8. Hardware Acceleration and Coprocessing

ED51006-1.2

This chapter discusses how you can use hardware accelerators and coprocessing to create more efficient, higher throughput designs in SOPC Builder. This chapter discusses the following topics:

- Accelerating Cyclic Redundancy Checking (CRC)
- Creating Nios II Custom Instructions
- Using the C2H Compiler
- Creating Multicore Designs
- Pre- and Post-Processing
- Replacing State Machines

Hardware Acceleration

Hardware accelerators implemented in FPGAs offer a scalable solution for performance-limited systems. Other alternatives for increasing system performance include choosing higher performance components or increasing the system clock frequency. Although these other solutions are effective, in many cases they lead to additional cost, power, or design time.

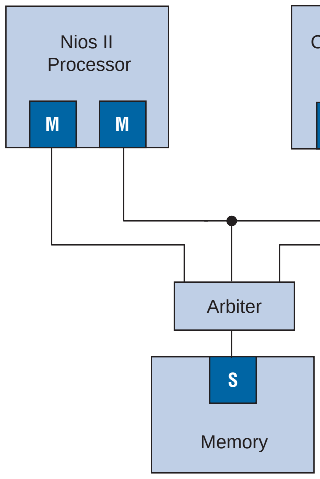
Accelerating Cyclic Redundancy Checking (CRC)

CRC is significantly more efficient in hardware than software; consequently, you can improve the throughput of your system by implementing a hardware accelerator for CRC. In addition, by eliminating CRC from the tasks that the processor must run, the processor has more bandwidth to accomplish other tasks.

Figure 8–1 illustrates a system in which a Nios® II processor offloads CRC processing to a hardware accelerator. In this system, the Nios II processor reads and writes registers to control the CRC using its Avalon® Memory-Mapped (Avalon-MM) slave port. The CRC component's Avalon-MM master port reads data for the CRC check from memory.

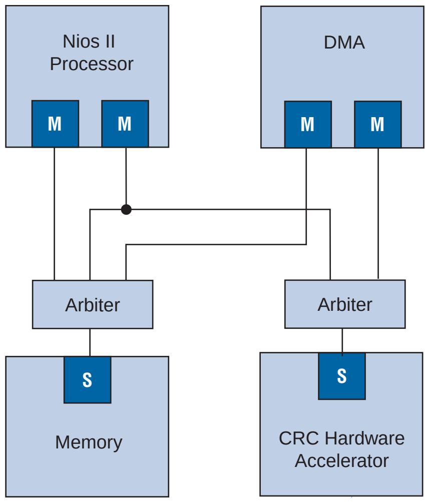
This design example and the HDL files to implement it are fully explained in the *SOPC Builder Component Development Walkthrough* chapter of the *SOPC Builder User Guide*.

Figure 8–1. A Hardware Accelerator for CRC



An alternative approach includes a dedicated DMA engine in addition to the Nios II processor. Figure 8–2 illustrates this design. In this system, the Nios II processor programs the DMA engine, which transfers data from memory to the CRC.

Figure 8–2. DMA and Hardware Accelerator for CRC



Although Figure 8–2 shows the DMA and CRC as separate blocks, you can combine them as a custom component which includes both an Avalon-MM master and slave port. You can import this component into your SOPC Builder system using the component editor.

To learn more about using component editor, refer to the *Component Editor* chapter of the *SOPC Builder User Guide*. You can find additional examples of hardware acceleration on Altera's Embedded Processor Design Examples web page.

Matching I/O Bandwidths

I/O bandwidth can have a large impact on overall performance. Low I/O bandwidth can cause a high-performance hardware accelerator to perform poorly when the dedicated hardware requires higher throughput than the I/O can support. You can increase the overall system performance by matching the I/O bandwidth to the computational needs of your system.

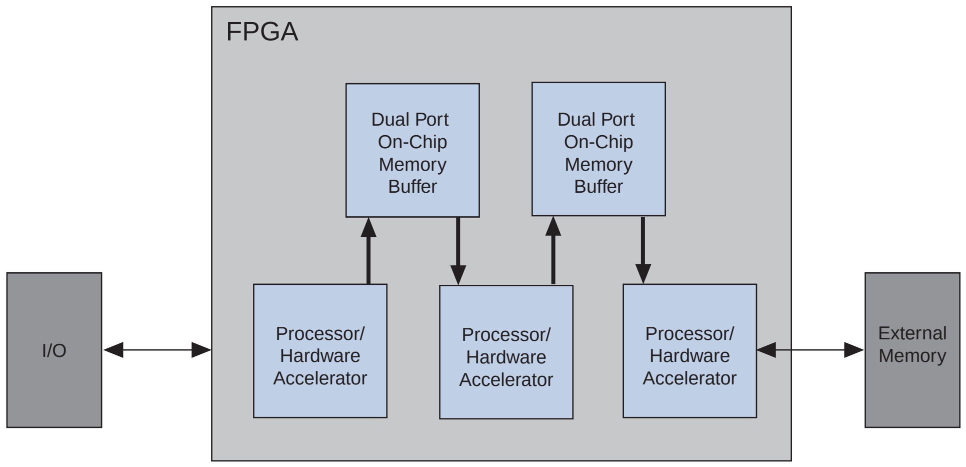
Typically, memory interfaces cause the most problems in systems that contain multiple processors and hardware accelerators. The following recommendations for interface design can maximize the throughput of your hardware accelerator:

- Match high performance memory and interfaces to the highest priority tasks your system must perform.
- Give high priority tasks a greater share of the I/O bandwidth if any memory or interface is shared.
- If you have multiple processors in your system, but only one of the processors provides real-time functionality, assign it a higher arbitration share.

Pipelining Algorithms

A common problem in systems with multiple Avalon-MM master ports is competition for shared resources. You can improve performance by pipelining the algorithm and buffering the intermediate results in separate on-chip memories. Figure 8–3 illustrates this approach. Two hardware accelerators write their intermediate results to on-chip memory. The third module writes the final result to an off-chip memory. Storing intermediate results in on-chip memories reduces the I/O throughput required of the off-chip memory. By using on-chip memories as temporary storage you also reduce read latency because on-chip memory has a fixed, low-latency access time.

Figure 8–3. Using On-Chip Memory to Achieve High Performance


FPGA
Dual Port On-Chip Memory Buffer
Dual Port On-Chip Memory Buffer
I/O
Processor/ Hardware Accelerator
Processor/ Hardware Accelerator
Processor/ Hardware Accelerator
External Memory

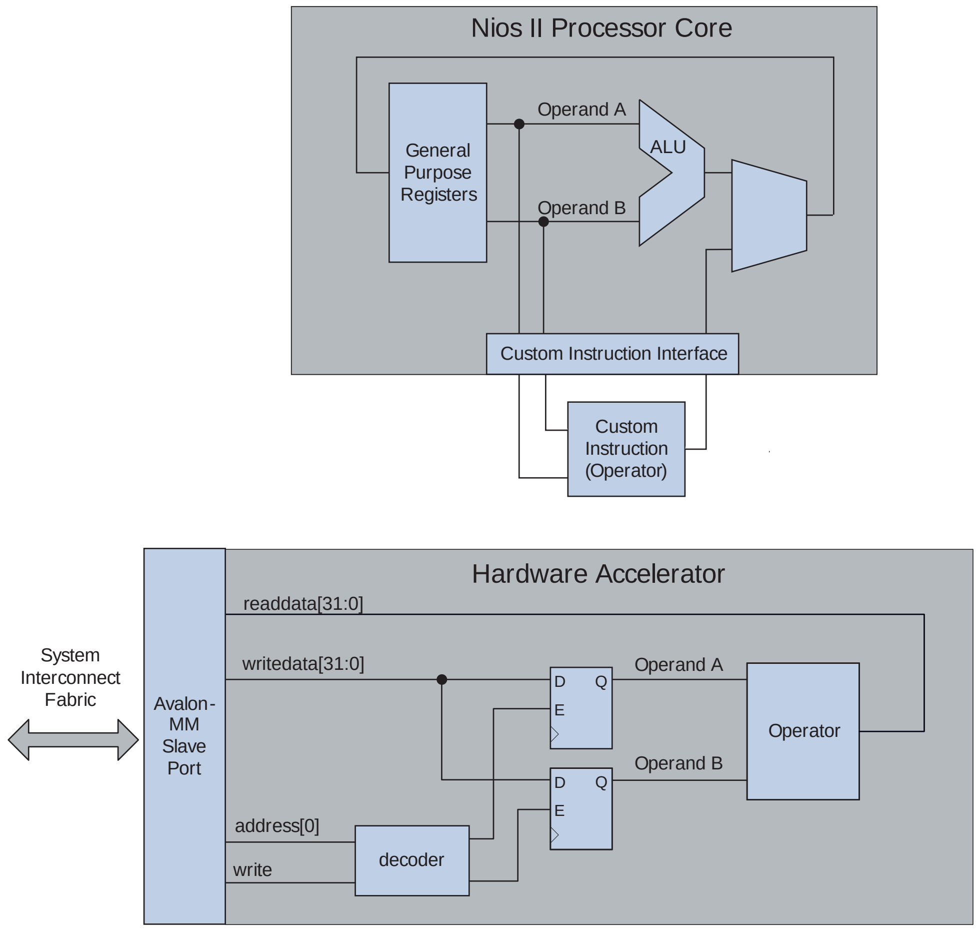
To learn more about the topics discussed in this section refer to the *System Interconnect Fabric for Memory-Mapped Interfaces* and *SOPC Builder Memory Subsystem Development Walkthrough* chapters of the *SOPC Builder User Guide*. To learn more about optimizing memory design refer to the *Memory System Design* chapter of the *Embedded Design Handbook.*

Creating Nios II Custom Instructions

The Nios II processor employs a RISC architecture which can be expanded with custom instructions. The Nios II processor includes a standard interface that you can use to implement your own custom instruction hardware in parallel with the arithmetic logic unit (ALU).

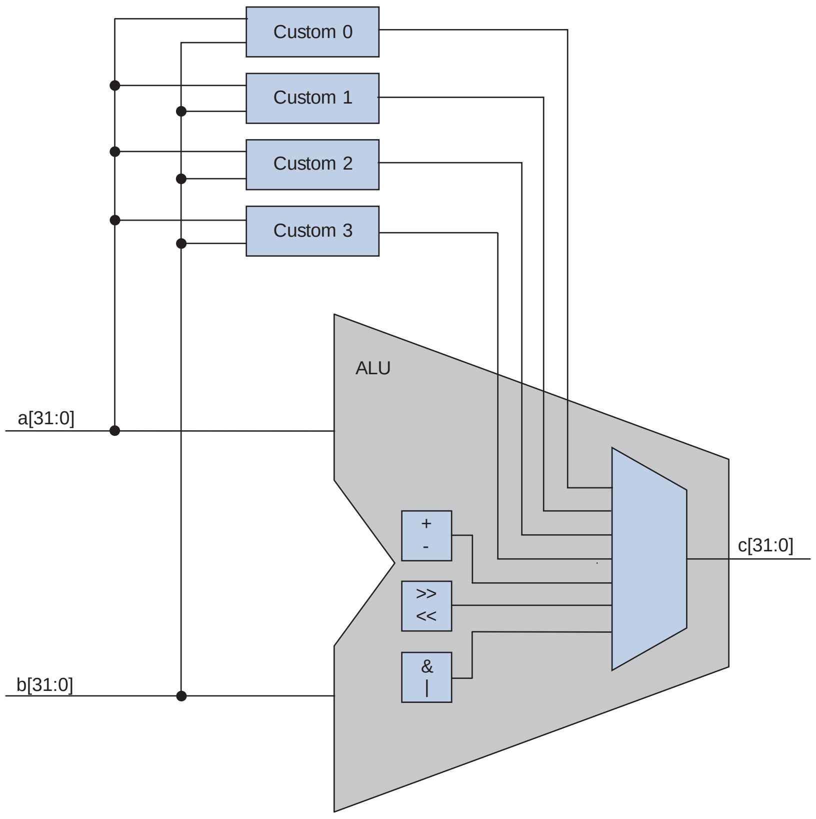
All custom instructions have a similar structure. They include up to two data inputs and one data output, and optional clock, reset, mode, address, and status signals for more complex multicycle operations. If you need to add hardware acceleration that requires many inputs and outputs, a custom hardware accelerator with an Avalon-MM slave port is a more appropriate solution. Custom instructions are blocking operations that prevent the processor from executing additional instructions until the custom instruction has completed. To avoid stalling the processor while your custom instruction is running, you can convert your custom instruction into a hardware accelerator with an Avalon-MM slave port. If you do so, the processor and custom peripheral can operate in parallel. Figure 8–4 illustrates the differences in implementation between a custom instruction and a hardware accelerator.

Figure 8–4. Implementation Differences between a Custom Instruction and Hardware Accelerator



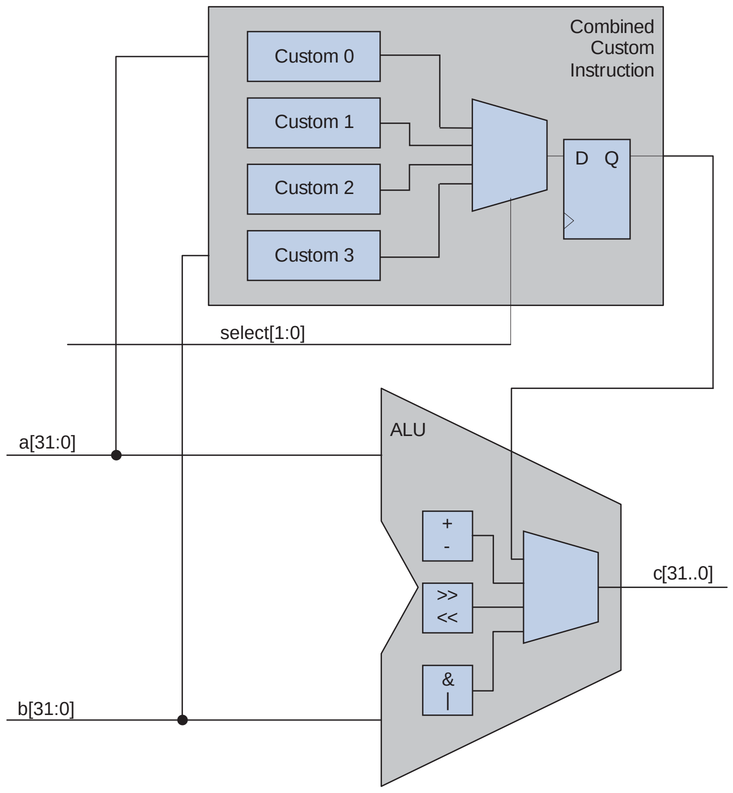
Because custom instructions extend the Nios II processor's ALU, the logic must meet timing or the f_{MAX} of the processor will suffer. As you add custom instructions to the processor, the ALU multiplexer grows in width as Figure 8–5 illustrates. This multiplexer selects the output from the ALU hardware (`c[31:0]` in Figure 8–5). Although you can pipeline custom instructions, you have no control over the automatically inserted ALU multiplexer. As a result, you cannot pipeline the multiplexer for higher performance.

Figure 8–5. Individual Custom Instructions



Instead of adding several custom instructions, you can combine the functionality into a single logic block as shown in Figure 8–6. When you combine custom instructions you use selector bits to select the required functionality. If you create a combined custom instruction, you must insert the multiplexer in your logic manually. This approach gives you full control over the multiplexer logic that generates the output. You can pipeline the multiplexer to prevent your combined custom instruction from becoming part of a critical timing path.

Figure 8–6. Combined Custom Instruction



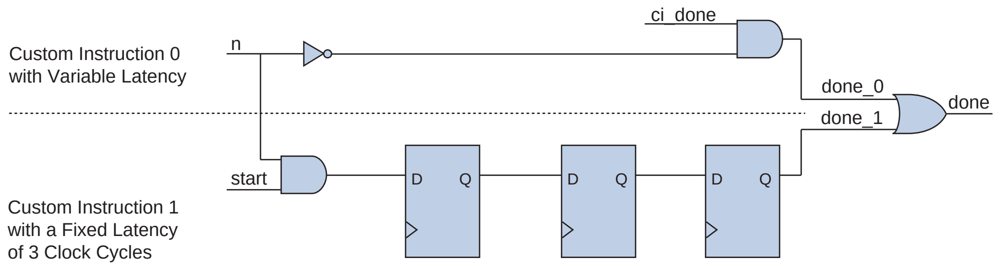
Notes to Figure 8–6:

(1) The Nios II compiler calls the select wires to the multiplexer *<n>*.

With multiple custom instructions built into a logic block, you can pipeline the output if it fails timing. To combine custom instructions, each must have identical latency characteristics.

Custom instructions are either fixed latency or variable latency. You can convert fixed latency custom instructions to variable latency by adding timing logic. Figure 8–7 shows the simplest method to implement this conversion by shifting the `start` bit by <*n*> clock cycles and logically `OR`ing all the `done` bits.

Figure 8–7. Sequencing Logic for Mixed Latency Combined Custom Instruction



Each custom instruction contains at least one custom instruction slave port, through which it connects to the ALU. A custom instruction slave is that slave port: the slave interface that receives the data input signals and returns the result. The custom instruction master is the master port of the processor that connects to the custom instruction.

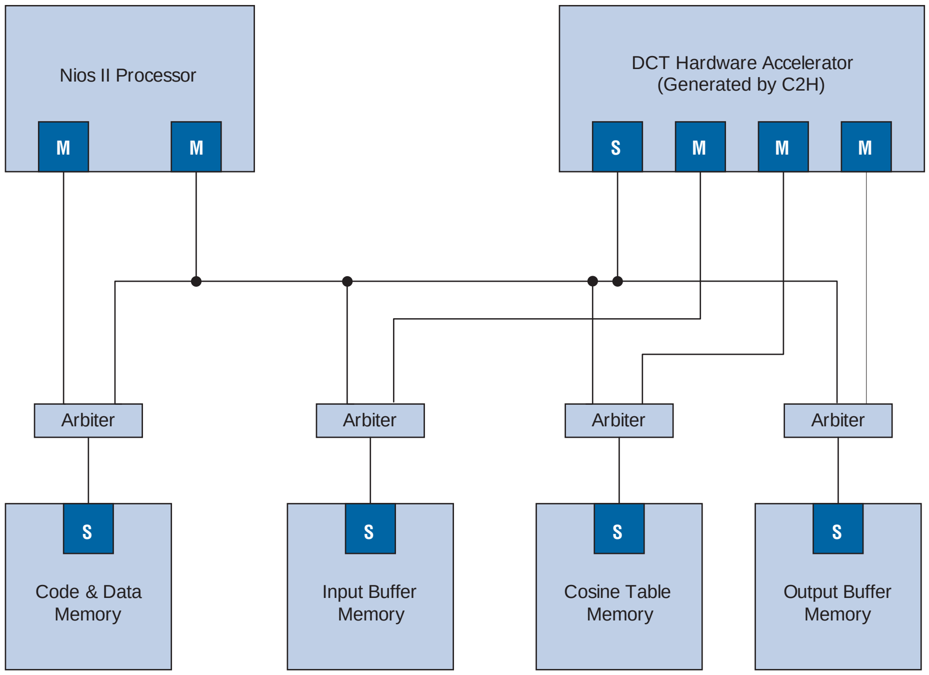
For more information about creating and using custom instructions refer to the *Nios II Custom Instruction User Guide*.

Using the C2H Compiler

You can use the Nios II C2H Compiler to compile your C source code into HDL synthesizable source code. SOPC Builder automatically places your hardware accelerator in your system. SOPC Builder automatically connects all the master ports to the necessary memories and connects the Nios II processor data master port to the accelerator slave port which is used to transfer data.

Choose the Nios II C2H Compiler instead of custom instructions when your algorithm requires access to memory. The C2H Compiler creates Avalon-MM masters that access memory. If your algorithm accesses several memories, the C2H Compiler creates a master per memory access, allowing you to benefit from the concurrent access feature of the system interconnect fabric. You can also use the C2H Compiler to create hardware accelerators that are non-blocking so that you can use the accelerator in parallel with other software functionality.

Figure 8–8. C2H Discrete Cosine Transform (DCT) Block Diagram



In figure Figure 8–8 the two-dimensional DCT algorithm is accelerated to offload a Nios II processor. The DCT algorithm requires access to input and output buffers as well as a cosine lookup table. Assuming that each resides in separate memories, the hardware accelerator can access all three memories concurrently.

For more information please refer to the *Nios II C2H Compiler User Guide* and the *Optimizing C2H Compiler Results* chapter in the *Embedded Design Handbook*. There are also C2H examples available on the Altera website.

Coprocessing

Partitioning system functionality between a Nios II processor and hardware accelerators or between multiple Nios II processors in your FPGA can help you control costs. The following sections demonstrate how you can use coprocessing to create high performance systems.

Creating Multicore Designs

Multicore designs combine multiple processor cores in a single FPGA to create a higher performance computing system. Typically, the processors in a multicore design can communicate with each other. Designs including the Nios II processor can implement inter-processor communication, or the processors can operate autonomously.

When a design includes more than one processor you must partition the algorithm carefully to make efficient use of all of the processors. The following example includes a Nios II-based system that performs video over IP, using a network interface to supply data to a discrete DSP processor. The original design overutilizes the Nios II processor. The system performs the following steps to transfer data from the network to the DSP processor:

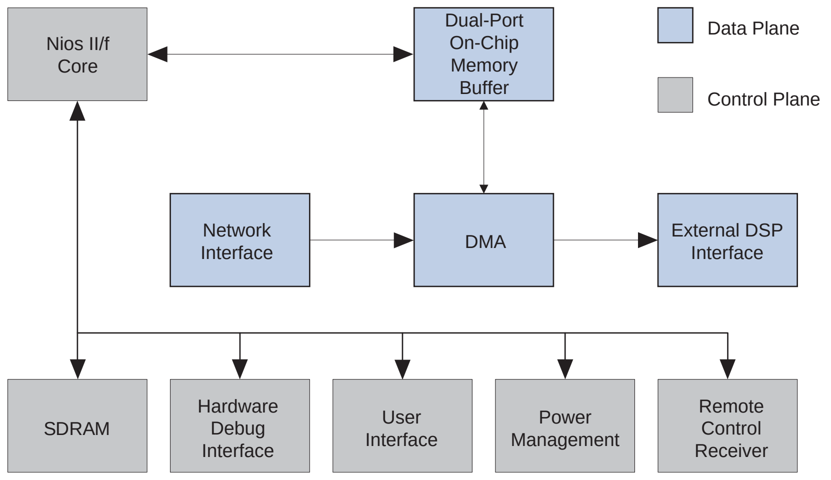
1. The network interface signals when a full data packet has been received.
2. The Nios II processor uses a DMA engine to transfer the packet to a dual-port on-chip memory buffer.
3. The Nios II processor processes the packet in the on-chip memory buffer.
4. The Nios II processor uses the DMA engine to transfer the video data to the DSP processor.

In the original design, the Nios II processor is also responsible for communications with the following peripherals that include Avalon-MM slave ports:

- Hardware debug interface
- User interface
- Power management
- Remote control receiver

Figure 8–9 illustrates this design.

Figure 8–9. Over-Utilized Video System

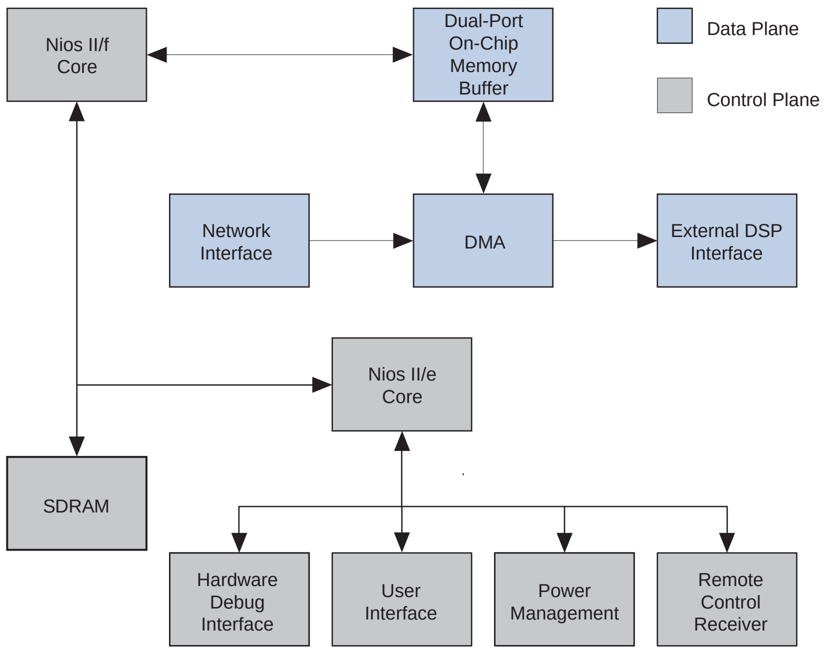


Adding a second Nios II processor to the system, allows the workload to be divided so that one processor handles the network functionality and the other the control interfaces. Figure 8–10 illustrates the revised design.

Because the revised design has two processors, you must create two software projects; however, each of these software projects handles fewer tasks and is simpler to create and maintain. You must also create a mechanism for inter-processor communication. The inter-processor communication in this system is relatively simple and is justified by the system performance increase.

For more information about designing hardware and software for inter-processor communication, refer to the *Creating Multiprocessor Nios II Systems Tutorial* and the *Peripherals* section of the *Embedded Peripherals IP User Guide.* Refer to the *Nios II Processor Reference Handbook* for complete information about this soft core processor. A Nios II Multiprocessor Design Example is available on the Altera website.

Figure 8–10. High Performance Video System



In Figure 8–10, the second Nios II processor added to the system performs primarily low-level maintenance tasks; consequently, the Nios II/e core is used. The Nios II/e core implements only the most basic processor functionality in an effort to trade off performance for a small hardware footprint. This core is approximately one-third the size of the Nios II/f core.

To learn more about the three Nios II processor cores refer to the *Nios II Core Implementation Details* chapter in the *Nios II Processor Reference Handbook*.

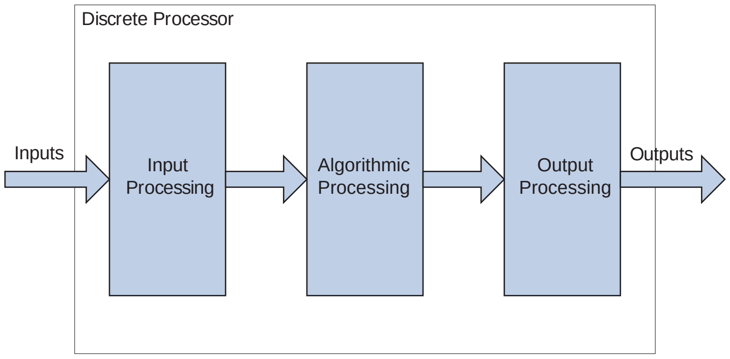
Pre- and Post-Processing

The high performance video system illustrated in Figure 8–10 distributes the workload by separating the control and data planes in the hardware. Figure 8–11 illustrates a different approach. All three stages of a DSP workload are implemented in software running on a discrete processor. This workload includes the following stages:

- Input processing—typically removing packet headers and error correction information
- Algorithmic processing and error correction—processing the data
- Output processing—typically adding error correction, converting data stream to packets, driving data to I/O devices

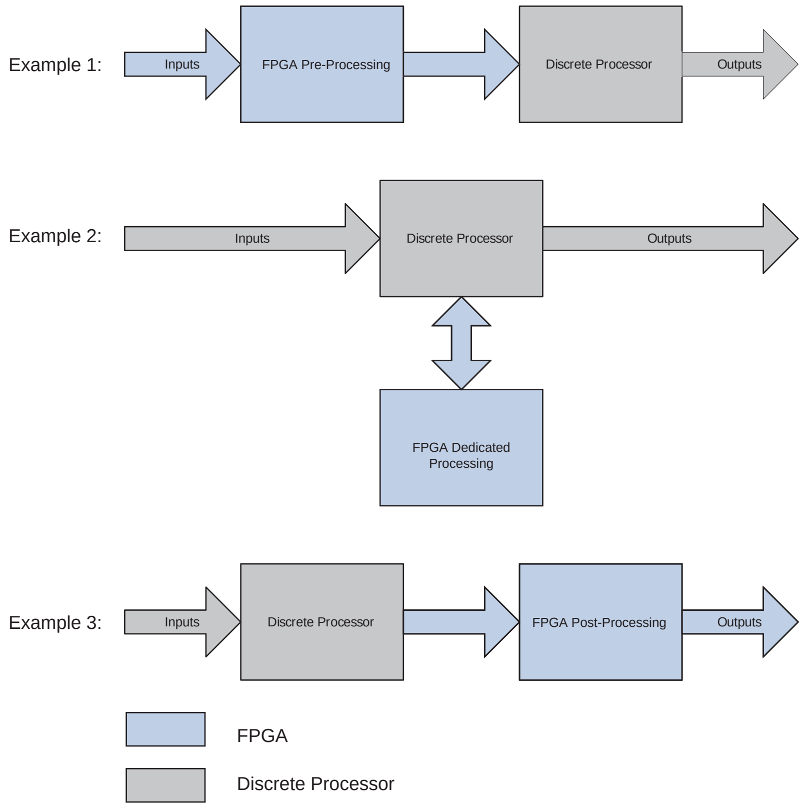
By off loading the processing required for the inputs or outputs to an FPGA, the discrete processor has more computation bandwidth available for the algorithmic processing.

Figure 8–11. Discrete Processing Stages



If the discrete processor requires more computational bandwidth for the algorithm, dedicated coprocessing can be added. Figure 8–12 below shows examples of dedicated coprocessing at each stage.

Figure 8–12. Pre- Dedicated, and Post-Processing



Replacing State Machines

You can use the Nios II processor to implement scalable and efficient state machines. When you use dedicated hardware to implement state machines, each additional state or state transition increases the hardware utilization. In contrast, adding the same functionality to a state machine that runs on the Nios II processor only increases the memory utilization of the Nios II processor.

A key benefit of using Nios II for state machine implementation is the reduction of complexity that results from using software instead of hardware. A processor, by definition, is a state machine that contains many states. These states can be stored in either the processor register set or the memory available to the processor; consequently, state machines that would not fit in the footprint of a FPGA can be created using memory connected to the Nios II processor.

When designing state machines to run on the Nios II processor, you must understand the necessary throughput requirements of your system. Typically, a state machine is comprised of decisions (transitions) and actions (outputs) based on stimuli (inputs). The processor you have chosen determines the speed at which these operations take place. The state machine speed also depends on the complexity of the algorithm being implemented. You can subdivide the algorithm into separate state machines using software modularity or even multiple Nios II processor cores that interact together.

Low-Speed State Machines

Low-speed state machines are typically used to control peripherals. The Nios II/e processor pictured in Figure 8–10 on page 8–10 could implement a low speed state machine to control the peripherals.

Even though the Nios II/e core does not include a data cache, Altera recommends that the software accessing the peripherals use data cache bypassing. Doing so avoids potential cache coherency issues if the software is ever run on a Nios II/f core that includes a data cache.

For information about data cache bypass methods, refer to the *Processor Architecture* chapter of the *Nios II Processor Reference Handbook*.

State machines implemented in SOPC Builder require the following components:

- A Nios II processor
- Program and data memory
- Stimuli interfaces
- Output interfaces

The building blocks you use to construct a state machine in SOPC Builder are no different than those you would use if you were creating a state machine manually. One noticeable difference in the SOPC Builder environment is accessing the interfaces from the Nios II processor. The Nios II processor uses an Avalon-MM master port to access peripherals. Instead of accessing the interfaces using signals, you communicate via memory-mapped interfaces. Memory-mapped interfaces simplify the design of large state machines because managing memory is much simpler than creating numerous directly connected interfaces.

For more information about the Avalon-MM interface, refer to the *Avalon Interface Specifications*.

High-Speed State Machines

You should implement high throughput state machine using a Nios II/f core. To maximize performance, focus on the I/O interfaces and memory types. The following recommendations on memory usage can maximize the throughput of your state machine:

- Use on-chip memory to store logic for high-speed decision making.
- Use tightly-coupled memory if the state machine must operate with deterministic latency. Tightly-coupled memory has the same access time as cache memory; consequently, you can avoid using cache memory and the cache coherency problems that might result.

For more information about tightly-coupled memory, refer to the *Cache and Tightly-Coupled Memory* chapter of the *Nios II Software Developer's Handbook*.

Subdivided State Machines

Subdividing a hardware-based state machine into smaller more manageable units can be difficult. If you choose to keep some of the state machine functionality in a hardware implementation, you can use the Nios II processor to assist it. For example, you may wish to use a hardware state machine for the high data throughput functionality and Nios II for the slower operations. If you have partitioned a complicated state machine into smaller, hardware based state machines, you can use the Nios II processor for scheduling.

Document Revision History

Table 8–1 shows the revision history for this document.

Table 8–1. Document Revision History

Date	Version	Changes
July 2011	1.2	Updated references and clarified meaning of “custom instruction slave” and “custom instruction master”.
June 2008	1.1	Corrected Table of Contents.
March 2008	1.0	Initial release.