5. Optimizing Nios II C2H Compiler Results

ED51005-1.2

The Nios® II C2H Compiler is a powerful tool that generates hardware accelerators for software functions. The C2H Compiler enhances design productivity by allowing you to use a compiler to accelerate software algorithms in hardware. You can quickly prototype hardware functional changes in C, and explore hardware-software design tradeoffs in an efficient, iterative process. The C2H Compiler is well suited to improving computational bandwidth, as well as memory throughput. It is possible to achieve substantial performance gains with minimal engineering effort.

The structure of your C code affects the results you get from the C2H Compiler. Although the C2H Compiler can accelerate most ANSI C code, you might need to modify your C code to meet resource usage and performance requirements. This document describes how to improve the performance of hardware accelerators, by refactoring your C code with C2H-specific optimizations.

Prerequisites

To make effective use of this chapter, you should be familiar with the following topics:

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

To familiarize yourself with the basics of the C2H Compiler, refer to the *Nios II C2H Compiler User Guide*, especially the *Introduction to the C2H Compiler* and *Getting Started Tutorial* chapters. To learn about defining, generating, and compiling Nios II systems, refer to the *Nios II Hardware Development Tutorial*. To learn about Nios II software projects, refer to the *Nios II Software Development Tutorial*, available in the Nios II IDE help system. To learn about data caching, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Cost and Performance

When writing C code for the C2H Compiler, you can optimize it relative to several optimization criteria. Often you must make tradeoffs between these criteria, which are listed below:

- Hardware cost—C2H accelerators consume hardware resources such as LEs, multipliers, and on-chip memory. This document uses the following terms to describe the hardware cost of C language constructs:
  - Free—the construct consumes no hardware resources.
  - Cheap—the construct consumes few hardware resources. The acceleration obtained is almost always worth the cost.
  - Moderate—the construct consumes some hardware resources. The acceleration obtained is usually worth the cost.
  - Expensive—the construct consumes substantial hardware resources. The acceleration obtained is sometimes worth the cost, depending on the nature of the application.
- Algorithm performance—A C2H accelerator performs the same algorithm as the original C software executed by a Nios II processor. Typically the accelerator uses many fewer clock cycles than the software implementation. This document describes the algorithm performance of C constructs as fast or slow. The concept of algorithm performance includes the concepts of latency and throughput. These concepts are defined under "Cycles Per Loop Iteration (CPLI)" on page 5–12.
- Hardware performance impact—Certain C language constructs, when converted to logic by the C2H Compiler, can result in long timing paths that can degrade f_{MAX} for the entire system or for the clock domain containing the C2H accelerator. This document clearly notes such situations and offers strategies for avoiding them.

Overview of the C2H Optimization Process

It is unlikely that you can meet all of your optimization goals in one iteration. Instead, plan on making the one or two optimizations that appear most relevant to your cost and performance issues. When you profile your system with the optimized accelerator, you can determine whether further optimizations are needed, and then you can identify the next most important optimization issue to address. By optimizing your accelerator one step at a time, you apply only the optimizations needed to achieve your goals.

Getting Started

The most important first step is to decide on a clear performance goal. Depending on your application, you may require a specific performance level from your algorithm. If you have already selected a target device, and if other hardware in the system is well defined, you might have specific hardware cost limitations. Alternatively, if you are in early phases of development, you might only have some general guidelines for conserving hardware resources. Finally, depending on your design needs and the f_{MAX} of your existing design, you might be concerned with possible f_{MAX} degradation. Refer to "Meeting Your Cost and Performance Goals" on page 5–3 for more information about cost and performance criteria.

The next step is to develop your algorithm in C, and, if possible, test it conventionally on the Nios II processor. This step is very helpful in establishing and maintaining correct functionality. If the Nios II processor is not fast enough for in-circuit testing of your unaccelerated algorithm, consider simulation options for testing.

When you are confident of your algorithm's correctness, you are ready to accelerate it. This first attempt provides a set of baseline acceleration metrics. These metrics help you assess the overall success of the optimization process.

Altera recommends that you maintain two copies of your algorithm in parallel: one accelerated and the other unaccelerated. By comparing the results of the accelerated and unaccelerated algorithms, you immediately discover any errors which you might inadvertently introduce while optimizing the code.

Iterative Optimization

The iteration phase of C2H Compiler optimization consists of these steps:

1. Profile your accelerated system.
2. Identify the most serious performance bottleneck.
3. Identify an appropriate optimization from the "Optimization Techniques" on page 5–14 section.
4. Apply the optimization and rebuild the accelerated system.

For instructions on profiling Nios II systems, refer to *AN391: Profiling Nios II Systems*.

Meeting Your Cost and Performance Goals

Having a clear set of optimization goals helps you determine when to stop optimization. Each time you profile your accelerated system, compare the results with your goals. You might find that you have reached your cost and performance goals even if you have not yet applied all relevant optimizations.

If your optimization goals are flexible, consider keeping track of your baseline acceleration metrics, and the acceleration metrics achieved at each optimization step. You might wish to stop if you reach a point of diminishing returns.

Factors Affecting C2H Results

This section describes key differences in the mapping of C constructs by a C compiler and the Nios II C2H Compiler. You must understand these differences to create efficient hardware.

C code originally written to run on a processor does not necessarily produce efficient hardware. A C compiler and the Nios II C2H Compiler both use hardware resources such as adders, multipliers, registers, and memories to execute the C code. However, while a C compiler assumes a sequential model of computing, the C2H Compiler assumes a concurrent model of computing. A C compiler maps C code to instructions which access shared hardware resources. The C2H Compiler maps C code to one or more state machines which access unique hardware resources. The C2H Compiler pipelines the computation as much as possible to increase data throughput. Example 5–1 illustrates this point.

Example 5–1. Pipelined Computation

```
int sumfunc(int a, int b, int c, int d)
{
   int sum1 = a + b;
   int sum2 = c + d;
   int result = sum1 + sum2;
   return result;
}
```

The `sumfunc()` function takes four integer arguments and returns their sum. A C compiler maps the function to three add instructions sharing one adder. The processor executes the three additions sequentially. The C2H Compiler maps the function to one state machine and three adders. The accelerator executes the additions for `sum1` and `sum2` concurrently, followed by the addition for `result`. The addition for `result` cannot execute concurrently with the `sum1` and `sum2` additions because of the data dependency on the `sum1` and `sum2` variables.

Different algorithms require different C structures for optimal hardware transformation. This chapter lists possible optimizations to identify in C code. Each C scenario describes the best methods to refactor the C code. The "Optimization Techniques"section discusses how to address the following potential problem areas:

- Memory Accesses and Variables
- Arithmetic and Logical Operations
- Statements
- Control Flow
- Subfunction Calls
- Resource Sharing
- Data Dependencies
- Memory Architecture

Memory Accesses and Variables

Memory accesses can occur when your C code reads or writes the value of a variable. Table 5–1 provides a summary of the key differences in the mapping of memory accesses between a C compiler and the C2H Compiler.

A C compiler generally allocates many types of variables in your data memory. These include scalars, arrays, and structures that are local, static, or global. When allocated in memory, variables are relatively cheap due to the low cost per bit of memory (especially external memory) and relatively slow due to the overhead of load or store instructions used to access them. In some situations, a C compiler is able to use processor registers for local variables. When allocated in processor registers, these variables are relatively fast and expensive.

The C2H Compiler allocates local scalar variables in registers implemented with logic elements (LEs), which have a moderate cost and are fast.

A C compiler maps pointer dereferences and array accesses to a small number of instructions to perform the address calculation and access to your data memory. Pointer dereferences and array accesses are relatively cheap and slow.

The C2H Compiler maps pointer dereferences and array accesses to a small amount of logic to perform the address calculation and creates a unique Avalon® Memory-Mapped (Avalon-MM) master port to access the addressed memory. This mapping is expensive due to the logic required to create an Avalon-MM master port. It is slow or fast depending on the type of memory connected to the port. Local arrays are fast because the C2H Compiler implements them as on-chip memories.

Table 5–1. Memory Accesses

C Construct	C Compiler Implementation	C2H Implementation
Local scalar variables	Allocated in memory (cheap, slow) or allocated in processor registers (expensive, fast)	Allocated in registers based on logic elements (LEs) (moderate cost, fast)
Uninitialized local array variables	Allocated in memory (cheap, slow)	Allocated in on-chip memory. (expensive, fast)
Initialized local array variables	Allocated in memory (cheap, slow)	Allocated in memory (cheap, slow)
All other types of variables	Allocated in memory (cheap, slow)	Allocated in memory (cheap, slow)
Pointer dereferences and nonlocal array accesses	Access normal data memory (cheap, slow)	Avalon-MM master port (expensive, slow or fast)

Arithmetic and Logical Operations

Table 5–2 provides a summary of the key differences in the mapping of arithmetic and logical operations between a C compiler and the C2H Compiler.

A C compiler maps arithmetic and logical operations into one or more instructions. In many cases, it can map them to one instruction. In other cases, it might need to call a function to implement the operation. An example of the latter occurs when a Nios II processor that does not have a hardware multiplier or divider performs a multiply operation.

The C2H Compiler implements the following logical operations simply as wires without consuming any logic at all.

- Shifts by a constant
- Multiplies and divides by a power of two constant
- Bitwise `AND`s and `OR`s by a constant

As a result, these operations are fast and free. The following is an example of one of these operations:

```
int result = some_int >> 2;
```

A C compiler maps this statement to a right shift instruction. The C2H Compiler maps the statement to wires that perform the shift.

Table 5–2. Arithmetic and Logical Operations

C Construct	C Compiler Implementation	C2H Implementation
Shift by constant or multiply or divide by power of 2 constant. *(1)* Example: `y = x/2;`	Shift instruction (cheap, fast)	Wires (free, fast)
Shift by variable Example: `y = x >> z;`	Shift instruction (cheap, fast)	Barrel shifter (expensive, fast)
Multiply by a value that is not a power of 2 (constant or variable Example: `y = x * z;`	Multiply operation (cheap, slow)	If the Quartus II software can produce an optimized multiply circuit (cheap, fast); otherwise a multiply circuit (expensive, fast)
Divide by a value that is not a power of 2 (constant or variable) Example: `y = x/z;`	Divide operation (cheap, slow)	Divider circuit (expensive, slow)
Bitwise AND or bitwise OR with constant Example: `y = x | 0xFFFF;`	AND or OR instruction (cheap, fast)	Wires (free, fast)
Bitwise AND or bitwise OR with variable Example: `y = x & z;`	AND or OR instruction (cheap, fast)	Logic (cheap, fast)

Notes to Table 5–2:

(1) Dividing by a *negative* power of 2 is expensive.

Statements

A C compiler maps long expressions (those with many operators) to instructions. The C2H Compiler maps long expressions to logic which could create a long timing path. The following is an example of a long expression:

```
int sum = a + b + c + d + e + f + g + h;
```

A C compiler creates a series of add instructions to compute the result. The C2H Compiler creates several adders chained together. The resulting computation has a throughput of one data transfer per clock cycle and a latency of one cycle.

A C compiler maps a large function to a large number of instructions. The C2H Compiler maps a large function to a large amount of logic which is expensive and potentially degrades f_{MAX}. If possible, remove from the function any C code that does not have to be accelerated.

A C compiler maps mutually exclusive, multiple assignments to a local variable as store instructions or processor register writes, which are both relatively cheap and fast. However, the C2H Compiler creates logic to multiplex between the possible assignments to the selected variable. Example 5–2 illustrates such a case.

Example 5–2. Multiple Assignments to a Single Variable

```
int result;
if (a > 100)
{
   result = b;
}
else if (a > 10)
{
   result = c;
}
else if (a > 1)
{
   result = d;
}
```

A C compiler maps this C code to a series of conditional branch instructions and associated expression evaluation instructions. The C2H Compiler maps this C code to logic to evaluate the conditions and a three-input multiplexer to assign the correct value to `result`. Each assignment to `result` adds another input to the multiplexer. The assignments increase the amount of the logic, and might create a long timing path. Table 5–3 summarizes the key differences between the C compiler and C2H Compiler in handing C constructs.

Table 5–3. Statements

C Construct	C Compiler Implementation	C2H Implementation
Long expressions	Several instructions (cheap, slow)	Logic (cheap, degrades f_{MAX})
Large functions	Many instructions (cheap, slow)	Logic (expensive, degrades f_{MAX})
Multiple assignments to a local variable	Store instructions or processor register writes (cheap, fast)	Logic (cheap, degrades f_{MAX})

Control Flow

Table 5–4 provides a summary of the differences in the mapping of control flow between a C compiler and the C2H Compiler.

If Statements

The C2H compiler maps the expression of the `if` statement to control logic. The statement is controlled by the expression portion of the `if` statement.

Loops

Loops include `for` loops, `do` loops, and `while` loops. A C compiler and the C2H Compiler both treat the expression evaluation part of a loop just like the expression evaluation in an `if` statement. However, the C2H Compiler attempts to pipeline each loop iteration to achieve a throughput of one iteration per cycle. Often there is no overhead for each loop iteration in the C2H accelerator, because it executes the loop control concurrently with the body of the loop. The data and control paths pipelining allows the control path to control the data path. If the control path (loop expression) is dependent on a variable calculated within the loop, the throughput decreases because the data path must complete before control path can allow another loop iteration.

The expression, `while (++a < 10) { b++ };` runs every cycle because there is no data dependency. On the other hand, `while (a < 10) { a++ };` takes 2 cycles to run because the value of `<a>` is calculated in the loop.

A C compiler maps `switch` statements to the equivalent `if` statements or possibly to a jump table. The C2H Compiler maps `switch` statements to the equivalent `if` statements.

Table 5–4. Control Flow

C Construct	C Compiler Implementation	C2H Implementation
If statements	A few instructions (cheap, slow)	Logic (cheap, fast)
Loops	A few instructions of overhead per loop iteration (cheap, slow)	Logic (moderate, fast)
Switch statements	A few instructions (cheap, slow)	Logic (moderate, fast)
Ternary operation	A few instructions (cheap, slow)	Logic (cheap, fast)

Subfunction Calls

A C compiler maps subfunction calls to a few instructions to pass arguments to or from the subfunction and a few instructions to call the subfunction. A C compiler might also convert the subfunction into inline code. The C2H Compiler maps a subfunction call made in your top-level accelerated function into a new accelerator. This technique is expensive, and stalls the pipeline in the top-level accelerated function. It might result in a severe performance degradation.

However, if the subfunction has a fixed, deterministic execution time, the outer function attempts to pipeline the subfunction call, avoiding the performance degradation. In Example 5–3, the subfunction call is pipelined.

Example 5–3. Pipeline Stall

```
int abs(int a)
{
   return (a < 0) ? –a : a;
}
int abs_sum(int* arr, int num_elements)
{
   int i;
   int result = 0;
   for (i = 0; i < num_elements; i++)
   {
      result += abs(*arr++);
   }
   return result;
}
```

Resource Sharing

By default, the C2H Compiler creates unique instances of hardware resources for each operation encountered in your C code. If this translation consumes too many resources, you can change your C code to share resources. One mechanism to share resources is to use shared subfunctions in your C code. Simply place the code to be shared in a subfunction and call it from your main accelerated function. The C2H Compiler creates only one instance of the hardware in the function, shared by all function callers.

Example 5–4 uses a subfunction to share one multiplier between two multiplication operations.

Example 5–4. Shared Multiplier

```
int mul2(int x, int y)
{
   return x * y;
}
int muladd(int a, int b, int c, int d)
{
   int prod1 = mul2(a, b);
   int prod2 = mul2(c, d);
   int result = prod1 + prod2;
   return result;
}
```

Data Dependencies

A data dependency occurs when your C code has variables whose values are dependent on the values of other variables. Data dependency prevents a C compiler from performing some optimizations which typically result in minor performance degradation. When the C2H Compiler maps code to hardware, a data dependency causes it to schedule operations sequentially instead of concurrently, which can cause a dramatic performance degradation.

The algorithm in Example 5–5 shows data dependency.

Example 5–5. Data Dependency

```
int sum3(int a, int b, int c)
{
   int sum1 = a + b;
   int result = sum1 + c;
   return result;
}
```

The C2H Compiler schedules the additions for `sum1` and `result` sequentially due to the dependency on `sum1`.

Memory Architecture

The types of memory and how they are connected to your system, including the C2H accelerator, define the memory system architecture. For many algorithms, appropriate memory architecture is critical to achieving high performance with the C2H Compiler. With an inappropriate memory architecture, an accelerated algorithm can perform more poorly than the same algorithm running on a processor.

Due to the concurrency possible in a C2H accelerator, compute-limited algorithms might become data-limited algorithms. To achieve the highest levels of performance, carefully consider the best memory architecture for your algorithm and modify your C code accordingly to increase memory bandwidth.

For the following discussion, assume that the initial memory architecture is a processor with a data cache connected to an off-chip memory such as DDR SDRAM.

The C code in Example 5–6 is data-limited when accelerated by the C2H Compiler because the `src` and `dst` dereferences both create Avalon-MM master ports that access the same Avalon-MM slave port. An Avalon-MM slave port can only handle one read or write operation at any given time; consequently, the accesses are interleaved, limiting the throughput to the memory bandwidth.

Example 5–6. Memory Bandwidth Limitation

```
void memcpy(char* dst, char* src, int num_bytes)
{
   while (num_bytes-- > 0)
   {
      *dst++ = *src++;
   }
}
```

The C2H Compiler is able to achieve a throughput of one data transfer per clock cycle if the code is modified and the appropriate memory architecture is available. The changes required to achieve this goal are covered in "Efficiency Metrics" on page 5–11.

Data Cache Coherency

When a C2H accelerator accesses memory, it uses its own Avalon-MM master port, which bypasses the Nios II data cache. Before invoking the accelerator, if the data is potentially stored in cache, the Nios II processor must write it to memory, thus avoiding the typical cache coherency problem. This cache coherency issue is found in any multimaster system that lacks support for hardware cache coherency protocols.

When you configure the C2H accelerator, you choose whether or not the Nios II processor flushes the data cache whenever it calls the accelerated function. If you enable this option, it adds to the overhead of calling the accelerator and causes the rest of the C code on the processor to temporarily run more slowly because the data cache must be reloaded.

You can avoid flushing the entire data cache. If the processor never shares data accessed by the accelerator, it does not need to flush the data cache. However, if you use memory to pass data between the processor and the accelerator, as is often the case, it might be possible to change the C code running on the processor to use uncacheable accesses to the shared data. In this case, the processor does not need to flush the data cache, but it has slower access to the shared data. Alternatively, if the size of the shared data is substantially smaller than the size of the data cache, the processor only needs to flush the shared data before calling the accelerator.

Another option is to use a processor without a data cache. Running without a cache slows down all processor accesses to memory but the acceleration provided by the C2H accelerator might be substantial enough to result in the overall fastest solution.

DRAM Architecture

Memory architectures consisting of a single DRAM typically require modification to maximize C2H accelerator performance. One problem with the DRAM architecture is that memory performance degrades if accesses to it are nonsequential. Because the DRAM has only one port, multiple Avalon-MM master ports accessing it concurrently prevent sequential accesses by one Avalon-MM master from occurring.

The default behavior of the arbiter in an SOPC Builder system is round-robin. If the DRAM controller (such as the Altera Avalon SDRAM controller) can only keep one memory bank open at a time, the master ports experience long stalls and do not achieve high throughput. Stalls can cause the performance of any algorithm accelerated using the C2H Compiler to degrade if it accesses memory nonsequentially due to multiple master accesses or nonsequential addressing.

For additional information about optimizing memory architectures in a Nios II system, refer to the *Cache and Tightly-Coupled Memory* in the *Nios II Software Developer's Handbook*.

Efficiency Metrics

There are several ways to measure the efficiency of a C2H accelerator. The relative importance of these metrics depends on the nature of your application. This section explains each efficiency metric in detail.

Cycles Per Loop Iteration (CPLI)

The C2H report section contains a CPLI value for each loop in an accelerated function. The CPLI value represents the number of clock cycles each iteration of the loop takes to complete once the initial latency is overcome. The goal is to minimize the CPLI value for each loop to increase the data throughput. It is especially important that the innermost loop of the function have the lowest possible CPLI because it executes the most often.

The CPLI value does not take into account any hardware stalls that might occur. A shared resource such as memory stalls the loop if it is not available. If you nest looping structures the outer loops stall and, as a result, reduce the throughput of the outer loops even if their CPLI equals one. The "Optimization Techniques" on page 5–14 section offers methods for maximizing the throughput of loops accelerated with the C2H Compiler.

Optimizations that can help CPLI are as follows:

- Reducing data dependencies
- Reducing the system interconnect fabric by using the `connect_variable` pragma

f_{MAX} is the maximum frequency at which a hardware design can run. The longest register-to-register delay or critical path determines f_{MAX}. The Quartus II software reports the f_{MAX} of a design after each compilation.

Adding accelerated functions to your design can potentially affect f_{MAX} in two ways: by adding a new critical path, or by adding enough logic to the design that the Quartus II fitter fails to fit the elements of the critical path close enough to each other to maintain the path's previous delay. The optimizations that can help with f_{MAX} are as follows:

- Pipelined calculations
- Avoiding division
- Reducing system interconnect fabric by using the `connect_variable` pragma
- Reducing unnecessary memory connections to the Nios II processor

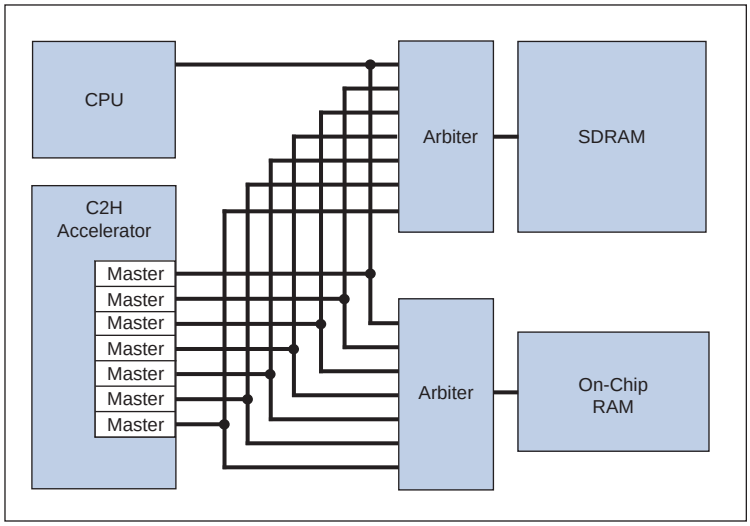
FPGA Resource Usage

Because an accelerated function is implemented in FPGA hardware, it consumes FPGA resources such as logic elements and memory. Sometimes, an accelerator consumes more FPGA resources than is desired or expected. Unanticipated resource usage has the disadvantage of consuming resources that are needed for other logic and can also degrade system f_{MAX}.

Avalon-MM Master Ports

The number of Avalon-MM master ports on the accelerator can heavily influence logic utilization. The C2H report, which the Nios II IDE displays after accelerating the function, reports how many Avalon-MM ports are generated. Multiple master ports can help increase the parallelization of logic when attached to separate memories. However, they have a cost in logic, and can also promote the creation of excessive arbitration logic when connected to the same memory port, as shown in Figure 5–1.

Figure 5–1. Too Many Master Ports



Embedded Multipliers

Multiplication logic is often available on the FPGA as dedicated hardware or created using logic elements. When you use dedicated hardware, be aware that having a large amount of multiplication logic can degrade the routing of your design because the fitter cannot place the multiplier columns to achieve a better fit. When creating multiplication logic from logic elements, be aware that this is expensive in resource usage, and can degrade f_{MAX}.

If one of the operands in a multiplication is a constant, the Quartus II software determines the most efficient implementation. Example 5–7 shows a optimization the Quartus II software might make:

Example 5–7. Quartus II Software Optimization for Multiplication by a Constant

```
/* C code mulitplication by a constant */
c = 7 × a;

/* Quartus II software optimization */
c = (4 × a) + (2 × a) + a;
```

Because the optimized equation includes multiplications by a constant factor of 2, the Quartus II software turns them into 2 shifts plus a add.

Embedded Memory

Embedded memory is a valuable resource for many hardware accelerators due to its high speed and fixed latency. Another benefit of embedded memory is that it can be configured with dual ports. Dual-ported memory allows two concurrent accesses to occur, potentially doubling the memory bandwidth. Whenever your code declares an uninitialized local array in an accelerated function, the C2H Compiler instantiates embedded memory to hold its contents. Use embedded memory only when it is appropriate; do not waste it on operations that do not benefit from its high performance.

Optimization tips that can help reduce FPGA resource use are:

- Using wide memory accesses
- Keeping loops rolled up
- Using narrow local variables

Data Throughput

Data throughput for accelerated functions is difficult to quantify. The Altera development tools do not report any value that directly corresponds to data throughput. The only true data throughput metrics reported are the number of clock cycles and the average number of clock cycles it takes for the accelerated function to complete. One method of measuring the data throughput is to use the amount of data processed and divide by the amount of time required to do so. You can use the Nios II processor to measure the amount of time the accelerator spends processing data to create an accurate measurement of the accelerator throughput.

Before accelerating a function, profile the source code to locate the sections of your algorithm that are the most time-consuming. If possible, leave the profiling features in place while you are accelerating the code, so you can easily judge the benefits of using the accelerator. The following general optimizations can maximize the throughput of an accelerated function:

- Using wide memory accesses
- Using localized data

For more information about profiling Nios II systems, refer to *AN391: Profiling Nios II Systems*.

Optimization Techniques

Pipelining Calculations

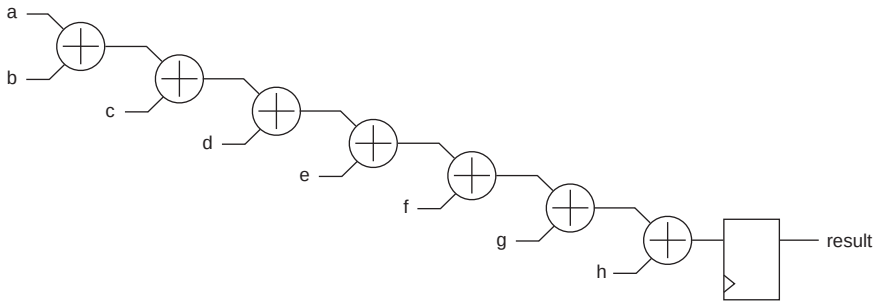
Although condensing multiple mathematical operations to a single line of C code, as in Example 5–8, can reduce the latency of an assignment, it can also reduce the clock speed of the entire design.

Example 5–8. Non-Pipelined Calculation (Lower Latency, Degraded f_{MAX})

```
int result = a + b + c + d + e + f + g + h;
```

Figure 5–2 shows the hardware generated for Example.

Figure 5–2. Non-Pipelined Calculations



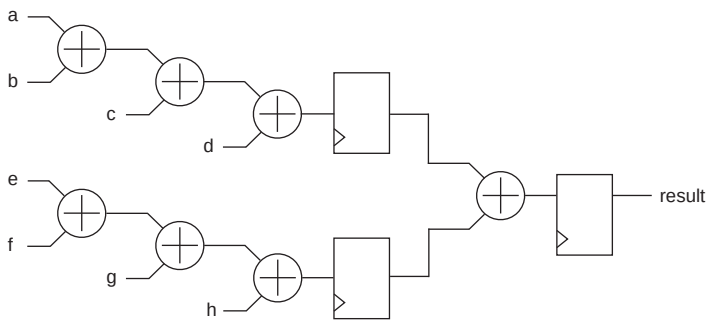
Often, you can break the assignment into smaller steps, as shown in Example 5–9. The smaller steps increase the loop latency, avoiding f_{MAX} degradation.

Example 5–9. Pipelined Calculation (Higher Latency, No f_{MAX} Degradation)

```
int result_abcd = a + b + c + d;
int result_efgh = e + f + g + h;
int result = result_abcd + result_efgh;
```

Figure 5–3 shows the hardware generated for Example 5–9.

Figure 5–3. Pipelined Calculations



Increasing Memory Efficiency

The following sections discuss coding practices that improve C2H performance.

Using Wide Memory Accesses

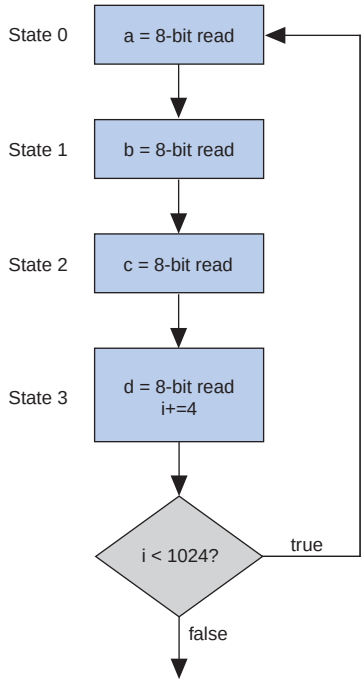
When software runs on a processor with a data cache, byte and halfword accesses to DRAM become full word transfers to and from the cache to guarantee efficient use of memory bandwidth. By contrast, when you make byte and halfword DRAM accesses in a C2H accelerator, as shown in Example 5–10, the Avalon-MM master port connected to the DRAM uses narrow accesses and fails to take advantage of the full data width of the memory.

Example 5–10. Narrow Memory Access (Slower Memory Access)

```
unsigned char narrow_array[1024];
char a, b, c, d;
for(i = 0; i < 1024; i+=4)
{
   a = narrow_array[i];
   b = narrow_array[i+1];
   c = narrow_array[i+2];
   d = narrow_array[i+3];
}
```

Figure 5–4 shows the hardware generated for Example 5–10.

Figure 5–4. Narrow Memory Access



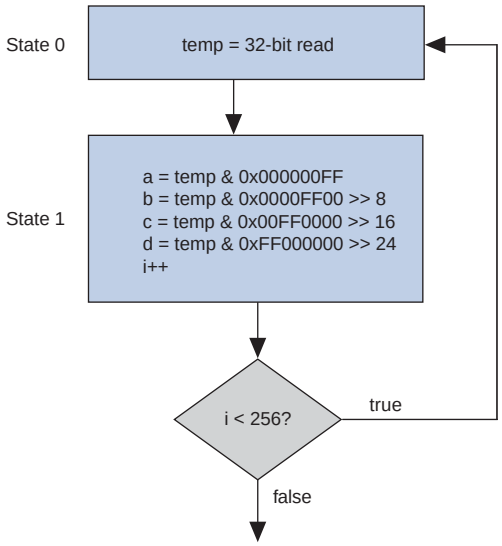
In a situation where multiple narrow memory accesses are needed, it might be possible to combine those multiple narrow accesses into a single wider access, as shown in Example 5–11. Combining accesses results in the use of fewer memory clock cycles to access the same amount of data. Consolidating four consecutive 8-bit accesses into one 32-bit access effectively increases the performance of those accesses by a factor of four. However, to ensure the Nios II `gcc` compiler accesses the data correctly on single-byte boundaries, the C2H Compiler must perform the shift operations shown in Example 5–11 consistently, to ensure that data is accessed according to its alignment in the longer data word.

Example 5–11. Wide Memory Access (Faster Memory Access)

```
unsigned int *wide_array = (unsigned int *) narrow_array;
unsigned int temp;
for(i = 0; i < 256; i++)
{
   temp = wide_array[i];
   a = (char)( temp and 0x000000FF);
   b = (char)( (temp and 0x0000FF00) >> 8);
   c = (char)( (temp and 0x00FF0000) >> 16);
   d = (char)( (temp and 0xFF000000) >> 24);
}
```

Figure 5–5 shows the hardware generated for Example 5–11.

Figure 5–5. Wide Memory Access



Segmenting the Memory Architecture

Memory segmentation is an important strategy to increase the throughput of the accelerator. Memory segmentation leads to concurrent memory access, increasing the memory throughput. There are multiple ways to segment your memory and the method used is typically application specific. Refer to Example 5–12 for the following discussions of memory segmentation optimizations.

Example 5–12. Memory Copy

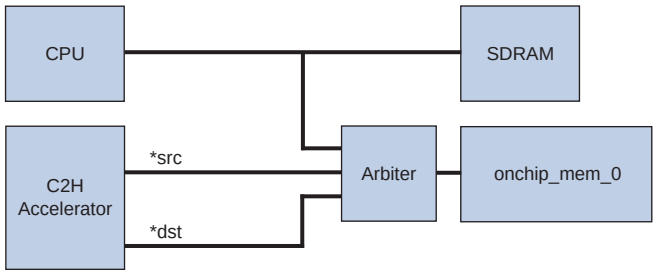
```
void memcpy(char* dst, char* src, int num_bytes)
{
   while (num_bytes-- > 0)
   {
      *dst++ = *src++;
   }
}
```

If the `src` and `dst` memory regions can be moved from the DRAM to an on-chip or off-chip SRAM, better performance is possible. To add on-chip memories, use SOPC Builder to instantiate an on-chip memory component (called **onchip_mem_0** in this example) with a 32-bit wide Avalon-MM slave port. Add the following pragmas to your C code before `memcpy`:

```
#pragma altera_accelerate connect_variable memcpy/dst to onchip_mem_0
#pragma altera_accelerate connect_variable memcpy/src to onchip_mem_0
```

The pragmas state that `dst` and `src` only connect to the **onchip_mem_0** component. This memory architecture offers better performance because SRAMs do not require large bursts like DRAMs to operate efficiently and on-chip memories operate at very low latencies. Figure Figure 5–6 shows the hardware generated for Example 5–12 with the data residing in on-chip RAM.

Figure 5–6. Use On-Chip Memory - Partition Memory for Better Bandwidth



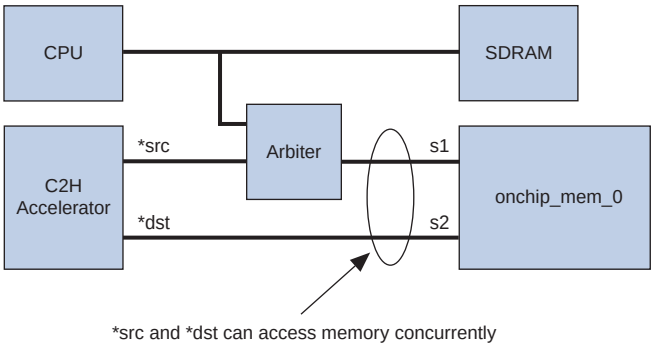
However, both master ports still share the single-port SRAM, **onchip_mem_0**, which can lead to a maximum throughput of one loop iteration every two clock cycles (a CPLI of 2). There are two solutions to this problem: Either create another SRAM so that each Avalon-MM master port has a dedicated memory, or configure the memories with dual-ports. For the latter solution, open your system in SOPC Builder and change the **onchip_mem_0** component to have two ports. This change creates slave ports called `s1` and `s2` allowing the connection pragmas to use each memory port as follows:

```
#pragma altera_accelerate connect_variable memcpy/dst to onchip_mem_0/s1
```

```
#pragma altera_accelerate connect_variable memcpy/src to onchip_mem_0/s2
```

These pragmas state that `dst` only accesses slave port `s1` of the **onchip_mem_0** component and that `src` only accesses slave port `s2` of the **onchip_mem_0** component. This new version of `memcpy` along with the improved memory architecture achieves a maximum throughput of one loop iteration per cycle (a CPLI of 1). Figure 5–7 shows the hardware generated for Example 5–12 with the data stored in dual-port on-chip RAM.

Figure 5–7. Use Dual-Port On-Chip Memory



Using Localized Data

Pointer dereferences and array accesses in accelerated functions always result in memory transactions though an Avalon-MM master port. Therefore, if you use a pointer to store temporary data inside an algorithm, as in Example 5–13, there is a memory access every time that temporary data is needed, which might stall the pipeline.

Example 5–13. Temporary Data in Memory (Slower)

```
for(i = 0; i < 1024; i++)
{
   for(j = 0; j < 10; j++)
   {
      /* read and write to the same location */
      AnArray[i] += AnArray[i] * 3;
   }
}
```

Often, storing that temporary data in a local variable, as in Example 5–14, increases the performance of the accelerator by reducing the number of times the accelerator must make an Avalon-MM access to memory. Local variables are the fastest type of storage in an accelerated function and are very effective for storing temporary data.

Although local variables can help performance, too many local variables can lead to excessive resource usage. This is a tradeoff you can experiment with when accelerating a function with the C2H Compiler.

Example 5–14. Temporary Data in Registers (Faster)

```
int temporary;
for(i = 0; i < 1024; i++)
{
   temporary = AnArray[i]; /* read from location i */
   for(j = 0; j < 10; j++)
   {
      /* read and write to a registered value */
      temporary += temporary * 3;
   }
   AnArray[i] = temporary; /* write to location i */
}
```

Reducing Data Dependencies

The following sections provide information on reducing data dependencies.

Use __restrict__

By default, the C2H Compiler cannot pipeline read and write pointer accesses because read and write operations may occur at the same memory location. If you know that the `src` and `dst` memory regions do not overlap, add the `__restrict__` keyword to the pointer declarations, as shown in Example 5–15.

Example 5–15. __restrict__ Usage

```
void memcpy(char* __restrict__ dst, char* __restrict__ src, int
num_bytes)
{
   while (num_bytes-- > 0)
   {
      *dst++ = *src++;
   }
}
```

The `__restrict__` declaration on a pointer specifies that accesses via that pointer do not alias any memory addresses accessed by other pointers. Without `__restrict__`, the C2H Compiler must schedule accesses to pointers strictly as written which can severely reduce performance.

It is very important that you verify that your algorithm operates correctly when using `__restrict__` because this option can cause sequential code to fail when accelerated. The most common error is caused by a read and write pointer causing overlapping accesses to a dual port memory. You might not detect this situation when the function executes in software, because a processor can only perform one access at a time, however by using `__restrict__` you are allowing the C2H Compiler to potentially schedule the read and write accesses of two pointers to occur concurrently.

The most common type of data dependency is between scalar data variables. A scalar data dependency occurs when an assignment relies on the result of one or more other assignments. The C code in Example 5–16 shows a data dependency between `sum1` and `result`:

Example 5–16. Scalar Data Dependency

```
int sum3(int a, int b, int c)
{
   int sum1 = a + b;
   int result = sum1 + c;
   return result;
}
```

A C compiler attempts to schedule the instructions to prevent the processor pipeline from stalling. There is no limit to the number of concurrent operations which you can exploit with the C2H Compiler. Adding all three integers in one assignment removes the data dependency, as shown in Example 5–17.

Example 5–17. Scalar Data Dependency Removed

```
int sum3(int a, int b, int c)
{
   int result = a + b + c;
   return result;
}
```

The other common type of data dependency is between elements of an array of data. The C2H Compiler treats the array as a single piece of data, assuming that all accesses to the array overlap. For example, the `swap01` function in Example 5–18 swaps the values at index 0 and index 1 in the array pointed to by <*p*>.

Example 5–18. Array Data Dependency

```
void swap01(int* p)
{
   int tmp = p[0];
   p[0] = p[1];
   p[1] = tmp;
}
```

The C2H Compiler is unable to detect that the p[0] and p[1] accesses are to different locations so it schedules the assignments to p[0] and p[1] sequentially. To force the C2H Compiler to schedule these assignments concurrently, add __restrict__ to the pointer declaration, as shown in Example 5–19.

Example 5–19. Array Data Dependency Removed

```
void swap01(int* p)
{
   int* __restrict__ p0 = andp[0];
   int* __restrict__ p1 = andp[1];
   int tmp0 = *p0;
   int tmp1 = *p1;
   *p0 = tmp1;
   *p1 = tmp0;
}
```

Now, the C2H Compiler attempts to perform the assignments to p[0] and p[1] concurrently. In this example, there is only a significant performance increase if the memory containing p[0] and p[1] has two or more write ports. If the memory is single-ported then one access stalls the other and little or no performance is gained.

A form of scalar or array data dependency is the in-scope data dependency. Example 5–20 exhibits an in-scope dependency, because it takes pointers to two arrays and their sizes and returns the sum of the contents of both arrays.

Example 5–20. In-Scope Data Dependency

```
int sum_arrays(int* arr_a, int* arr_b,
int size_a, int size_b)
{
   int i;
   int sum = 0;
   for (i = 0; i < size_a; i++)
   {
      sum += arr_a[i];
   }
   for (i = 0; i < size_b; i++)
   {
      sum += arr_b[i];
   }
   return sum;
}
```

There is a dependency on the sum variable which causes C2H accelerator to execute the two loops sequentially. There is no dependency on the loop index variable <*i*> between the two loops because the algorithm reassigns <*i*> to 0 in the beginning of the second loop.

Example 5–21 shows a new version of sum_arrays that removes the in-scope dependency:

Example 5–21. In-Scope Data Dependency Removed

```
int sum_arrays(int* arr_a, int* arr_b,
int size_a, int size_b)
{
   int i;
   int sum_a = 0;
   int sum_b = 0;
   for (i = 0; i < size_a; i++)
   {
      sum_a += arr_a[i];
   }
   for (i = 0; i < size_b; i++)
   {
      sum_b += arr_b[i];
   }
   return sum_a + sum_b;
}
```

Using separate sum variables in each loop removes the in-scope dependency. The accelerator adds the two independent sums together at the end of the function to produce the final sum. Each loop runs concurrently although the longest loop determines the execution time. For best performance, connect arr_a and arr_b to a memory with two read ports or two separate memories.

Sometimes it is not possible to remove data dependencies by simple changes to the C code. Instead, you might need to use a different algorithm to implement the same functionality. In Example 5–22, the code searches for a value in a linked list and returns 1 if the value found and 0 if it is not found. Arguments to the search function are the pointer to the head of the linked list and the value to match against.

Example 5–22. Pointer-Based Data Dependency

```
struct item
{
   int value;
   struct item* next;
};
int search(struct item* head, int match_value)
{
   struct item* p;
   for (p=head; p != NULL; p=p->next)
   {
      if (p->value == match_value)
      {
         return 1; // Found a match
      }
   }
   return 0;  // No match found
}
```

The C2H Compiler is not able to achieve a throughput of one comparison per cycle due to the `p=p->next <p>` does not occur until the next pointer location has been read from memory, causing a latency penalty to occur each time the loop state machine reaches this line of C code. To achieve better performance, use a different algorithm that supports more parallelism. For example, assume that the values are stored in an array instead of a linked list, as in Example 5–23. Arguments to the new search function are the pointer to the array, the size of the array, and the value to match against.

Example 5–23. Pointer-Based Data Dependency Removed

```
int search(int* arr, int num_elements, int match_value)
{
   for (i = 0; i < num_elements; i++)
   {
      if (arr[i] == match_value)
      {
         return 1;  // Found a match
      }
   }
   return 0;  // No match found
}
```

This new search function achieves a throughput of one comparison per cycle assuming there is no contention for memory containing the `arr` array. Prototype such a change in software before accelerating the code, because the change affects the functionality of the algorithm.

Reducing Logic Utilization

The following sections discuss coding practices you can adopt to reduce logic utilization.

Using "do-while" rather than "while"

The overhead of `do` loops is lower than those of the equivalent `while` and `for` loops because the accelerator checks the loop condition after one iteration of the loop has executed. The C2H Compiler treats a `while` loop like a `do` loop nested in an `if` statement. Example 5–24 illustrates code that the C2H Compiler transforms into a `do` loop and nested `if` statement.

Example 5–24. while Loop

```
int sum(int* arr, int num_elements)
{
   int result = 0;
   while (num_elements-- > 0)
   {
      result += *arr++;
   }
   return result;
}
```

Example 5–25 is the same function rewritten to show how the C2H Compiler converts a `while` loop to an `if` statement and a `do` loop.

Example 5–25. Converted while Loop

```
int sum(int* arr, int num_elements)
{
   int result = 0;
   if (num_elements > 0)
   {
      do
      {
         result += *arr++;
      } while (--num_elements > 0);
   }
   return result;
}
```

Notice that an extra `if` statement outside the `do` loop is required to convert the `while` loop to a `do` loop. If you know that the `sum` function is never called with an empty array, that is, the initial value of `num_elements` is always greater than zero, the most efficient C2H code uses a `do` loop instead of the original `while` loop. Example 5–26 illustrates this optimization.

Example 5–26. do Loop

```
int sum(int* arr, int num_elements)
{
   int result = 0;
   do
   {
      result += *arr++;
   } while (--num_elements > 0);
   return result;
}
```

Using Constants

Constants provide a minor performance advantage in C code compiled for a processor. However, they can provide substantial performance improvements in a C2H accelerator.

Example 5–27 demonstrates a typical add and round function.

Example 5–27. Add and Round with Variable Shift Value

```
int add_round(int a, int b, int sft_amount)
{
   int sum = a + b;
   return sum >> sft_amount;
}
```

As written above, the C2H Compiler creates a barrel shifter for the right shift operation. If `add_round` is always called with the same value for `sft_amount`, you can improve the accelerated function's efficiency by changing the `sft_amount` function parameter to a `#define` value and changing all your calls to the function. Example 5–28 is an example of such an optimization.

Example 5–28. Add and Round with Constant Shift Value

```
#define SFT_AMOUNT 1
int add_round(int a, int b)
{
   int sum = a + b;
   return sum >> SFT_AMOUNT;
}
```

Alternatively, if `add_round` is called with a few possible values for `sft_amount`, you can still avoid the barrel shifter by using a `switch` statement which just creates a multiplexer and a small amount of control logic. Example 5–29 is an example of such an optimization.

Example 5–29. Add and Round with a Finite Number of Shift Values

```
int add_round(int a, int b, int sft_amount)
{
   int sum = a + b;
   switch (sft_amount)
   {
      case 1:
         return sum >> 1;
      case 2:
         return sum >> 2;
   }
   return 0;  // Should never be reached
}
```

You can also use these techniques to avoid creating a multiplier or divider. This technique is particularly beneficial for division operations because the hardware responsible for the division is large and relatively slow.

Leaving Loops Rolled Up

Sometimes developers unroll loops to achieve better results using a C compiler. Because the C2H Compiler attempts to pipeline all loops, unrolling loops is unnecessary for C2H code. In fact, unrolled loops tend to produce worse results because the C2H Compiler creates extra logic. It is best to leave the loop rolled up. Example 5–30 shows an accumulator algorithm that was unrolled in order to execute faster on a processor.

Example 5–30. Unrolled Loop

```
int sum(int* arr)
{
   int i;
   int result = 0;
   for (i = 0; i < 100; i += 4)
   {
      result += *arr++;
      result += *arr++;
      result += *arr++;
      result += *arr++;
   }
   return result;
}
```

This function is passed an array of 100 integers, accumulates each element, and returns the sum. To achieve higher performance on a processor, the developer has unrolled the inner loop four times, reducing the loop overhead by a factor of four when executed on a processor. When the C2H Compiler maps this code to hardware, there is no loop overhead because the accelerator executes the loop overhead statements concurrently with the loop body.

As a result of unrolling the code, the C2H Compiler creates four times more logic because four separate assignments are used. The C2H Compiler creates four Avalon-MM master ports in the loop. However, an Avalon-MM master port can only perform one read or write operation at any given time. The four master ports must interleave their accesses, eliminating any advantage of having multiple masters.

Example 5–30 shows how resource sharing (memory) can cause parallelism to be nullified. Instead of using four assignments, roll this loop up as shown in Example 5–31.

Example 5–31. Rolled-Up Loop

```
int sum(int* arr)
{
   int i;
   int result = 0;
   for (i = 0; i < 100; i++)
   {
      result += *arr++;
   }
   return result;
}
```

This implementation achieves the same throughput as the previous unrolled example because this loop can potentially iterate every clock cycle. The unrolled algorithm iterates every four clock cycles due to memory stalls. Because these two algorithms achieve the same throughput, the added benefit of the rolling optimization is savings on logic resources such as Avalon-MM master ports and additional accumulation logic.

Using ++ to Sequentially Access Arrays

The unrolled version of the `sum` function in Example 5–30 uses `*arr++` to sequentially access all elements of the array. This procedure is more efficient than the alterative shown in Example 5–32.

Example 5–32. Traversing Array with Index

```
int sum(int* arr)
{
   int i;
   int result = 0;
   for (i = 0; i < 100; i++)
   {
      result += arr[i];
   }
   return result;
}
```

The C2H Compiler must create pointer dereferences for both `arr[i]` and `*arr++`. However, the instantiated logic is different for each case. For `*arr++` the value used to address memory is the pointer value itself, which is capable of incrementing. For `arr[i]` the accelerator must add base address `arr` to the counter value `i`. Both require counters, however in the case of `arr[i]` an adder block is necessary, which creates more logic.

Avoiding Excessive Pointer Dereferences

Any pointer dereference via the dereference operator * or array indexing might create an Avalon-MM master port. Avoid using excessive pointer dereference operations because they lead to additional logic which might degrade the f_{MAX} of the design.

Any local arrays within the accelerated function instantiate on-chip memory resources. Do not declare large local arrays because the amount of on-chip memory is limited and excessive use affects the routing of the design.

Avoiding Multipliers

Embedded multipliers have become a standard feature of FPGAs; however, they are still limited resources. When you accelerate source code that uses a multiplication function, the C2H accelerator instantiates a multiplier. Embedded multiplier blocks have various modes that allow them to be segmented into smaller multiplication units depending on the width of the data being used. They also have the ability to perform multiply and accumulate functionality.

When using multipliers in accelerated code validate the data width of the multiplication to reduce the logic. The embedded multiplier blocks handle 9 by 9 (`char *char`), 18 by 18 (`short *short`), and 36 by 36 (`long *long`) modes which are set depending on the size of the largest width input. Reducing the input width of multiplications not only saves resources, but also improves the routing of the design, because multiplier blocks are fixed resources. If multiplier blocks are not available or the design requires too many multiplications, the Quartus II software uses logic elements to create the multiplication hardware. Avoid this situation if possible, because multipliers implemented in logic elements are expensive in terms of resources and design speed.

Multiplications by powers of two do not instantiate multiplier logic because the accelerator can implement them with left shift operations. The C2H Compiler performs this optimization automatically, so it is not necessary to use the << operator. When multiplication is necessary, try to use powers of two in order to save logic resources and to benefit from the fast logic created for this operation. An assignment that uses a multiplication by a power of two becomes a register-to-register path in which the data is shifted in the system interconnect fabric.

When multiplying by a constant the Quartus II software optimizes the LEs either using memory or logic optimizations. Example 5–33 shows an optimization for multiplication by a constant.

Example 5–33. Multiplication by Constants

```
/* This multiplication by a constant is optimized */
y = a × 3;

/*The optimization is shift and add: (2*a + a = 3*a) */
y = a << 1 + a
```

C2H offloads the intelligence of multiplies, divides, and modulo to Quartus II synthesis to do the right thing when possible.

Avoiding Arbitrary Division

If at all possible, avoid using arbitrary division in accelerated functions, including the modulus % operator. Arbitrary division occurs whenever the divisor is unknown at compile time. True division operations in hardware are expensive and slow.

Example 5–34. Arbitrary Division (Expensive, Slow):

```
z = y / x;  /* x can equal any value */
```

The exception to this rule is division by denominators which are positive powers of two. Divisions by positive powers of two simply become binary right-shift operations. Dividing by two can be accomplished by shifting the value right one bit. Dividing by four is done by shifting right two bits, and so on. If the accelerated function uses the / division operator, and the right-hand argument is a constant power of two, the C2H Compiler converts the divide into a fixed-bit shift operation. In hardware, fixed-bit shift operations result in only wires, which are free.

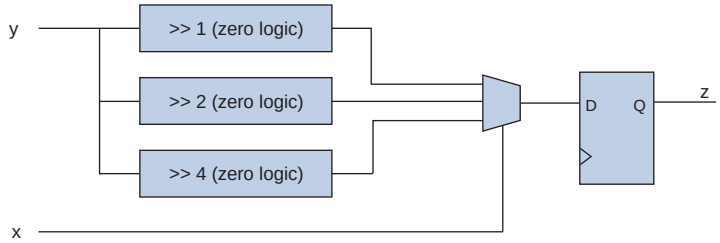
If a division operation in an accelerated function always uses a denominator that is a power of two, but can use various multiples of two, you can use a ternary operation to convert the divides to the appropriate fixed-bit shift, as shown in Example 5–35.

Example 5–35. Division using Shifts with a Ternary Operator (Cheap, Fast)

```
z = (x == 2)? y >> 1:((x == 4)? y >> 2: y >> 4);
```

Figure 5–8 shows the hardware generated for Example 5–35.

Figure 5–8. . Ternary Shift Divide



The logic created by this optimization is relatively cheap and fast, consisting of a multiplexer and minimal control logic. Because the assignments to z are just shifted copies of y the multiplexer is the only logic in the register-to-register path. If there are many possible denominator values, explore the tradeoff between latency and frequency discussed in the"Improving Conditional Frequency" on page 5–37.

The other possible optimization to avoid generating an expensive and slow division circuit is to implement a serial divider. Serial dividers have a high latency, but tend not to degrade f_{MAX}. Another benefit of using serial division is the relatively low cost of the hardware generated because the operations performed are on bits instead of words.

You can use macros in **c2h_division.h** and **c2h_modulo.h** to implement serial division or modulo operations in your own system. These files are available on the Nios II literature page. A hyperlink to the software files appears next to *Optimizing Nios II C2H Compiler Results* (this document), at **www.altera.com/literature/lit-nio2.jsp.** The two header files are distributed in a zip file.

Using Masks

Both the C compiler for a 32-bit processor and the C2H Compiler convert data types smaller than integers to 32-bit integers. If you want to override this default behavior to save logic and avoid degrading the f_{MAX} of the design, add a bitwise AND with a mask. In Example 5–36, the C2H Compiler promotes b1 and b2 to 32-bit integers when performing the addition so that it instantiates a 32-bit adder in hardware. However, because b1 and b2 are unsigned characters, the sum of b1 and b2 is guaranteed to fit in nine bits, so you can mask the addition to save bits. The C2H Compiler still instantiates a 32-bit adder but Quartus II synthesis removes the unnecessary bits, resulting in a 9-bit adder in hardware.

Example 5–36. Using Bitwise And with a Mask

```
unsigned int add_chars(unsigned char b1, unsigned char b2)
{
  return (b1 + b2) & 0x1ff;
}
```

This optimization can cause a failure if you mistakenly reduce the width of the calculation so that needed data resolution is lost. Another common mistake is to use bit masks with signed data. Signed data, stored using 2's complement format, requires that the accelerator preserve and extend the sign bit through the masking operation.

Using Powers of Two in Multi-Dimensional Arrays

A conventional C compiler implements a multidimensional array as a one-dimensional array stored in row-major order. For example, a two-dimensional array might appear as follows:

```
#define NUM_ROWS 10
#define NUM_COLS 12
int arr2d[NUM_ROWS][NUM_COLS];
```

The first array index is the row and the second is the column. A conventional C compiler implements this as a one-dimensional array with NUM_ROWS x NUM_COLS elements. The compiled code computes the offset into the one-dimensional array using the following equation:

```
offset = row * NUM_COLS + col;
```

The C2H Compiler follows this implementation of multidimensional arrays. Whenever your C code indexes into a multidimensional array, an implicit multiplication is created for each additional dimension. If the multiplication is by a power of two, the C2H Compiler implements the multiplication with a wired shift, which is free. If you can increase that dimension of the array to a power of two, you save a multiplier. This optimization comes at the cost of some memory, which is cheap. In the example, just make the following change:

```
#define NUM_COLS 16
```

To avoid all multipliers for multidimensional array accesses of <*n*> dimensions, you must use an integer power of two array size for each of the final <*n-1*> dimensions. The first dimension can have any length because it does not influence the decision made by the C2H Compiler to instantiate multipliers to create the index.

Using Narrow Local Variables

The use of local variables that are larger data types than necessary can waste hardware resources in an accelerator. Example 5–37 includes a variable that is known to contain only the values 0–229. Using a `long int` variable type for this variable creates a variable that is much larger than needed. This type of optimization is usually not applicable to pointer variables. Pointers always cost 32 bits, regardless of their type. Reducing the type size of a pointer variable affects the size of the data the pointer points to, not the pointer itself. It is generally best to use large pointer types to take advantage of wide memory accesses. Refer to "Using Wide Memory Accesses" on page 5–16 for details.

Example 5–37. Wide Local Variable i Costs 32 Bits

```
int i;
int var;
for(i = 0; i < 230; i++)
{
   var += *ptr + i;
}
```

An `unsigned char` variable type, as shown in Example 5–38, is large enough because it can store values up to 255, and only costs 8 bits of logic, whereas a `long int` type costs 32 bits of logic. Excessive logic utilization wastes FPGA resources and can degrade system f_{MAX}.

Example 5–38. Narrow Local Variable i Costs 8 Bits

```
unsigned char i;
int var;
for(i = 0; i < 230; i++)
{
   var += *ptr + i;
}
```

Optimizing Memory Connections

The following sections discuss ways to optimize memory connectivity.

Removing Unnecessary Connections to Memory Slave ports

The Avalon-MM master ports associated with the `src` and `dst` pointers in Example 5–39 are connected to all of the Avalon-MM slave ports that are connected to the processor's data master. Typically, the accelerator does not need to access all these slave ports. This extra connectivity adds unnecessary logic to the system interconnect fabric, which increases the hardware resources and potentially creates long timing paths, degrading f_{MAX}.

The C2H Compiler supports pragmas added to your C code to inform the C2H Compiler which slave ports each pointer accesses in your accelerator. For example, if the `src` and `dst` pointers can only access the DRAM (assume it is called **dram_0**), add these pragmas before `memcpy` in your C code.

```
#pragma altera_accelerate connect_variable memcpy/dst to dram_0
```

```
#pragma altera_accelerate connect_variable memcpy/src to dram_0
```

Example 5–39. Memory Interconnect

```
void memcpy(char* dst, char* src, int num_bytes)
{
   while (num_bytes-- > 0)
   {
      *dst++ = *src++;
   }
}
```

These pragmas state that `dst` and `src` only access the **dram_0** component. The C2H Compiler connects the associated Avalon-MM ports only to the **dram_0** component.

Reducing Avalon-MM Interconnect Using #pragma

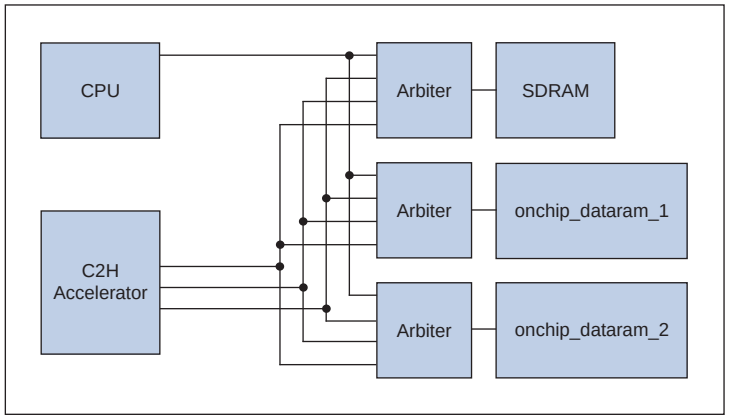
Accelerated functions use Avalon-MM ports to access data related to pointers in the C code. By default, each master generated connects to every memory slave port that is connected to the Nios II data master port. This connectivity can result in large amounts of arbitration logic when you generate an SOPC Builder system, which is expensive and can degrade system f_{MAX}. In most cases, pointers do not need to access every memory in the system.

You can reduce the number of master-slave port connections in your SOPC Builder system by explicitly specifying the memories to which a pointer dereference must connect. You can make connections between pointers and memories with the `connect_variable` pragma directive, as shown in Example 5–40. In Figure 5–9, three pointers, `output_data`, `input_data1`, and `input_data2` are connected to memories named `sdram`, `onchip_dataram1`, and `onchip_dataram2`, respectively. Using the `connect_variable` pragma directive ensures that each of the accelerated function's three Avalon-MM master ports connects to a single memory slave port. The result is a more efficient overall because it has no unnecessary master-slave port connections.

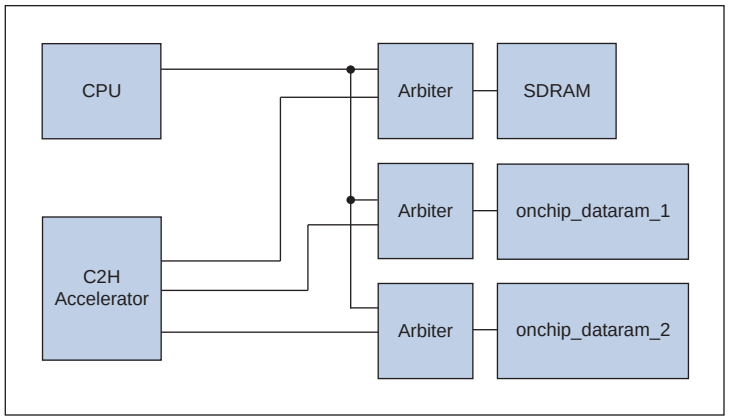
Example 5–40. Reducing Memory Interconnect

```
#pragma altera_accelerate connect_variable my_c2h_function/output_data to sdram

#pragma altera_accelerate connect_variable my_c2h_function/input_data1 to onchip_dataram1
#pragma altera_accelerate connect_variable my_c2h_function/input_data2 to onchip_dataram2

void my_c2h_function( int *input_data1,
                      int *input_data2,
                      int* output_data )
{
  char i;
  for( i = 0; i < 52;  i++ )
  {
    *(output_data + i) = *(input_data1 + i) + *(input_data2 + i);
  }
}
```

Figure 5–9. Pragma Connect



All connections (unnecessary arbitration logic)



Only necessary connections (less arbitration logic)

Removing Unnecessary Memory Connections to the Nios II Processor

As part of your optimization, you might have added on-chip memories to the system to allow an accelerated function access to multiple pointers in parallel, as in "Segmenting the Memory Architecture" on page 5–18. During implementation and debug, it is important that these on-chip memories have connections to both the appropriate accelerator Avalon-MM master port and to the Nios II data master port, so the function can run in both accelerated and non-accelerated modes. In some cases however, after you are done debugging, you can remove the memory connections to the Nios II data master if the processor does not access the memory when the function is accelerated. Removing connections lowers the cost and avoids degrading system f_{MAX}.

Optimizing Frequency Versus Latency

The following sections describe tradeoffs you can make between frequency and latency to improve performance.

Improving Conditional Latency

Algorithms that contain `if` or `case` statements use registered control paths when accelerated. The C2H Compiler accelerates the code show in Example 5–41 in this way.

Example 5–41. Registered Control Path

```
if(testValue < Threshold)
{
   a = x;
}
else
{
   a = y;
}
```

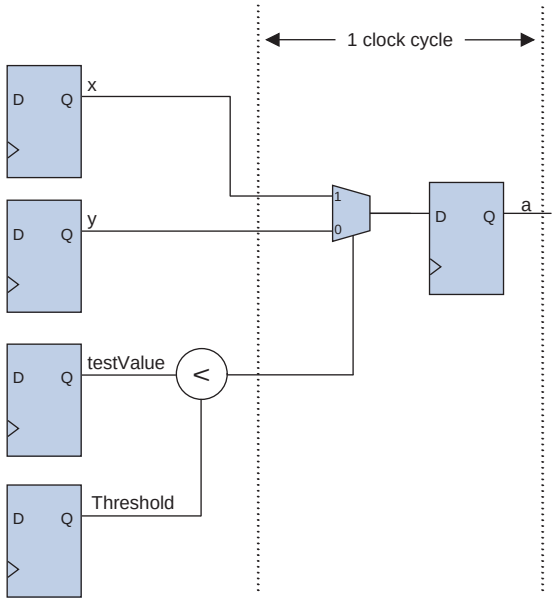
You can modify your software to make use of the ternary operator, (`?:`), as in Example 5–42, to reduce the latency of the control path. The ternary operator does not register signals on the control path, so this optimization results in lower latency at the expense of f_{MAX}. This optimization primarily helps reduce the CPLI of the accelerator when a data dependency prevents the conditional statement from becoming fully pipelined. Do not use this optimization if the CPLI of the loop containing the conditional statement is already equal to one.

Example 5–42. Unregistered Control Path

```
a = (testValue < Threshold)? x : y;
```

Figure 5–10 shows the hardware generated for Example 5–42.

Figure 5–10. Conditional Latency Improvement



Improving Conditional Frequency

If you wish to avoid degrading f_{MAX} in exchange for an increase in latency, consider removing ternary operators. By using an `if` or `case` statement to replace the ternary operator the control path of the condition becomes registered and shortens the timing paths in that portion of the accelerator. In the case of the conditional statement being executed infrequently (outside of a loop), this optimization might prove a small price to pay to increase the overall frequency of the hardware design.

Example 5–43 and Example 5–44 show how you can rewrite a ternary operator as an `if` statement.

Example 5–43. Unregistered Conditional Statement

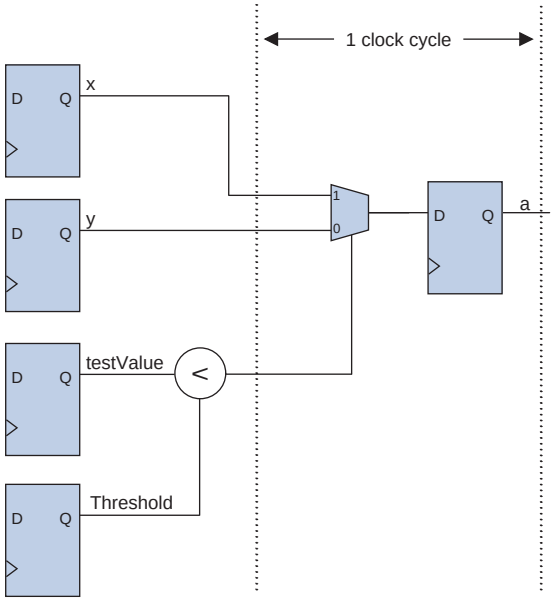
```
a = (testValue < Threshold)? x : y;
```

Example 5–44. Registered Conditional Statement

```
if(testValue < Threshold)
{
   a = x;
}
else
{
   a = y;
}
```

Figure 5–11 shows the hardware the C2H Compiler generates for Example 5–44.

Figure 5–11. Conditional Frequency Improvement



Improving Throughput

To increase the computational throughput, focus on two main areas: achieving a low CPLI, and performing many operations within one loop iteration.

Avoiding Short Nested Loops

Because a loop has a fixed latency before any iteration can occur, nesting looping structures can lead to unnecessary delays. The accelerator incurs the loop latency penalty each time it enters the loop. Rolling software into loops adds the possible benefit of pipelining, and the benefits of this pipelining usually outweigh the latency associated with loop structures. Generally, if the latency is greater than the maximum number of iterations times the CPLI then the looping implementation is slower. You must take into account that leaving a loop unrolled usually increases the resource usage of the hardware accelerator.

Example 5–45. Nested Loops

```
for(loop1 = 0; loop1 < 10; loop1++) /* Latency = 3, CPLI = 2 */
{
  /* statements requiring two clock cycles per loop1 iteration */
  for(loop2 = 0; loop2 < 5; loop2++)  /* Latency = 10, CPLI = 1 */
  {
     /* statements requiring one clock cycle per loop2 iteration */
  }
}
```

Assuming no memory stalls occur, the total number of clock cycles is as follows:

$$Innerloop = latency + (iterations - 1)(CPLI + innerlooptime)$$

$$Innerloop = 10 + 4(1 + 0)$$

$$Innterloop = 14cycles$$

$$Outerloop = 3 + 9(2 + 14)$$

$$Outerloop = 147cycles$$

Due to the high latency of the inner loop the total time for this example is 147 clock cycles.

Example 5–46. Single Loop

```
for(loop1 = 0; loop1 < 10; loop1++) /* Latency = 3, CPLI = 7 */
{
  /* statements requiring two clock cycles per loop1 iteration */
  /* statements that were previously contained in loop2 */
}
```

Assuming no memory stalls occur, the total number of clock cycles is as follows:

$$Outerloop = latency + (iterations - 1)(CPLI + innerlooptime)$$

$$Outerloop = 3 + 9(7 + 0)$$

$$Outerloop = 66cycles$$

The inner loop (`loop2`) has been eliminated and consequently is 0 in these equations. Combining the inner loop with the outer loop dramatically decreases the total time to complete the same outer loop. This optimization assumes that unrolling the inner loop resulted in adding five cycles per iteration to the outer loop. The combination of the loops would most likely result in a hardware utilization increase which you must take into consideration.

Removing In-place Calculations

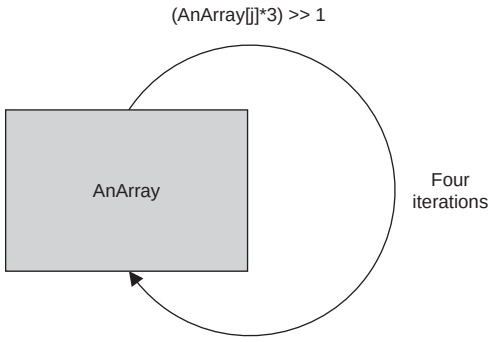
Some software algorithms perform in-place calculations, in which results overwrite the input data as they are calculated. This technique conserves memory, but produces suboptimal performance when compiled to a C2H accelerator. Example 5–47 shows such an algorithm. Unfortunately this approach leads to memory stalls because in-place algorithms read and write to the same memory locations.

Example 5–47. Two Avalon-MM Ports Using The Same Memory

```
for(i = 0; i < 4; i++)
{
   for(j = 0; j < 1024; j++)
   {
      AnArray[j] = (AnArray[j] * 3) >> 1;
   }
}
```

Figure 5–12 shows the dataflow in hardware generated for Example 5–47.

Figure 5–12. In-Place Calculation



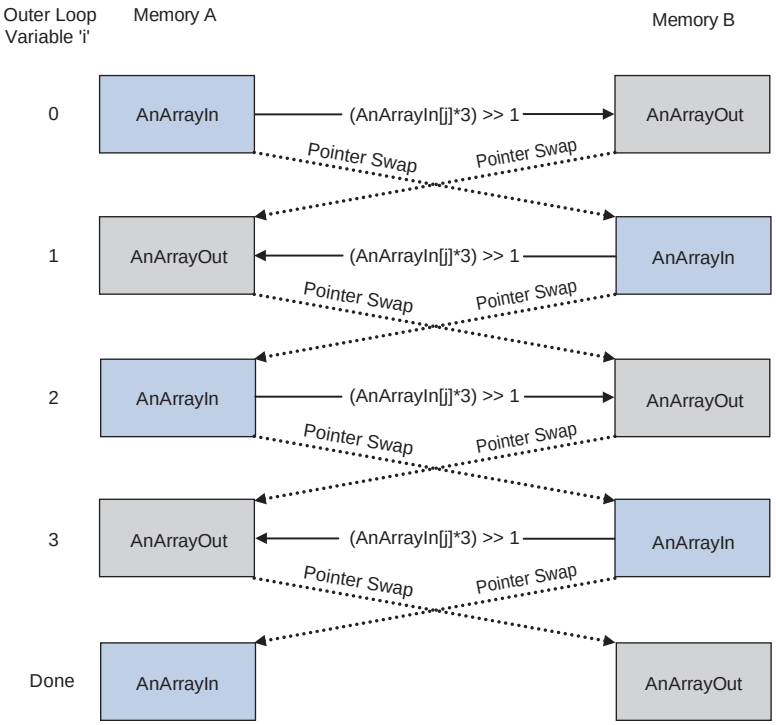
To solve this problem, remove the in-place behavior of the algorithm by adding a "shadow" memory to the system, as shown in Example 5–48. Instead of the input and output residing in the same memory, each uses an independent memory. This optimization prevents memory stalls because the input and output data reside in separate memories.

Example 5–48. Two Avalon-MM Ports Using Separate Memories

```
int * ptr;
for(i = 0; i < 4; i++)
{
   for(j = 0; j < 1024; j++)
   {
      /* In from one memory and out to the other */
      AnArrayOut[j] = (AnArrayIn[j] * 3) >> 1;
   }
   /* Swap the input and output pointers and do it all over again */
   ptr = AnArrayOut;
   AnArrayOut = AnArrayIn;
   AnArrayIn = ptr;
}
```

Figure 5–13 shows the dataflow of hardware generated for Example 5–48.

Figure 5–13. In-Place Calculation



You can also use this optimization if the data resides in on-chip memory. Most on-chip memory can be dual-ported to allow for simultaneous read and write access. With a dual-port memory, the accelerator can read the data from one port without waiting for the other port to be written. When you use this optimization, the read and write addresses must not overlap, because that could lead to data corruption. A method for preventing a read and a write from occurring simultaneously at the same address is to read the data into a variable before the write occurs.

Replacing Arrays

Often software uses data structures that are accessed via a base pointer location and offsets from that location, as shown in Example 5–49. When the hardware accelerator accesses the data in these structures, memory accesses result.

Example 5–49. Individual Memory Accesses

```
int a = Array[0];
int b = Array[1];
int c = Array[2];
int d = Array[3];
```

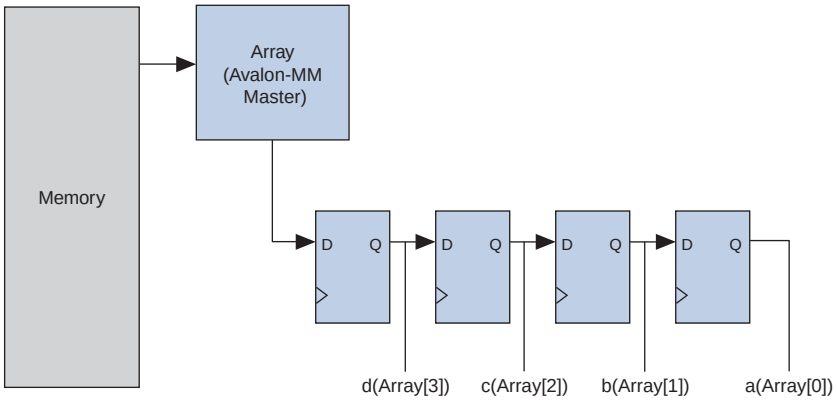
You can replace these memory accesses using a single pointer and registers, as in Example 5–50. The overall structure of the hardware created resembles a FIFO.

Example 5–50. FIFO Memory Accesses

```
/* initialize variables */
int a = 0;
int b = 0;
int c = 0;
int d = 0;
for(i = 0; i < 4; i++)
{
   d = Array[i];
   c = d;
   b = c;
   a = b;
}
```

Figure 5–14 shows the hardware generated for Example 5–50.

Figure 5–14. Array Replacement



Using Polled Accelerators

When you create a hardware accelerator using the C2H Compiler, it creates a wrapper file that is linked at compile time, allowing the main program to call both the software and hardware versions of the algorithm using the same function name. The wrapper file performs the following three tasks:

- Writes the passed parameters to the accelerator
- Polls the accelerator to determine when the computation is complete
- Sends the return value back to the caller

Because the wrapper file is responsible for determining the status of the accelerator, the Nios II processor must wait for the wrapper code to complete. This behavior is called blocking.

The hardware accelerator blocks the Nios II processor from progressing until the accelerator has reached completion. The wrapper file is responsible for this blocking action. Using the same pragma statement to create the interrupt include file, you can access the macros defined in it to implement a custom polling algorithm common in systems that do not use a real time operating system.

Instead of using the interrupt to alert Nios II that the accelerator has completed its calculation, the software polls the `busy` value associated with the accelerator. The macros necessary to manually poll the accelerator to determine if it has completed are in the include file created under either the **Debug** or **Release** directory of your application project. These macros are shown in Table 5–5.

Table 5–5. C2H Accelerator Polling Macros

Purpose	Macro Name
Busy value	`ACCELERATOR_<Project Name>_<Function Name>_BUSY( )`
Return value	`ACCELERATOR_<Project Name>_<Function Name>_GET_RETURN_VALUE( )`

While the accelerator is busy, the rest of the software must not attempt to read the return value because it might be invalid.

Using an Interrupt-Based Accelerator

The blocking behavior of a polled accelerator might be undesirable if there are processing tasks which the Nios II processor can carry out while the accelerator is running. In this case, you can create an interrupt-based accelerator.

Create the hardware accelerator with the standard flow first, because interrupts add an extra level of complexity. Before proceeding to the interrupt flow, debug the system to make sure the accelerator behaves correctly. Add enhancements in polled mode, as well.

To use the hardware accelerator in a non-blocking 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 wrapper file containing all the macros needed to use the accelerator and service the interrupts it generates. 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 you must click the **Generate** button to regenerate your SOPC Builder system.

The macros necessary to service the accelerator interrupt are in the **include** file created under either the **Debug** or **Release** directory of your application project. These macros are shown in Table 5–6.

Table 5–6. C2H Accelerator Interrupt Macros

Purpose	Macro Name
IRQ level value	ACCELERATOR_<*Project Name*>_<*Function Name*>_IRQ( )
Return value	ACCELERATOR_<*Project Name*>_<*Function Name*>_GET_RETURN_VALUE( )
Interrupt clear	ACCELERATOR_<*Project Name*>_<*Function Name*>_CLEAR_IRQ( )

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

Glossary

This chapter uses the following terminology:

- **Accelerator throughput**—the throughput achieved by a C2H accelerator during a single invocation. Accelerator throughput might be less than peak throughput if pipeline stalls occur. Accelerator throughput does not include latency. See also CPLI, Throughput, Peak throughput, and Application throughput.
- **Application throughput**—the throughput achieved by a C2H accelerator in the context of the application, involving multiple accelerator invocations and including the number of cycles of latency.
- **Barrel shifter –** hardware that shifts a byte or word of data an arbitrary number of bits in one clock cycle. Barrel shifters are fast and expensive, and can degrade f_{MAX}.

- **Cache coherency**—the integrity of cached data. When a processor accesses memory through a cache and also shares that memory with a coprocessor (such as a C2H accelerator), it must ensure that the data in memory matches the data in cache whenever the coprocessor accesses the data. If the coprocessor can access data in memory that has not been updated from the cache, there is a cache-coherency problem.
- **Compute-limited**—describes algorithms whose speed is restricted by how fast data can be processed. When an algorithm is compute-limited, there is no benefit from increasing the efficiency of memory or other hardware. See also Data-limited.
- **Control path**—a chain of logic controlling the output of a multiplexer
- **CPLI**—cycles per loop iteration. The number of clock cycles required to execute one loop iteration. CPLI does not include latency.
- **Critical timing path**—the longest timing path in a clock domain. The critical timing path limits f_{MAX} or the entire clock domain. See also timing path.
- **Data dependency**—a situation where the result of an assignment depends on the result of one or more other assignments, as in Example 5–5.
- **Data-limited**—describes algorithms whose speed is restricted by how fast data can be transferred to or from memory or other hardware. When an algorithm is data-limited, there is no benefit from increasing processing power. See also Compute-limited.
- **DRAM**—dynamic random access memory. It is most efficient to access DRAM sequentially, because there is a time penalty when it is accessed randomly. SDRAM is a common type of DRAM.
- **Latency**—a time penalty incurred each time the accelerator enters a loop.
- **Long timing path**—a critical timing path that degrades f_{MAX}.
- **Peak throughput**—the throughput achieved by a C2H accelerator, assuming no pipeline stalls and disregarding latency. For a given loop, peak throughput is inversely proportional to CPLI. See also throughput, accelerator throughput, application throughput, CPLI, latency.
- **Rolled-up loop**—A normal C loop, implementing one algorithmic iteration per processor iteration. See also unrolled loop.
- **SDRAM**—synchronous dynamic random access memory. See DRAM.
- **SRAM**—static random access memory. SRAM can be accessed randomly without a timing penalty.
- **Subfunction**—a function called by an accelerated function. If `apple()` is a function, and `apple()` calls `orange()`, `orange()` is a subfunction of `apple()`. If `orange()` calls `banana()`, `banana()` is also a subfunction of `apple()`.
- **Throughput**—the amount of data processed per unit time. See also accelerator throughput, application throughput, and peak throughput.
- **Timing path**—a chain of logic connecting the output of a hardware register to the input of the next hardware register.

- **Unrolled loop**—A C loop that is deconstructed to implement more than one algorithmic iteration per loop iteration, as illustrated in Example 5–30. See also Rolled-up loop.

Document Revision History

Table 5–7 shows the revision history for this document.

Table 5–7. Document Revision History

Date	Version	Changes
July 2011	1.2	Updated references and added information in "Using Wide Memory Accesses" on page 5–16.
June 2008	1.1	Corrected Table of Contents.
March 2008	1.0	Initial release. This chapter was previously released as *AN420: Optimizing Nios II C2H Compiler Results*.