USING C-TO-HARDWARE ACCELERATION IN FPGAS FOR WAVEFORM BASEBAND PROCESSING

David Lau (Altera Corporation, San Jose, CA, dlau@altera.com)
Jarrod Blackburn, (Altera Corporation, San Jose, CA, jblackbu@altera.com)
Charlie Jenkins (Altera Corporation, San Jose, CA, chjenkin@altera.com)

ABSTRACT

Software-defined radio (SDR) architectures typically include general-purpose CPUs (GPPs), digital signal processing (DSP) ASSPs and FPGAs that process different waveforms, functions, and algorithms. GPPs typically handle network protocol processing and management functions. Historically, DSPs handled transceiver baseband processing and encoding, while FPGAs provided high-performance IF up/down conversion and preconditioning functions. Now FPGAs, when used with embedded soft-core processors, have absorbed the DSP baseband processing and some GPP functionality as well, providing a smaller, lower power solution. However, meeting the baseband performance requirements requires aggressive use of hardware acceleration. In this paper, we discuss an efficient methodology for hardware acceleration of SDR waveforms, the creation and use of hardware acceleration units, and a tool that automates the flow. The Altera® Nios® II C-to-Hardware (C2H) Acceleration Compiler is a coprocessor generation tool that converts performance-critical ANSI C functions into hardware accelerator modules with direct memory access. Results are presented showing performance gains of 13–73X over software only, offering a promising solution for rapid development of high-performance SDR systems.

1. INTRODUCTION

In the past, FPGAs were used as a convenient interconnect layer between chips in a system. In SDRs, FPGAs are now being used as programmable up/down intermediate frequency and signal processing hardware that boost performance while providing lower cost and lower power. Typical implementations of SDR modems include a GPP, DSP, and FPGA. Today's latest generation FPGAs can also be used to offload the GPP or DSP with application-specific hardware acceleration units. Soft-core microprocessors can easily extend their functionality with custom logic and hardware acceleration coprocessors added to the system. Furthermore, by using general-purpose routing resources available in the FPGA, these hardware acceleration units can run in parallel to further enhance the total computational throughput of the system.

Three different methods for hardware acceleration of SDR waveforms have been previously discussed at this conference: custom instructions, arithmetic coprocessing units, and application-specific instruction-set processors [3]. In this paper, we will focus on arithmetic coprocessing units and the automated design flow made possible by Altera's Nios II C2H Compiler. This compiler provides a pure-software design flow, automatically moving user-specified performance-critical functions from software running on the FPGA processor into pipelined, optimized hardware accelerators. These accelerators have direct access to the processor's memory subsystem and can sustain extremely high bandwidth through parallel transactions to an arbitrary number of buffers.

Sections 2 and 3 provide background information on SDR and system architecture. An efficient methodology for developing hardware coprocessors using a slave-side arbitration interconnect is discussed in Sections 4 and 5. In Sections 6 and 7, the automation of this flow with the Nios II C2H Compiler is discussed, followed by optimization strategies in Section 8, user test results in Section 9 and the summary in Section 10.

2. SOFTWARE-DEFINED RADIO

The concept behind SDR is that more waveform processing can be implemented in reprogrammable digital hardware so a single platform can be used for multiple waveforms. With the proliferation of wireless standards, future wireless devices will need to support multiple air interfaces and modulation formats. SDR technology enables such functionality in wireless devices by using a reconfigurable hardware platform across multiple standards.

SDR is the underlying technology behind the Joint Tactical Radio System (JTRS) initiative to develop software-programmable radios that enable seamless, real-time communication across the U.S. military services, and with coalition forces and allies. The functionality and expandability of the JTRS is built upon an open architecture

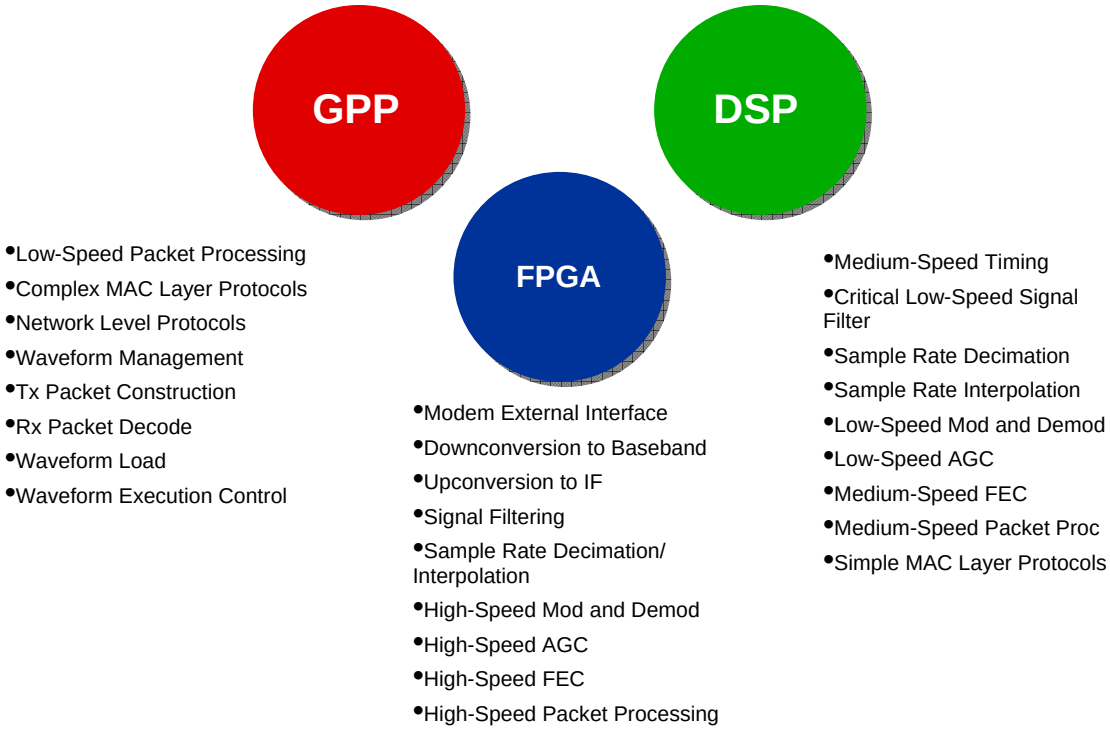

Figure 1. Example Architecture Splitting SDR Functions Across GPP, DSP, and FPGA

framework called the software communications architecture. The JTRS terminals must support dynamic loading of one or more of 25 specified air interfaces or waveforms, typically more complex than those used in the civilian sector. To achieve all these requirements in a reasonable form factor requires extensive, yet different processing powers.

3. SDR SYSTEM ARCHITECTURE

Most SDR systems utilize GPP, DSP, and FPGA in their architectures. These general-purpose processing resources can be used for different parts of the overall SDR system, and Figure 1 shows the typical functions found in an SDR divided across each of these devices. However, there is a significant amount of overlap between each of these elements. For example, an algorithm running on the DSP could be implemented in the GPP, albeit more slowly, or rewritten in HDL and run much faster in an FPGA as a coprocessor or hardware acceleration unit.

4. HARDWARE COPROCESSORS

Typical system design flow begins with a pure-software specification running on a CPU, then uses profiling to determine algorithmic bottlenecks. These bottlenecks can be reduced in several ways, including the creation of custom hardware accelerators that offload the CPU, exploiting parallelism in the algorithm to achieve significant performance increases. This method works best when dealing with complex computation-intensive algorithms that operate on large blocks of data. Unlike custom instructions, hardware accelerators are autonomous—once activated, they can run without intervention from the CPU, creating thread-level parallelism as well as instruction-level parallelism.

Accelerator modules contain data master ports that connect directly to the CPU's memory subsystem. With an FPGA-based interconnect (such as Altera's Avalon® memory mapped and streaming interfaces), it is possible to enable an arbitrary number of simultaneous transfers by arbitrating between multiple masters on the slave side. Accelerator modules can then be created with multiple master ports that simultaneously access different memory buffers, allowing much higher memory bandwidth than that achievable by a CPU with a single data master.

5. SYSTEM INTERCONNECT

The FPGA-based Avalon system interconnect fabric is automatically generated by Altera's SOPC Builder system integration tool. Similar to a fully switched system bus in functionality, it uses dynamically generated switches and wires to interconnect modules. These can be soft intellectual property (IP) blocks, custom user logic, or interfaces to off-chip peripherals. This approach overcomes the bottleneck

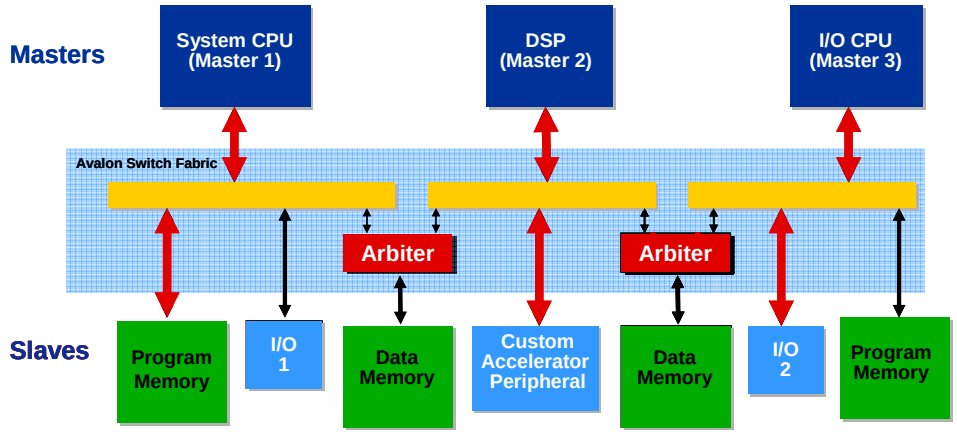

Figure 2. Avalon Switch Fabric Connecting Masters and Slaves in a System

encountered in the traditional shared-bus model, where only one master can issue a request at any given time. As shown in Figure 2, the switch fabric's automatically generated slave-side arbiters allow multiple masters to sustain transfers simultaneously to different slaves.

This is important in multi-CPU systems and systems with coprocessing units, as it mitigates the memory bottleneck created by bus contention. When data is separated into multiple partitions and/or dual-port buffers, it is possible to create multimaster systems in which little or no arbitration delays are incurred.

The Avalon switch fabric supports a rich feature set, including dynamic bus sizing, burst management, address decoding, datapath multiplexing, wait-state insertion, pipelining, clock domain crossing, and off-chip interfaces. (For more information on SOPC Builder, see [1], [2], and [5].)

6. AUTOMATED HARDWARE ACCELERATION

The Nios II C2H Compiler automates a significant portion of the design flow outlined above, by generating coprocessors that offload and enhance performance of a microprocessor running software written in pure ANSI C. It is tightly integrated into the software build flow and SOPC Builder system generation tool, allowing true pushbutton acceleration of performance-critical functions. The tool automatically integrates the accelerator into the hardware and software projects, providing a pure-software development environment for managing hardware/software partitioning.

Recursion, floating-point types, and the `goto` control statement are the only major exclusions from standard C. Pointers, arrays, structures, and enums, as well as all other loop types and control structures (including `break`, `continue`, and `return` statements) are fully supported. The Nios II C2H Compiler uses SOPC Builder 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 GUI for the compiler is the CPU's software integrated development environment (IDE). By supporting pointers and unextended 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.

Figure 3 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. It also shows the generation and selective linking of the accelerator driver (discussed in section 4) into the executable file. When prototyping and optimizing accelerators, running this complete process is not required. Options in the IDE provide for switching between the software-only build process, software plus Nios II C2H Compiler analysis/reporting, or software plus complete hardware. This allows for fast debug and optimization iterations during the early stages of development, as well as automated integration of the entire hardware flow during later stages.

7. DIRECT MEMORY ACCESS

The Nios II C2H Compiler also provides a unique solution with full support for pointers and array accesses. This is possible due to the integration with SOPC Builder, which gives the accelerated function access to the same memory map that it had when running in software. This is also

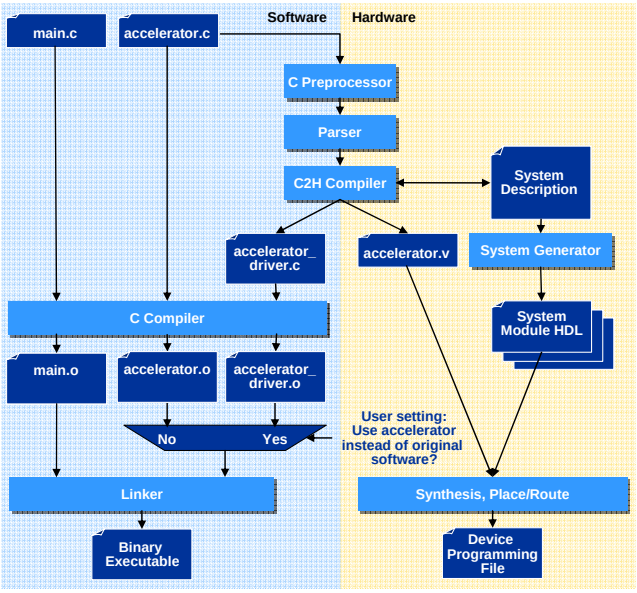

Figure 3. C2H Integration With Software Build

necessary for easy transfer of data between the accelerator and the CPU, as well as the other peripherals in the system.

The solution presented here imposes no restrictions on the bandwidth into and out of the accelerator, other than the bandwidth limitations of the connected memories. When the Nios II C2H Compiler creates hardware for a function, 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 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. Since an accelerator can have an arbitrary number of master ports, the memory bandwidth is limited only by the number of slave ports to which they can connect.

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 register banks and 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.

8. OPTIMIZING CODE FOR ACCELERATION

Test users found that they could increase their speed significantly (3–7X) by simply performing optimizations in the C code, such as applying the restrict qualifier to pointers

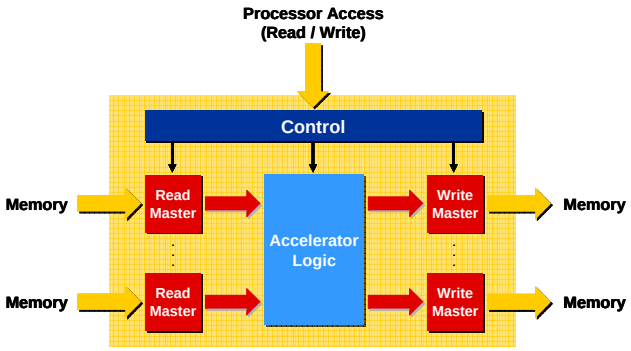

Figure 4. Accelerator Master Ports and Control Slave

(introduced in ISO C 99, specifying that a particular pointer will not alias another), reducing loop-carried dependencies, and consolidating control paths where possible. These three techniques considerably increased the amount of parallelism that the Nios II C2H Compiler could extract from the target function.

However, despite these significant optimizations, performance was limited not by computational speed, but by availability of data. The problem was no longer compute bound, but memory bound. Because the accelerator was attempting to operate on multiple data structures 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, the accelerators could use many master ports simultaneously to quickly stream data in and out of the buffers.

Combining code-optimization techniques by 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 I/O. The Nios II C2H Compiler methodology allows for elimination of both techniques, thus providing computational performance, efficient scheduling and pipelining, and detailed critical-path reporting, as well as increased memory bandwidth, and generation of dedicated master ports and buffers.

However, not all algorithms and C functions are suitable for hardware acceleration. Parallel/speculative execution and loop pipelining are two key ways by which performance is increased. Therefore, if the algorithm (or its implementation) limits the compiler in performing these two tasks, then only minimal speedup factors will result. For example, code that contains sequentially dependent operations in disjoint control paths (such as complex peripheral servicing routines) will not significantly benefit from hardware acceleration. These 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. Therefore, it is very important for developers to appropriately manage software/ hardware tradeoffs when introducing accelerators into a design, as sequential tasks stay in the CPU and iterative computation-intensive tasks are moved to hardware.

Table 1. User Test Results

Algorithm	Speedup (vs. Nios II CPU only)	Additional System Resource
FFT	17.1X	1.13X
FIR	31.8X	1.48X
QAM	25.3X	.74X
Viterbi	17.2X	.54X
Autocorrelation	41.0X	1.42X
Bit Allocation	42.3X	1.52X
Convolutional Encoder	13.3X	1.33X
Image Rotation	24.0X	2.08X
High Pass Filter	42.9X	1.81X
Matrix Rotation	73.6X	1.06X
RBG to CMYK	41.5X	.84X
RBG to YIQ	39.9X	1.58X

9. RESULTS

Table 1 presents performance and area results for twelve common signal-processing algorithms. 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 additional system resource column shows the incremental cost (in equivalent cost of logic elements for Nios II units) of adding the accelerator and on-chip hardware blocks such as multipliers, and memory buffers.

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

10. CONCLUSIONS

In this paper, we have discussed the benefits of using FPGAs for hardware acceleration to provide the computation power, as well as the portability and reconfigurability necessary in SDRs. Additionally, we reviewed the Nios II C2H Compiler, which automates the creation of hardware coprocessing units and allows the designer to easily manage hardware/software partitioning in a pure ANSI C environment. By creating accelerators that are efficiently pipelined to exploit parallelism in the algorithm, as well as multiple master ports for fetching and storing shared data, the Nios II C2H Compiler addresses both computational and memory bottlenecks while providing seamless switching between hardware and software implementations. Results for algorithms pertinent to SDR exhibit significant speedups with efficient resource utilization.

11. REFERENCES

[1] Altera Corp, *Quartus® II Version 6.0 Handbook, Volume 4: SOPC Builder*, Altera Corp., San Jose, CA, 2006.

[2] Altera Corp, Avalon Interface Specification, Altera Corp., San Jose, CA, 2005.

[3] D. Lau, J. Blackburn, and J. Seely, "The Use of Hardware Acceleration in SDR Waveforms," in *Proc. 2006 Software Defined Radio Technical Conference (SDR '05)*, (Orlando, FL, November 13-17, 2005).

[4] D. Lau, O. Pritchard, and P. Molson, "Automated Generation of Hardware Accelerators with Direct Memory Access from ANSI/ISO Standard C Functions," in *Proc. 2006 IEEE Symposium on Field-Programmable Custom Computing Machines (FCCM '06)*, (Napa, CA, April 24-26, 2006).

[5] D. Lau and O. Pritchard, "Rapid System-On-A-Programmable-Chip Development and Hardware Acceleration of ANSI C Functions," in *Proc. 16th International Conference on Field Programmable Logic and Applications (FPL 2006)*, (Madrid, Spain, August 28-30, 2006).