Application Note

Using DMA for Data Communications on Altera Nios II Platforms

Ralph Bodenner, Senior Applications Engineer
Impulse Accelerated Technologies, Inc.
Scott Thibault, President
Green Mountain Computing Systems, Inc.

Overview

This application note describes how to use shared memory for hardware-software communications on the Altera Nios II platform. The example presented in this application note is a simple image filter that performs an edge-detection function. You can modify this example as needed to create other, more complex image filters or other similar functions.

The key concept demonstrated in this application note is the use of shared memory. Shared memory, or direct memory access (DMA), can be a useful way to improve application performance for certain types of streaming applications and can also be useful for operating on non-streaming data shared between hardware and software processes. Note that the use of memory is platform-specific: the performance tradeoffs and available memory resources will vary greatly, depending on the FPGA platform you are targeting.

The Image Filter Application

The specific convolution performed in this example is an edge-detection function, in which a 3x3-pixel "window" is assembled and processed for each pixel in the source image. The algorithm is represented by two pipelined hardware processes that decompose the image data into three rows of image data and process those rows to calculate a resulting value from the 3x3-pixel window. Two additional hardware processes are used to read and write image data from shared memory and to present the data to the image processing algorithm as a stream. The filter operation is diagrammed in Figure 1.

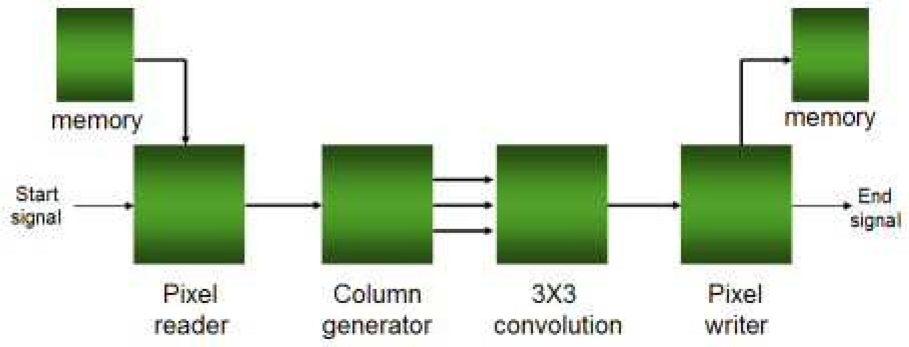

Figure 1. *Edge detection image filter with DMA*

These four processes and the stream, memory, and signal components are described using Impulse C and interconnected using the configuration function shown below:

```
void config_img(void *arg)
{
    int error;
    co_signal startsig, donesig;
    co_memory shrmem;
    co_stream istream, row0, row1, row2, ostream;
    co_process reader, writer;
    co_process cpu_proc, prep_proc, filter_proc;

    startsig = co_signal_create("start");
    donesig  = co_signal_create("done");
    shrmem   = co_memory_create("image", "heap0",
                                 IMG_WIDTH * IMG_HEIGHT * sizeof(uint16));
    istream  = co_stream_create("istream", INT_TYPE(32), IMG_HEIGHT/2);
    row0     = co_stream_create("row0",    INT_TYPE(32), 4);
    row1     = co_stream_create("row1",    INT_TYPE(32), 4);
    row2     = co_stream_create("row2",    INT_TYPE(32), 4);
    ostream  = co_stream_create("ostream", INT_TYPE(32), IMG_HEIGHT/2);

    cpu_proc    = co_process_create("cpu_proc",  (co_function)call_fpga,
                                                3, shrmem, startsig, donesig);
    reader      = co_process_create("reader",    (co_function)to_stream,
                                                3, startsig, shrmem, istream);
    prep_proc   = co_process_create("prep_proc", (co_function)prep_run,
                                                4, istream, row0, row1, row2);
    filter_proc = co_process_create("filter",    (co_function)filter_run,
                                                4, row0, row1, row2, ostream);
    writer      = co_process_create("writer",    (co_function)from_stream,
                                                3, ostream,  shrmem,   donesig);

    co_process_config(reader,      co_loc, "pe0");
    co_process_config(prep_proc,   co_loc, "pe0");
    co_process_config(filter_proc, co_loc, "pe0");
    co_process_config(writer,      co_loc, "pe0");

    IF_SIM(error = cosim_logwindow_init();)
}
```

Note that a fifth process (**call_fpga**) is included that represents the controlling software application running on the embedded Nios II processor. (The processes themselves are not shown here, but complete source code can be found in the CoDeveloper Examples\Altera\NiosII\ImageFilterDMA directory.)

Why Use DMA?

Using DMA can improve the performance of some hardware/software applications. The image filter application is an excellent candidate for the use of DMA for two reasons:

- A large block of data (the image pixels) must be processed in hardware
- The data is shared between hardware (the filter) and software running on the embedded processor (the controlling application)

When sharing data between hardware and software processes, whether using streams or shared memories, it is important to remember that data transactions run over a system bus (the Avalon bus, in this case). In general, it is more efficient to move a large block of data in a few shared memory operations than it is to move many small blocks of data in many operations over the bus. Once the data have reached the hardware processes, they move through very efficiently on direct process-to-process connections using streams and signals.

In the image filter application, image data are acquired by the software running on the Nios II processor and then read, in row-sized blocks, from shared memory by the hardware image filter. Once in the hardware, the pixel data are streamed quickly through a series of pipelined processes that perform the edge-detection function and write the resulting image back to shared memory.

An alternative implementation of the image filter application uses streams to move pixel data from software to the hardware filter. Due to bus transfer overhead, this purely streaming implementation is much slower than an application that uses DMA to pull data into the hardware filter. Internal testing by Impulse Accelerated Technologies showed the DMA image filter gaining a 41x performance advantage over the pure streaming implementation (21 vs. 875 ms execution time for a 512x512-pixel image). This is due to the overhead associated with polling of streams on the Avalon bus.

There are situations where using DMA may not provide a performance boost. A DMA transaction on the Avalon bus blocks all other bus traffic, so if your application has many peripherals or shared memories vying for access to the bus then the overall performance of the application may suffer. With a high level of bus contention, it may be more efficient to use streams or smaller DMA transfers for hardware/software communication.

Using the DMA Interface

To use DMA in your Impulse C application, you must create a shared memory object and call Impulse C library functions in your hardware and software processes to read and write blocks of memory.

Creating a Shared Memory Object

A shared memory is defined as a **co_memory** type and created in the Impulse C configuration function using the **co_memory_create** function, which takes three parameters:

```
    co_memory co_memory_create(const char *name, const char *loc, size_t size);
```

- const char * **name**: A programmer-defined name, used for identifying the memory..
- const char * **loc**: Location of the memory in hardware.
- size_t **size**: Size of the shared memory, in bytes.

The **loc** parameter determines which physical memory is used for shared memory operations. The Nios II Platform Support Package supports one memory location ("heap0"), which is interpreted as the memory whose space is allocated with the **malloc** system call. In the Altera Nios II IDE, you may specify the physical memory component that the processor uses for data memory—this memory is "heap0" in Impulse C. Several **co_memory** objects may share a single physical memory, though the image filter application only requires one shared memory to store pixel data:

```
    shrmem   = co_memory_create("image", "heap0", IMG_WIDTH * IMG_HEIGHT
                                                 * sizeof(uint16));
```

(Hardware resources, such as shared memories, are defined in the platform definition file of a Platform Support Package. See the file **Architectures/altera_nios2.xml** in your CoDeveloper installation to see how "heap0" is related to **malloc** in the Nios II platform.)

Using DMA in Software Processes

In software processes, the DMA interface consists of calls to three Impulse C functions whose names begin with "co_memory_": **co_memory_readblock**, **co_memory_writeblock**, and **co_memory_ptr**. When executed in a Nios II application, these functions operate on shared memory via the embedded processor.

The **co_memory_writeblock** and **co_memory_readblock** functions have the same set of parameters, as follows:

```
    void co_memory_writeblock(co_memory mem, unsigned int offset, void *buf, size_t
buffersize);

    void co_memory_readblock(co_memory mem, unsigned int offset, void *buf, size_t
buffersize);
```

- co_memory **mem**: The shared memory to operate on, created with **co_memory_create**.
- unsigned int **offset**: Byte offset (address) to begin the read/write operation from.
- void ***buf**: Destination or source, respectively, of the data read or written.
- size_t **buffersize**: Number of bytes to read or write.

co_memory_ptr returns the base address of the shared memory and is useful for iterating over shared memory using pointer arithmetic:

```
    void *co_memory_ptr(co_memory mem);
```

For example, this code, from a software process in the image filter application, writes the input image to shared memory:

```
uint16 *inp, *datain;
unsigned short testimage16[1024] = /* ... */ ;
datain = co_memory_ptr(imgmem);
k = 0;
for ( i = 0; i < IMG_HEIGHT; i++ ) {
    for ( j = 0; j < IMG_WIDTH; j++ ) {
        if ( (j%32) == 0 ) inp = testimage16 + (i % 32) * 32;
            datain[k++] = *(inp++);
    }
}
```

Using DMA in Hardware Processes

Use the **co_memory_readblock** and **co_memory_writeblock** functions in hardware processes the same way you use them in software processes, with one subtle difference: the **buf** parameter must be a C array identifier, not the address of a variable or an array element. The following syntax is acceptable:

```
co_int32 buffer[64];
co_memory_readblock(mem, 0, buffer, 64 * sizeof(co_int32));
```

These two examples, however, will cause CoBuilder to report an error:

```
co_int32 datum;
co_int32 list[64];
co_memory_readblock(mem, 0, &datum, 64 * sizeof(co_int32));
// Inefficient use of shared memory and also illegal
for ( i = 0; i < 64; i++ ) {
    co_memory_readblock(mem, i * sizeof(co_int32), &list[i], sizeof(co_int32));
}
```

Note that **co_memory_ptr** is not suitable for use in hardware. Calling this function in a hardware process will cause CoBuilder to report an error.

CoBuilder translates local memory buffers and **co_memory** function calls into RTL. A platform-specific RAM entity will be instantiated in RTL for every C array in a hardware process that serves as a buffer for a shared memory transaction. For example, the C array variable **row** will be translated into a block RAM called **to_stream_row_RAM**:

```
#define IMG_WIDTH 512
/* img_hw.c */
void to_stream(co_signal go, co_memory imgmem, co_stream output_stream)
{
    uint32 offset = 0;
    uint32 row[IMG_WIDTH / 2];
    /* ... */
    for ( i = 0; i < IMG_HEIGHT; i++ ) {
        co_memory_readblock(imgmem, offset, row, IMG_WIDTH * sizeof(int16));
    /* ... */
}
```

```
-- img_comp.vhd
entity to_stream_row_RAM is
  port (
    rst,clk : in std_ulogic;
    we : in std_ulogic;
    w_addr : in std_ulogic_vector(7 downto 0);
    r_addr : in std_ulogic_vector(7 downto 0);
    din : in std_ulogic_vector(31 downto 0);
    dout : out std_ulogic_vector(31 downto 0)
  );
end to_stream_row_RAM;
```

Such RAM blocks are connected to a shared Avalon DMA interface (**image_if**). The name of the DMA interface component, defined in **subsystem.vhd**, is derived from the value of **co_memory_create's name** parameter.

Calls to the Impulse C **co_memory_writeblock** and **co_memory_readblock** functions are translated into RTL as DMA operations that write or read data between the shared memory and the hardware RAM blocks, over the bus.

Connecting Hardware and Software

The shared memory base address is the information that ties the software and hardware interfaces together. **co_memory_create** allocates space for the shared memory and obtains the base address. For each memory, the hardware interface contains a configuration register to which the base address is written at runtime by the **co_initialize** function generated by CoBuilder in **co_init.c**. Hardware operations on shared memory add the **offset** to the base address stored in this register. Software processes access the base address internally via the **co_memory** object.

Additional Resources

Complete source code for this example is available in the CoDeveloper installation under the **Examples/Altera/NiosII/ImageFilterDMA** directory.

A detailed description of each of the Impulse C **co_memory** functions is available in the *Impulse C User's Guide*, accessible from CoDeveloper's Help menu.

Summary

This application note has described how to use the Impulse C shared memory (DMA) interface for data communication between hardware and software processes. Using DMA can result in dramatic increases in performance for many types of applications, and should be evaluated as an alternative to streams for software-to-hardware data movement. DMA is platform-specific, however, so be sure to consult your Platform Support Package documentation for specific information and examples.