

Washington University in St. Louis
Washington University Open Scholarship

All Computer Science and Engineering Research

Computer Science and Engineering

Report Number: WUCSE-2012-40

2012

Common DMA Engine Interface

Authors: Roger Alessi

Circuit boards with Field Programmable Gate Arrays (FPGAs) have a historically diverse set of standards for communicating with other devices. This provides a challenge for developers creating FPGA applications and makes migration of applications from one board or FPGA to another difficult. Many board manufacturers, including Xilinx and GiDEL, create boards with Peripheral Component Interconnect Express (PCIe) buses. PCIe provides a low-level standard for transferring data between a Central Processing Unit (CPU) and an FPGA, and these manufacturers have designed Direct Memory Access (DMA) engines to implement the standard. However, each manufacturer's DMA engines do not share a standard interface, leading to the same difficulties switching between boards or FPGAs. The goal of this project is to reach a common interface for the diverse set of available DMA engines to improve application development productivity, particularly for the Autopipe environment, an environment for developing multi-device streaming applications.

... **Read complete abstract on page 2.**

Follow this and additional works at: http://openscholarship.wustl.edu/cse_research



Part of the [Computer Engineering Commons](#), and the [Computer Sciences Commons](#)

Recommended Citation

Alessi, Roger, "Common DMA Engine Interface" Report Number: WUCSE-2012-40 (2012). *All Computer Science and Engineering Research*.

http://openscholarship.wustl.edu/cse_research/81

Common DMA Engine Interface

Complete Abstract:

Circuit boards with Field Programmable Gate Arrays (FPGAs) have a historically diverse set of standards for communicating with other devices. This provides a challenge for developers creating FPGA applications and makes migration of applications from one board or FPGA to another difficult. Many board manufacturers, including Xilinx and GiDEL, create boards with Peripheral Component Interconnect Express (PCIe) buses. PCIe provides a low-level standard for transferring data between a Central Processing Unit (CPU) and an FPGA, and these manufacturers have designed Direct Memory Access (DMA) engines to implement the standard. However, each manufacturer's DMA engines do not share a standard interface, leading to the same difficulties switching between boards or FPGAs. The goal of this project is to reach a common interface for the diverse set of available DMA engines to improve application development productivity, particularly for the Autopipe environment, an environment for developing multi-device streaming applications.

2012-40

Common DMA Engine Interface

Authors: Roger Alessi

Abstract: Circuit boards with Field Programmable Gate Arrays (FPGAs) have a historically diverse set of standards for communicating with other devices. This provides a challenge for developers creating FPGA applications and makes migration of applications from one board or FPGA to another difficult.

Many board manufacturers, including Xilinx and GiDEL, create boards with Peripheral Component Interconnect Express (PCIe) buses. PCIe provides a low-level standard for transferring data between a Central Processing Unit (CPU) and an FPGA, and these manufacturers have designed Direct Memory Access (DMA) engines to implement the standard. However, each manufacturer's DMA engines do not share a standard interface, leading to the same difficulties switching between boards or FPGAs.

The goal of this project is to reach a common interface for the diverse set of available DMA engines to improve application development productivity, particularly for the Autopipe environment, an environment for developing multi-device streaming applications.

Type of Report: MS Project Report

Common DMA Engine Interface

Master's Project Report

Roger Alessi

May 7th, 2012

Introduction

Circuit boards with Field Programmable Gate Arrays (FPGAs) have a historically diverse set of standards for communicating with other devices. This provides a challenge for developers creating FPGA applications and makes migration of applications from one board or FPGA to another difficult.

Many board manufacturers, including Xilinx and GiDEL, create boards with Peripheral Component Interconnect Express (PCIe) buses. PCIe provides a low-level standard for transferring data between a Central Processing Unit (CPU) and an FPGA, and these manufacturers have designed Direct Memory Access (DMA) engines to implement the standard. However, each manufacturer's DMA engines do not share a standard interface, leading to the same difficulties switching between boards or FPGAs.

The goal of this project is to reach a common interface for the diverse set of available DMA engines to improve application development productivity, particularly for the Autopipe environment, an environment for developing multi-device streaming applications.

Resources

The two primary boards for this project are GiDEL's ProcStar III AP-4 and Xilinx's ML605.

GiDEL board

The ProcStar III AP-4 houses four Altera (an FPGA manufacturer) Stratix-3 FPGAs and 12 external memories (3 memories per FPGA: 2 SODIMM slots and an on-board SDRAM), using an 8-lane PCIe bus as its primary method of communication with the host CPU. GiDEL also provided access to drivers, Intellectual Property (IP) cores, and licenses for their board.

Xilinx board

The ML605 houses a Xilinx Virtex-6 FPGA and a DDR3 memory interface. It has an 8-lane PCIe bus as well. Xilinx provided source code utilizing their DMA engine IP core and an example performance test application.

Novo-G forum

The Novo-G forum is a collaboration of 11 universities and several manufacturers to explore the power of FPGA-based applications and tool in high performance computing. The group centers around the Novo-G, a supercomputer composed of 24 of GiDEL's ProcStar III boards and 96 Stratix III FPGAs. Although it is significantly

smaller in size and number of devices than traditional CPU based supercomputers, its performance matches that of other supercomputers.

The Novo-G forum, particularly the University of Florida group, provided source code and an example DMA engine application for the ProcStar III board. They also provided debugging expertise.

Autopipe

Autopipe is an environment for developing streaming applications using multi-device systems, including systems contains CPUs and FPGAs connected by a PCIe bus. An Autopipe user specifies blocks of functionality that make up his program, as seen in Figure 1. The application has its function divided over the devices available by mapping these blocks to the different devices (including mapping multiple blocks to the same device). Autopipe provides native code to implement the functionality of the block on the given device, as well as First-In-First-Out queues (FIFOs) and interconnect resources (IRs) to connect the functional blocks. Interconnect resources include shared-memory buffers for CPU to CPU communication and PCIe DMA engines for CPU to FPGA communication.

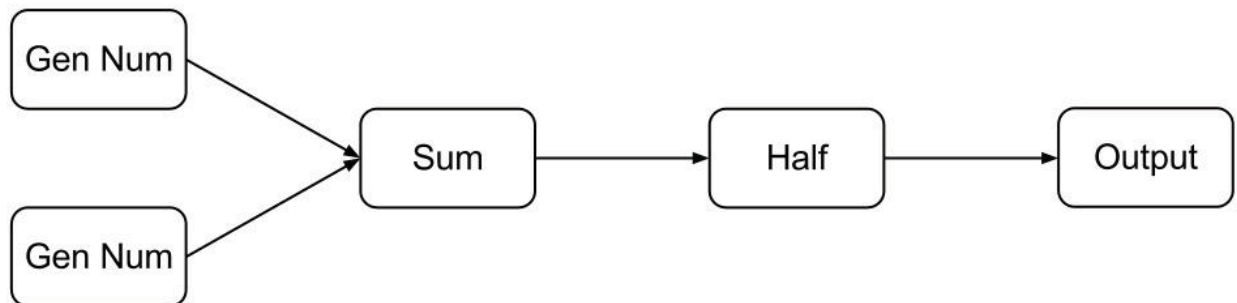


Figure 1: Toy Autopipe application for averaging two streams of numbers.

While an Autopipe user doesn't need to concern him with the underlying implementation of the system designed an IR for Autopipe requires a deeper understanding. If we mapped the toy Autopipe application of Figure 3 so that the Sum block were on an FPGA, and the others were on a CPU, the application would actually look like Figure 2.

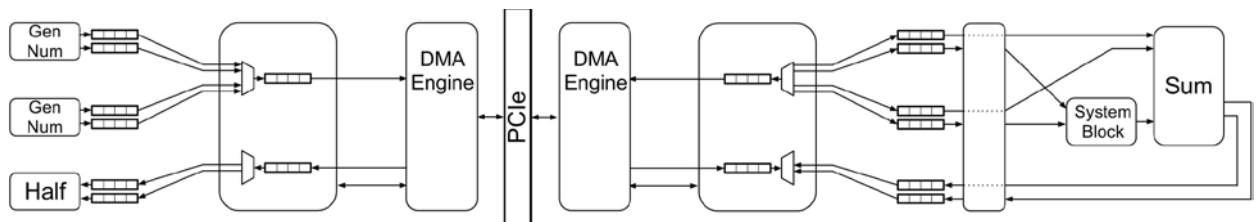


Figure 2: Autopipe infrastructure for toy application with sum block on FPGA.

Autopipe applications have two implemented channels for each logical path between blocks: a data channel and signal channel. The data channel's purpose is self-evident. The purpose of the signal channel is to pass user and system signals between blocks. Signals are used to indicate the end of new data, system restarts, and for buffer management and deadlock avoidance. Users can also create their own signals for information that is not general data. These two channels are represented by the pair of FIFOs leaving and entering the block.

To traverse the PCIe bus, elements from the GenNum blocks enter a FIFO (like they would if traveling to another block on the same CPU). The DMA engine (or application, depending on the specification of the DMA engine) physically multiplexes these FIFOs together, and the DMA engine transfer the data across the PCIe bus. There the DMA engine (or application) demultiplexes the data and stores it in FIFOs.

Autopipe processes certain system signals at a system level; hence, all signal FIFOs are filtered as they enter and leave the FPGA. Additionally, each Autopipe block on the FPGA is accompanied by a system block that processes system signals on behalf of that block.

The process whereby data travels from the FPGA to the CPU is symmetric to the process described above.

A common interface

In order to reduce the diversity of PCIe DMA engine interfaces, we created a common interface, and integrated existing engine interfaces into the unified design.

Objectives

An appropriate interface will meet three objectives: DMA engine interchangeability, application ignorance, and encapsulation of responsibilities. Seamlessly interchanging DMA engines allows applications using the interface to be portable; they can use any DMA engine that supports the common interface. Similarly, an application shouldn't be aware of the inner workings of the DMA engine it is relying on. Reliance on certain DMA engine properties also reduces portability. Finally encapsulating the details of the DMA engine removes that responsibility from the application developer, and inversely, the DMA engine developer can ignore details of applications that use it.

Interface

The common interface separates the DMA engine and the application at the FIFOs, as seen in Figure 3. By encapsulating the DMA engine at the FIFOs, from the perspective of the

entire DMA engine can be logically viewed as a series of FIFOs. The application never has to be aware of that fact that data is crossing a PCIe bus at all.

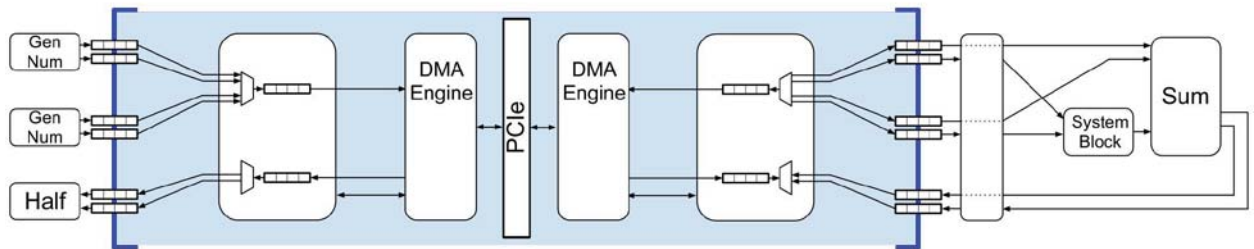


Figure 3: Common interface encapsulation.

This common interface provides DMA engine and application independence. The FIFOs remain in the design regardless of what application or DMA engine the developer uses. This interchangeability disappears at higher or lower levels of encapsulation. At lower levels, DMA specific actions must be taken into account, such as whether or not the data must be multiplexed before being passed to the DMA engine. At higher levels, unique parts of the application become incorporated into the DMA engine encapsulation. For example, the signal filtering done by Autopipe on the FPGA would be included in the DMA engine encapsulation if a higher level than the FIFOs were chosen as the interface.

Additionally, since all interconnect resources mimic the same structure (a FIFO), this interface could be applied to multiple IRs, further increasing modularity. For example, the shared-memory buffer IR for inter-CPU communication is already a FIFO; it could easily be incorporated into the same interface.

Autopipe and the interface

Because of the modularity of this common interface, Autopipe applications can treat the IR as an autopipe block. The software initializes the DMA engine by passing it the appropriate FIFOs, and call normal go and push functions on it. [INLINE CITATION TO AUTOPIPE FUNCTIONALITY] The application is never aware of the special functionality of the DMA engine, nor does it have to make special calls to it.

On the hardware side, a similar situation occurs. The Autopipe entity receives one set of FIFO ends, and the DMA engine receives the set of other ends. Autopipe never has to be aware of how data moves in and out of them; it can treat the FIFOs as if they were simply connected to other Autopipe blocks.

GiDEL DMA Engine

With the interface design complete, the next focus is integrating a current DMA engine into the interface as a proof of concept. GiDEL's ProcStar III board provides a working DMA engine with available support (from GiDEL and the Novo-G forum). In addition successfully integrating this DMA engine significantly expands Autopipe's user population by making Autopipe available to the Novo-G forum. Integrating the GiDEL DMA Engine required three major components: software design, hardware design, and testing.

Software Design

The Novo-G forum provides the C++ App class, an interface to GiDEL's driver software for the DMA Engine. The App class provides simple method calls for allocating PCIe safe buffers, reading and writing memory mapped registers on the FPGA, and performing large DMA transfers to and from the FPGA.

The App class does not require that data be multiplexed before DMA transfers. Therefore each input and output FIFO has its own dedicated transfer buffer, and each transfer call can be made independently. While the transfer buffer requires an extra copy, it also hides the specific workings of the DMA engine from the application; the application only interfaces with a normal FIFO. Because the only requirement to add a new FIFO to the interface is an additional transfer buffer, scaling the design increases memory requirements linearly but does not increase design complexity.

Hardware Design

The hardware design for the DMA engine differs from the generic Autopipe representation. Figure 4 illustrates the components of the hardware design. GiDEL and the Novo-G forum provided the components to the left of the blue divider. Components to the right of the divider were created to integrate the DMA engine with the FIFO interface.

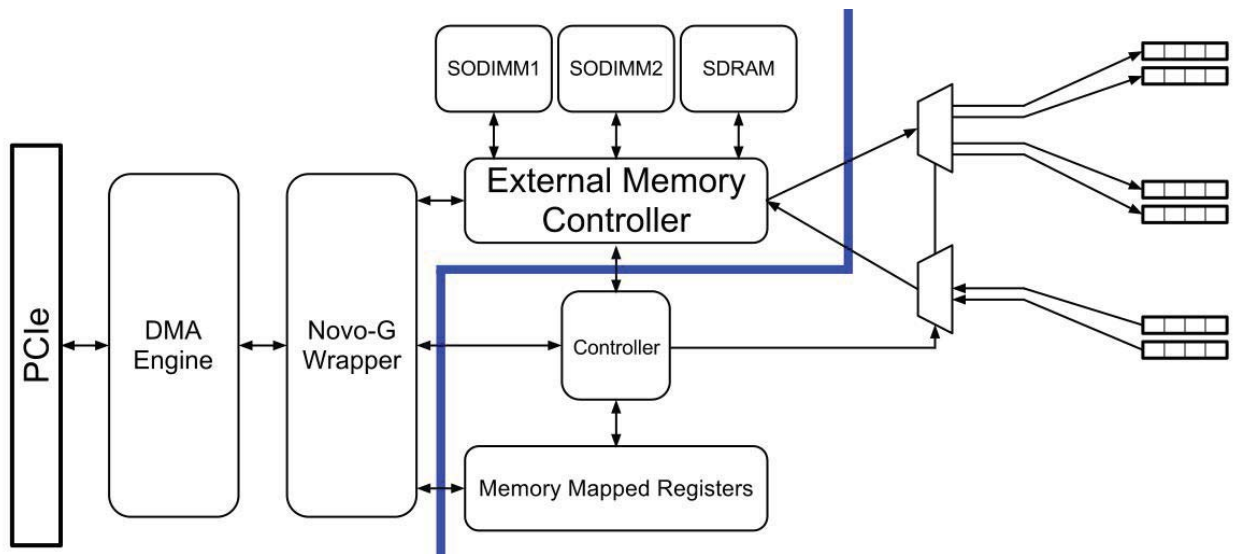


Figure 4: GiDEL hardware block diagram. Blocks to the left of the blue line were provided.

GiDEL and the Novo-G forum provide the DMA engine and a wrapper entity that control low level signaling. The DMA engine reads and writes data transfers to two locations: it writes to or reads from the memory mapped registers via programmed I/O, or it writes to or reads from the external memories associated with the FPGA.

The controller relies on the memory mapped registers to communicate with the software. Some of the registers act as handshaking signals, allowing the controller of software to confirm a transfer completion. The other signals provide information on the volume of transfers. Because the buffers are a set size and a DMA transfer may not fill a buffer, it is important to know how much data is valid in the buffer. This prevents garbage data from entering the data stream.

Application data rests in external memory. The controller partitions the memory into one transfer buffer per FIFO. Because memory is large (approximately 2GB) and individual transfer buffers are small (approximately 4KB), scalability is not an issue. The controller can access these partitions with burst reads or writes. While these bursts mimic FIFO actions, there are several reasons why the controller transfers the data to FIFOs instead of directly to the application. First, using the external memories as the interface with the application breaks the encapsulation of the DMA engine. Not all DMA engines deposit data in external memory banks, so this approach would force applications to have a unique interface for this DMA engine. Secondly, access to external memory is limited. The external memory controller does not allow multiple reads or writes from the same external memory to occur at the same time. Multiple data reads or writes would cause bottlenecks in the application's performance.

The controller interfaces with an external memory controller to transfer data to the FIFOs. Both input data to the FPGA and output data from the FPGA are handled in similar ways but by different processes.

The input state machine handles all data written to the FPGA. Its state diagram is in Figure 5. When the state is steady (a.k.a not transitioning) the state machine performs a steady operation. At the 'done' state, the state machine performs a round-robin with passover selection process to find the next input buffer with available data. At any 'read' state, the controller reads the contents of the buffer into the FIFO. Transitions manipulate the external memory controller to prepare for the next steady state. When transitioning to a 'read', for example, the external memory controller receives the starting address, size of the transfer and the go command. When transitioning back to 'done', the controller saves information about the buffer, such as whether or not the buffer is empty, how much data is left in the buffer if the FIFO filled, and what the beginning address of that data is. It also tells the external memory control to finish the last read prematurely so it can initiate a new one later. The controller confirms transitions with software via the memory mapped registers.

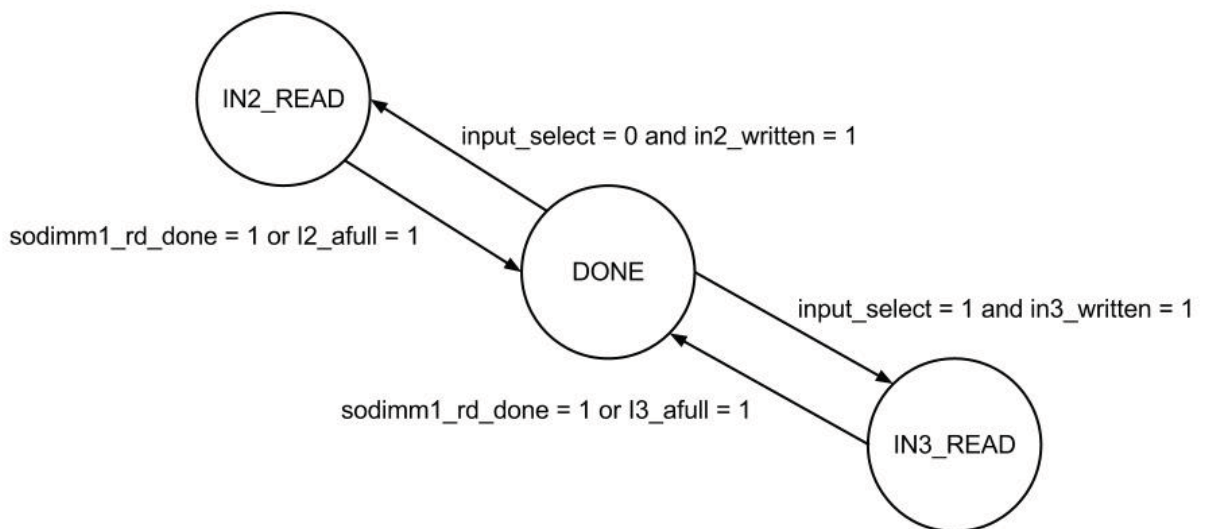


Figure 5: Input state diagram

The input state machine scales well with additional inputs. Although Figure 5 only shows two inputs, each input one requires one unique state. The only additional complexity is the selection method, which is a simple counter. This makes it easy to scale to a large number of input FIFOs. In the case where the number of FIFOs is exceedingly large, the other external memories can be used by cloning the state machine for that memory. The FIFOs can be spread across two or three round-robin cycles instead of one.

The output state machine, whose state diagram is in Figure 6, behaves similarly to the input state diagram in most ways. It uses a round-robin passover selection scheme to select the next FIFO to read data from into the external memory transfer buffer. Transitions trigger control signals that interact with external memory controller to initiate transfers, and the controller confirms transitions with the software via the memory mapped register. Because the process of writing data to the transfer buffer and then initiating the DMA transfer requires two sequential steps, an additional state exists, 'dma_write.'

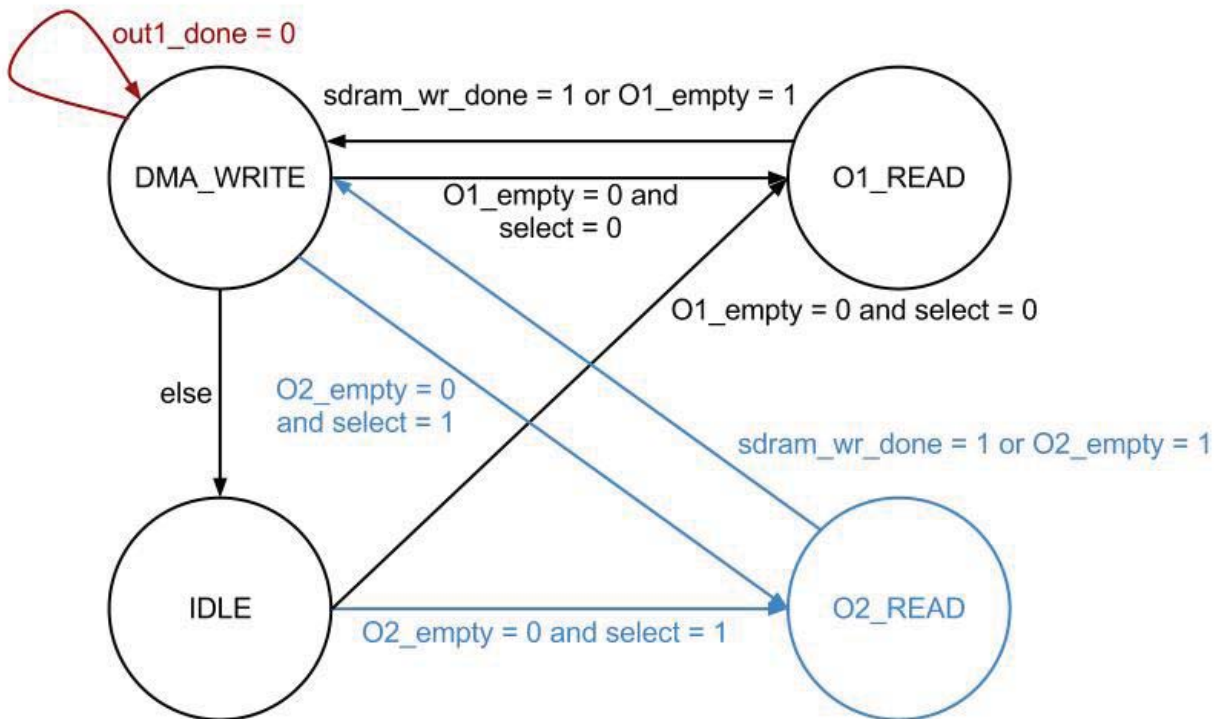


Figure 6: Output state diagram. Blue illustrates increasing scale of state diagram. Red represents the first path chosen if possible from that state.

Although the output state machine requires an extra state to initiate the DMA transfer, all outputs can share the same 'dma_write' state, ensuring that only one additional state is required per output FIFO. Therefore, the output state machine scales at the same rate as the input state machine.

Testing

The GiDEL DMA engine was tested in components and as a whole to ensure functional correctness.

Software Testing

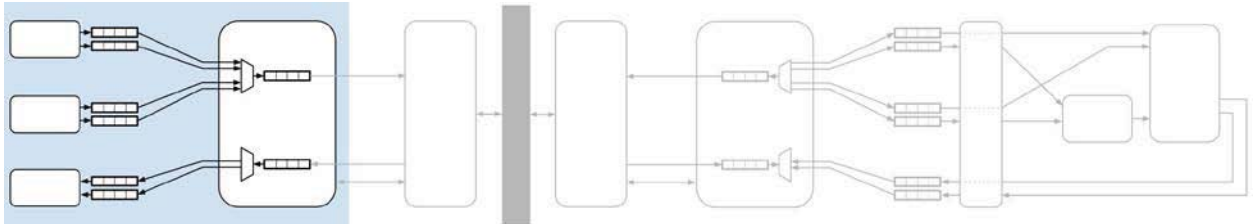


Figure 7: Software test scope

The purpose of the test bench was to ensure that if the IR could be treated as a FIFO, the Autopipe application would still function correctly. To test the software components of the application, the test treated the IR as the Autopipe application does: as an Autopipe block. The software test bench modified the IR so that it performed the duties of the Sum Autopipe block. The modified IR added data from the GenNum blocks and then pushed the data onto the FIFO to the Half block. The Autopipe application acted as expected; the application correctly averaged the stream of data.

DMA engine testing

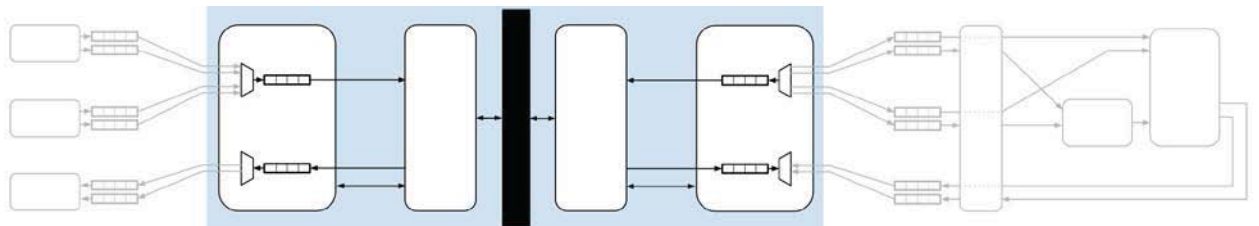


Figure 8: DMA engine testing scope

The DMA engine test ensured that the core DMA engine was functionally correct and worked as expected. The test used Novo-G example DMA engine application to test this functionality. The example application uses memory mapped registers to communicate with the FPGA and performed a read and write DMA transfer of a large volume of data to and from the FPGA. This tests all of the core functionality of the DMA engine, although it doesn't necessarily expose potential issues with a streaming application such as an Autopipe application.

The test exposed many unexpected problems. Although it was an example application designed to work immediately, it relied on several third party libraries. The debugging involved in the test revolved around building the infrastructure to support these libraries. This included installing the PROCWizard libraries (a portion of the supplied

GiDEL drivers), accessing the other GiDEL Hardware Description Language (HDL) library, acquiring the GiDEL licenses to use their IP, and uses the correct Altera compilers to create the requisite bitfile to load onto the FPGA. Although the debugging was significant, it enabled a straightforward build plan to generate the actual Autopipe application.

Hardware Testing

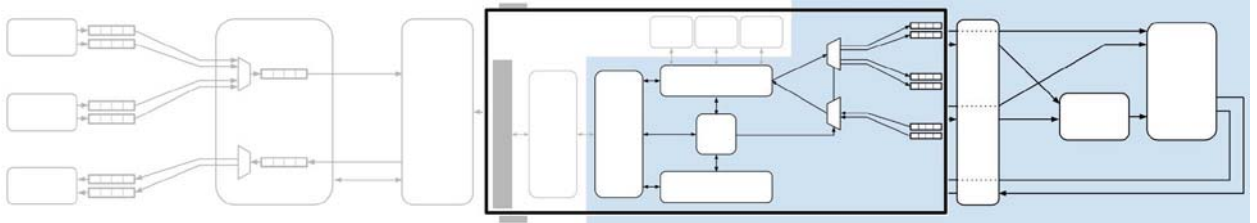


Figure 9: Hardware Testing

The hardware test used Modelsim, a hardware simulation engine, to verify the correctness of the hardware. The Novo-G forum provided a test bench that roughly imitated the DMA engine and external memory, with some limitations. The size of the artificial memory was small relative to the actual memories, reducing the area for testing separate buffers. Memory could not be rewritten after initialization, preventing the testing of a true streaming application, and once memory was written, it could not be read. On the other hand, the test bench allowed memory mapped register driving of the hardware component of the application, which provided significant testing.

By updating registers at intervals via the test bench, the hardware runs through every state in both input and output state machines. The test bench starts by asserting that data has been written to buffers (which were initialized with sequential data values), which starts the input state machine. Because the buffer is larger than the FIFO, it reaches an almost full state and tests how the controller handles a partially read buffer. It also tests the input state machines selection process as it cycles between the two inputs as it repeatedly fills buffers. When both buffers empty, the input state machine successfully returns to its resting polling state.

Once data proceeded through the Autopipe Sum block, the output FIFO started to fill. The output state machine tested all of its transitions. When it received data it started to read the data in, and it transitioned when the FIFO emptied. It initiated DMA transfers before starting to read more data in and could transition back to idle or directly to reading more data. When all of the data had passed through and the last DMA transfer was performed, it successfully returned to the resting state.

The test bench confirmed that all of the output data successfully reached the output buffer, and that all of the data was correct.

Interaction Testing

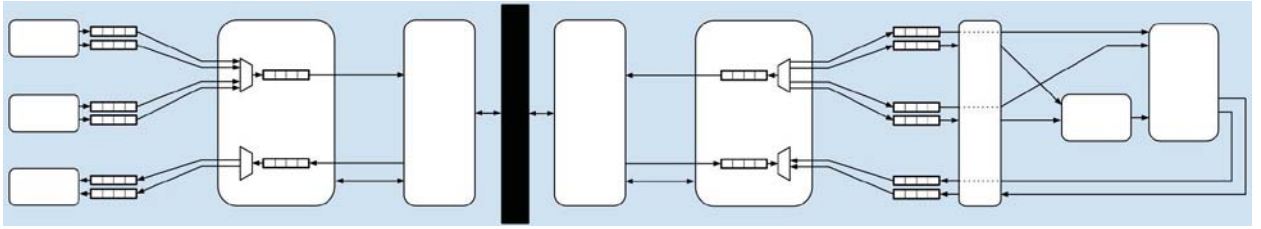


Figure 10: Interaction Testing

The purpose of interaction testing is to ensure that all of the components integrate together, demonstrating that the DMA engine and interface are functionally correct. The Autopipe example application is run its entirety, relying on the DMA engine to communicate successfully between the CPU and the FPGA. Due to the lack of availability, testing in this regard has been minimal. Tests indicated that the DMA engine transferred data to the FPGA, and the FPGA returned data. The data was not the expected volume, and the data values were not examined. Subsequent runs failed to return data and eventually led to a kernel panic. To be successful more testing will be needed to ensure full functionality.

Xilinx DMA engine

The example DMA engine Xilinx provides can also be integrated into the common interface, providing increased modularity and portability to the Autopipe system. Although the Xilinx DMA engine shares many common traits with the GiDEL DMA engine, it has several unique features that require a different design to conform to the common interface.

Software Design

The software interface Xilinx provides is very similar to the GiDEL model, as can be seen in Figure 11. Both perform DMA transfers to user-space buffers. Therefore, like the GiDEL model, we can easily scale to support many FIFOs, with one transfer buffer per FIFO. Additionally, the software design uses Xilinx's status and control registers in the same way that it uses GiDEL's memory mapped registers: as handshaking and volume data signals.

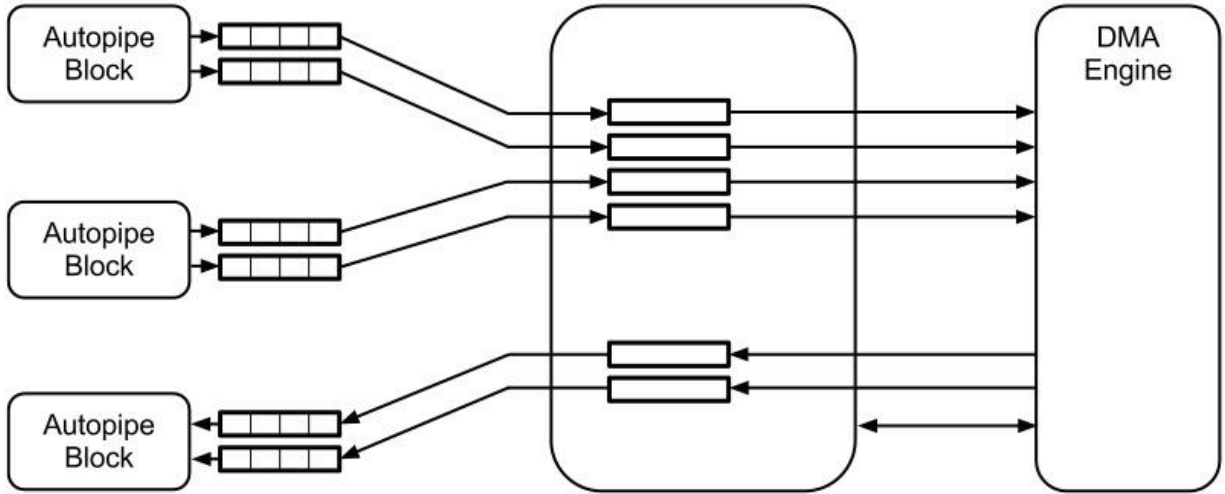


Figure 11: Xilinx Software Model

In addition to the GiDEL software design approach of polling the FIFOs and status registers to check for input, the Xilinx software can send interrupts when DMA transfers complete. While the initial software design should imitate the working GiDEL design, future design iterations could explore the benefits of interrupts instead of continuous polling for triggering data management actions.

Hardware Design

The hardware design illustrated in Figure 12 for the Xilinx DMA engine differs greatly from the GiDEL design, primarily because the Xilinx base code takes responsibility for controlling the handshaking and volume control register signals but requires the design to handle multiplexing the channels.

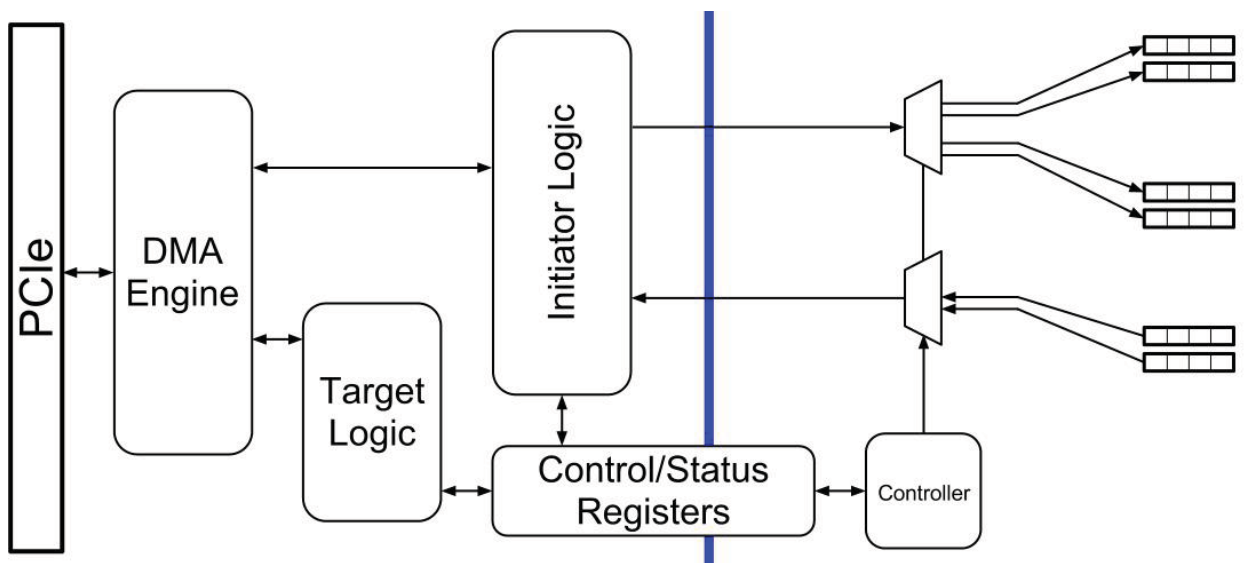


Figure 12: Xilinx Hardware Model

The design is unique in that it has two controllers: the initiator logic provided by Xilinx and a controller created by this design. The Xilinx initiator handles the sequence of packets that must be sent for a given transfer based on the data stored in the control registers. The controller has several responsibilities. For all transfers, it must ensure that the correct FIFO input is selected for. In addition, for transferring data from the FPGA to the CPU, the controller must update the control registers with the size (of a single element) and count of the data to be sent.

While the design complexity does not increase as the number of FIFOs increases (due to the fact the multiplexer simply has more options to select from), there is potential for a performance bottleneck if the transfers can't be handled quickly enough and many FIFOs fill with data.

Scope

In addition to creating a standard for PCIe DMA engine interfaces, the common interface improves the Autopipe environment by defining clear boundaries for interconnect resources and improving the modularity of the infrastructure. The DMA engines implementing the common interface will greatly expand Autopipe's potential user base.

GiDEL

The GiDEL DMA engine is fully integrated with the common interface. The DMA engine requires a GiDEL board and Altera's Stratix-3 or Stratix-4 FPGA, but GiDEL has committed itself to maintaining compatibility and providing a stable code platform to work from. Because of this, the GiDEL DMA engine should remain compatible and useful for many generations of GiDEL products.

The major benefit of Autopipe having access to ProcStar-III boards is that grants Autopipe access to Novo-G the FPGA centered supercomputer. Novo-G is an ideal platform for Autopipe applications due to its mix of FPGAs and CPUs and the potential to deploy applications across a large number of devices. Members of the Novo-G forum have also expressed desire in using the Autopipe environment on their own boards.

Xilinx

Xilinx supports its PCIe DMA engine on Virtex-6, Virtex-5, Spartan-6 and Spartan-3 FPGAs. Because this DMA engine is not tied to a specific board, it has potential to provide Autopipe on a wide variety of boards from a variety of manufacturers.

In the short term, the physics department at Washington University in St. Louis is seeking funding from the United States Department of Energy to use Xilinx FPGAs and the

Autopipe environment to construct a gamma ray telescope. The Xilinx DMA engine will be critical to constructing a successful application.

Conclusion

The common interface provides a common standard across multiple brands of FPGAs and boards. The encapsulation of the DMA engine increases modularity and portability, two aspects the Autopipe system will benefit greatly from. The fully integrated GiDEL DMA engine provides many opportunities for Autopipe to increase its user base, and the design for the Xilinx DMA engine will further increase Autopipe's versatility. Creating the interface and integrating a DMA engine into it was not without obstacles, and it taught many valuable lessons.

Obstacles

The largest obstacle I encountered was getting GiDEL's DMA engine example application working. The application relied on a framework of GiDEL libraries, and they had to be installed and configured before progress could be made. This involved working through a diverse set of errors. Even more diverse than the errors were the solutions to the errors: general HDL debugging, modifying makefiles to access libraries, adding HDL libraries to a Quartus project, installing device drivers, acquiring licenses to use GiDEL IP, and seeking expertise on the correct Quartus installation. On the plus side, working through these obstacles with the example application provided a framework for building working projects that could use the DMA engine, simplifying the process of successfully creating the Autopipe project executable and bitfile.

Setting up Modelsim to test hardware portions of the design also proved to be challenging. Modelsim proved challenging because of the same third party library issue. Although the problem was the same, Modelsim and Quartus have different methods of including necessary files, so a new solution was needed. Once all the required files were available, the test benches had to be redone to send the proper signals to the controller, and the memory controller had to be modified to hold the appropriate data. In addition the memory controller did not accurately represent the memory controller of the actual application, so it had to be modified to provide more accurate testing data.

The final major obstacle I encountered revolved around coordinating the programmed I/O between the software and hardware. Since software was reading and writing over a bus, timing assumptions couldn't be made about when it would read or write a memory mapped register. This put very strict requirements on the values the memory mapped registers could hold at any given time. Additionally, to test that the memory mapped registers were

working correctly, the Modelsim test bench had to be carefully written to simulate the different possible timing scenarios.

Learning

One of the areas I lack expertise in when I began the project was working with third party tools and software. While the issues I experienced when working with these resources quickly became major sticking points in the project, I gained valuable experience about how to understand third party requirements, how to install third party resources, and most importantly where to go for help when I didn't know the answer. Each of the problems I encountered required a very different solution, which reinforced exploring several different approaches to solve a problem.

While I did not do low level DMA transfers and PCIe signal processing, the project has taught me a lot about how DMA designs work and what the PCIe protocol is. This gave me a much deeper understanding of how devices communicate with each other. This knowledge helps round out my understanding of computer systems as a whole, giving me a more complete view of what a computer is.

I also received a new perspective on the Autopipe system. Although I have frequently been an Autopipe user and even block designer, I had never done infrastructure development with Autopipe. It revealed many of the complexities of the system that I had previously overlooked, such as how the push call actually works, and it helped me understand some of the design decisions that have been made in Autopipe.

The experience of working on a long term project will be very useful in the future. Practicing organization, self-motivation, and problem-solving skills with the goal of creating a fully functioning component of a large project will help me when as I begin my career.

Project Files

All project materials, including reference source will be made available on the Autopipe wiki.

Acknowledgements

Thank you to my many collaborators who helped make this project possible: Jonathan Beard, Shyam Chander, Robert Kirchgessner, Joe Lancaster, Ed Richter, Berkley Shands, and Joe Wingbermuehle.

Thank you also to my advisor and mentor Roger Chamberlain. Without your guidance, patience, and understanding, I would not be where I am today.

This project was funded by two NSF grants.