White Paper

Automated Generation of Hardware Accelerators With Direct Memory Access From ANSI/ISO Standard C Functions

Introduction

Forty years after its original statement, Moore's Law continues to provide designers with devices of increasing density at lower cost, enabling systems of greater size and complexity. Implementing such systems has become a challenge as developers attempt to scale conventional methods of using register-transfer-level HDL to build their circuits. This problem is commonly referred to as the design gap—the difference between device density and developer productivity. As Moore's Law outpaces advancements in design tools and methodologies, the gap widens.

However, the increased density and performance of new devices permits movement to a higher level of abstraction. Designers can no longer afford to concern themselves with low-level circuit implementation details. The EDA industry has identified this need, and many tools and methodologies for synthesis of logic from high-level languages have emerged in the past decade, attempting to address it.

Traditionally, the problem of generating stand-alone hardware modules has been addressed by C-to-gates methodologies. A very different approach is to generate coprocessors that off-load and enhance performance of a microprocessor running software written in C. This methodology addresses several important issues: (1) tight integration with a software design flow, including true push-button acceleration of critical computations prototyped or already running in C on a conventional microprocessor or digital signal processor; (2) direct connection of generated hardware accelerators into the processor's memory map; (3) seamless support for pointers and arrays; (4) efficient latency-aware scheduling and pipelining of memory transactions.

An implementation of this methodology is possible only with a supporting ecosystem of tools, which will be demonstrated in this paper. The Altera® Nios® II C-to-Hardware Acceleration (C2H) Compiler generates, from pure ANSI/ISO standard C functions, hardware accelerators that have direct access to memory and other peripherals in the processor's system. It uses an existing commercial system integration tool to connect the accelerator to the processor and any other peripherals in the system. This gives the accelerator direct access to a memory map identical to that of the CPU, allowing seamless support for pointers and arrays when migrating from software to hardware.

The cockpit for the Nios II C2H Compiler is the CPU's software integrated development environment (IDE). By supporting pointers and un-extended ANSI/ISO C, the compiler allows developers to quickly prototype a function in software running on the processor, then switch to a hardware-accelerated implementation with the push of a button.

One of the biggest problems in the EDA industry is the incompatibility between hardware and software design flows. This methodology solves that problem by enabling development of embedded software and embedded hardware accelerators, as well as management of hardware/software trade-offs, from one environment in one language.

Supporting Tools

The methodology described in this paper relies heavily on the tool chain that integrates the hardware accelerator with an associated processor, as well as all other peripherals in a complete embedded system. This section provides some background information on these tools. Later sections will discuss, in detail, the roles of these tools in supporting the Nios II C2H Compiler.

System Integration

The Nios II C2H Compiler makes use of SOPC Builder, a system integration tool that allows designers to specify and connect components such as processors, memories, and peripherals, using a graphical user interface (GUI). Based on the component list and connectivity matrix, SOPC Builder is able to output HDL for all components in the system, as well as a top-level design file that contains automatically generated interconnect and arbitration logic. It also automatically generates configured ModelSim® projects, scripts, and testbenches.

SOPC Builder permits extension of its component library with custom user components. C2H accelerators are custom, automatically generated SOPC Builder components that are inserted and connected in the system automatically. When HDL is generated for the system, the accelerator is connected, just like any other system component, to the rest of the system using a programmatically generated system interconnect fabric.

System Interconnect

Altera's Avalon® interconnect fabric is a collection of signals and logic that connects components into a memory-mapped register system. It resembles a typical shared system bus in its functionality, but instead uses dynamically generated wires, registers, and slave-side arbiters, increasing performance. Components interface to the rest of the system through Avalon master and slave ports, which, combined with the matrix of master-slave connections maintained by SOPC Builder, encapsulate the entire system interconnect.

Any master or slave port can have an arbitrary number of connections. For example, a CPU's data master port might connect to eight slaves in the system, all belonging to different memory and peripheral modules. Similarly, the slave port on a memory module might be mastered by two CPUs, a direct memory access (DMA) device, and two hardware accelerators. The Avalon interconnect fabric manages these connections by allowing for simultaneous multimastering through slave-side arbitration. It inserts arbitration modules in front of each slave port that manage requests from the different masters, and abstracts the system's interconnect details from master and slave ports alike. Figure 1 shows the Avalon interconnect fabric connecting multiple slaves and masters in a system.

The Nios II C2H Compiler leverages the sophisticated feature set of the Avalon interconnect fabric, which includes address decoding, data-path multiplexing, wait-state insertion, pipelining, dynamic bus sizing, arbitration for simultaneous multimastering, burst management, clock domain crossing, and off-chip interfaces.

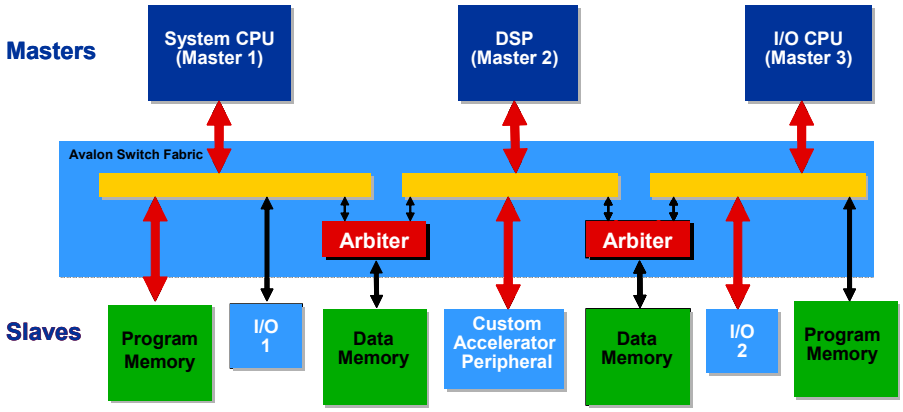

Figure 1. Avalon Interconnect Fabric Connecting Masters and Slaves in a System

Soft RISC Processor

The compiler integrates with the Nios II soft embedded microprocessors. The Nios II processor is a general-purpose RISC architectures with full 32-bit instruction set, data path, and address space, capable of performance up to 225 MIPS (Dhrystones 2.1).

C2H accelerators are coprocessors that are fully integrated into the Nios II processor and its entire software tool chain. An important distinction must be made here—C2H accelerators are *not* custom instructions (although custom instructions are a fully supported feature of the processor). Custom instructions extend the arithmetic logic unit of a processor by accepting two scalar operands and returning a scalar value. C2H accelerators are complete coprocessors that accept an unlimited number of arguments (which can be scalars, arrays, structures, etc. passed by value or reference), and are capable of manipulating memory and other peripherals in the system. While it is perfectly viable to build a tool that automatically generates custom instructions from descriptions in C, this methodology was chosen because its scope and potential for increase in overall performance is much greater.

Although this methodology currently only supports Nios II processors, it is not necessarily processor-specific. It could very well be applied in other environments, and integrated into the software development flows of other microprocessors. However, the CPU's Eclipse-based IDE and integration with SOPC Builder greatly facilitate generation of the hardware coprocessors that extend it.

Design Flow

The C2H design flow begins with a C/C++ project in the software IDE, containing an ANSI C function that has been written for acceleration, or identified as a bottleneck in existing code and selected for acceleration. The flow proceeds as follows:

1. Prototype and debug in software IDE.
2. Select target function(s) for acceleration in IDE.
3. Run C2H analysis on accelerated functions to generate detailed report file containing resource usage, performance information, warnings about parts of code that the Nios II C2H Compiler was unable to optimize, and critical data paths (such as a loop-carried dependency between two pointers passed in as input arguments). This process takes a few seconds to a few minutes, depending on the complexity of the function—on the order of a conventional software compilation.
4. Optimize and iterate.
5. Enable the IDE option to generate an updated device programming file. This runs the system generation tool to generate logic for the entire system, and runs synthesis, placement, and routing on the design.
6. Program hardware image into the device.
7. Verify performance by running the software project, which is now making use of the hardware accelerators, in the IDE.

This design flow illustrates the user's ability to drive the entire software/hardware partitioning process from the software IDE. Implementation details of connecting the hardware block to the system and enabling it to communicate with the CPU, memory, and other peripherals become an abstraction.

Additionally, because the same ANSI/ISO C language is used for software and hardware development, the user can switch at any time between running the function in software, quickly analyzing it for acceleration, and running the complete process to generate an updated hardware image. This gives the user the tools to rapidly iterate over debugging and optimizing the function, without experiencing the long delays of synthesis and placement.

Figure 2 shows how the Nios II C2H Compiler integrates into the software build process in the IDE. The left half of the flowchart shows the standard C compilation of `main.c` and `accelerator.c`, as it occurs without acceleration. The right half of the flowchart shows the hardware compilation process invoked when a function in `accelerator.c` is accelerated with the compiler. It also shows the generation and selective linking of the accelerator driver (discussed in the following section) into the executable file. When prototyping and optimizing accelerators, the developer does not need to run this complete process—options in the IDE provide for switching between (1) the software-only build process, (2) software + C2H analysis/reporting, and (3) software + complete hardware (shown in the flowchart). This allows for fast debug and optimization iterations during early stages of development, as well as automated integration of the entire hardware flow during later stages.

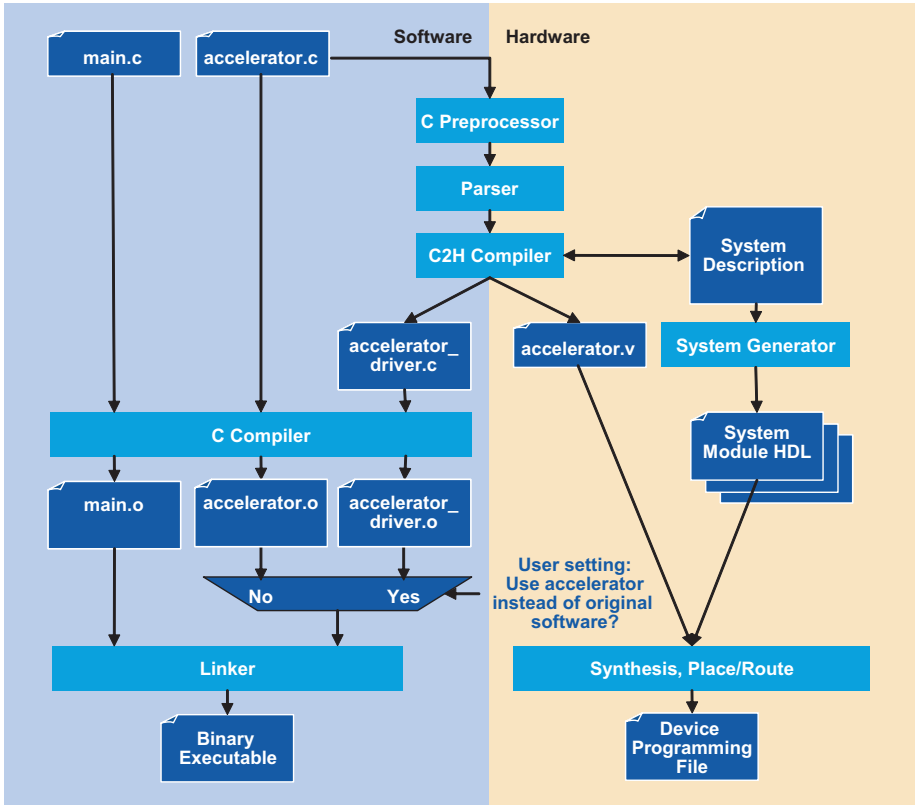

Figure 2. C2H Integration of Hardware Compilation Into the Software Build Process

Accelerator Interface to the CPU

Each accelerator has a slave port that acts as a control interface to its main processor. The slave port contains a START bit, as well as a STATUS bit. It also has a bank of registers that accept values of the accelerated function's input arguments. If the function returns a value, the slave port contains a read-only register that holds it. Because the hardware module is an SOPC Builder component, the CPU can access these registers simply by issuing read and write operations to their locations. Figure 3 shows the CPU mastering an accelerator control slave in a simple system.

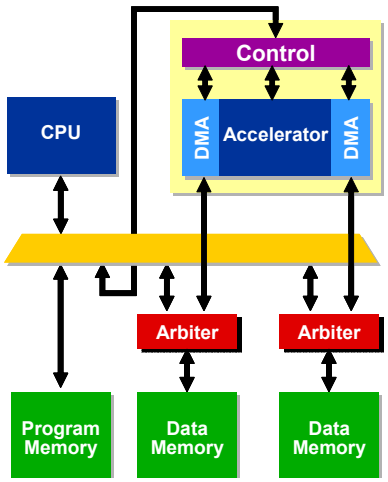

Figure 3. Accelerator Interfaces to the CPU and System Memory

For each accelerator, the Nios II C2H Compiler emits a C driver that has the same signature as the original function. This driver is compiled into the software project along with the original source code for the function. The linker, depending on whether or not the user has selected to run the function in hardware, chooses the appropriate definition of the function. This process is completely automated—the user need only build the project, and the tool chain proceeds according to the user's settings.

There are two modes of accelerator operation: polled and interrupt-driven. If the user has specified the polled mode, the driver consists of four operations:

1. Load the input arguments into the accelerator's register bank.
2. Set the START bit.
3. Poll the STATUS bit until the hardware signals completion.
4. Read the return value and exit.

After the driver function exits, the processor then resumes execution of the program, just as if the function had run in software.

If the user has specified interrupt-driven mode, the process to initialize and start the accelerator remains the same but the polling-loop and return-value fetch are removed. The compiler adds a hardware interrupt port to the accelerator that is asserted upon completion, and emits a header file that contains macros for clearing the interrupt and reading the return value for the function. Using these macros, the designer can write interrupt service routines that allow concurrent operation of the processor and multiple hardware accelerators. Furthermore, the hardware can be integrated for use with a real-time operating system.

Direct Memory Access

Handling pointers is one of the most formidable challenges to a complete, elegant C-to-gates solution. In software, pointers and indirection are very well-defined—every variable declared (statically or dynamically) has an address, and dereferencing a pointer is merely a load or store operation at its address. In a stand-alone hardware block, the definition becomes blurred. There is no natural address space, and a single variable can map to many different registers and wires due to pipelining and speculative execution. The difficulty of translating into hardware constructs such as pointers has motivated many researchers in the area to avoid them by restricting their languages.

By limiting support for pointers, the utility of the language becomes significantly lower, as most nontrivial algorithms use arrays and pointers to operate on memory data structures. This forces the user to spend time overhauling the original code to an acceptable form that does not use any unsupported constructs.

In order for pointer operations to be meaningful and to migrate seamlessly into hardware, the accelerated function must have access to the same memory map that it did when running in software. This is also necessary for easy transfer of data between the accelerator and the CPU, as well as the rest of the peripherals in the system.

The solution presented in this paper imposes no restrictions on the bandwidth into and out of the accelerator. When the Nios II C2H Compiler accelerates a function (converts it to hardware), it generates Avalon master ports for pointer and array operations, as well as operations that access static and global variables allocated on the stack or heap. These master ports allow access to memory and other peripherals in the system, and are capable of operating completely independently, in parallel. While one or more masters fetch data from input buffers, others write data to output buffers—all on the same clock cycle. Figure 4 shows an accelerator with multiple read and write masters.

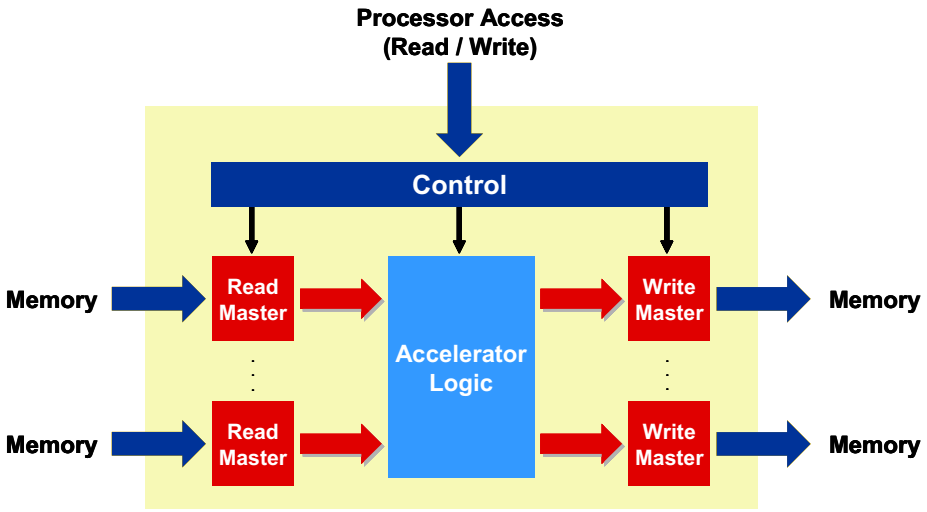

Figure 4. Accelerator Master Ports (and Control Slave)

Although initial versions of the Nios II C2H Compiler used the brute-force method of generating nearly one master port for every external memory operation, it now uses "points-to" information, as well as scheduling information to eliminate unnecessary masters. For example, if two pointers simultaneously attempt to access the same slave port, the accesses are rescheduled sequentially and only one master port is generated. Similarly, if two pointers are determined to be in mutually exclusive control paths (such as in separate blocks of an if/else statement) or are scheduled in a way that precludes simultaneous execution, they share a master port.

Optimal accelerator performance is achieved when large data structures used in critical loops are stored in system modules with separate slave ports. This way, the Nios II C2H Compiler is able to use dedicated master ports that connect to each of the slaves and transfer data concurrently. This is easily achieved by using on-chip memory blocks embedded in the FPGA fabric, which can either be instantiated manually into the system or inferred by declaring an array in the accelerated function.

Accelerator Compilation Process

The Nios II C2H Compiler operates in four main stages: (1) control/data flow graph analysis, (2) pointer analysis, (3) scheduling and pipelining, and (4) HDL generation. A discussion of each stage is provided in this section.

Control/Data Flow Graph Analysis

The Nios II C2H Compiler extracts instruction-level parallelism from the user's source code through construction and analysis of control and data flow graphs. DFG edges are analyzed using methods similar to those described by Coutinho, Jiang, and Luk in 2005 (see the Further Information section). A number of compiler optimizations are performed, including strength reduction, loop inversion, constant folding, constant propagation, and common subexpression elimination.

Pointer Analysis

Pointer analysis is of paramount importance to the performance of any compiler targeting a parallel architecture. Inability to determine that two pointers do not alias can sacrifice significant amounts of performance, because the two operations must be forced to execute sequentially, rather than in parallel. Although there are many cases where it is impossible for static analysis to extract precise points-to information, most pointers are resolvable at compile time.

The Nios II C2H Compiler does not attempt to resolve the exact location or variable that a pointer references, but instead simply attempts to determine whether or not two pointers overlap. This analysis is performed by extracting and then comparing the polynomial decomposition for the address expressions of each pointer operation. If two

expressions have a nonzero constant offset, then they do not overlap. This analysis is performed iteratively for pointer operations in loops, as the compiler implements pipelining of successive iterations and, therefore, must consider loop-carried edges.

In many cases, this analysis technique is sufficient for extracting parallelism from conventional C code. However, the Nios II C2H Compiler supports two simple methods by which the developer can provide additional information about pointers in their accelerated functions.

The Restrict Pointer Type Qualifier

The `restrict` type qualifier was introduced in the ISO C99 specification, and provides a means by which the user can instruct the compiler to ignore any aliasing implications of dereferencing the qualified pointer. It is not a Nios II C2H Compiler language extension. `restrict` is most widely known for its role in delineating `memcpy` from `memmove`—the pointer arguments to `memcpy` are qualified with `restrict`, whereas the pointer arguments to `memmove` are not.

Connection Pragma

The Nios II C2H Compiler introduces a single optional pragma that is used to specify connections between accelerator master ports and other slave ports in the system. However, this pragma is a placeholder for a more elegant solution, which is defined in the ISO TR18037 Technical Report on Extensions for the Programming Language C to support embedded processors. The technical report proposes support for system-specific space qualifiers to associate memory spaces with declarations. Altera plans to adopt these methods as TR18037 gains widespread recognition.

Latency-Aware Scheduling and Pipelining

Execution of operations in an accelerator is controlled by a hierarchy of finite state machines. One state machine is generated for every loop and subroutine contained in the target function. Thus, scheduling and pipelining is a matter of arranging operations inside this hierarchy and assigning state numbers to them based on the optimized data flow graph.

Similar to other C-to-gates methodologies, most assignments are translated into combinational logic representing the RHS expression, then registered for pipelining. There are two exceptions to this rule. First, complex arithmetic operations such as multiplication, division, barrel shifting, and pointer operations are extracted into separately pipelined subexpressions. Second, operations that are trivial to perform in hardware simply by manipulation of wires (such as constant-distance shifts, AND/OR by constant, etc.) are not registered.

Speculative Execution

When possible, operations in control blocks are executed speculatively and multiplexed using their condition expression at the merging of control paths. Operations such as volatile and write pointers cannot execute speculatively, and are scheduled with a dependency on their condition expressions.

Memory Latency-Aware Pipelining

When scheduling pointer operations that require memory transactions, the Nios II C2H Compiler queries the system description to determine read latency of connected slave ports. Based on the memory latency, the compiler is able to increase the computation time for the pointer operation, allowing other nondependent operations to be evaluated while the accelerator waits for `readdata` to return.

For example, consider the following function:

```
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;
}
```

Assume that `ptr_in` is connected to a slave with two cycles read latency. When scheduling this function, the Nios II C2H Compiler places the calculation of both `xy` and `xy_plus_z` in parallel with the read operation, because it knows to expect two cycles of latency before receiving data from memory. Figure 5 illustrates the dependencies and state assignments for this example.

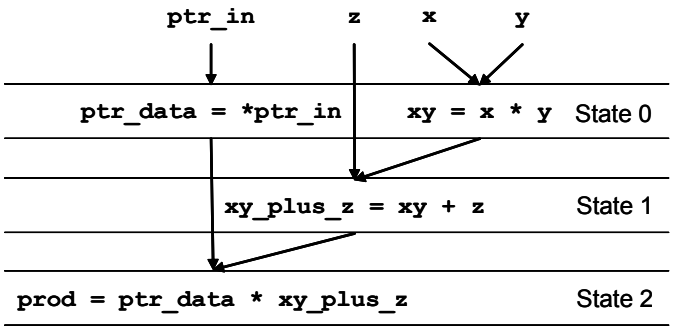

Figure 5. Latency-Aware Scheduling of a Read Operation With Two Cycles Latency

Latency-aware scheduling and pipelining is a powerful feature. If the compiler were unaware of specific memory latencies, then accelerators would either have to stall the pipeline in any case where data was not immediately ready, or insert an arbitrary number of wait states, which would be unnecessary in all other cases. The methodology nearly eliminates pipeline stalling due to memory latency by increasing the compiler's visibility into the run-time behavior of connected peripherals.

The two types of memory-based pipeline stalls that can occur are those that are unpredictable at compile time. They occur when: (1) a slave port has a variable amount of latency, or (2) another master port has been granted access to the slave during the time at which the accelerator requests it. In these two cases, the interconnect fabric signals the master to wait, and the accelerator stalls its pipeline. The compiler is often able to reduce these types of stalls by (1) pipelining variable-latency slaves based on their maximum number of outstanding read operations, and (2) consolidating master ports and appropriately scheduling memory operations that access the same slave port (so that two accelerator master ports do not end up contending with each other). However, though these optimizations reduce stalls, they are not ideal solutions. If possible, the designer should place critical data structures in fixed-latency (preferably low-latency) modules that have their own slave ports. This will allow for optimal latency-aware scheduling, and eliminate the bottleneck due to multiple-master contention for the same slave.

Loop Iteration Pipelining

The Nios II C2H Compiler pipelines execution of successive loop iterations by allowing multiple states in the loop's state machine to be active simultaneously. The frequency at which new iterations can enter the pipeline is determined by analysis of loop-carried dependencies. For example, if operation A, which occurs in the first state of the loop, is dependent on the result of operation B from the previous iteration, then new iterations cannot enter the pipeline until the state in which B is valid. This analysis is performed after the operations in a loop have been scheduled (assigned state numbers). It is possible to separate these two processes because state number assignment is concerned with dependency between two operations in the same iteration, whereas the loop iteration frequency calculation is concerned with dependency between two operations of different iterations.

Loop iteration pipelining enables the accelerator to issue multiple outstanding read operations to latent slaves, effectively hiding the memory latency in many cases. This is illustrated in the following example:

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

Assume again that the pointer is connected to a slave with two cycles read latency. The read operation will be scheduled on State 0 of the loop, and the accumulation of `sum` will be scheduled on State 2. However, iterations of the loop can enter through the pipeline once every cycle. Table 1 shows execution of the two operations during multiple iterations of this loop. After the initial latency of three clock cycles is overcome, the accelerator completes one iteration on every cycle. In case the accelerator state machine stalls during execution of the loop, the compiler generates a FIFO buffer behind the read master that collects incoming readdata values until they are used by downstream operations.

Table 1. Loop Iteration Pipelining Hiding Memory Latency of a Read Operation

Time	Iteration 0	Iteration 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	One cycle per loop iteration: memory latency hidden by loop pipelining.		...	(State 0) `readdata = *list++`
N+1			...	(State 1) (empty latency state)
N+2			...	(State 2) `sum += readdata`

HDL Generation

As the final step in the compilation process, the Nios II C2H Compiler generates synthesizable Verilog or VHDL for the data/control path, state machines, and Avalon interface ports. It also updates the SOPC Builder system description with Avalon interface information and optionally runs system generation to connect the hardware accelerator to other modules.

Results

To test the efficiency of the Nios II C2H Compiler, a few embedded software/hardware engineers were each given some common algorithms found in embedded computing. Their goal was to achieve the highest possible performance after 8 to 24 hours of debugging and optimization. The test users found that they could achieve integer speedup factors (3–7X) simply by performing a few optimizations in their C code, such as applying the `restrict` qualifier to pointers, reducing loop-carried dependencies, and consolidating control paths where possible. These three techniques significantly increased the amount of parallelism that the Nios II C2H Compiler was able to extract from the target function.

However, despite these significant optimizations, there came a point where performance was limited not by computational speed, but by availability of data. The problem was no longer compute-bound—it was memory bound. Because the accelerator was attempting to operate on multiple data buffers that were all stored in the CPU's data memory, the bandwidth of the single memory slave became the performance bottleneck. Users were able to overcome this limitation simply by adding on-chip memory buffers to their systems and using them to store critical data structures. Similarly, by using dedicated memory modules for input and output buffers, they allowed their accelerators to use many master ports simultaneously to quickly stream data in and out of the buffers.

When combined, code-optimization techniques of reducing dependencies and moving critical arrays into dedicated memory buffers proved extremely successful in increasing accelerator performance. These two techniques addressed the two types of potential bottlenecks—computation and memory. The methodology allows for elimination of both: (1) efficient scheduling and pipelining, as well as detailed critical-path reporting, for computational performance; and (2) generation of dedicated master ports and buffers, for increased memory bandwidth.

However, not all algorithms and C functions are suitable for hardware acceleration. Parallel/speculative execution and loop pipelining are two key ways in which performance is increased. Therefore, if the algorithm (or its implementation) limits the compiler's ability to do these two things, then only minimal speedup factors will result. For example, code that contains sequentially dependent operations in disjunct control paths (such as complex peripheral servicing routines) will not significantly benefit from hardware acceleration. These types of tasks are better-suited for running on a processor. In contrast, functions with simple control paths, and critical computations consolidated into a small number of inner loops that execute uninterrupted for many iterations are ideal candidates for conversion to hardware coprocessors. This means that it is very important for the developer to appropriately manage software/hardware trade-offs when introducing accelerators into a design—sequential tasks stay in the CPU and iterative computation-intensive tasks are moved to hardware.

Table 2 presents performance and area results for eight algorithms common in embedded computing. Speedup is calculated as the total algorithm compute time in software running on the Nios II processor divided by total compute time running in the accelerator. The system resource increase takes into account the logic element (LE) equivalent cost of on-chip hardware blocks such as multipliers and memories, and shows the incremental cost of adding the accelerator and buffers.

Table 2. User Test Results

Algorithm	Speedup (vs. Nios II CPU)	System F_{MAX} (MHz)	System Resource Increase
Autocorrelation	41.0X	115	142%
Bit Allocation	42.3X	110	152%
Convolutional Encoder	13.3X	95	133%
Fast Fourier Transform (FFT)	15.0X	85	208%
High Pass Filter	42.9X	110	181%
Matrix Rotation	73.6X	95	106%
RBG to CMYK	41.5X	120	84%
RBG to YIQ	39.9X	110	158%

This investigation shows that after one to three days of work, considerable performance gains of 13–73X can be achieved with the Nios II C2H Compiler, for approximately one to two times the increase in system resources. Furthermore, this experiment was performed in early stages of Nios II C2H Compiler development, before implementation of analysis and reporting functionality, which significantly reduces design time. Many optimizations have since been implemented that cause the compiler to be much more efficient with resource utilization.

Conclusion

This paper describes a methodology for converting ANSI/ISO standard C functions into custom hardware accelerators that integrate tightly with a general-purpose soft processor. When implemented as a plug-in to the

processor's IDE, it unifies the software and hardware design flows and allows for rapid debug and optimization iterations. By leveraging an existing infrastructure for system generation, a method was demonstrated for seamless integration of accelerators into the CPU's memory map and abstraction of the details of system connectivity. Based on this integration, this paper showed techniques that provide full support of pointers, as well as latency-aware pipelining of memory operations.

Also presented was an implementation of this methodology, the Nios II C2H Compiler, which is now available from Altera. User tests of the Nios II C2H Compiler revealed that it is possible, within hours, to achieve double-digit increases in performance after C2H acceleration.

Further Information

- Altera's Nios II embedded processors, version 6.0:
  www.altera.com/nios2
- Altera's Nios II C2H Compiler:
  www.altera.com/c2h
- ISO TR18037 Technical Report on Extensions to Support Embedded Processors:
  www.open-std.org/jtc1/sc22/wg14
- Coutinho, Jose Gabriel F., Jiang, Jun, and Luk, Wayne. Interleaving Behavioral and Cycle-Accurate Descriptions for Reconfigurable Hardware Compilation. In Proceedings of The IEEE Symposium on Field-Programmable Custom Computing Machines. (FCCM '05) (Napa, CA, April 18-20, 2005)
  www.fccm.org
- This white paper is based on an IEEE FCCM '06 conference paper, which is © IEEE.

101 Innovation Drive
San Jose, CA 95134
(408) 544-7000
http://www.altera.com