop.

In general, the goal of optimizing C code for the C2H Compiler is to reduce both loop latency and CPLI to as close to 1 as possible for all loops in the accelerator. Loop latency and CPLI values assume no stalling, however, which is not always realistic (see section "Stalling" on page 3–39). Inner loops, subfunction calls, and slow memory accesses all cause stalling at runtime, which cannot be reflected in CPLI and loop latency. Conceptually, C2H optimization techniques are similar to traditional C compilers: Focus effort on minimizing loop latency and CPLI for critical inner loops first; reduce execution time of subfunction calls; and use fast memory for performance-critical sections of code.

Subfunction Pipelining

Each subfunction is implemented as a state machine, and a subfunction call translates to a particular state within the containing function or loop. In other words, a subfunction translates to a state machine within a state machine. The fact that it is a distinct state machine allows it to be a shared resource within the containing function.

If the subfunction does not contain loops or shared data dependencies, the C2H Compiler can pipeline the subfunction. The subfunction has its own state machine, but the datapath is pipelined as if it were the body of a loop. When the outer state machine reaches the state to call the subfunction, it can continue to execute other operations in parallel with the inner state machine. Data sets from multiple subfunction calls are pipelined in the subfunction's state machine.The code in Example 3–38 contains a subfunction which is pipelined by the C2H Compiler.

Example 3–38. Pipelined Subfunction

```
int MAX(int a, int b)
{
  return ((a > b)? a : b);
}

#pragma altera_accelerate connect_variable MAX_loop/a to sdram
#pragma altera_accelerate connect_variable MAX_loop/b to
onchip_ram_64_kbytes
int MAX_loop(int * __restrict__ a, int * __restrict__ b)
{
  int i, c = 0;
  for (i = 0; i < 1024; i++)
  {
    c += MAX(a[i], b[i]);
  }
  return c;
}
```

If the subfunction performs a memory access that stalls, then the outer state machine also stalls.

Pipelined subfunctions provide a useful option for controlling shared resources. For further information, see "Resource Sharing".

Resource Sharing

The C2H Compiler is capable of sharing resources which consume significant amounts of logic. A resource only becomes shared if it is under-utilized. In other words, the C2H Compiler only shares a resource if the performance of accelerator is not affected. Table 3–6 lists all of the resources that can be shared automatically by the C2H Compiler.

Table 3–6. Sharable Resource		
Dereference or Operator	**Description**	**Required Conditions**
*var var[]	Memory Access	- Multiple dereferences to access data from the same Avalon-MM memory port - Multiple dereferences occurring within the same level in the algorithm (function or loop) - Loop CPLI must not increase
*	Multiply	- Promoted data width (32 or 64 bit) must be the same for all multiplications. The promoted data width is shown in the C2H report. - Multiplications occurring within the same level in the algorithm (function or loop) - Both operands must be either signed or unsigned. - Loop CPLI must not increase
/	Divide	- Promoted data width (32 or 64 bit) must be the same for all divisions. The promoted data width is shown in the C2H report. - Divisions occurring within the same level in the algorithm (function or loop) - Both operands must be either signed or unsigned. - Loop CPLI must not increase
%	Modulo	- Promoted data width (32 or 64 bit) must be the same for all modulo operations. The promoted data width is shown in the C2H report. - Modulo operations occurring within the same level in the algorithm (function or loop) - Both operands must be either signed or unsigned. - Loop CPLI must not increase
<<	Left Shift	- Promoted data width (32 or 64 bit) must be the same for all left shift operations. The promoted data width is shown in the C2H report. - Left shift operations occurring within the same level in the algorithm (function or loop) - Both operands must be either signed or unsigned. - Loop CPLI must not increase
>>	Right Shift	- Promoted data width (32 or 64 bit) must be the same for all right shift operations. The promoted data width is shown in the C2H report. - Right shift operations occurring within the same level in the algorithm (function or loop) - Both operands must be either signed or unsigned. - Loop CPLI must not increase

The resource sharing technique used for memory accesses differs slightly from all other sharable resources. Memory accesses which share the same Avalon-MM master port use byte enables to control the width of the access. The master port width is equal to the widest data type being accessed. The other sharable resources do not use byte enables, so operator inputs and result must be of equal width for the resource to be shared. 8-bit and 16-bit operators are automatically promoted to be 32 bits wide, so as long as both operands are either signed or unsigned, the operator can be shared.

As previously mentioned, resources are not shared if the performance of the hardware accelerator degrades. For example, if the algorithm is capable of performing two multiplications in parallel, the C2H Compiler does not share a multiplier resource. The C2H Compiler optimizes for performance by default and enables sharing only when appropriate.

The exception to this rule is when two pointer dereferences occur that access data from the same Avalon-MM memory port. The C2H Compiler knows that memory arbitration already limits the performance of the accelerator and so the resources are shared. This exception is bound to Avalon-MM ports since multi-port components can be accessed concurrently. If you use the appropriate connection pragma statements and `__restrict__` qualifier, you can ensure that memory accesses occur concurrently instead of becoming a shared Avalon-MM master port.

The C2H Compiler only shares resources if they reside at the same level within a loop or function. Figure 3–26 shows an algorithm which contains shared and independent resources.

Figure 3–26. Shared And Independent Resources
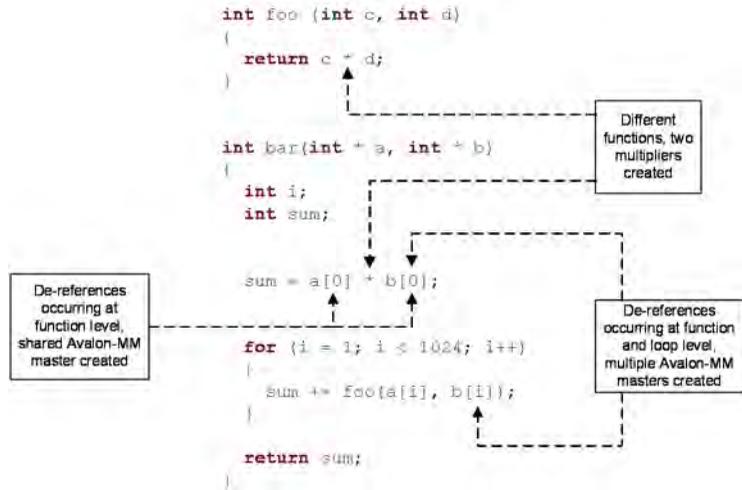
Another way of managing shared resources is to place the code that uses the resource in a subfunction. For example, to ensure that a math-intensive function uses no more than three multipliers, you could place the multiply operation in three subfunctions `mul1()`, `mul2()` and `mul3()`. With pipelined subfunctions, the latency overhead of this approach is not excessive. For further details, see "Subfunction Calls" on page 3–41.

4. Understanding the C2H View

Introduction

This chapter discusses the Altera Nios II C-to-Hardware Acceleration (C2H) Compiler view available in the Nios II IDE. Understanding the C2H view allows you to estimate the resource usage and the performance of the accelerator. You can use this information to perform optimizations to reduce the logic resource size or increase the performance of the accelerator.

Overview

You can use the C2H view to do the following:

- Add or update hardware accelerators
- Control how the C2H Compiler compiles your C code to hardware
- Display information about hardware accelerators

The C2H view displays performance information about your accelerated functions. This information makes it easy to select the best configuration for the next compile.

The C2H view contains two sections:

- **Generation/Compilation Configurations** – the C2H view allows you to specify various generation and compilation configurations for your accelerated functions. The configurations allow you to control the build flow of the entire software project and the individual accelerated functions.
- **Build Report** – The C2H Compiler creates the build report during a software compilation when it analyzes functions or generates accelerators. Using the build report you can view the resource usage and scheduling information for each accelerated function.

Generation/Compilation Configurations

The following sections discuss the two levels of configurations that you can use to control the build flow.

Project Build Configurations

The project build configurations allow you to control linker settings for a project with multiple accelerators. The project build configurations also control the integration of the accelerators with SOPC Builder and the Quartus II software.

Table 4–1 *summarizes the project build configurations.

Table 4–1. Project Build Configurations (Part 1 of 2)	
Configuration	**Meaning**
Use software implementation for all accelerators	You can use this global override configuration to force the linker to use the software implementation of all the existing accelerators in the system. You can use this configuration to perform functional changes to your algorithm without having to regenerate your system or compile the hardware design. This is a global configuration, and affects all accelerated functions. If you only wish to revert to the software implementation of an individual accelerated function, use the function build configurations section. When you use the software implementation, the hardware accelerator remains in the system, but is not used.
Use the existing accelerators	This global override configuration allows you to use accelerators that already exist in a system. If you previously switched to **Use software implementation for all accelerators**, this configuration lets you revert back to the accelerator hardware. In this configuration, the C2H Compiler does not regenerate the accelerator even if you make changes to the accelerator source. You can use this configuration near the end of the product development cycle to help prevent accidental hardware regeneration.
Analyze all accelerators	If you select **Analyze all accelerators**, the C2H Compiler analyzes the functions you have marked for acceleration, and produces a report of expected accelerator performance. It does not compile the accelerators. If you have existing accelerators (from a previous compile), they are untouched. In other words, **Analyze all accelerators** is the same as **Use the existing accelerators**, except that it also produces a report. **Analyze all accelerators** lets you quickly display build information without regenerating the SOPC Builder system. This configuration does not overwrite the existing accelerator logic, it simply analyzes the source code.

Table 4–1. Project Build Configurations (Part 2 of 2)

Configuration	Meaning
Build software and generate SOPC Builder system	This configuration allows you to update the function accelerators in your system by forcing SOPC Builder to regenerate the logic. SOPC Builder runs in the background and you can view the process of the logic regeneration from the Nios II IDE console view. Do not open SOPC Builder until logic generation is complete. The Nios II IDE coordinates the software and hardware builds. Once the SOPC Builder regeneration is complete, the Nios II IDE displays the updated build information in the C2H view. The logic regeneration only occurs if the accelerator did not previously exist or the accelerated function source code changes. In order for these changes to take effect you must compile the hardware design using the Quartus II software.
Build software, generate SOPC Builder system, and run Quartus II compilation	This configuration is a superset of **Build software and generate SOPC Builder system**. Once SOPC Builder generates the system, Quartus II runs in the background and compiles the hardware design. While the Quartus II software is running in the background you can view the progress from the Nios II IDE console view.

Function Build Configurations

The function build configurations let you control the linking of individual accelerated functions of a software project and the cache coherency settings. This control is important for systems with multiple accelerators per software project since cache coherency might not be an issue for all the accelerated functions. While you work on a single accelerator, you can compile the other accelerated functions in software-only mode, speeding up C2H compilation.

Table 4–2 describes the available function build configurations.

Resources

The resources section of the build report shows information about the resource usage of the accelerated function. The following are the resources that can appear in this section of the build report:

■ Avalon-MM master ports
■ Multipliers
■ Dividers
■ Barrel shifters (variable shift)

Each of these resources has information about how they are configured in the hardware and the line of source code that mapped to them. In the following discussion of these resources, refer to Example 4–1.

Table 4–2. Function Build Configurations	
Configuration	**Meaning**
Use hardware accelerator in place of software implementation. Flush data cache before each call	Use this configuration if you are not certain whether cache coherency is an issue in your system. In this configuration, every time the software calls the accelerated function, the wrapper function flushes the entire Nios II data cache, to prevent cache coherency issues. The C2H Compiler inserts flush code into the wrapper function so no source code modification is necessary. Since flushing the cache is a fixed overhead, if you have strict processing time requirements you need to study the system architecture and determine if this operation is necessary.
Use hardware accelerator in place of software implementation	This configuration uses the hardware accelerator without flushing the data cache before each invocation. Use this configuration with algorithms that do not require Nios II data cache flushing. Do not use this configuration if you have not studied your algorithm to determine if it could have cache coherency problems. The accelerated function might create cache coherency problems in certain corner cases. To prevent cache coherency problems, use one or more of the following techniques in your code: ● Allocate all shared data in uncached Nios II memory space. Refer to "*Bit-31 Cache Bypass*" in the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*. *(1)* ● Flush all shared data before calling the accelerated function. ● Place all shared data in a tightly coupled memory. ● Manage cache coherency in a multiprocessor system by establishing a cache coherency protocol between the processor controlling the accelerated function and all other processors. ● Use cache bypass macros to access all shared data. ● Do not share memory between the hardware accelerator and the Nios II processor ● Refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook* to learn more about cache coherency.
Use software implementation	Much like the project wide configuration, this causes the C2H Compiler to link the software implementation of the accelerated function. Unlike the project wide configuration, this only affects a single accelerated function and not the entire software project. You can use this configuration to prototype changes to your algorithm without having to regenerate or recompile the hardware. This project configuration does not remove the existing accelerator from the system.

Note to Table 4–2:

(1) Use caution passing uncached pointers to a C2H accelerator. There are two unusual situations that require special consideration: (a) If the accelerator performs direct arithmetic or comparisons on the address value of a pointer, it must account for the possibility that bit 31 is set. (b) If the accelerated function masters a component whose true address is >= 0x80000000, and uses the same Avalon-MM master port to connect to the memory that it shares with the Nios II processor, uncached pointers to the shared memory might result in spurious Avalon-MM transfers to addresses >= 0x80000000.

Example 4–1. Vector Power Calculation

```
#pragma altera_accelerate connect_variable power_calculation/voltage to onchipRAM1
#pragma altera_accelerate connect_variable power_calculation/current to onchipRAM1
#pragma altera_accelerate connect_variable power_calculation/power to onchipRAM2
void power_calculation ( short * __restrict__ voltage,
                         short * __restrict__ current,
                         short * __restrict__ power,
                         short downscale,
                         int length)
{
    int i;
    for(i = 0; i < length; i++)
    {
        *power++ = (*voltage++ * *current++) >> downscale;
    }
}
```

Example 4–1 requires Avalon-MM read and write master ports to perform memory accesses. It also requires a multiplier and a barrel shifter to perform the right shift operation. The pragma statements inform the C2H Compiler that the input data is stored in a memory called onchipRAM1 and the output data is to be stored in onchipRAM2. When the C2H Compiler compiles this function, the Nios II IDE generates a build report as shown in Figure 4–1.

Figure 4–1. Build Report - Resources
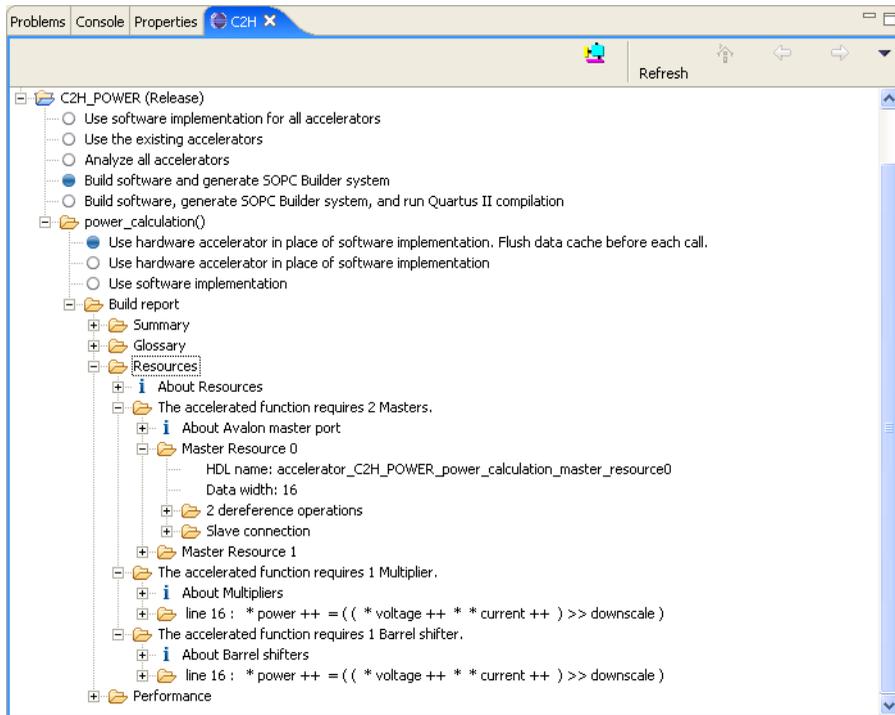
The resources section contains a subsection for each type of resource. The report shows Avalon-MM master port resources in a different layout from other operator resources due to the differences between the functionality.

Avalon-MM Master Port Resources

The C2H Compiler uses Avalon-MM master ports to implement dereference operations (memory accesses). The resources section of the C2H build report shows each Avalon-MM master port created by the C2H Compiler. The C2H Compiler creates the optimal number of master ports for the memory accesses it finds in the code. The C2H Compiler conserves Avalon-MM resources by creating a single Avalon-MM master port to access memory for multiple dereference operations if there is no performance disadvantage. In Example 4–1, the data pointed to by `voltage` and `current` reside in the same single-ported physical

memory, making it impossible for the accelerator to read both variables on the same clock cycle. The C2H Compiler creates a single Avalon-MM master port to access both values using interleaved accesses.

Figure 4–2. Avalon-MM Master Port Resources
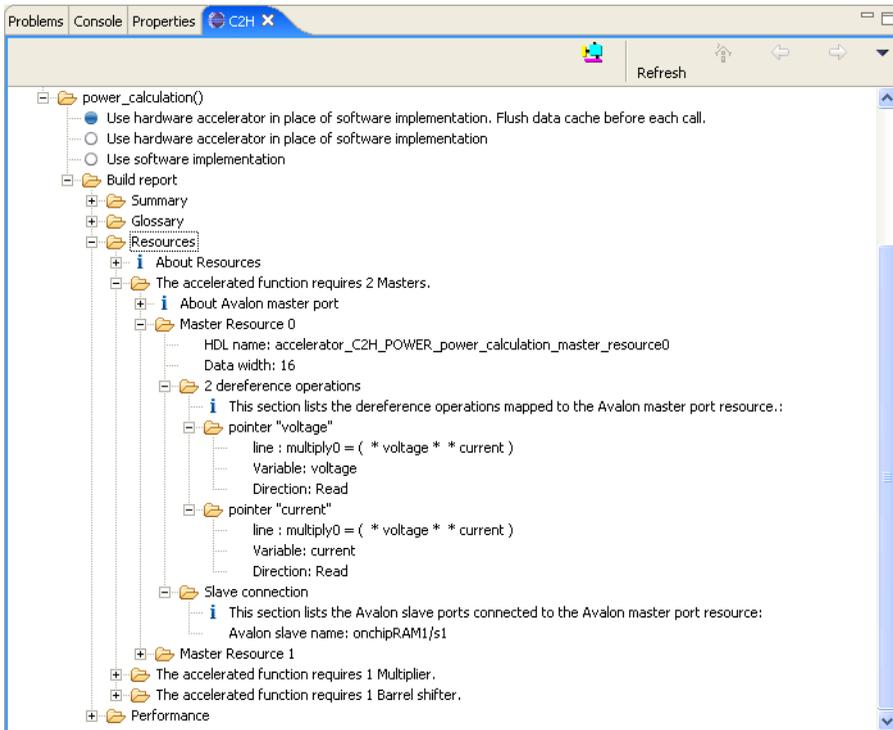
When memory accesses share a single Avalon-MM master port, the reported data width is that of the largest data type being accessed. The report shows each dereference operation for the shared Avalon-MM master port resource. In Example 4–1 the pointer `power` requires a separate Avalon-MM master port resource because it resides in a different memory than the input values.

For each dereference operation, the report shows the source line on which the C statement appears. It also shows the variable being dereferenced, and the data direction (read or write). Any one statement is either a read

or a write. However, when an Avalon-MM master port is shared among two or more dereference statements, it might need to support both directions.

In Example 4–1 on page 4–5, the connection pragmas forced the C2H Compiler to create a single, shared Avalon-MM master port called Master Resource 0. If the connection pragmas were omitted from the example software, all dereference operations would have resulted in a single Avalon-MM master port resource connecting to all Avalon-MM memory slave ports.

For more information about connection pragmas, refer to "*Optimizing Memory Connections*" in the *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*.

Mathematical Operator Resources

As mentioned in Chapter 3, C-to-Hardware Mapping Reference, the data types used in the C code determine the width of the logic which the C2H Compiler generates. The resources section of the C2H build report shows the width of all the resources listed. The resources section also shows the degree of pipelining on each operator resource. This information helps you understand the accelerator scheduling information shown in the Performance section.

Figure 4–3 illustrates the information presented in the C2H build report for mathematical operator resources in Example 4–1 on page 4–5.

Figure 4–3. Multiplier Resources
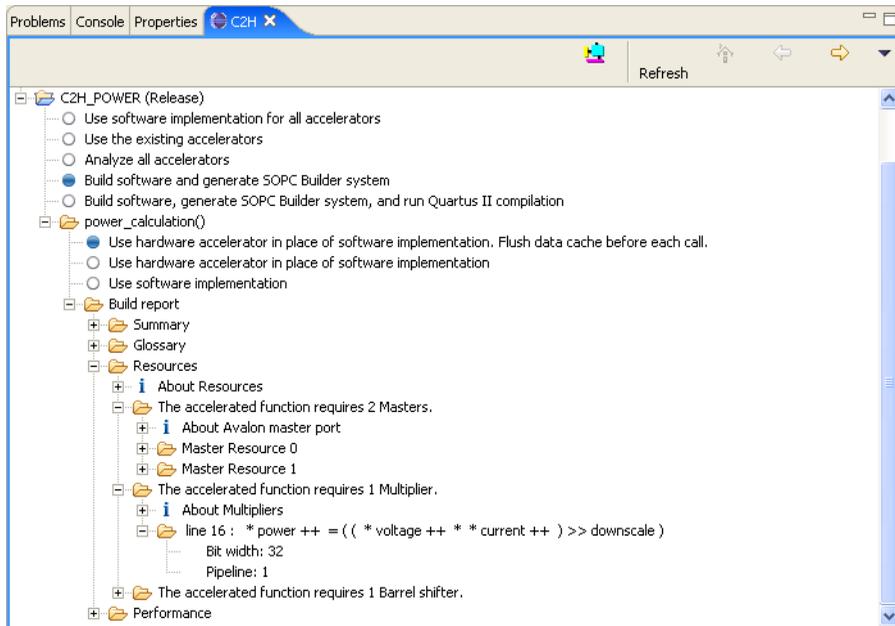
☞ The resource usage does not reflect the final resource utilization of the compiled hardware. When ANSI C code is compiled, small integer data types are promoted to the int data type. In Figure 4–3 we can see that the multiplier is 32 bits wide even though the operands are short (16 bits). The C2H Compiler performs the same integer data promotion, creating a 32-bit multiplier. When the Quartus II software compiles the hardware design, the synthesized multiplier is 16 bits in width.

The pipeline value associated with the resource specifies the number of clock cycles that the hardware logic requires for the calculation to complete. Pipelined logic can typically operate at higher clock frequencies due to the additional latency introduced. The C2H Compiler factors in the pipelining of the hardware and schedules the accelerated function accordingly to maximize data throughput. When the report does not show a pipeline value for a resource, that means that the operator is purely combinational, with no latency.

Performance

The performance section of the C2H build report details information about each loop in the accelerated function. For each loop shown, the report contains the following information:

■ File name and source line number
■ Loop latency
■ Cycles per loop iteration (CPLI)
■ Scheduling information per assignment
■ Scheduling information per state

In the following discussion of information shown in the performance section, refer to Example 4–2.

Example 4–2. CRC32 (Ethernet CRC)

```
#pragma altera_accelerate connect_variable\
    crc_calculation/data to onchipRAM1
#pragma altera_accelerate connect_variable\
    crc_calculation/table to onchipRAM2
unsigned long crc_calculation
  ( unsigned char * __restrict__ data,
    unsigned long * __restrict__ table,
    unsigned long length)
{
  unsigned long i, crc = 0xFFFFFFFF;
  unsigned char lut_addr;
  for (i = 0; i < length; i++)
  {
    lut_addr = (crc & 0xFF) ^ *data++;
    crc = (crc >> 8) ^ table[lut_addr];
  }
  return (crc ^ 0xFFFFFFFF);
}
```

Source Line Number

The performance section of the build report shows a source line number for each loop statement. The line number is determined by the beginning of the loop, which is the source line containing the opening block delimiter ({). If your coding style places the loop keyword (do, while, or for) on a separate line from the delimiter, the report shows the line where the delimiter appears.

Loop Latency

As mentioned in Chapter 3, C-to-Hardware Mapping Reference, the loop latency is a fixed time overhead, incurred each time the accelerator enters the loop. The loop latency is the number of states needed to set up conditions for efficient loop iteration.

☞ It is important to remember that if the accelerator re-enters the loop multiple times, it incurs the loop latency each time.

Cycles Per Loop Iteration (CPLI)

The CPLI is a measure of the data throughput of a loop. This section lists the critical loop variable and the assignments in the critical path. The critical loop variable represents the most significant contribution to the CPLI value of a loop. In Example 4–2, the critical loop variable is crc.

The critical path represents the longest data dependency in the loop. The CPLI value quantifies the length of the critical path. You can use this section to help you understand the scheduling and optimize the algorithm to reduce the CPLI. The assignments shown in this section contain not only the critical path variable but also all other assignments that take place on the same states as the critical path variable.

The concept of the critical path might appear confusing at first. However, if you have a good understanding of the algorithm, it is not difficult to find the critical path.

Figure 4–4 shows the CPLI report for Example 4–2. In the CPLI report, a path is identified by its starting and ending states. In Example 4–2, as we will see, the critical path turns out to be states 6 through 11. The report identifies this path as 6--->11.

Figure 4–4. CPLI Report

The report identifies two assignments containing critical path states. The data dependency graph for these statements is shown in Figure 4–5. The graph shows that `data` does not depend on any other statement in the loop. However, variables `crc`, `lut_addr` and `table` are involved in a mutually dependent chain of calculations in the critical path. `crc`, `lut_addr` and `table` are therefore the critical path variables. The C2H Compiler displays one critical loop variable (`crc`) as a way of identifying the critical path.

Figure 4–5. CRC Dependency Graph

Since the critical path does not involve the pointer `data` the only operations are on local scalar types and a read operation from the array `table`. The calculation of `lut_addr` only depends on the scalar critical loop variable `crc`, while the calculation of `crc` depends on a memory reference to critical loop array variable `table`.

Each time the C2H accelerator finishes calculating the value of `crc` for loop iteration *n*, it can start calculating `crc` for iteration *n+1*. On the same clock cycle, it can also start calculating the value of `lut_addr` for iteration *n+2*. This means that the accelerator always gets a one-loop head start on calculating `lut_addr`. Thus, although the accelerator requires `lut_addr` to calculate `crc`, `lut_addr` does not limit the loop speed, because it is always ready as soon as `crc` is.

The report shows that the critical path is either `0--->6`, or `6--->11`.

Since the C2H Compiler pipelines the logic contained in loops, multiple states are active concurrently. Figure 4–6 represents the pipelined timing of Example 4–2 on page 4–10. Notice that since the assignment of `crc` is the critical path, the accelerator begins each execution of that statement as

soon as the previous execution is complete. Not surprisingly, the critical path statement is what limits the speed of the loop, and hence what determines CPLI.

Figure 4–6. CRC Critical Path Scheduling by Assignment

Notice, also, that line 13 (`lut_addr = (crc & 0xFF) ^ *data++`) appears to take more clock cycles than line 14 (`crc = (crc >> 8) ^ table[lut_addr]`). The C2H Compiler "stretches out" the calculation of `lut_addr` so that it is available exactly when it is needed. The memory access in line 14 is nonetheless the limiting operation.

Scheduling Information

There are two ways of presenting the loop scheduling information: per assignment, and per state.

Scheduling Information Per Assignment

Typically the number of assignments in a loop is fewer than the number of states mapped for the hardware state machine that controls the loop. When there are fewer assignments than states, this method of interpreting the scheduling information is often easier.

In Example 4–2, there are four assignments in the loop. This section displays all of the assignments whether they occur on states that are a part of the critical path or not. As shown in the previous section the CPLI of this loop is six due to the critical path variable `crc`. Figure 4–7 illustrates the information shown when the example is compiled.

Figure 4–7. CRC Scheduling Per Assignment

Using the methodology from "Cycles Per Loop Iteration (CPLI)", you can create a chart such as Figure 4–8, corresponding to Example 4–2 on page 4–10.

Figure 4–8. CRC Assignment Sequence
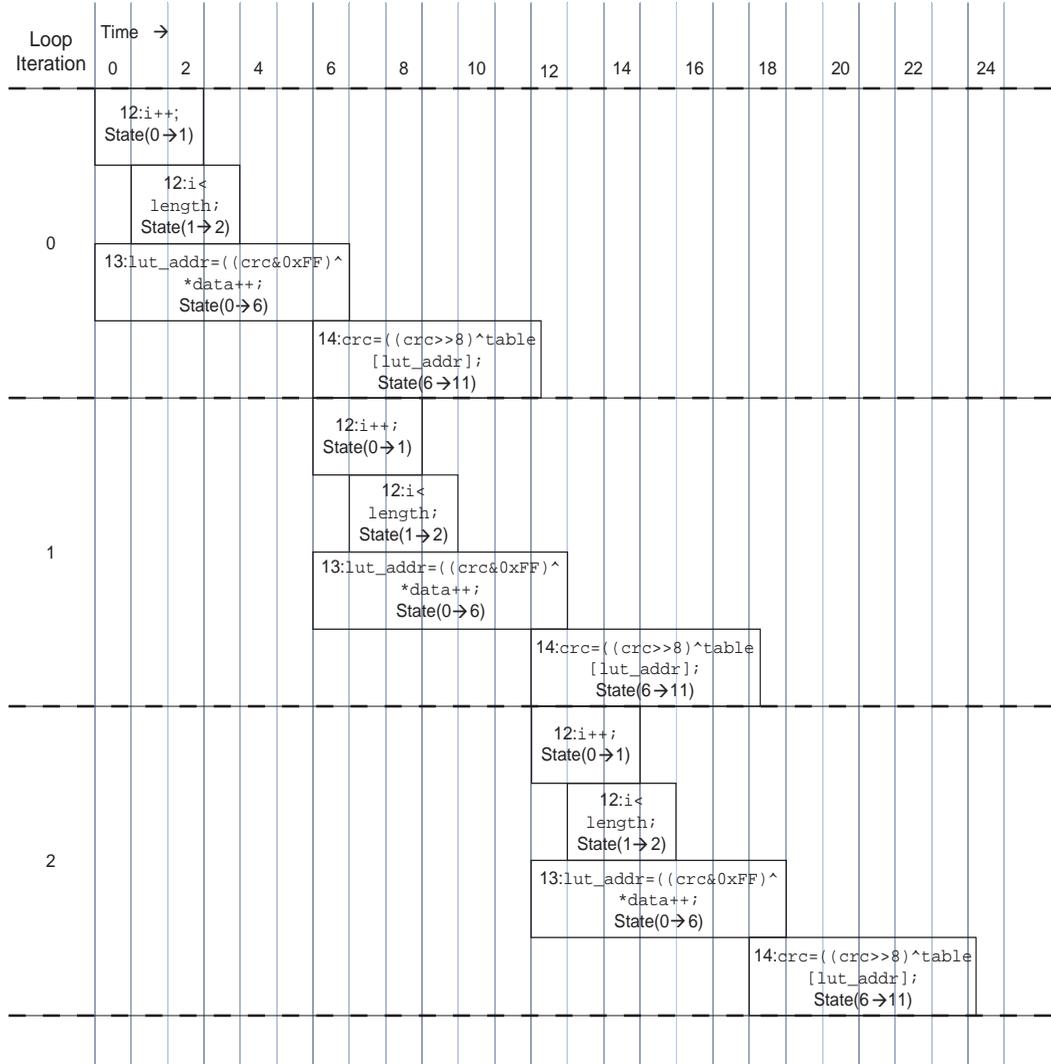
Scheduling Information Per State

As an alternative to viewing the loop scheduling information per assignment, you can look at **Scheduling information per state**.

This section shows you the assignments that occur during each state. This is the opposite of the previous section however it can sometimes be easier to interpret the information presented. When Example 4–2 is compiled, the states are mapped and presented in this section of the C2H build report, as shown in Figure 4–9.

Figure 4–9. CRC Scheduling Per State

Figure 4–10. Schedule by State

Time	Iteration 0	Iteration 1

0	(State 0)
	i++;
	lut_addr=(crc&0xFF)^*data++;	

1	(State 1)
	i++;
	(i<length);
	lut_addr=(crc&0xFF)^*data++;	

2	(State 2) (i<length); lut_addr=(crc&0xFF)^*data++;	CPLI
3	(State 3) lut_addr=(crc&0xFF)^*data++;	
4	(State 4) lut_addr=(crc&0xFF)^*data++;	
5	(State 5) lut_addr=(crc&0xFF)^*data++;	

6	(State 6)
	lut_addr=(crc&0xFF)^*data++;
	crc=(crc>>8)^table[lut_addr];
		(State 0)
		i++;
		lut_addr=(crc&0xFF)^*data++;

7	(State 7)
	crc=(crc>>8)^table[lut_addr];
		(State 1)
		i++;
		(i<length);
		lut_addr=(crc&0xFF)^*data++;

8	(State 8) crc=(crc>>8)^table[lut_addr];	(State 2) (i<length); lut_addr=(crc&0xFF)^*data++;
9	(State 9) crc=(crc>>8)^table[lut_addr];	(State 3) lut_addr=(crc&0xFF)^*data++;
10	(State 10) crc=(crc>>8)^table[lut_addr];	(State 4) lut_addr=(crc&0xFF)^*data++;
11	(State 11) crc=(crc>>8)^table[lut_addr];	(State 5) lut_addr=(crc&0xFF)^*data++;

12		(State 6)
		lut_addr=(crc&0xFF)^*data++;
		crc=(crc>>8)^table[lut_addr];

13		(State 7) crc=(crc>>8)^table[lut_addr];
14		(State 8) crc=(crc>>8)^table[lut_addr];
15	CPLI	(State 9) crc=(crc>>8)^table[lut_addr];
16		(State 10) crc=(crc>>8)^table[lut_addr];
17		(State 11) crc=(crc>>8)^table[lut_addr];

In the case of Example 4–2, a total of 12 states is required to schedule the loop. Figure 4–10 outlines the same information presented in Figure 4–8, organizing it by state to show how multiple states execute concurrently:

Further Reading

For more advice on using the information presented in the C2H view, refer to the *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*.

Creating an Accelerator from the Command Line

The Nios II software build tools support the Nios II C2H Compiler on the command line with the **nios2-c2h-generate-makefile** command. This command creates a C2H makefile fragment that specifies all accelerators and accelerator options for an application.

☞ C2H Compiler projects created on the command line cannot be imported into the Nios II Software Build Tools for Eclipse. Altera recommends creating new C2H accelerators with the Nios II IDE.

The **nios2-c2h-generate-makefile** command usage is as follows:

```
nios2-c2h-generate-makefile \
    --sopcinfo=<SOPC Builder System File> [OPTIONS]
```

☞ This command creates a new **c2h.mk** each time it is called, overwriting the existing **c2h.mk**.

Table 5–1 lists the command line arguments for the **nios2-c2h-generate-makefile** command.

Table 5–1. nios2-c2h-generate-makefile Command Line Arguments (Part 1 of 2)	
Argument Name	**Meaning**
`--sopcinfo`	The path to the SOPC Builder system file (**.sopcinfo**).
`--app-dir`	Directory to place the application Makefile and ELF. If omitted, it defaults to the current directory.

`--accelerator`	Specifies a function to be accelerated. This argument accepts up to four comma-separated values:
	● Target function name
	● Target function file
	● Link hardware accelerator instead of original software. 1 or 0. Defaults to 1.
	● Flush data cache before each call. 1 or 0. Defaults to 1.
	Examples:
	`--accelerator=doDMA,../../DMA.c,1,0`
	`--accelerator=analyze,../../finite.c`

`--enable_quartus`	Building the application compiles the associated Quartus II project. Defaults to 0.

Table 5–1. nios2-c2h-generate-makefile Command Line Arguments (Part 2 of 2)

Argument Name	Meaning
`--analyze_only`	Disables hardware generation, SOPC Builder system generation, and Quartus II compilation for all accelerators in the application. Building the project with this option only updates the report files. Defaults to 0.
`--use_existing_accelerators`	Disables all hardware generation steps. The build behaves as if **c2h.mk** did not exist, with the exception of possible accelerator linking as specified in the `--accelerator` option. Defaults to 0.

Example 5–1 shows a typical **nios2-c2h-generate-makefile** command line.

Example 5–1. nios2-c2h-generate-makefile command line

```
nios2-c2h-generate-makefile \
    --sopcinfo=../../NiosII_stratix_1s40_standard.sopcinfo \
    --app_dir=./ \
    --accelerator=doDMA,DMA.c \
    --accelerator=analyze,../../../finite.c,1,0 \
    --use_existing_accelerators
```

For more detail about **nios2-c2h-generate-makefile**, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

☞ You must use the `--c2h` flag when calling **nios2-app-generate-makefile** in order to make your application with C2H. This flag causes the static C2H make rules to be included in your application makefile. These rules in turn include the **c2h.mk** fragment generated by this command.

For more information about **nios2-app-generate-makefile**, refer to the *Nios II Software Build Tools Reference* chapter of the *Nios II Software Developer's Handbook*.

C2H Performance Metrics

The C2H Compiler produces a detailed report during software compilation. This report shows hardware structure, resource usage, and throughput. Using the build report you can view the resource usage and scheduling information for each accelerated function.

The report is saved as an XML file in the application directory. The name of the report file is *<function_name>*.**prop**, where *<function_name>* is the name of the accelerated function.

For details of the report's contents, refer to "Resources" on page 4–3 and "Performance" on page 4–10 of Chapter 4, Understanding the C2H View.

6. Pragma Reference

Introduction

The C2H Compiler uses pragmas that allow user control of master-slave connections and arbitration shares. This chapter describes these pragmas in detail.

The C language specification dictates that when a compiler implementation encounters a `pragma` directive it does not recognize, the compiler ignores the pragma. By using pragmas, you can write directives to optimize the C2H Compiler results, without making the C code incompatible with other compilers.

Connection Pragma

The C2H Compiler provides a connection pragma that associates a pointer variable with an Avalon-MM slave port, which is typically a memory. A pointer variable can translate to one or more master ports, depending on how many times it is dereferenced in the C code. The connection pragma directs the C2H Compiler to connect all master ports generated for a particular variable to a specific slave port in the SOPC Builder system, reducing arbitration logic.

The connection pragma syntax is as follows:

```
#pragma altera_accelerate connect_variable \
    <function name>/<variable name> to \
    <module>[/<slave name>] [arbitration_share <shares>]
```

<function name> and *<variable name>* are the exact names of the accelerated function and the pointer variable. *<module>* is the exact name of the component instance, as specified in SOPC Builder. *<slave name>* is optional. If provided, *<slave name>* is the exact name of a specific slave port on *<module>*; if not provided, the master port connects to all slave ports on *<module>*. *<shares>* is a positive integer from 1 to 100.

Define the connection pragma in the same file as the function to be accelerated, outside the function body.

To connect a variable's master ports to multiple slave ports, you can use multiple pragmas. If you use the connection pragma for a specific variable, the C2H Compiler connects only the slave ports specified in pragma statements.

Reducing Arbitration Logic

Example 6–1 illustrates use of the connection pragma to connect two master ports for the variable `my_ptr` to the memory module named `onchip_buffer`.

Example 6–1. Pragma Connecting Master Ports to a Slave Port

```
#pragma altera_accelerate connect_variable foo/my_ptr to onchip_buffer

int foo(int *my_ptr)
{
  int x = *my_ptr;
  my_ptr[8] = 23;
}
```

Example 6–2 illustrates using multiple pragmas to connect a pointer variable's master ports to multiple slave ports.

Example 6–2. Pragma Connecting a Master Ports to Multiple Slave Ports

```
#pragma altera_accelerate connect_variable foo/my_ptr to onchip_buffer_0
#pragma altera_accelerate connect_variable foo/my_ptr to ext_ram_bridge
#pragma altera_accelerate connect_variable foo/my_ptr to sdram
#pragma altera_accelerate connect_variable \
                   foo/my_ptr to onchip_buffer_1/s2

int foo(int *my_ptr)
{
  int x = *my_ptr;
  my_ptr[8] = 23;
}
```

In addition to reducing arbitration logic, the connection pragma helps the C2H Compiler determine if two pointers overlap. If the memory connections for two separate variables are mutually exclusive, the compiler concludes that the pointers are never dependent on each other. For more information, refer to "Pointer Aliasing" on page 3–32.

Optimizing Sequential Memory Access with Arbitration Shares

Arbitration shares benefit memories that have higher efficiency when accessed sequentially, such as SDRAM. You can use arbitration shares to reduce interruptions to sequences of transfers with a specific slave. For example, if a master-slave connection has an arbitration share value of ten, then the arbitrator grants at least ten consecutive transfers to the

master port when it begins a sequence of transfer requests. The arbitration share of a shared Avalon-MM master port is the sum of the arbitration shares of all master-slave pairs associated with the master port.

The connection pragma with additional terms for arbitration share is defined as follows, where *<shares>* is a positive integer from 1 to 100:

```
#pragma altera_accelerate connect_variable \
    <function name>/<variable name> to \
    <module>[/<slave name>] arbitration_share <shares>
```

Example 6–3 connects the variable x in function myfunc to the memory module named sdram with an arbitration share of 16.

Example 6–3. Pragma Specifying Arbitration Share

```
#pragma altera_accelerate connect_variable myfunc/x to sdram \
    arbitration_share 16
```

Flow Control Pragma

Avalon-MM transfers with flow control force a master port to obey flow control signals controlled by a slave port. For example, a slave FIFO might assert flow control signals to prevent write transfers when the FIFO memory is full. The C2H Compiler provides a flow control pragma which enables flow control for all master ports related to a specific pointer variable.

The flow control pragma is defined as follows:

```
#pragma altera_accelerate \
  enable_flow_control_for_pointer
  <function name>/<variable name>
```

The flow control pragma must be placed outside the function to accelerate in the same file. *<function name>* and *<variable name>* are the exact names of the accelerated function and the pointer variable.

☞   Using the flow control pragma might result in an accelerator that functions differently from the original function running on the Nios II processor.

👣   For details about Avalon-MM flow control, refer to the *Avalon Memory-Mapped Interface Specification*.

Interrupt Pragma

To use a hardware accelerator in interrupt mode, add the following line to your function source code:

```
#pragma altera_accelerate \
    enable_interrupt_for_function <function name>
```

At the next software compilation, the C2H Compiler creates a new header file containing all the macros needed to use the accelerator and service the interrupts it generates.

This pragma causes the function (which is assumed to be a top-level accelerated function, not an accelerated subfunction) to be an interrupt-mode accelerator. Specifically, the following things change:

■ The accelerator's control slave has an IRQ signal, which is asserted every time the function has completed execution.
■ The polling loop in the generated driver file is removed. When the function is called, the CPU immediately returns after launching the accelerator.
■ A header file is generated, providing macros and definitions required for you to write an ISR. The macros are summarized in Table 6–1.

Table 6–1. C2H Accelerator Interrupt Macros

Purpose	Macro Name
Return value	`ACCELERATOR_<Project Name>_<Function Name>_GET_RETURN_VALUE()`
Interrupt clear	`ACCELERATOR_<Project Name>_<Function Name>_CLEAR_IRQ()`
Check status	`ACCELERATOR_<Project Name>_<Function Name>_BUSY()`

An example of this header file is shown in Example 6–4 for an accelerated function called `coprocess()` in a Nios II IDE project called **my_project**. The file is generated in *<Project Path>/<Configuration>*, where *<Project Path>* is the software project directory, and *<Configuration>* is the project configuration name (Release or Debug). The file name is **ACCELERATOR_*<Project Name>*_*<Function Name>*_IRQ.h**, where *<Project Name>* is the name of the project (usually the same as *<Project Path>)*, and *<Function Name>* is the name of the function you are accelerating.

Example 6–4. Interrupt Header File

```
#ifndef ALT_C2H_COPROCESS_IRQ_H

#define ALT_C2H_COPROCESS_IRQ_H
#include "io.h"

#include "c2h_accelerator_base_addresses.h"

#define ACCELERATOR_MY_PROJECT_COPROCESS_GET_RETURN_VALUE() \
    (( int ) IORD_32DIRECT ( \
            ACCELERATOR_MY_PROJECT_COPROCESS_CPU_INTERFACE0_BASE, \
            (1*sizeof(int))))
#define ACCELERATOR_MY_PROJECT_COPROCESS_CLEAR_IRQ() \
    ( IOWR_32DIRECT ( \
            ACCELERATOR_MY_PROJECT_COPROCESS_CPU_INTERFACE0_BASE, \
            (0*sizeof(int)), 0))
#define ACCELERATOR_MY_PROJECT_COPROCESS_BUSY() \
    ( IORD_32DIRECT ( \
            ACCELERATOR_MY_PROJECT_COPROCESS_CPU_INTERFACE0_BASE, \
            ((0*sizeof(int))) & 1) ^ 1)

#endif /* ALT_C2H_COPROCESS_IRQ_H */
```

The hardware accelerator does not have an IRQ level so you must open the system in SOPC Builder and manually assign this value. After assigning the IRQ level press the generate button because this is a change outside of the Nios II IDE. You only have to do this manual step once. In addition, you can use the `accelerate_my_project_coprocess_busy` macro in a non-interrupt based system in which the user code pulls for the `done` bit, rather than using the automatically generated C wrapper.

Refer to the *Exception Handling* chapter of the *Nios II Software Developer's Handbook* for more information about creating interrupt service routines.

Unshare Pointer Pragma

As discussed in *"Resource Sharing"* in the *C-to-Hardware Mapping Reference* chapter, the C2H compiler automatically shares a master port for multiple pointer dereference operations that connect to the same slave port or group of slave ports. In certain cases, this causes a reduction in performance. For example, in Example 6–5 both `ptr_a` and `ptr_b` must be connected to both onchip_memory_0 and onchip_memory_1, but they never access the same memory at the same time. By default, the C2H compiler will attempt to share a single master between `ptr_a` and `ptr_b`, preventing these dereference operations from being scheduled concurrently and possibly degrading performance.

Example 6–5. Automatically Shared Master Port

```
#pragma altera_accelerate connect_variable ptr_a to onchip_memory_0
#pragma altera_accelerate connect_variable ptr_a to onchip_memory_1

#pragma altera_accelerate connect_variable ptr_b to onchip_memory_0
#pragma altera_accelerate connect_variable ptr_b to onchip_memory_1

if (x)
{
   ptr_a = ONCHIP_MEMORY_1_BASE;
   ptr_b = ONCHIP_MEMORY_2_BASE;
}
else
{
   ptr_a = ONCHIP_MEMORY_2_BASE;
   ptr_b = ONCHIP_MEMORY_1_BASE;
}
/* ... perform some dereference operations with ptr_a and ptr_b ... */
```

To overcome this problem, use the `unshare_pointer` pragma, which instructs the compiler to always optimize for speed, and never defer operations for the purposes of resource scheduling. The syntax is as follows, for a pointer `my_ptr` in function `my_func`:

```
#pragma altera_accelerate unshare_pointer my_func/my_ptr
```

Introduction

The Nios II C-to-Hardware Acceleration (C2H) Compiler supports a large subset of the ANSI C language as described in Chapter 5 and Chapter 6 of the *ISO/IEC 9899:1999(E) Specification*. The current Nios II C2H Compiler does not support the C++ programming language or the library functions described in Chapter 7 of the *ISO/IEC 9899:1999(E) Specification*.

This chapter describes Nios II C2H Compiler restrictions, including unsupported ANSI C language syntax, semantics, and constraints.

Language

This section refers to Chapter 6 of the *ISO/IEC 9899:1999(E) Specification*. Section and paragraph numbers from the *ISO/IEC 9899:1999(E) Specification* are cited in parentheses.

Declarations

The C2H Compiler supports the majority of data types used in the C programming language. The following sections describe C2H restrictions on C declarations.

Unsupported Types (Section 6.7.2, Paragraph 1)

The following types are not supported by the C2H Compiler:

- `float` - section 6.3.1.5
- `double` - section 6.3.1.5
- `_Complex` – section 6.3.1.7
- `_Bool` – section 6.3.1.2
- `_Imaginary` – section 6.3.1.7

The following types are supported if specific conditions are met:

- Floating constants are supported only after casting to a supported type.

   For example, the following code casts π to an integer constant:

   ```
   constant int pi - (int) 3.142957142957;
   ```

■ Escape character sequences in character constants are supported if they are used as string literals rather than character constants.

The following declaration is supported because it employs a string literal:

```
char *newline = "\n";
```

The following declaration is not supported because it employs a character constant:

```
char newline = '\n';
```

■ Composite concatenation is not supported in initialization statements.

The C2H Compiler does not support the initialization statement which concatenates two strings, such as the following:

```
char s[] = "this" " string";
```

The following declaration is supported:

```
char s[] = "this string";
```

Bit-Field Declarations (Section 6.7.2.1)

Structs are supported; however, bit-field declarations used in packed structs are not supported.

For example, the following packed struct defining a 3-bit field is not supported:

```
struct my_struct
{
  unsigned int tbits:3;
} ms;
ms.tbits = 0xFF;
```

You can use a mask of appropriate length to identify the significant bits:

```
struct my_struct
{
  unsigned int tbits;
} ms;
ms.tbits = 0xFF & 7;
```

Array Initialization (Section 6.7.8, Paragraph 3)

Array initialization is supported; however, the array size must be established before initializing individual elements of the array.

The following code, which assigns a single element of an array without establishing its size, is not supported:

```
int a[] = {[5]=2};
```

The following initialization, which establishes the array size before initializing a single element, is supported:

```
int a[6]; /* establish array size */
a[5]=2; /* assign element 5 */
```

It is also possible to initialize the entire array with a single statement, as follows:

```
int a[6]={0,0,0,0,0,2}; /* init a[]*/
int b[]={0,0,0,0,0,2}; /* init b[]*/
```

Delayed Declaration

The C2H Compiler does not support delayed declaration of variables.

For example, the following code, which first declares an array of unspecified size and later provides the size, is not supported:

```
int a[];
int a[20];
```

You can establish the size of the array when it is declared:

```
int a[20];
```

Expressions

The C2H Compiler does not support the following C operators.

Unary Operator (Address Operator) (Section 6.5.3.2, Paragraph 1)

The unary & operator used as an address operator is not supported.

The following example, which passes the address of a as the argument to the function analyze(), is not supported:

```
void foo()
{
  int a=0;
  int c=analyze(&a);
}
int analyze( int * p );
```

You can substitute the following code, which initializes the pointer outside of the accelerator:

```
int *pa = &a;
int foo()
{
  return analyze(pa);
}
int analyze( int * p );
```

Logical Expressions

All expressions in logical operations are evaluated. The parser does not stop evaluation if the first expression of a compound statement is true.

For example, in the following statement, both i and j decrement, even if i is nonzero:

```
if (i-- || j--)
```

In the following code fragment, the C2H Compiler evaluates the divide by 0, which causes an error:

```
int i = 2 || 1 / 0 ;
```

Functions

The following sections list restrictions on functions.

Function Arguments

This section lists restrictions on arguments to functions.

Composite Types (Section 6.2.7)
The C2H Compiler does not support function arguments of different but compatible types in function declarations that refer to the same entity.

For example, the following code shows two definitions of `my_func()` with compatible arguments, which is not supported:

```
int my_func(int (*)(), double (*)[3]);
int my_func(int (*)(char *), double (*)[]);
```

These two declarations can be combined into a single composite function prototype that is compatible with the previous declarations:

```
int my_func(int (*)(char *), double (*)[3]);
```

Ellipsis (Section 6.7.5.3, Paragraph 9)
The ellipsis function argument is not supported.

The following function includes an incompletely specified parameter list, which is not supported:

```
void foo(int a, short b,...);
```

The previous example can be replaced with a function declaration that completely specifies the parameter list:

```
void foo(int a, short b, char *a);
```

Struct and Union (Section 6.7.2.1, Paragraph 1)
The C2H Compiler does not support passing struct or union arguments to a function by value. There are two ways to include structs or union types in the C source:

■ Pass a pointer to a struct or union as an argument to the function.
■ Define the struct or union globally outside the accelerated function.

The following code, which passes a struct `MyStruct` as an argument, is not supported:

```
void doDMA(struct s MyStruct);
```

The previous example can be replaced with code that defines the struct `s` outside of the function call:

```
struct s MyStruct;
void doDMA();
```

Function Pointers (Section 6.7.5.3, Paragraph 8)
Function pointers are supported if used to point to functions that exist inside the hardware accelerator. The C2H Compiler does not support function pointers used as input or output arguments to an accelerator.

Example 7–1 defines three sub-functions, sub_plus_one(),
sub_plus_two() and sub_plus_three(). A fourth function,
c2h_fnc(), returns a pointer to one of the three sub-functions,
depending on the value of the input argument one_two_or_three. The
C2H Compiler supports this use of function pointers as long as all four
functions are part of the hardware accelerator.

Example 7–1. Use of Function Pointers Inside the C2H Accelerator

```
int sub_plus_one(int in)
{
  return in + 1;
}

int sub_plus_two(int in)
{
  return in + 2;
}

int sub_plus_three(int in)
{
  return in + 3;
}

int c2h_fnc(int in, int one_two_or_three)
{
  int (*fp)(int);

  fp = ((one_two_or_three == 3) ? sub_plus_three :
       ((one_two_or_three == 2) ? sub_plus_two :
       sub_plus_one));

  return fp(in);
}
```

Function Argument Types (Section 6.9.1, Paragraph 13)
Functions must define the types of the arguments passed.

For example, the following declaration of foo() is not supported
because the arguments a, b, and c are not typed:

```
void foo(a,b,c);
```

The following function declaration which defines the argument types
inside the function argument list is supported:

```
void foo(char a, char b, char c);
```

Function Prototypes (Section 6.9.1, Paragraph 14)

A function prototype cannot be encapsulated within a function.

For example, the following code is not supported:

```
void doDMA(int a)
{
  void analyze(int i);
  ...
}
```

The C2H Compiler supports separate declarations of functions, as follows:

```
void analyze(int i);
void doDMA(int a)
{
  ...
}
```

Recursive Function Calls (Section 6.5.2.2, Paragraph 11)

Recursive function calls are not supported.

Example 7–2 shows an unsupported recursive implementation of the factorial function.

Example 7–2. Recursive Implementation of Factorial Function

```
int factorial(int x)
{
  if (x>1) return factorial(x-1) * x;
  else return x;
}
```

You can replace recursive functions with equivalent code that implements the function without using recursion. Example 7–3 shows an equivalent implementation of the factorial function without using recursion.

Example 7–3. Nonrecursive Implementation of Factorial Function

```
int factorial(int x)
{
  int tmp = 1, i;
  for (i = 0 ;i<x;i++)
  {
    tmp *= (i+1);
  }
  return tmp;
}
```

Function Specifiers (Section 6.7.4)

The `inline` function specifier is ignored in the C2H design flow. The build process uses the **nios2-elf-gcc** option `-fno-inline`.

Functions Declared Without a Return Type

The C2H compiler does not support functions without an explicitly declared return type. If you are using the implicit `int` return type, declare the return type explicitly. If your function has no return value, declare it as `void`.

Miscellaneous Unsupported Features

The C2H Compiler does not support the features of ANSI C listed in this section.

Goto (Section 6.8.6.1)

The `goto` keyword is not supported.

Identifiers (Section 6.4.2.2)

The predefined identifier `__func__` is not supported.

Trigraph Sequences (Section 5.2.1.1)

The use of trigraph sequences to reduce the standard C character set to the smaller ISO 646 character set is not supported.

The following function call uses unsupported trigraph "??<" in place of "{":

```
int cmpchar(char *c2)
```

```
{
  char *c1 = "??<";
  return (c1!=c2);
}
```

Directives (Section 6.10)

The C2H Compiler does not support the directives listed in this section.

Error Directive (Section 6.10.5)
The error directive is not supported.

```
#error
```

Predefined Macro Names (Section 6.10.8)
All the predefined macros of Section 6.10.8 are not supported. These predefined macros include:

```
__bool_true_false_are_defined
__cplusplus
__DATE__
__FILE__
__LINE__
__STDC_, 6.11.9
__STDC__
__STDC_CONSTANT_MACROS
__STDC_FORMAT_MACROS
__STDC_HOSTED__
__STDC_IEC_559_
__STDC_IEC_559_COMPLEX__
__STDC_ISO_10646__
__STDC_LIMIT_MACROS
__STDC_VERSION__
__TIME__
_Complex_I
_Imaginary_I
_IOFBF
_IOLBF
_IONBF
```

Other Restrictions

The C2H Compiler does not support external subfunctions. You must locate the subfunction in the same source file as the accelerated function. This is because, unlike the #include construct, a C external function reference requires the presence of a linker. The C2H Compiler has no linker.

Additional Information

Referenced Documents

This user guide references the following documents:

- *Quartus II Handbook, volume 4: SOPC Builder*
- *Overview* chapter of the *Nios II Software Developer's Handbook*
- *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*
- *Exception Handling* chapter of the *Nios II Software Developer's Handbook*
- *Using the Nios II Integrated Development Environment* appendix to the *Nios II Software Developer's Handbook*
- *Avalon Memory-Mapped Interface Specification*
- *AN 320: OpenCore Plus Evaluation of Megafunctions*
- *AN 391: Profiling Nios II Systems*
- *Optimizing Nios II C2H Compiler Results* chapter of the *Embedded Design Handbook*
- *Nios II Hardware Development Tutorial*
- *Nios II Software Development Tutorial* available in the Nios II integrated development environment (IDE) help system
- *Accelerating Nios II Systems with the C2H Compiler Tutorial*

Revision History

The table below displays the revision history for the chapters in this user guide.

Chapter	Date	Document Version	Changes Made
1	November 2009	1.6	● C2H Compiler not supported by Nios II Software Build Tools for Eclipse ● C2H Compiler generates little-endian hardware
5			C2H accelerators cannot be imported into the Nios II Software Build Tools for Eclipse
7	November 2008	1.5	Document restriction on implicit function return values.
All	May 2008	1.4	SOPC Builder system file implemented as **.sopcinfo** type (instead of **.sopc** type).
2			Include makefile fragments when copying project.
6			Added `unshare_pointer` pragma.
2	October 2007	1.3	Added details about how to delete IDE project containing accelerator
5			● **nios2-c2h-generate-makefile** accepts an SOPC Builder version 7.0 or later system file.
7			List limitation on external subfunctions
All	May 2007	1.2	Updated for changes between 6.0 and 7.1 ● Pipelined subfunctions. ● Extended resource sharing. ● Support for the software build tools. ● Interrupt generation.
5			Additional chapter on Nios II software build tools
6			Additional chapter on pragmas
7			Move from chapter 5 to chapter 7
4	August 2006	1.1	Additional chapter on C2H view
5			Move from chapter 4 to chapter 5
All	May 2006	1.0	First publication

How to Contact Altera

For the most up-to-date information about Altera products, refer to the following table.

Information Type	Contact *(1)*
Technical support	www.altera.com/support
Technical training	www.altera.com/training custrain@altera.com
Altera literature services	literature@altera.com
Non-technical support (General)	nacomp@altera.com
(Software Licensing)	authorization@altera.com

Note to table:
(1) You can also contact your local Altera sales office or sales representative.

Typographic Conventions

This document uses the typographic conventions shown below.

Visual Cue	Meaning
Bold Type with Initial Capital Letters	Command names, dialog box titles, checkbox options, and dialog box options are shown in bold, initial capital letters. Example: **Save As** dialog box.
bold type	External timing parameters, directory names, project names, disk drive names, filenames, filename extensions, and software utility names are shown in bold type. Examples: f_{MAX}, **\qdesigns** directory, **d:** drive, **chiptrip.gdf** file.
Italic Type with Initial Capital Letters	Document titles are shown in italic type with initial capital letters. Example: *AN 75: High-Speed Board Design.*
Italic type	Variable names are shown in italic type. Example: *<file name>*, *<project name>***.pof** file.
Initial Capital Letters	Keyboard keys and menu names are shown with initial capital letters. Examples: Delete key, the Options menu.
"Subheading Title"	References to sections within a document and titles of on-line help topics are shown in quotation marks. Example: "Typographic Conventions."
Courier type	Signal and port names are shown in lowercase Courier type. Examples: data1, tdi, input. Active-low signals are denoted by suffix n, e.g., resetn. Anything that must be typed exactly as it appears is shown in Courier type. For example: c:\altera\. Also, references to C code are shown in Courier.
1., 2., 3., and a., b., c., etc.	Numbered steps are used in a list of items when the sequence of the items is important, such as the steps listed in a procedure.
■ ● •	Bullets are used in a list of items when the sequence of the items is not important.
☞	The hand points to information that requires special attention.

Visual Cue	Meaning
⚠ CAUTION	A caution calls attention to a condition or possible situation that can damage or destroy the product or the user's work.
⚠ WARNING	A warning calls attention to a condition or possible situation that can cause injury to the user.
↵	The angled arrow indicates you should press the Enter key.
👣	The feet direct you to more information on a particular topic.