X3-SD16 FrameWork Logic User Guide

X3-SD16 FrameWork Logic User Guide

The X3-SD16 FrameWork Logic User Guide was prepared by the technical staff of Innovative Integration on July 27, 2011.

For further assistance contact:

Innovative Integration
2390-A Ward Ave
Simi Valley, California 93065

PH: (805) 578-4260
FAX: (805) 578-4225

email: techsprt@innovative-dsp.com
Website: www.innovative-dsp.com

VSS \ Distributions \ X3-SD16 \ Documentation \ Manual \ X3-SD16Master.odm

Revision	*Changes*	*Author*	*Date*
1.0	Initial release for X3 family.	DPM	06/25/07
1.1	Revised into new manual format. Added X3-SD16 chapter.	DPM	07/24/11

Table of Contents

List of Tables

List of Figures

Introduction

This manual is written to assist in the creation, implementation and testing of custom logic for Innovative Integration products. The scope of this manual is limited to discussion of the logic development tools, example logic designs and logic libraries provided in the FrameWork Logic toolset.

Additional documentation on each product is provided for hardware features and software in other manuals. These are used in conjunction with this manual for product development and use.

Thank you for using our products. Your comments and input are appreciated so that we can improve our support and help you to be successful on your project. Email us at techsprt@innovative-dsp.com with your input or give us a call.

Prerequisite Experience and Required Tools

The designer is expected to have experience in VHDL and FPGA design to use the FrameWork Logic tools and code. All components in the FrameWork Logic are VHDL source code whenever provided and supported by VHDL models and test code. As a standard, the code is written in VHDL 1993 version which is widely used and supported.

The design tools used are listed here. We make an effort to keep the logic supported under the newest versions, but in many cases the logic must be reworked and retested to support the newest tool version. For each product, we have listed the required tool set that was used to create the logic.

Here is the tools set list we use for supporting the FrameWork Logic use and development.

Function	***Tool Vendor***	***Tool Name***
Synthesis, Place and Route	Xilinx	ISE 12.1 or above WebPack ISE 12.1
Simulation	Mentor Graphics	ModelSim 6.2c
Bit and PROM Image Creation	Xilinx	Impact 12.1
Logic Debug and Testing	Xilinx	ChipScope 12.1
Logic JTAG Cable	Xilinx	USB, Parallel Cable IV or others

Table 1. Supported Logic Development Tools

See MATLAB BSP manual for current MATLAB toolset.

The documentation for the development tools is provided by the tool vendor. All of them have on- line documentation and help that can acquaint you with their use. This manual makes no attempt to replace them, but rather supplement them with specifics on using them with FrameWork Logic application development.

While it is not expected that you are expert in these tools, these tools are used for FrameWork Logic development and are discussed in this manual. If you are using other tools, they should have similar capabilities.

Organization of this Manual

This manual covers the main topics in using the FrameWork Logic for Innovative Integration products for HDL development methods. The first few sections describe the HDL tools and development methods including synthesis, placement and routing, and simulation. Finally, the generating the logic image and debugging are discussed.

Each product supported by the logic is discussed, showing the details of the example logic are shown. The hardware interface components and application logic internals are shown.

Finally, the FrameWork Logic library components are shown.

The MATLAB tools and Board Support Packages (BSP) are covered in the *X3 MATLAB Board Support Package Guide.* There is one manual for each product that discusses the specifics of that BSP and development process. This manual is describes some of the underlying infrastructure logic that supports the MATLAB BSP.

Logic Directories and Files Organization

The logic files for all X3 products are organized by product with a library of common components that is used by each logic design. Each product design has both RTL and MATLAB support under its product directory.

The RTL for each design provides support for simulating, creating and debugging. Design-specific source files for in the source directory include the top-level and unique hardware interface items, while the library files contain components that may be used in all X3 products. Constraints for the design is provided in the source directory and includes all the physical and timing constraints required. Results of the compilation are included in the ISE and Linux directories showing the compilation and fitting results.

The MATLAB Board Support Package (BSP) is included in the MATLAB directory. These directories are the MATLAB BSP files and example projects that use Simulink with Xilinx System Generator. Refer the to X3 MATLAB BSP Manual for details on using MATLAB. Do not use these files for RTL work since they have MATLAB-specific features.

The Library directory has logic components supporting the X3 family. These components are used in many of the designs and are common to the family of products.

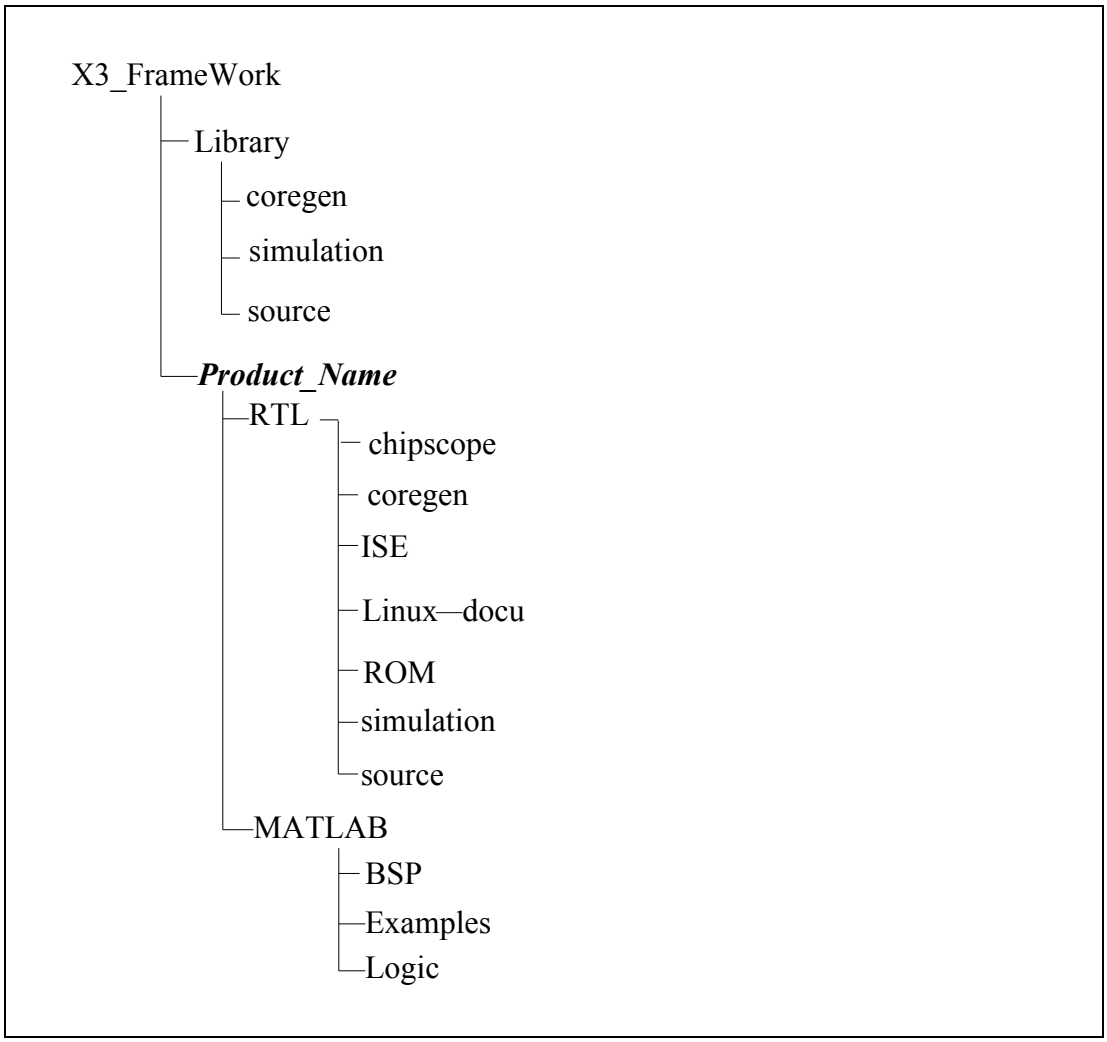


Figure 1. X3 FrameWork Logic Directory Structure

Directory	Files
./Product_Name/RTL/coregen	Logic cores specific to a design.
./Product_Name/RTL/chipscope	ChipScope cores and debug session files
./Product_Name/RTL/ISE	Xilinx ISE directory for a specific design
./Product_Name/RTL/Linux	Files for compiling in Linux environment and compile results (if available)
./Product_Name/RTL/ROM	The released logic image in EXO or BIT format
./Product_Name/RTL/simulation	Logic simulation files
./Product_Name/RTL/source	Logic source files for a specific design
./Product_Name/MATLAB/BSP	MATLAB Board Support Package
./Product_Name/MATLAB/Example	MATLAB Examples
./Product_Name/MATLAB/Logic	Logic source and compilation files for MATLAB BSP
./Library/RTL/coregen	Logic cores for library components
./Library/RTL/simulation	Simulation files common to all X3 designs
./Library/RTL/coregen	Logic source files for library components

Table 2. FrameWork Logic Directories and Organization

Logic Component Naming Conventions

For all hardware interface components, the standard naming convention is

ii_<product_name>_<function_name>

where <product_name> is the Innovative product this component is used on, and <function_name> is a descriptor of the component.

For example,

ii_sdf_adc

is the X3-SDF A/D hardware interface component.

Where to Get Help

In addition to this manual, the example design for each product is provided with an HTML document that allows designers to quickly navigate the design to understand the hierarchy, entities used, ports and source code.

For help on Innovative Integration hardware or software, there are separate help manuals and an on-line help system for the software tools. These manuals are provided on the CDs delivered with the product or on the web at http://www.innovative-dsp.com/support/productdocs.htm . At this site, you can download the product information, software and logic updates.

Help for other tools such as Xilinx or ModelSim is provided on-line with the tool. Xilinx also has an excellent Answers Database on the web (http://www.xilinx.com/support/mysupport.htm) and many examples of techniques used in FPGA design. This is the primary site for support on Xilinx- specific problems that can include tools problems and workarounds.

Technical support from Innovative Integration is available at

Web Site www.innovative-dsp.com (product manuals, software updates, firmware and discussion forums)

Email us at techsprt@innovative-dsp.com

Phone : ++1 805-578-4260 Monday through Friday, 8 AM to 5 PM Pacific Standard Time

Logic Development Process

The FrameWork Logic system supports two logic development methods: VHDL, MATLAB Simulink, or a combination. Each system offers benefits and have strengths that in some cases complement each other.

VHDL development is very flexible, allowing the developer the full freedom of a high level language that is expressive and extensible. The FrameWork Logic system provides VHDL components for hardware interfaces that allow the designer to quickly integrate custom VHDL code into the application logic. Other library components are offered that provide some common functions used in signal processing and control. Libraries from Xilinx and third parties are also used to provide broad support for signal processing, analytical and communications applications.

Development Tool	***Pro***	***Con***
VHDL	Expressive, extensible language. Gives complete flexibility to the designer.	Design and debug of DSP algorithms is more difficult and time consuming.
MATLAB Simulink	Allows design of complex DSP algorithms at a high level. Great visualization and analysis tools for design and debug.	Less capable of handling low-level details. Less visibility and control of logic design process.
VHDL + MATLAB	Best of both tools gives optimum flexibility where needed and high level design for complex DSP algorithms.	Multiple tools must be used resulting in a more complex development process.

Table 3. Logic Environment Pros and Cons

MATLAB Simulink offers a high-level block diagram approach to logic design that allows the designer to work at a higher, more abstract level. Signal processing algorithms can be quickly developed and simulated in MATLAB then directly ported to the logic hardware. Inside of the FrameWork Logic tools, the designer can concentrate on the algorithms because the system has a hardware interface layer that integrates the hardware with MATLAB cleanly and efficiently. Application development is dramatically sped up for complex signal processing algorithms because of the powerful capabilities within MATLAB for algorithm design, visualization and analysis.

Many applications find that a mix of VHDL and MATLAB offer the best of both worlds: high level signal processing development and the full flexibility of a high level language. It is common that unique data handling, triggering and interface functions may be better expressed in VHDL, but nothing beats the power of MATLAB for things like filter design, down-conversion and mathematical analysis of data. The designer can mix VHDL components, or MATLAB-generated components with one another in either environment and reap the benefits of each system.

Developing Using VHDL

Application logic development with the FrameWork Logic in VHDL follows the typical development cycle: code creation, simulation, physical implementation and test. This flow is summarized in the following diagram showing the Xilinx ISE tools and ModelSim as the primary development tools.

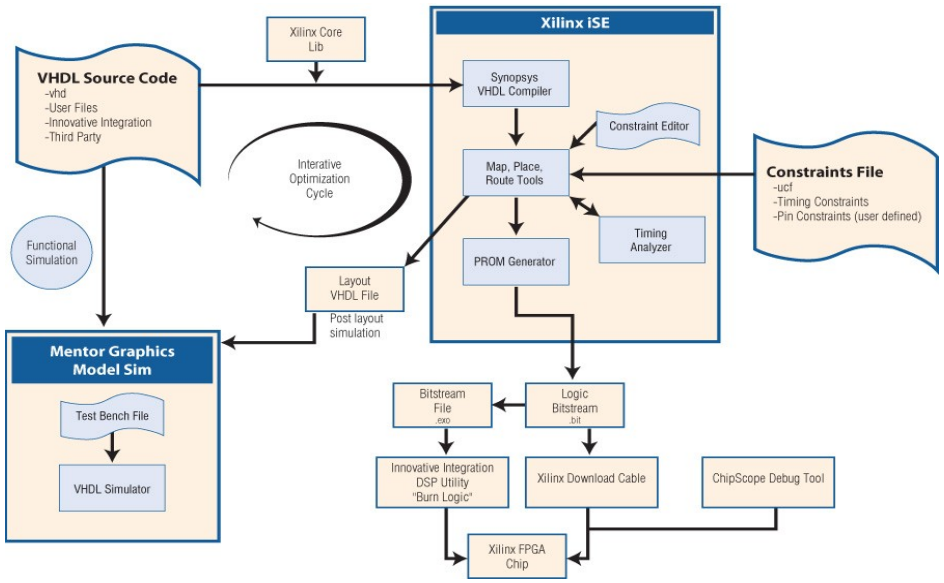


Figure 2. VHDL Development Process

The application development begins with the FrameWork Logic code for the product you are using. In many cases, the example application code provides a good starting point for your application logic. In most cases the application logic shows a basic data flow between the IO devices, such as A/D and D/A converters, to the logic and system. You can then build on top of the example logic by inserting your algorithms into the data flow along with unique triggering and other application-specific logic.

The FrameWork Logic provides a library of components for the hardware interfaces as well as others functions, an example application showing IO interfacing and data flow, design constraints, a simulation testbench, and a Xilinx ISE project or each example. This gives you the basic foundation to begin work. After you install the FrameWork Logic on your system, you should be able to recreate the logic and verify its operation. Once that is complete, you are ready to begin development.

At this point, you should have a look at the example logic and determine the best place to insert your logic and how you can best use the example in your development. If you can preserve many of the basic memory mappings, controls and system interfaces, you will then be able to use the example application software delivered with the product. That saves time for both you and the software developers.

In most cases, you will see that the logic is organized as a hardware interface layer composed of components that directly interface to the hardware and an application layer that is composed of the analysis, data handling and triggering functions.

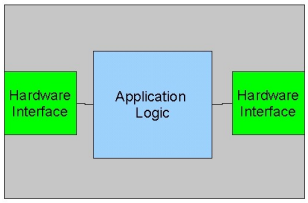


Figure 3. Logic Architecture Showing Hardware and Application Layers

The application layer is on a single clock domain so that it is easy to integrate functions into the design.

Code for your application layer design can be created in a number of ways: written in VHDL or Verilog, created in MATLAB, or included as a black box netlist from a third party such as Xilinx or others. If you design you logic to run on a single clock it is then easier to integrate into the application layer of the FrameWork Logic. This is usually possible because the other clocks in the design, such as the A/D sample clocks, or hardware-specific clocks are handled in the hardware interface layer. The use of a single clock in the application layer allows the designer to use the timing and physical constraints associated with the hardware interface components.

Using Xilinx ISE

The Xilinx ISE toolset is recommended for most logic developers. ISE provides code editing, core generation, synthesizing and fitting tools for the chips that is integrates all of the Xilinx tools for the project. An example project is shown here.

The existing project should be used as a starting point. This project has all the options set and file structure required to recreate the design. The Xilinx ISE project is in the ./RTL/ISE/product_name directory for each design. When you open the project, the summary screen is displayed showing the design statistics and current status.

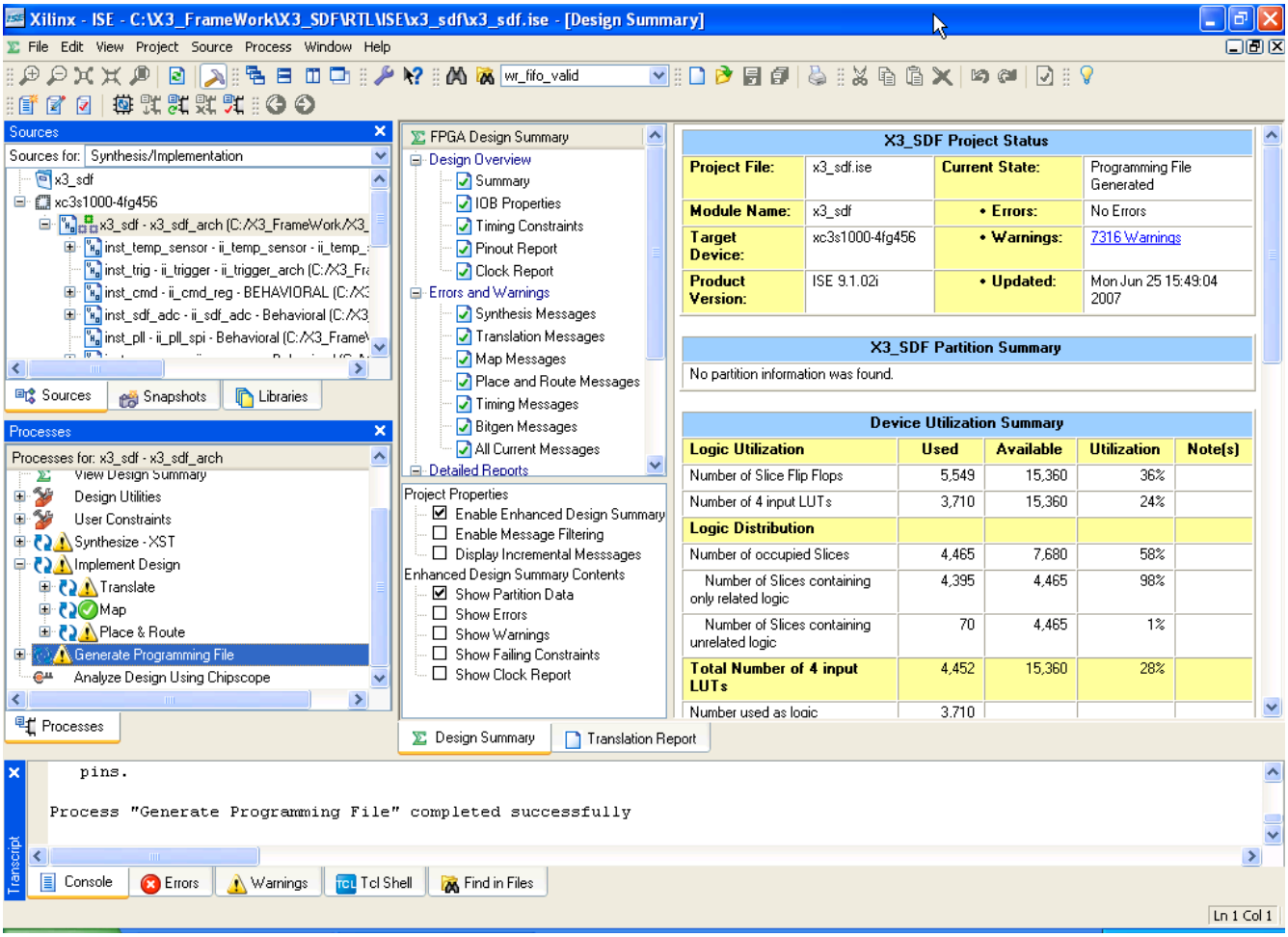


Figure 4. Example Xilinx ISE project

Note: The project options have been set to use the directory structure for the FrameWork logic design. It is important to use these options when compiling the project so that the cores and source code can be found. It is also required to preserve hierarchy on the design to use the constraints provided. In the NGDBuild options, the -sd directive is used to point at the core locations for the design. If you make a new project, be sure to point at the location of the coregen directories for the project and X3 library.

Using the FrameWork Library

The components in the FrameWork Logic library are divided into generic components that may be used in any design, and hardware-specific components.

The hardware-specific components are used in the designs for A/Ds, DACs, memories and the like that have unique interface protocols and timings. Constraints in the specific design for IO standards and specific timing requirements are usually required for use. The constraints for the hardware-specific components are found in the application example that includes that component.

All hardware-specific components have unique names such as *ii_sdf_adc*. The naming convention prevents inadvertent naming collisions with your design if you do not use a *ii_* prefix on your components. The hardware name is included in the name showing which design uses this component.

In the installation, you will find that hardware-specific components in the directory for that specific design. The generic components are in the ii_library directory. To use the components, you can copy them into the design you are creating, or reference the library directory.

Also, you may need to include packages supporting the components in your design. For example, *ii_cmd_reg* component requires *ii_x3_pkg* to be included. This is done by including these statements in the component and by compiling the package in your design.

```
library  work;
use  work.ii_x3_pkg.ALL;
```

One problem that frequently occurs is that the simulation requires the package be compiled for use. The script that Xilinx ISE produces seems to exclude these packages for compilation, so be sure to compile the required packages separately.

VHDL Simulation

Simulation is an important part of the logic development process. All designs that are targeted at using the large logic devices supported by the FrameWork Logic require simulation for successful implementation. Unless the design change is very simple, you should simulate design. If you don't, it is unlikely that you will successfully complete design in your lifetime.

For simulation, we are currently supporting ModelSim 6.2 PE. Other versions of ModelSim can be used by creating projects for that version.

The FrameWork Logic includes a test bench and models required for most simulations. In many cases the models are simple representations of the device that give a data pattern that is easy to follow through the simulations. More complex waveforms can always be substituted later for proving out the signal processing or data analysis portions of the design. In each design, the list of files shows the applicable test bench name and available models.

The testbench contains a set of simulation steps that exercise various functions on the FrameWork logic for basic interface testing. Behavioral procedures have been written to simulate the host timing for command channel and data link transfers that are useful in simulating data movement. Other features such as SRAM interface, alert log and triggering are demonstrated in the testbench.

As delivered, the FrameWork Logic example provides a basic example in the use of the hardware interface components, data flow through the design, and some simple triggering control. It is anticipated that you can use this example test bench as a starting point for your application logic simulation. Your logic can be added to the simulation in many cases without modifying the test bench since the application logic does not change the external pins on the design.

Before simulation can begin, the simulation libraries must be compiled for the chip you are targeting. If you are starting from Xilinx ISE, this is done by selecting the device in the Sources window, then double-clicking the "Compile HDL Simulation Libraries" process in the Processes window. This will compile the unisim, simprim and xilixcorelib files necessary for simulation. You may have to configure this process so that it points at your current ModelSim installation; this is done by right-clicking and setting the parameters. If you are working with ModelSim standalone, be sure to compile the libraries unisim, simprim and xilinxcorelib and add them to the libraries in ModelSim (vlib) before attempting to simulate.

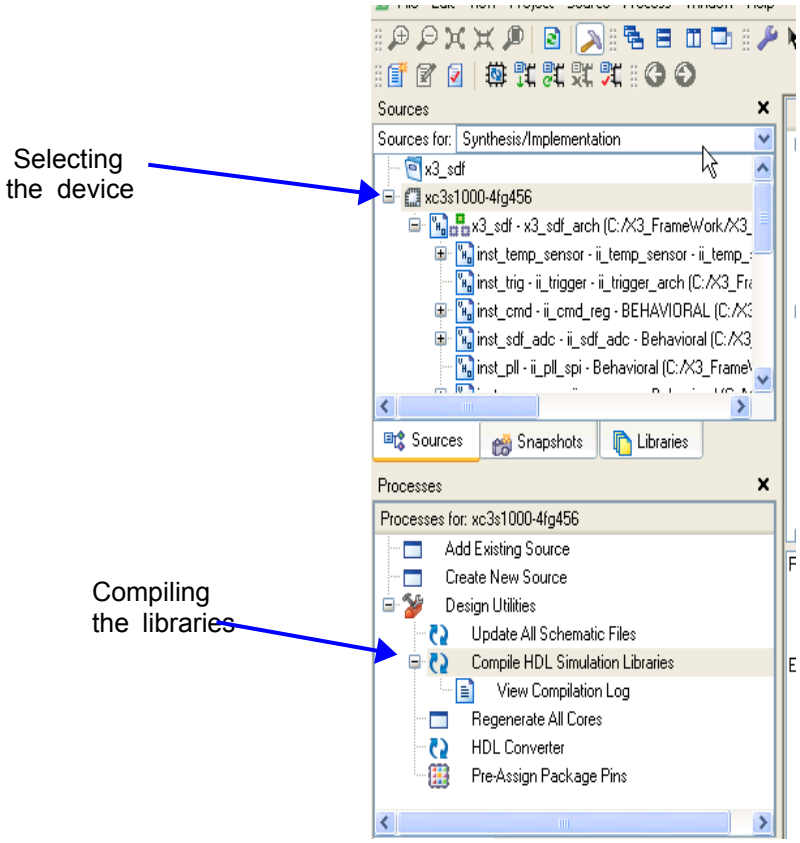


Figure 5. Compiling the Simulation Libraries

The simulation properties must be set to for ModelSim as shown here. The default timing should be 1 ps.

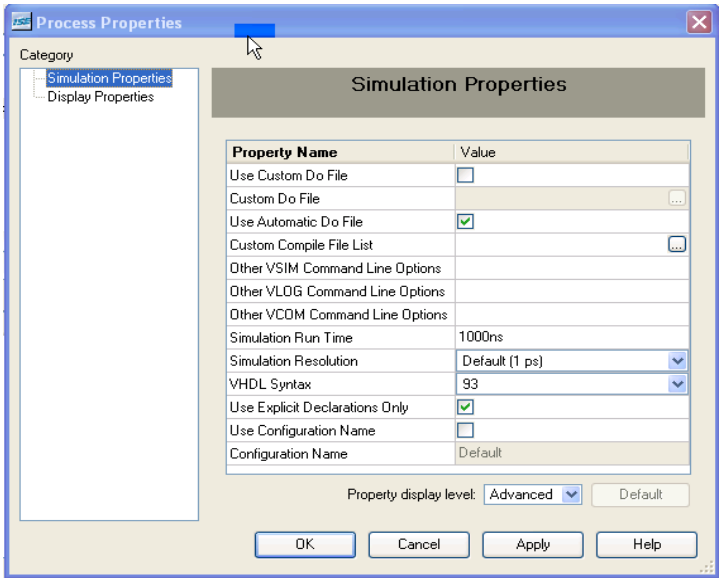


Figure 6. Setting Simulation Library Compilation Properties

Simulation can be started in either the Xilinx ISE environment, or by using the ModelSim tool in a standalone mode. In Xilinx ISE, select the Behavioral Simulation Mode in the Project window. This then will show the simulations that can be run in the Processes window using ModelSim. Usually, the functional simulation is best to use when you are creating code because the simulation will execute quickly. Once you have the right functionality, you can then use the timing simulation to verify performance at speed.

When you enter the ModelSim tool from Xilinx ISE, it will execute a default macro that compiles the files and begins the simulation. If you enter in standalone mode, you will need to compile the files within the ModelSim.

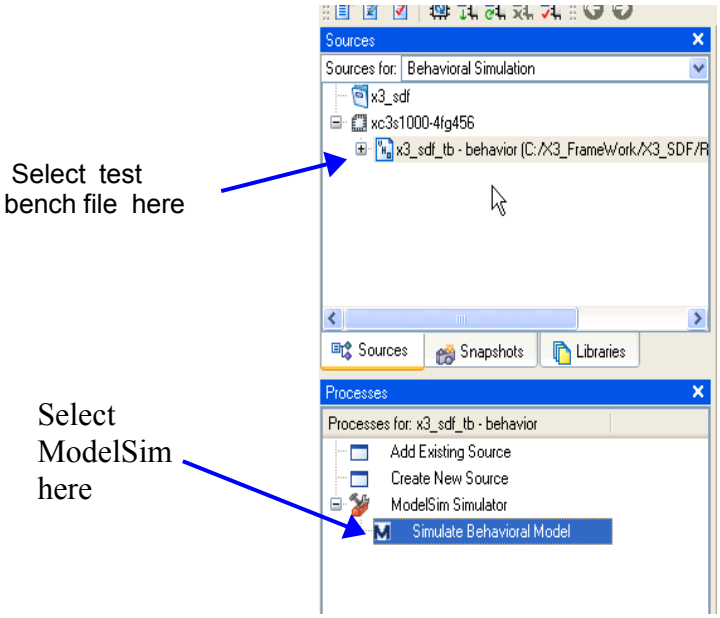


Figure 7. Invoking ModelSim from Xilinx ISE

In many cases a macro file is provided that compiles the files within ModelSim according to the macro file order. If you use this file, just add your files to the list in order of ascending hierarchy. These macro files have a .do extension; usually *project_name.do* for the project loading, and *wave.do* for the wave window format. You can reference this macro inside Xilinx ISE by changing the properties of the simulator window as shown here.

The advantage of using the custom DO file is that the additional packages, models and wave window setup can be easily automated when the simulator is invoked.

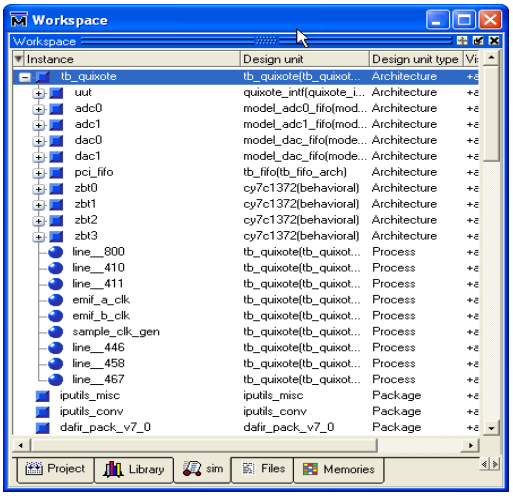


Figure 8. Example ModelSim Workspace After Loading

Once the design is loaded, the design hierarchy is shown in the Workspace window with the test bench at the top of the hierarchy. Here is an example.

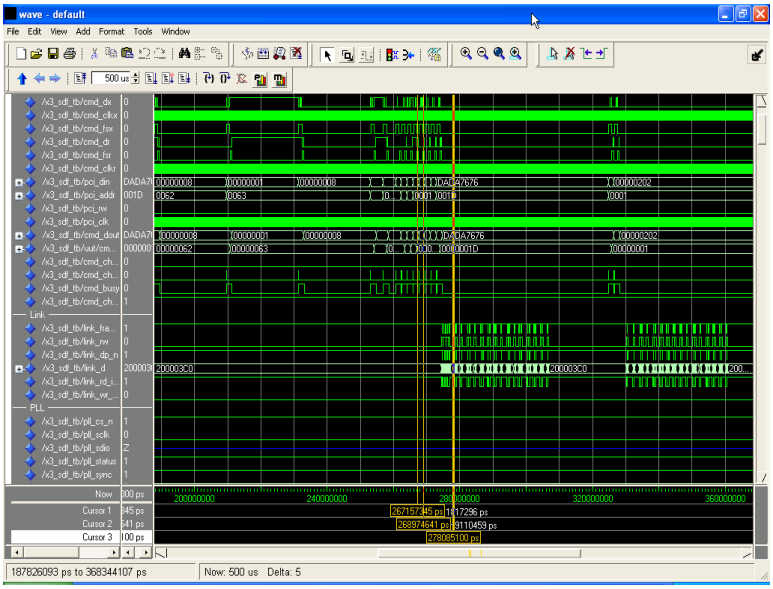


Figure 9. ModelSim Wave Window Example

Once you are inside the ModelSim environment, you should be able to use the tools to run simulations of the design. The wave window is many times the main focus since it gives a logic analyzer view of the design.

You can quickly view debug the design within ModelSim because you can probe the logic down to the lowest level. This visibility is often lost after synthesis and fitting because logic is minimized by the tools and may be trimmed out if unused, even if by accident. When you select an design unit within ModelSim Work Space window, the processes, signals and variable for that design unit are shown. You can add them to the window by selecting them and right-clicking to add to the wave window.

Some common simulation problems are

- libraries are wrong version – be sure to compile them from within Xilinx ISE before use
- wrong simulator resolution – be sure to use 1 ps resolution for all designs
- design won't load – be sure all models and packages were compiled
- naming conflicts – all instances must have a unique name
- source code won't compile – check that you compile with the correct version for 1993 to 2000 version of HDL

Logic Development using MATLAB Simulink

These tools are described in the *X3 MATLAB Board Support Package Manual*. Refer to that manual for details on logic development using MATLAB Simulink and Xilinx System Generator. The following description is just to orient you to what that tools are and how they may be useful in developing applications.

MATLAB Simulink provides a powerful method of developing logic using a high level design tool that integrates hardware into the MATLAB Simulink environment. Complex signal processing designs can be developed rapidly using the Simulink block diagrams interacting with the actual hardware in real time. Gateways between MATLAB Simulink and the hardware allow data to flow between the actual hardware and MATLAB, bringing the power of MATLAB to the logic development process.

Simulink blocks diagrams are directly translated into logic using the Xilinx System Generator tool. For each supported product, a hardware interface layer of Simulink components is provided that allows the hardware to be used in the the Simulink design. Simulink components from the various libraries provided by Mathworks, Xilinx and Innovative interface with this hardware interface layer for building the application logic on the product. The Xilinx place and route tools are used for the logic build as in any HDL project.

Here is a typical Simulink block diagram design. Notice the Xilinx icon in the upper right; this is the Xilinx System Generator control block. This block provides the link to the Xilinx place and route tools used to implement the logic. The other blocks are mixture of hardware interface components, such as the Quixote A/D converters, SRAM and DACs. The remaining blocks are Simulink functions for control, display and data formating.

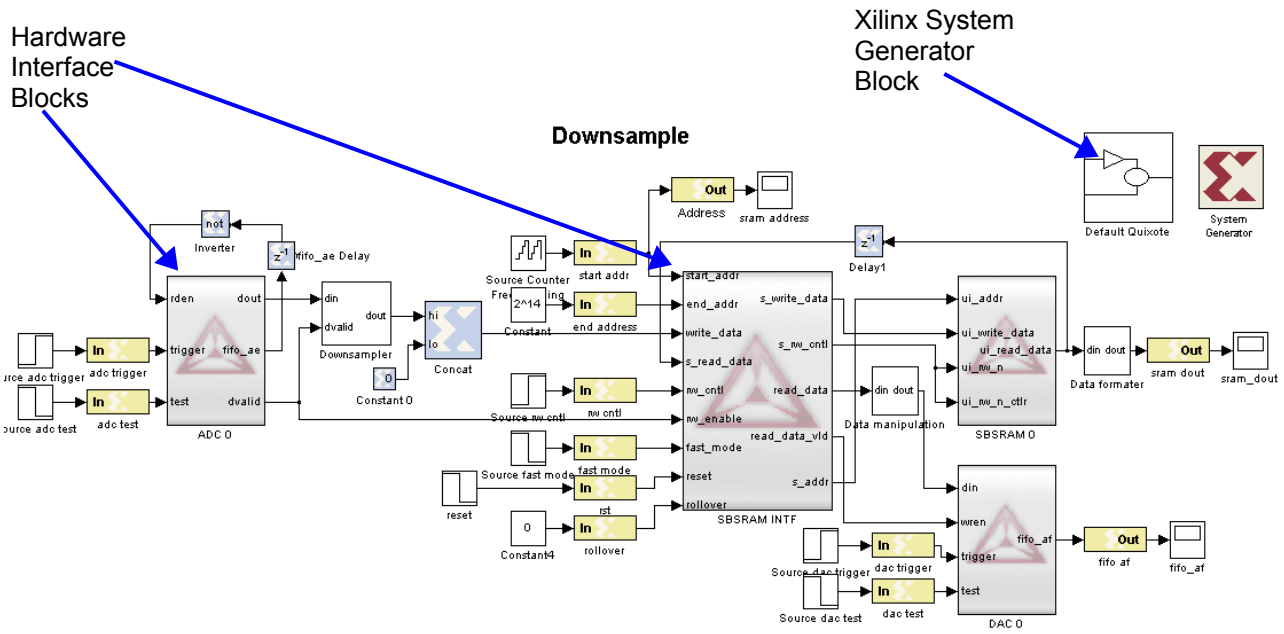


Figure 10. MATLAB Simulink Development

Making the Logic

The Xilinx ISE tools are used for the physical logic creation. For HDL designs, these tools are accessed through the ISE environment in the processes window as shown here. The main steps are translate (link), map and place & route.

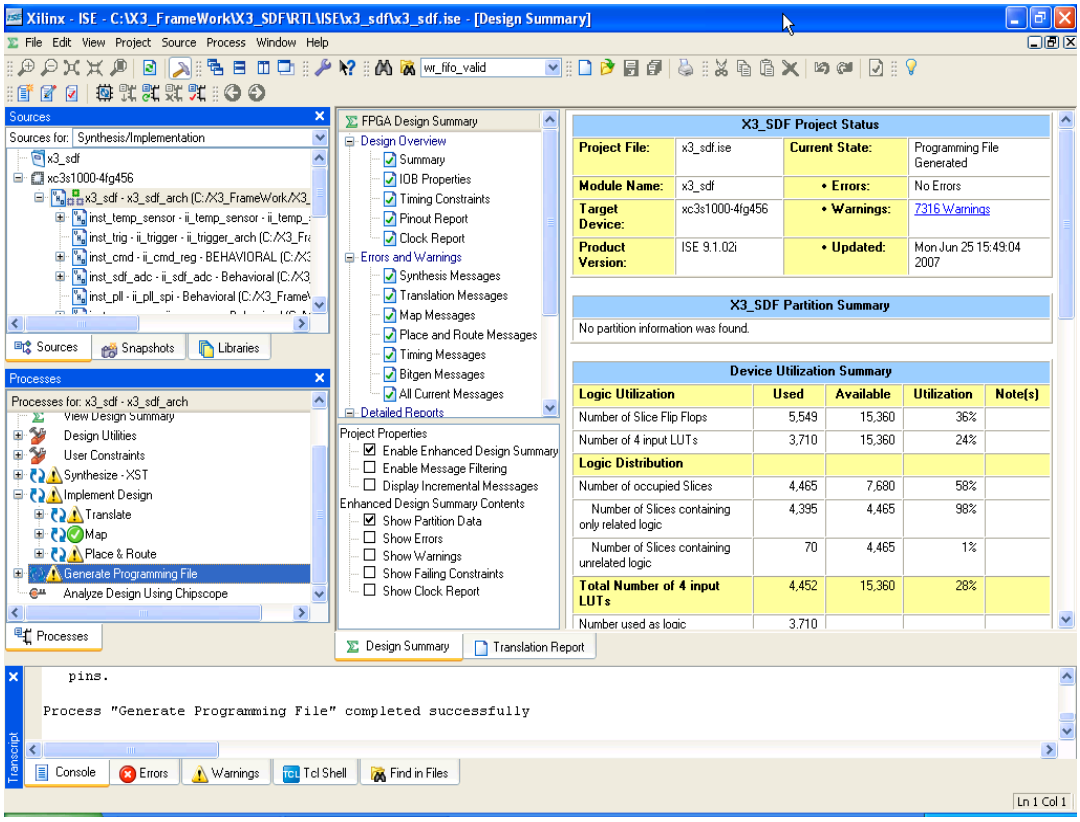


Figure 11. ISE Design Environment

There a many options for each of the implementation steps which are set in the individual project files for each FrameWork Logic example.

Since most of the chips on the products are very large, we have chosen to preserve the hierarchy of the design during the implementation so that area constraints and incremental design approach may be used. Area constraints allow the designer

to control the placement of logic on the FPGA chip for best timing control. With area constraints, the logic will be constrained to where you put it and in many cases helps the tool do a better job overall.

Place and Route Reports

The place and route implementation by the Xilinx tools results in several files you may want to review in case of problems. In ISE 9.1, a summary page is provided that gives a hyperlink to each report.

File extension	*Contents*	*What to Look For*
.BLD	The output from the NGDBuild process that link all the logic together	There should be no errors. This program issues numerous warnings but there should be no errors. Most link errors occur because of missing netlist files – Coregen put the files somewhere else, or a missing netlist from a black box.
.MAP	The output from the MAP process that does the logic mapping to the physical device. Removes unused logic and.	There should be no errors, but warnings are usually OK. Common problems range from incompatible logic mappings, impossible area constraints, and clock connections.
.PAR	The output of the Place and Route implementation process. Shows timing results and fit results.	Timing constraints should be met. Review the summary at the end of this report to see if timing is OK since it will complete no matter how bad it is. Also look for any incomplete routing.
.BGN	The output of the Bitgen tool.	Normally, this is not a problem. Occasionally though the design will route but not complete Bitgen will report this error.

Table 4. Xilinx Report Files Generated During Implementation

In many designs, you will have to resolve timing problems that are shown in the place and route process. Xilinx has several tools to help find the problems; Timing Analyzer is usually the place to start. This tool helps you to understand the reason the logic did not meet timing – too many connections, bad routing, etc. The tool suggests how to fix it. This is usually helpful but may mean you are back to functional simulation again to add pipelining or change the logic and must re-implement the design.

Once you achieve one good result, you may want to switch to incremental mode in the tool. This allows you to use the last good result for most of the design that is unchanged when minor fixes are made. For big changes however, you will need to reroute the whole chip.

Expect that the implementation process will be in the range of 10 to 30 minutes depending on your computer, how easy it is to meet constraints and how big the chip is. A tightly packed, fast big chip will take a while. A Spartan3 that is 85% full, running at 67 MHz takes about 25 minutes to route on a Core2 Duo with 1 GB of RAM. We have found that 64-bit Linux OS on 2.4 GHz Opteron processor is about 2x faster, although multiple processor are not currently supported by the Xilinx tools.

The final output of the implementation process is a .BIT file that represents the logic image. This file is loaded into the logic using the LogicLoader applet or the JTAG cable.

Loading Logic

There are usually two methods of loading logic into the target FPGA : JTAG and via the PCI Express interface using the logic loading applet or loader that is embedded in application software. JTAG is normally used during the development process because it is quick and easy to to use and loading can be done from the ChipScope debug environment. Logic loading using the PCI Express interface is used in final deployment or when debug tools are not needed.

JTAG

Logic images may be loaded using the JTAG interface to the FPGA using a Xilinx JTAG cable such as USB or Parallel Cable IV. This provides a convenient method of quickly loading the logic during the development process but is not usually used in deployed applications.

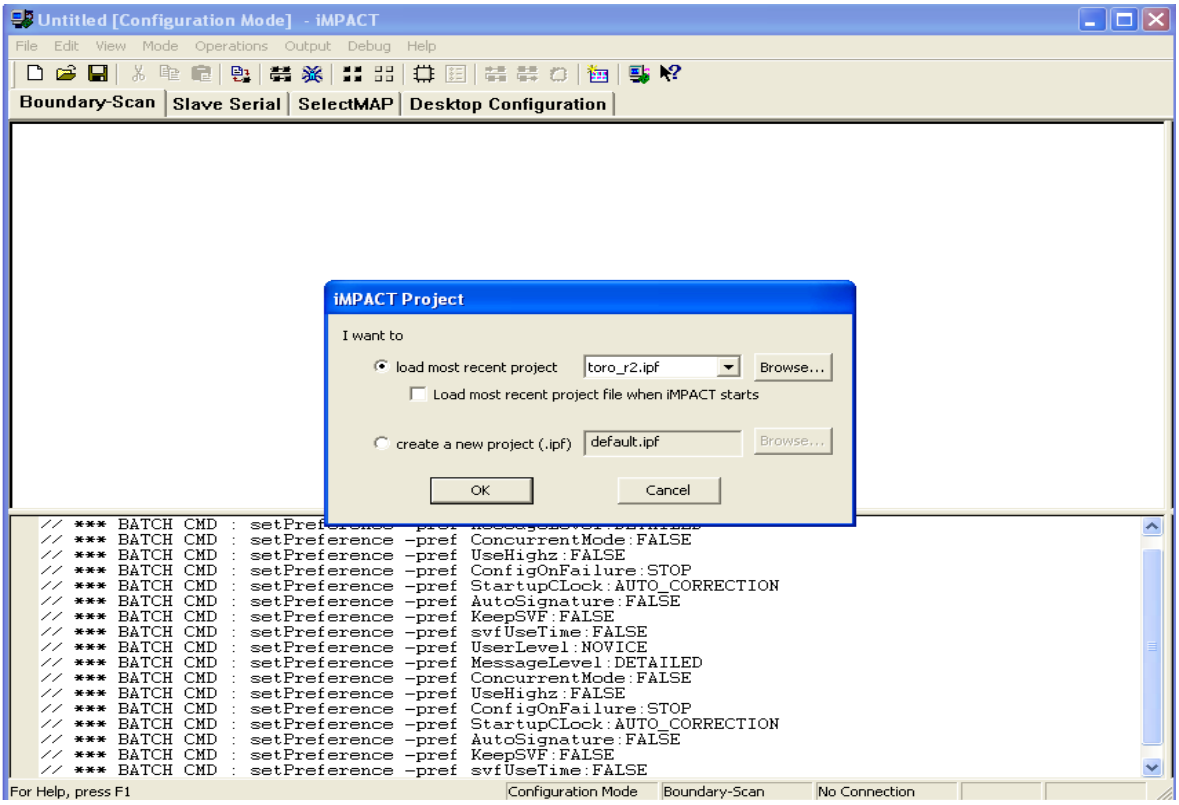


Figure 12. Getting Started with IMPACT

The Xilinx IMPACT application is used to load the logic into the application FPGA. The IMPACT tool may be invoked from the Xilinx ISE tool or as a standalone application. When you enter the tool, you will be prompted for either a stored

project or to create a new one. The first time you use this, you should create a new project. Later, you can save the project and it remembers the scan chain and files for the project.

After you select create a new project, the IMPACT wizard will direct you through the process of identifying the JTAG scan chain, assigning files and programming the devices. The first step is to select Operation Mode. For JTAG, pick configure devices.

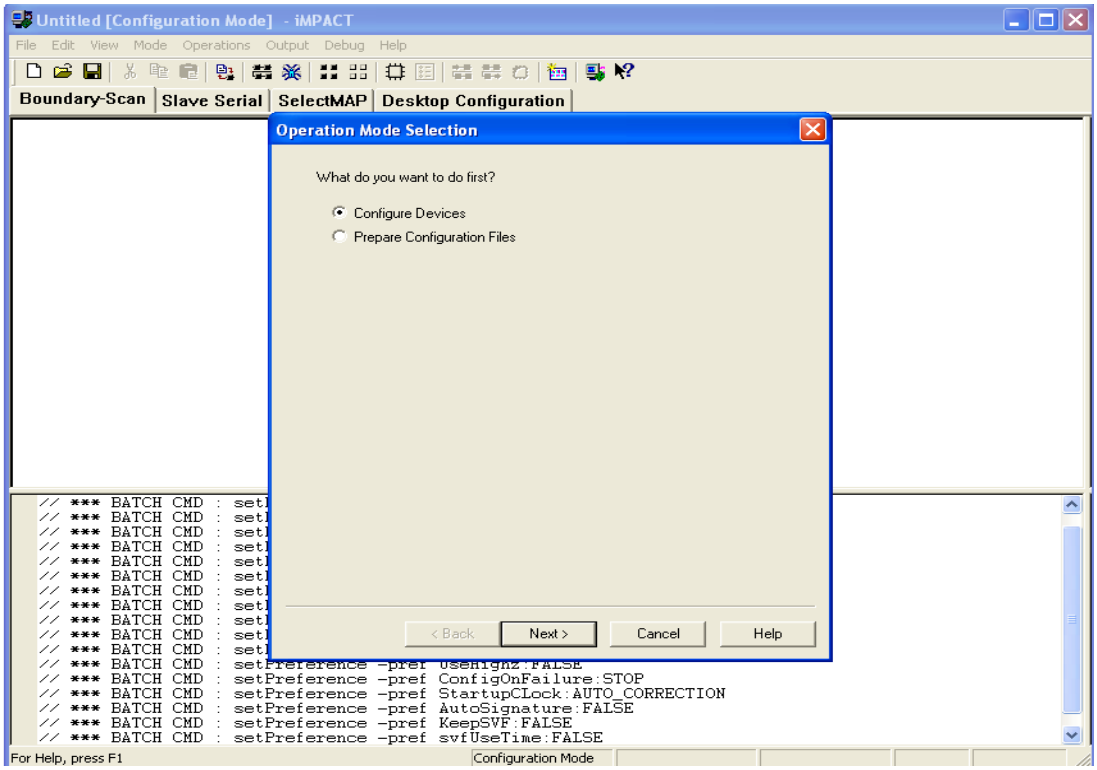


Figure 13. Choosing the Operation Mode for IMPACT

The next screen will prompt for what type of configuration to perform. Choose the Boundary Scan Mode as shown.

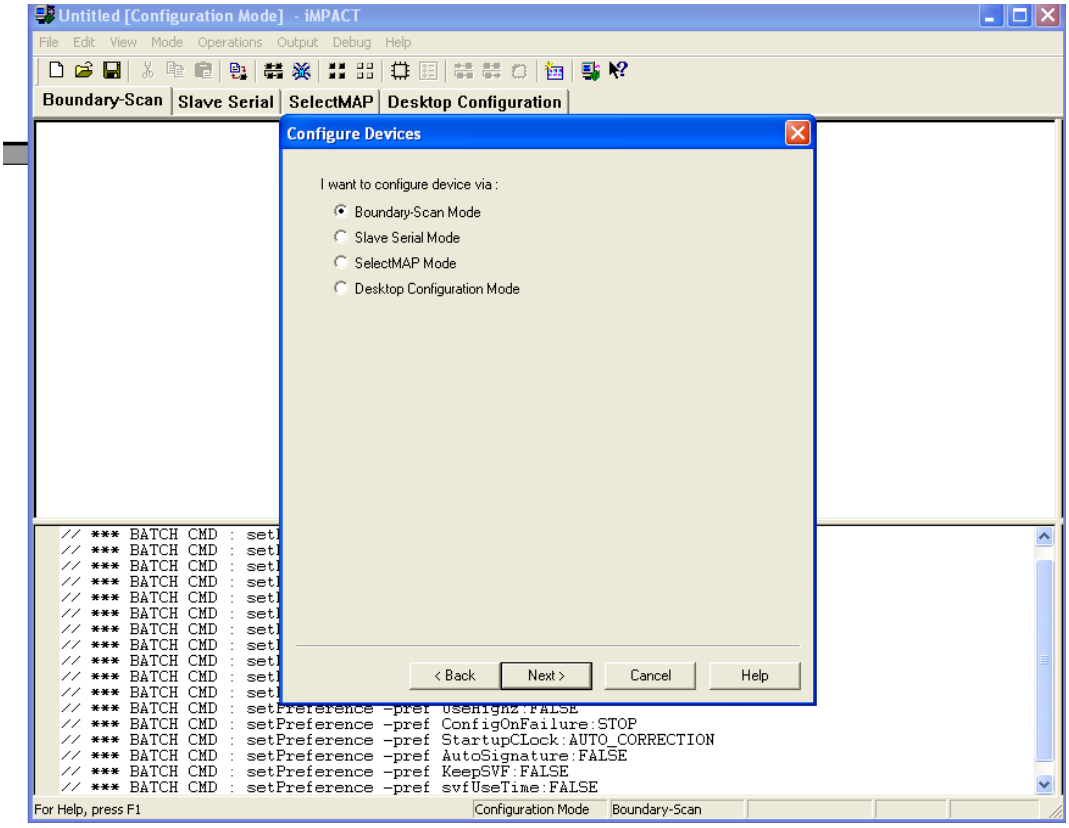


Figure 14. Choosing the Boundary Scan Mode for IMPACT

Next, IMPACT will prompt for the JTAG chain identification. Choose the automatic identify mode.

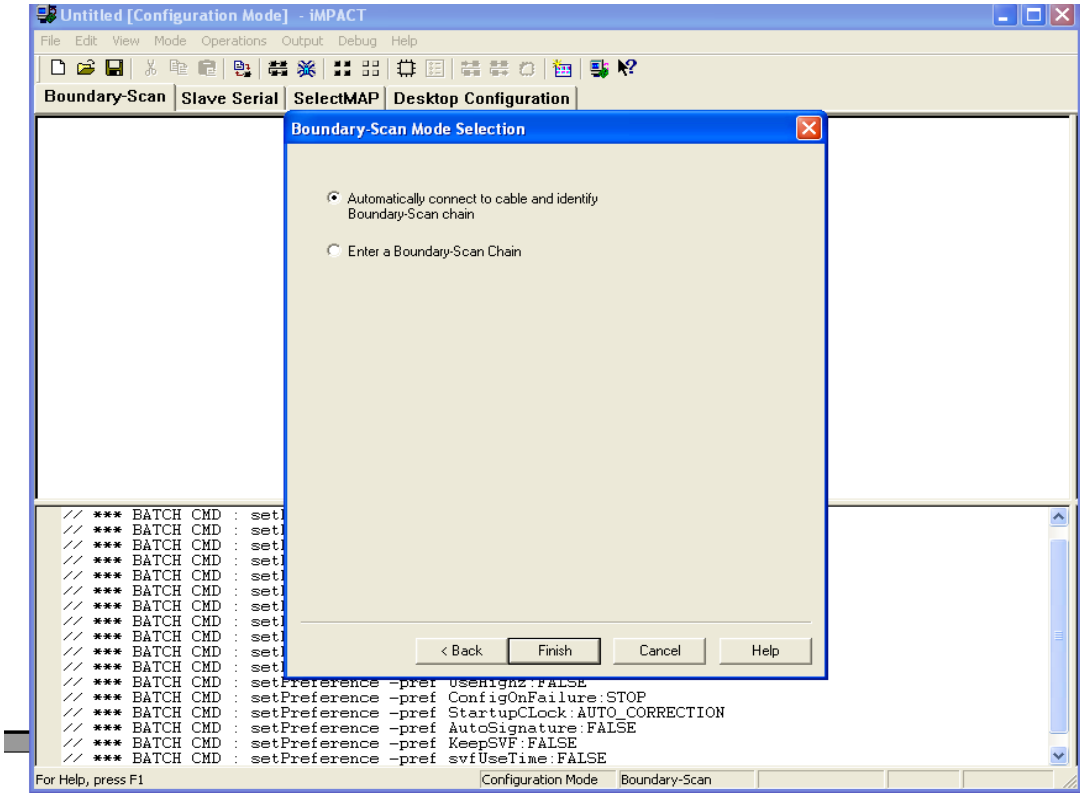


Figure 15. Automatic JTAG Chain Detection using IMPACT

If all goes well, IMPACT will find all the devices in the JTAG chain and display them as shown in this example. If not, check the cable connection to the board. The cable should be detected by IMPACT; if not, check that the port it is connected to on the PC is working and in the proper mode. If the chip is not detected, be sure you have the right scan path, that the board is powered up normally, and that IMPACT was able to connect to the scan path. Power everything off and try again if it fails and you don't see any obvious problems. You can also check your setup and software on a good card if you have one. If you don't, curse some then call tech support.

The next step is to assign a bit file to each device to be programmed. Right click each device and use the dialog to assign the BIT file for programming each device. **Double-check that you are using the correct BIT file – you could damage the chip if it gets the wrong logic**.

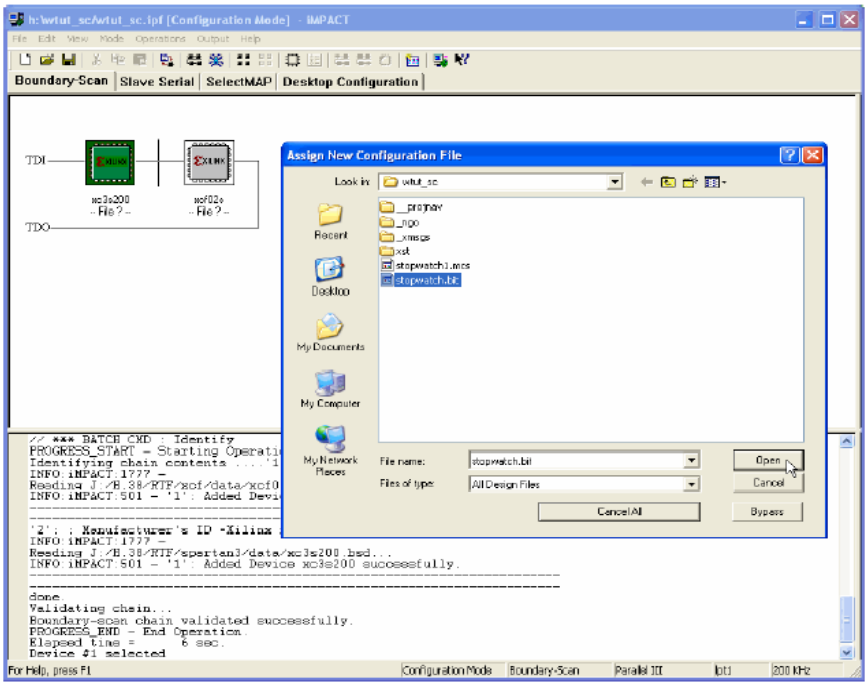


Figure 16. Selecting the Configuration Image Using IMPACT

The last step is to program the device. This is done by right clicking the device and selecting PROGRAM. This will only program the device you are selecting at that time. If it succeeds, then you will be notified by IMPACT as shown in the example screen.

NOTE: *We have found problems using the VERIFY function. It seems as though IMPACT has some problems doing this and will usually fail. It is our experience however that if the programming succeeds, then you will be OK. Xilinx employs a rigorous checksum routine that is very good and seems to guarantee that programming succeeded when it reports as such.*

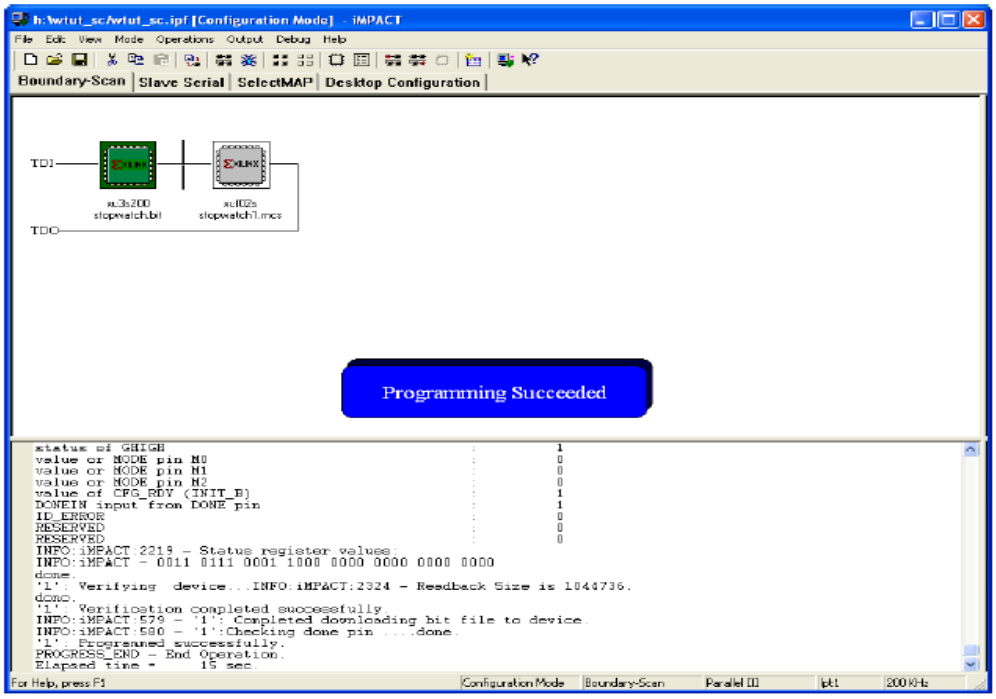


Figure 17. Programming Devices using JTAG under IMPACT

Loading the Logic Using BIT Images

Making the Device Image

The Xilinx ISE tools make a BIT file when the logic compile and fit is successful. The BIT file is the native Xilinx format for logic image. This BIT file is made when the Generate step is completed in the Xilinx ISE tool flow.

Logic Download

The X3 XMC family of products use a simple Windows application to download logic images to the card. This application can be used instead of the logic download cable used during development.

Note: The application logic must be loaded after each power-up since there is no on-card logic ROM.

The logic is loaded over the SelectMAP interface to the FPGA via the PCI bus. The logic may be loaded and reloaded at any time during operation.

The Velocia Download applet requires a logic image in BIT or EXO format. BIT format is normally used since it is created automatically by the Xilinx tools.

To load the logic image, select either EXO or BIT format in drop-down box. Then select the file for loading. You can navigate to this file using the standard techniques. Then press the Load button. Status indicators will first show the logic file being parsed, then the loading. Loading takes about 5 seconds for a 1M device. If anything goes wrong, the Event Log window reports a failure. Parsing failures usually mean that the file was not EXO format. A load failure is more serious in that the hardware failure somehow.

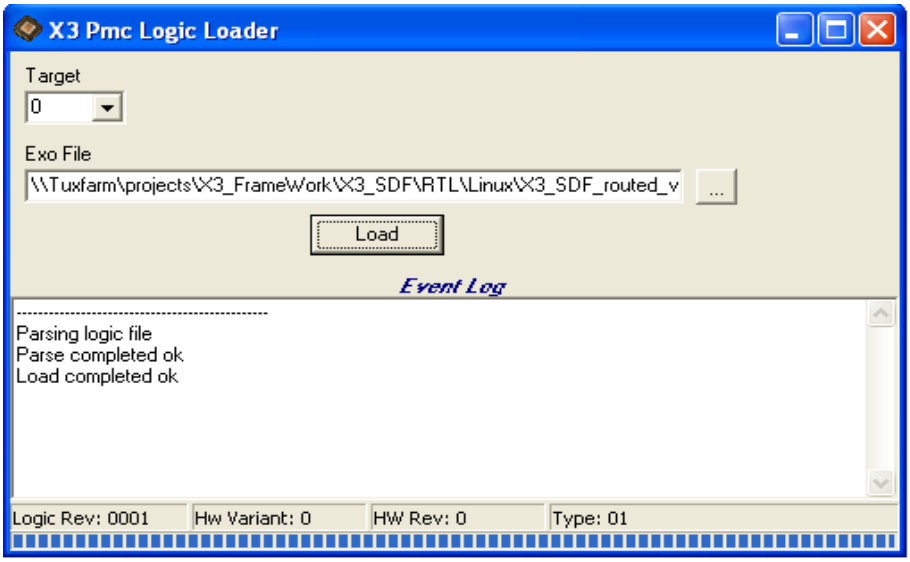


Figure 18. X3 Logic Loader Download Applet

The X3 LogicLoader applet shows the logic revisions, hardware revisions and other information on the lower line. The logic revision is the current version in the PCI interface FPGA. The hardware revision and hardware variant are set by the hardware PCB and PCIe interface FPGA.

Multiple cards in a single system are supported via the target number box. You can identify which card is which target by blinking its LED with the Finder applet.

Software tools for loading the logic from application software are provided for loading the logic also. These are detailed in the software manual. As an example, the SNAP application for the X3-SDF module provides a logic downloader that is embedded in the program.

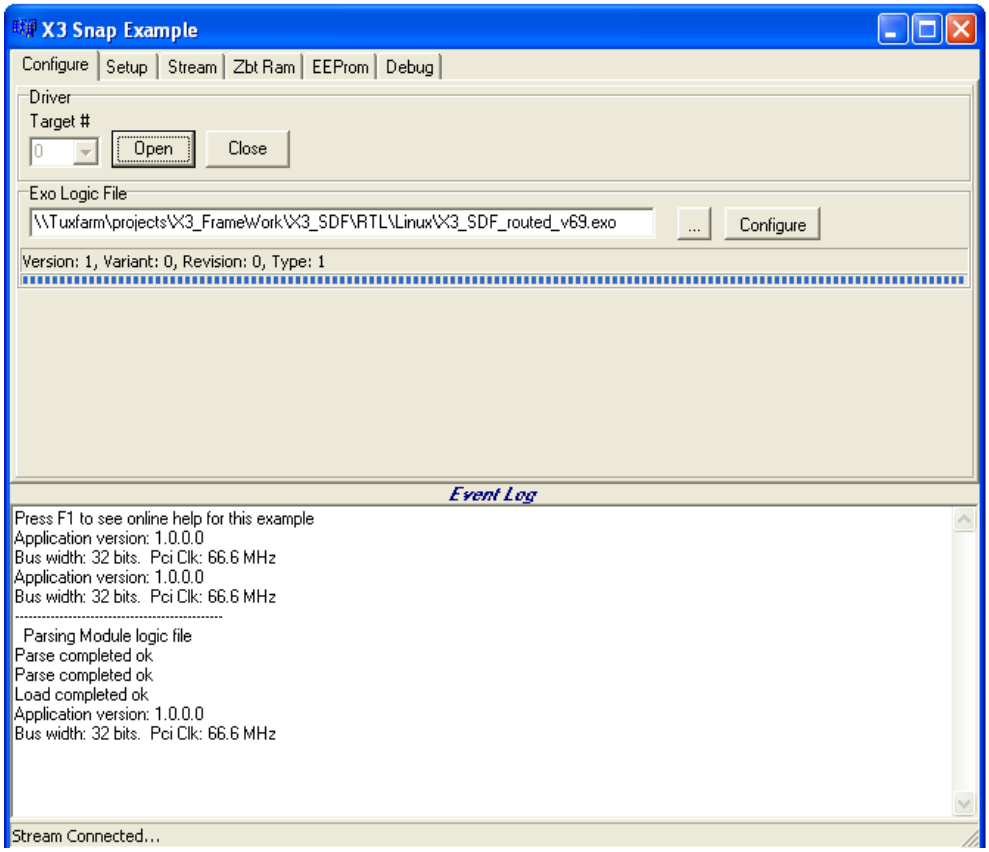


Figure 19. Example of Logic Loading from Application Software

Debugging

It is inevitable that the logic will require some debugging and it is best to have a strategy for debug before you actually use the hardware. Debugging on actual hardware is difficult because you have poor visibility into the FPGA internals.

For HDL designs, the best and easiest debug method is simulation for functional and timing problems. This gives the best visibility and interactivity to debug problems before the real hardware is tested. A good set of test cases that stress the design should be run prior to working on the real hardware. You will save time in the overall design process by doing a thorough job in simulation. Sermon over.

There are several techniques that have worked for us on projects: Xilinx ChipScope, built-in test modes, and judicious use of test points. Between these techniques and the capabilities of each method, it is usually possible to find and fix bugs that are either functional design errors or timing problems.

MATLAB Simulink developers can use the “hardware in the loop” features of system to debug the design at a high level. Simulink can be used to generate test data or for viewing and analyzing real hardware data. This is invaluable in debugging complex signal processing designs.

Here we will discuss a few of these techniques.

Built-in Test Modes

Another good way to debug your design is to have built-in test modes in the logic. If you plan ahead for test, then you can more quickly validate your design later and spot problems. When you finish the design, if the test generators and checkers can be left in the design, they are there later as production debug or test.

In many designs, test pattern or data generators are invaluable since they provide known data into the FPGA so that the output is known. If the data source is analog in the real design, substituting perfect data is nice because it helps spot problems that may be hidden in the noise. The test pattern may be an easily recognized stream, like incrementing numbers, that are easy to check in the logic or on the test equipment. Also, its easier to test the extreme cases of the design that may be difficult to reproduce with real signals.

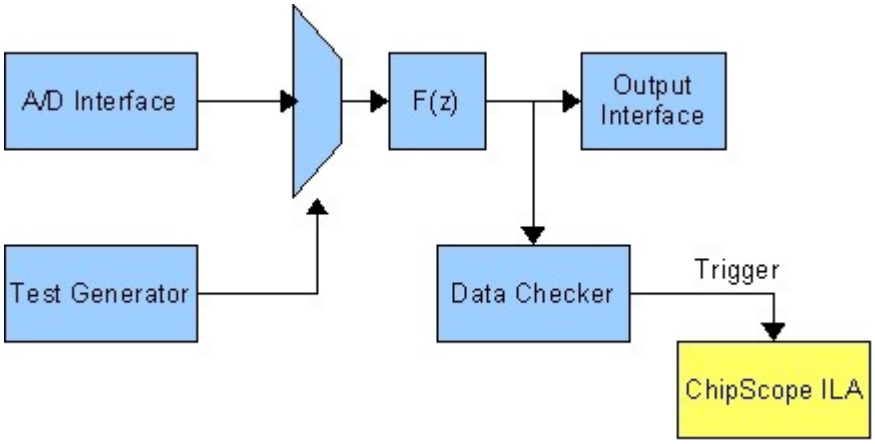


Figure 20. Typical Debug Block Diagram

Another built-in test method is to use data checkers in the logic sprinkled through the data chain help to spot the source of problems. If you have a missing timing constraint or a clock domain issue, these can be hard to catch since they may be rare. A data checker gives you a way to look for bad data and then trigger ChipScope or the logic analyzer. In many cases, rare errors are impossible to catch without this sort of data checker. This technique has saved time because the trigger at the bad data point allows you to inspect all the suspect signals and find the culprit.

Xilinx ChipScope

Xilinx offers an excellent tool for debugging FPGA designs call ChipScope. This tool works over the FPGA JTAG port using any of the standard Xilinx JTAG cables. Software on the PC connects to a ChipScope logic core that you embed in your logic. This is an optional tool from Xilinx and is not included in the standard ISE software. For its cost of under $1500, we have found it well worth the money.

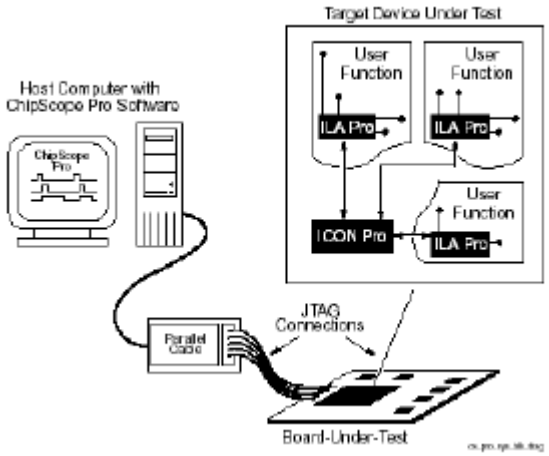


Figure 21. Debugging with ChipScope

The ChipScope core allows you to monitor internal FPGA signals using triggers and a sample clock. It is as though you can embed a logic analyzer in the logic itself, hence the ILA core name (integrated logic analyzer). Other ChipScope cores support Virtual IO (VIO) core, which allows you to monitor and control some internal signals, and cores for working with the PowerPC cores in some logic devices.

The ILA core is very configurable and it allows you to set the number of signals you can monitor, the trigger methods and the signals used for triggers is set up when you generate the ChipScope core. The core size is determined by the number of signals monitored and the number of samples stored. If the core gets too big, it will affect your design and tends to muddle the debug process. Sometimes it is better to have a small core that has a small footprint and does not interfere with the other logic for this reason.

The clock is used as the sample clock for the logic so it should be synchronous to the inputs signals or sufficiently fast to sample them accurately. If you sample signals on other clock domains, be aware that the clock used by the ILA core is used for the sampling of the signals so the signals will not precisely represent the real signal running on another clock domain.

You will interact with the ChipScope software over a JTAG cable to the target device. This link is limited to about 1-20 Mb/s depending on the target device JTAG chain, so it is not really real-time, but rather just a means to get the data from the FPGA to the ChipScope software. The signals are captured in the FPGA block RAMs so the record length is somewhat short being limited in most cases to 256 to 1K points. In some experiments though we have made larger captures of up to 16K points in large devices, useful for capturing a signal.

Because of these limitations in JTAG speed and capture size, it is important to devise triggering methods that allow you to catch the error condition. It is common to devise a piece of error detection logic that serves as a trigger to ChipScope to best use the capture RAM. It is possible to pre-trigger or post-trigger in the software which makes trigger design easier. You can also selectively store data so that the memory is preserved just for useful data by using an option on the trigger panel in ChipScope.

Here is one of the common cables used for debug, just for reference.

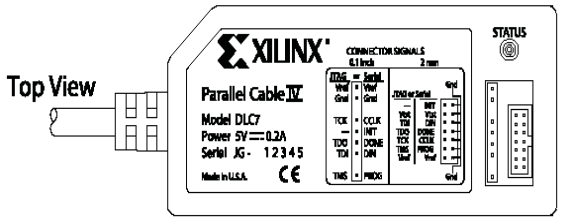


Figure 22. Xilinx Parallel IV Cable for Debug and Development

Figure 23. Ribbon cable for Xilinx JTAG (2mm, 5x2 female on each end, 6 inches length)

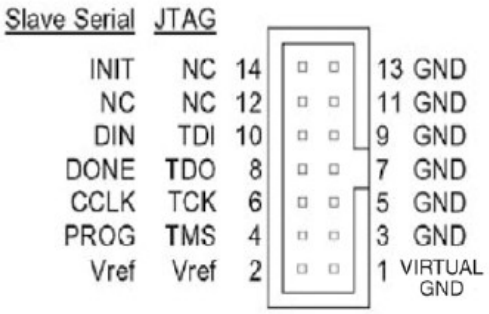


Figure 24. Xilinx Parallel Cable IV Pinout on IDC 5x2 2MM Header

CAUTION:

The user MUST make sure that Xilinx JTAG cable connector is plugged in the proper polarity to the Innovative target connector. If by mistake, the user connects the Xilinx cable incorrectly, this may damage the target card and Xilinx POD. See the hardware manual for each product to locate the connector and its pinout.

Declaration Of ChipScope Core in VHDL

The ChipScope core is simple to use. Just connect up the signal for observation to the data ports, the trigger signals to trigger and the clock. The number of ports and triggers is defined when you create the debug core in the ChipScope tool. The clock is used as the sample clock for the logic analyzer, so you must use a clock that is higher frequency than the signals you wish to observe.

Here are Chipscope cores we used in debug shown below.

```
component icon
  port
  (
    control0    : out std_logic_vector(35 downto 0);
    control1    : out std_logic_vector(35 downto 0);
    control2    : out std_logic_vector(35 downto 0)
  );
end component;


component ila
  port
  (
    control     : in    std_logic_vector(35 downto 0);
    clk         : in    std_logic;
    trig0       : in    std_logic_vector(31 downto 0)
  );
end component;

  component vio
  port
  (
    control     : in    std_logic_vector(35 downto 0);
    clk         : in    std_logic;
    async_in    : in    std_logic_vector(15 downto 0);
    async_out   :   out std_logic_vector(15 downto 0);
    sync_in     : in    std_logic_vector(39 downto 0);
    sync_out    :   out std_logic_vector(127 downto 0)
  );
end component;
```

Figure 25. ChipScope Core Declarations

Here is its instantiation during a debug session.

```
 -- chipscope for debug/testing

inst: icon
    port map
    (
```

```
      control0    => ila_control0,
      control1    => ila_control1,
      control2    => ila_control2
    );


  inst_ila0 : ila
    port map
    (
      control   => ila_control0,
      clk       => sys_clk,  -- fs_clk
      trig0     => ila0
    );

------ debug A/D interface
ila0(0) <= reset;
ila0(1) <= fs_clk;
ila0(2) <= adc_trigger;
ila0(3) <= ctl_reg(1)(10);  --sw trigger
ila0(4) <= adc_trigger_en;
ila0(5) <= trigger_tp(28);
ila0(6) <= trigger_tp(27);  --trigger_vld
ila0(9 downto 7) <= trigger_tp(31 downto 29);
ila0(10) <= trigger_tp(24);
ila0(11) <= adc_cs_n_s(0);
ila0(12) <= adc_rd_n_s(0);
ila0(13) <= trig_en;
ila0(14) <= trigger_tp(25); --trigger_edge
ila0(15) <= trigger_tp(26); --trigger_src
```

Figure 26. ChipScope Core Instantiation

In this case, the designer was using the system clock as the ChipScope core clock and had many of the A/D triggering signals connected to observe using ChipScope.

Once the core is in the design, you can then trigger on different conditions just as you would use a logic analyzer. If connect up all the signals in the problem area, only one compilation is needed to get the core into the design for debug. Once you get it all working, you have a logic analyzer inside the FPGA. Here's sample view.

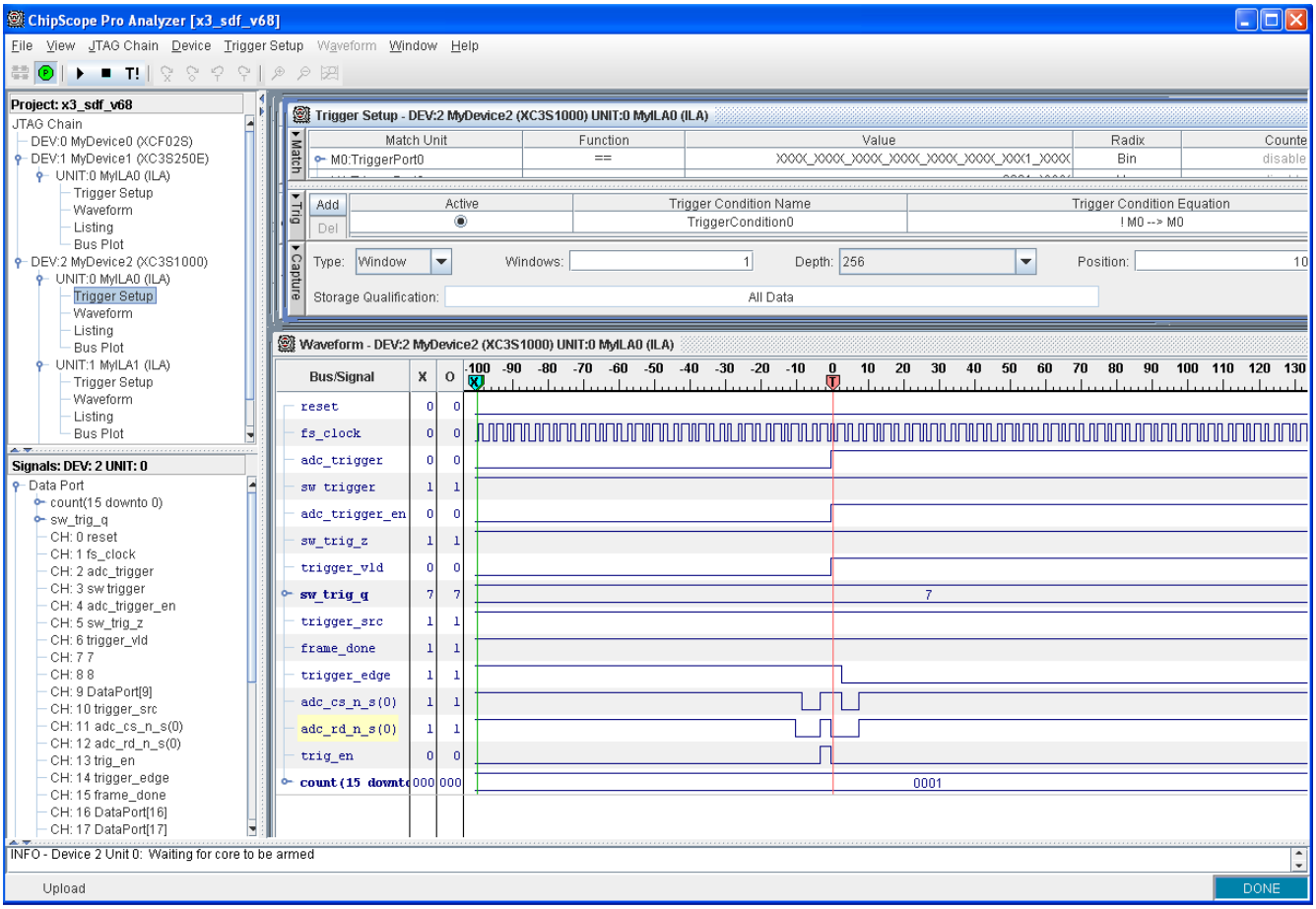


Figure 27. ChipScope Debug Example

As you can see, the screen is like a logic analyzer and the internal logic signals are being observed real-time. This visibility into the internal operation of the logic is invaluable in finding design problems and saves time.

X3-SD16 FrameWork Logic

Overview

This chapter of the manual discusses the FrameWork Logic for the X3-SD16, provides information on modifying the logic and details on hardware interfaces to the FPGA for the logic designer. This chapter is intended to support logic developers who intend to modify the X3-SD16 FrameWork Logic to add custom features.

The X3-SD16, a member of the X3 XMC module family, has 16 channels of 24-bit, 144 kSPS A/D, 16 channels of 24-bit, 192 kHz DAC, and a Xilinx Spartan3A DSP 1.8M FPGA for signal processing. As is common with the other X3 XMCs, the application FPGA has dual SRAM memory buffers, precision clock sources, a packet-protocol PCI Express interface and digital IO using P16.

The FrameWork logic for the X3-SD16 provides the hardware interface components and system controls for basic data acquisition and playback. The logic is structured so that signal processing can be added to the application logic in the application logic with minimum low-level device interfacing. The FrameWork logic supports VHDL and MATLAB development for the X3-SD16.

The X3-SD16 has two FPGAs: a Xilinx Spartan3A DSP 1.8M for application logic and a Spartan3E 250K dedicated to PCI interface and module control. The Spartan3A DSP 1.8M device provides a flexible logic core for signal processing, data analysis and module control. The Spartan 3E device is not normally modified and is not discussed in this manual.

Target Devices

The standard X3-SD16 is available with the following FPGA device:

X3-SD16 Variant	Logic Density	Device Used
X3-SD16 (80179-0)	1.8M gates	Xilinx XC3SD1800A-4FGG676C

Table 5. X3-SD16 FPGA Device Part Number

Notes: Higher speed grades may be special ordered. The logic density of 1.8M is an approximation of the gate counts. See the Xilinx data sheet for complete details.

Full data sheets and User Guides are available on the Xilinx web site at this address:

http://www.xilinx.com/xlnx/xweb/xil_publications_display.jsp?iLanguageID=1&category=-1209726&sGlobalNavPick=&sSecondaryNavPick=

Development Tools

The currently supported tool set is shown here. The Xilinx WebPack tools can also be used, which are available on the Xilinx website free of charge.

Function	*Tool Vendor*	*Tool Name*
Synthesis, Place and Route	Xilinx	ISE 12.3
Simulation	Mentor Graphics	ModelSim 6.2c
Bit and PROM Image Creation	Xilinx	Impact 12.3
Logic Debug and Testing	Xilinx	ChipScope 12.3
Logic JTAG Cable	Xilinx	Xilinx USB

Table 6. Logic Development Tools for X3-SD16

FrameWork Logic

The block diagram of the X3-SD16 hardware shows how the application FPGA is the main processing and control element of the XMC. In the application FPGA, the FrameWork logic for the X3-SD16 provides the hardware interfaces in the logic, data flow and controls. Each of the hardware devices, such as the A/D converters and memories, has a component in the logic that controls the device and provides the logic with a simplified data interface.

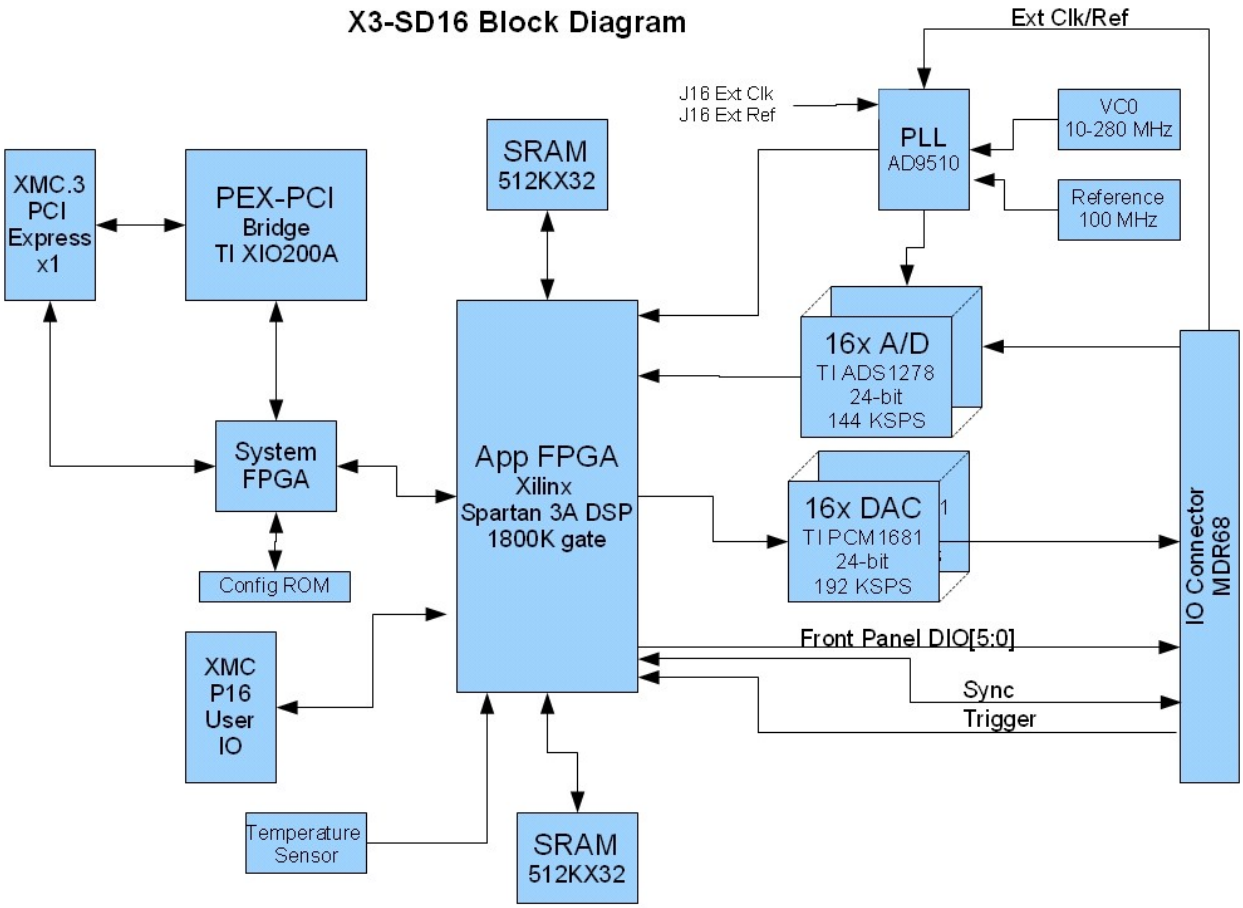


Figure 28. X3-SD16 Hardware Diagram

In the FrameWork logic, the data flow for acquisition is from the A/Ds, through the error correction and into the SRAM data buffer, then on to the packetizer and out to the the PCIe interface. The data flow for playback is similar in reverse: data flows in from the PCIe interface, to the packet deframer, into the data buffer, then out to the DAC interface. This logic flow provides basic data acquisition and playback functionality using many of the X3-SD16 features. The command channel is used for control and status reporting features such as the control of PLL, resets and data flow configuration.

Some of the other features, such as the computational SRAM, are connected in the logic solely for test and example use. While these are simple connections in the logic, they can be modified to provide higher performance, specialized functions.

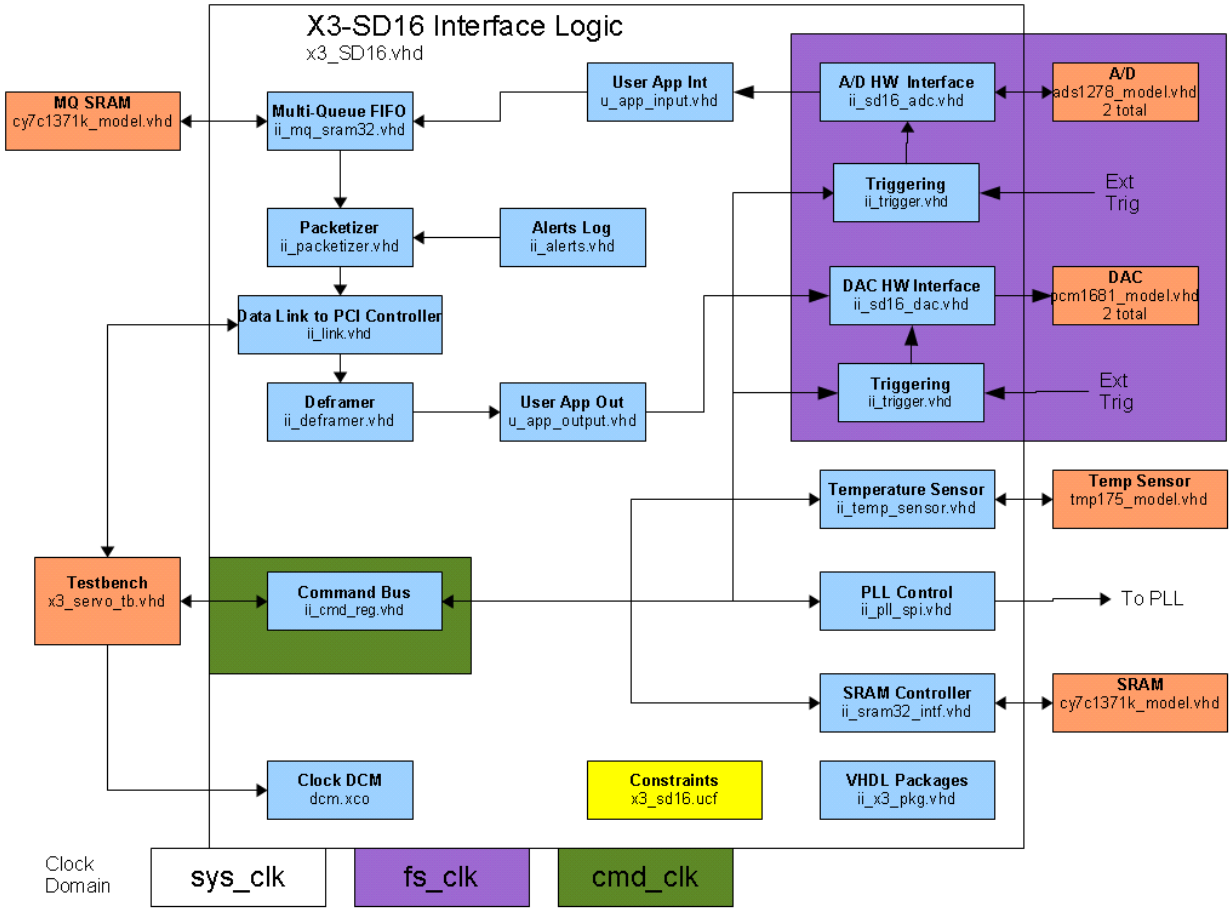
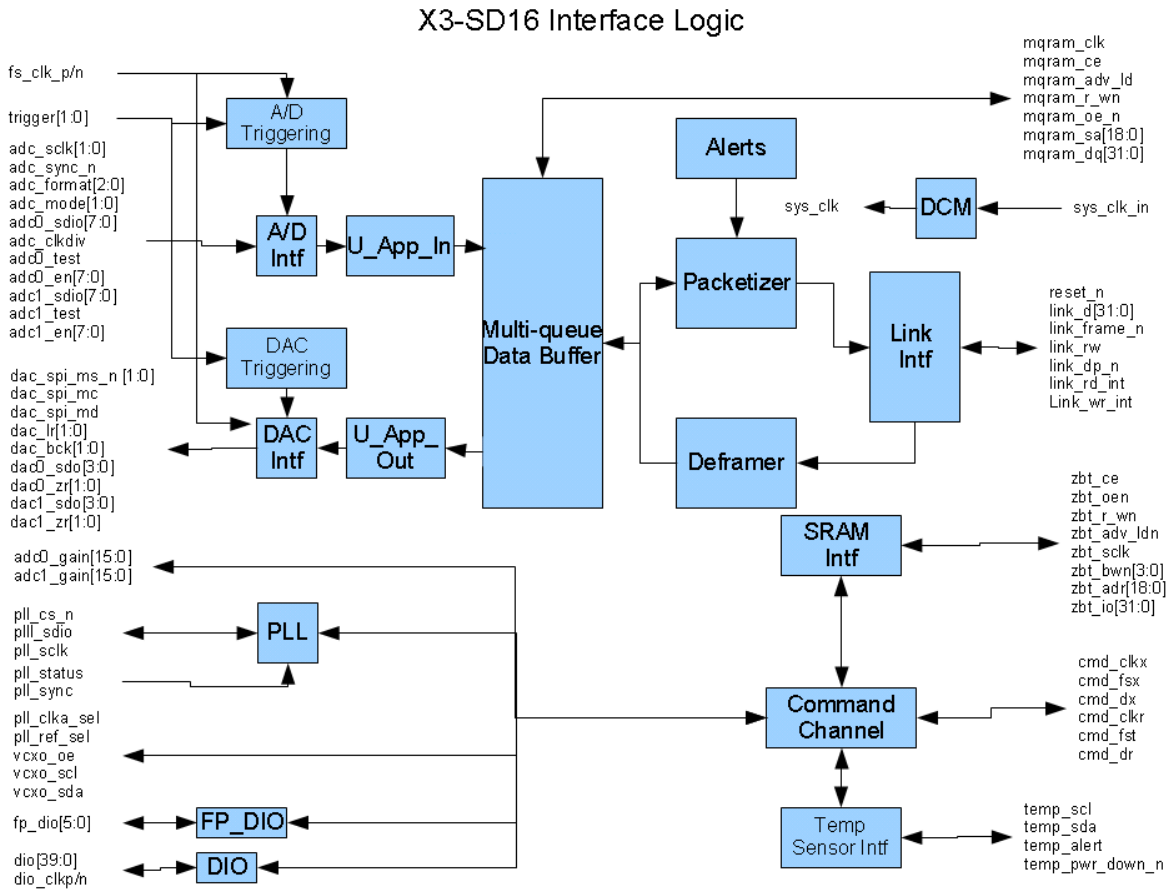


Figure 29. X3-SD16 FrameWork Logic Files

In the block diagram, several major components are shown: the A/D interface, DAC interface, the multi-queue VFIFO data buffer, packetizer, deframer, memory interface for SRAM, the data link to the PCIe controller, command bus and the alert log. The command bus interface controls and monitors the triggering, PLL and SRAM, as well as providing reset and status functions for the module to the PCI host. Simulation is driven by the testbench and uses simulation models for the memories, data link and command channel. Other models for the A/D and temperature sensor provide simple models to verify communications, but do not emulate all the features of the device.



The most important thing to notice about the logic design is that it is organized as a hardware interface layer composed of hardware interface components and an application core. The data flow is readily modified to add functions to the logic that analyze and process the data streams flowing between the hardware interfaces. In a typical application, signal processing can then be added by inserting new application logic into the data stream. This new application logic can be done with either VHDL or by using the MATLAB Simulink with Xilinx System Generator tool.

Figure 30. X3-SD16 FrameWork Logic Block Diagram

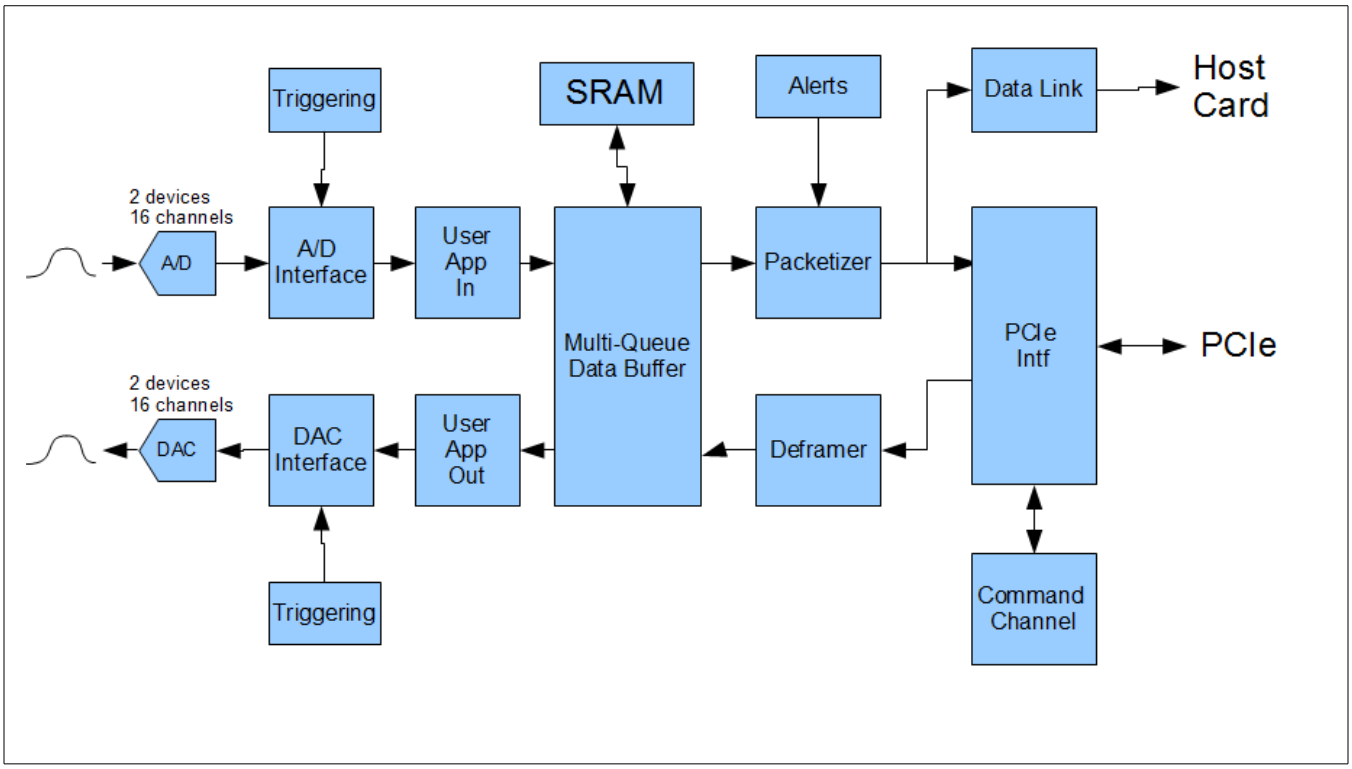


Figure 31. X3-SD16 Logic Data Flow

X3-SD16 FrameWork Logic Ports

The ports for the top level show the external connections of the logic and their functions. Signal directions are relative to the X3-SD16 application logic.

Port Name	Direction	Clock Domain	Function
hw_rev[3:0]	In	-	Hardware revision code from the PCB. Used to identify to the logic and system what the PCB revision is.
led	Out	sys_clk	Application LED output from the logic. A logic '0' turns the LED on.
rs tn	in	-	Master reset into the FPGA controlled by the PCI interface FPGA. Active low.
Clocks and Trigger			
sys_clk_in	in	sys_clk	System clock input, 67 MHz.
fs_clkp/n	In	fs_clk	Sample clock, differential input (LVPECL).

Port Name	Direction	Clock Domain	Function
pll_ref_sel	Out	sys_clk	Selects the clock to the PLL reference input. '0' = 100 MHz oscillator (default), '1' = PXIE_100M from P16.
pll_clka_sel	Out	sys_clk	Selects the input to PLL clk A from the external clock mux. '0' = front panel ext_clkp/n (default), '1' = PXIE_DSTARA from P16.
ext_trigger[1:0]	In	external	External trigger inputs
Front Panel Digital IO			
fp_dio[5:0]	Inout	sys_clk	Front Panel Digital IO
DAC Interface Controls			
dac_spi_ms_n[1:0]	Out	sys_clk	DAC SPI chip select, active low
dac_spi_mc	Out	sys_clk	DAC SPI clock
dac_spi_md	Out	sys_clk	DAC SPI data
dac_lr[1:0]	Out	fs_clk	DAC left/right data frame
dac_bck[1:0]	Out	fs_clk	DAC bit clocks
dac0_sdo[3:0]	Out	fs_clk	DAC 0 serial data
dac0_zr[1:0]	In	fs_clk	DAC 0 zero crossing detect
dac1_sdo[3:0]	Out	fs_clk	DAC 1 serial data
dac1_zr[1:0]	In	fs_clk	DAC 1 zero crossing detect
ADC Interface and Controls			
adc_sclk[1:0]	Out	fs_clk	5 A/D serial data clock
adc_fsync[1:0]	Out	fs_clk	A/D data frame syncs
adc_sync_n[1:0]	Out	fs_clk	A/D sync control
adc_format[2:0]	Out	sys_clk	A/D format
adc_mode[1:0]	Out	sys_clk	A/D mode
adc0_sdio[7:0]	In	fs_clk	A/D 0 serial data bus
adc_clkdiv	Out	fs_clk	A/D clock divisor control
adc0_test	Out	sys_clk	A/D 0 test (=1)
adc0_en	Out	fs_clk	A/D 0 enable
adc1_sdio[7:0]	In	fs_clk	A/D 1 serial data bus
adc1_test	Out	fs_clk	A/D 1 test (=1)
adc1_en	Out	fs_clk	A/D 1 enable

Port Name	Direction	Clock Domain	Function
adc0_gain[15:0]	Out	sys_clk	A/D 0 channel gain controls
adc1_gain[15:0]	Out	sys_clk	A/D 1 channel gain controls
PLL Interface			
pll_cs_n	Out	sys_clk	PLL chi select, active low.
pll_sclk	Out	sys_clk	PLL serial clock.
pll_sdio	Inout	sys_clk	PLL serial data.
pll_status	In	undefined	PLL status pin input.
pll_sync	Out	sys_clk	PLL synchronization control pin.
vcxo_oe	Out	sys_clk	VCXO output enable
vcxo_scl	Inout	sys_clk	VCXO I2C bus clock
vcxo_sda	Inout	sys_clk	VCXO I2C data
Multi-queue SRAM Interface			
mqram_ce	Out	sys_clk	Multi-Queue SRAM chip enable.
mqram_oen	Out	sys_clk	Multi-Queue SRAM output enable, active low.
mqram_r_wn	Out	sys_clk	Multi-QueueSRAM read/write.
mqram_sclk	Out	sys_clk	Multi-Queue SRAM clock.
mqram_sa[20:0]	Out	sys_clk	Multi-Queue SRAM address bus.
mqram_dq[31:0]	Inout	sys_clk	Multi-Queue SRAM data bus
ZBT SRAM			
zbt_ce	Out	sys_clk	SRAM chip enable.
zbt_oen	Out	sys_clk	SRAM output enable, active low.
zbt_r_wn	Out	sys_clk	SRAM read/write.
zbt_adv_ldn	Out	sys_clk	SRAM address advance/load.
zbt_sclk	Out	sys_clk	SRAM clock.
zbt_bwn[3:0]	Out	sys_clk	SRAM byte writes, active low.
zbt_adr[20:0]	Out	sys_clk	SRAM address bus.
zbt_io[31:0]	Inout	sys_clk	SRAM data bus
Link Interface			
link_d[31:0]	Inout	sys_clk	Link data bus.

Port Name	Direction	Clock Domain	Function
link_frame_n	In	sys_clk	Link data frame, active low.
link_rw	In	sys_clk	Link read/write(low) input.
link_dp_n	In	sys_clk	Link data phase, active low.
link_rd_intn	Out	sys_clk	Link read interrupt (data is ready for the PCIe controller), active low.
link_wr_intn	Out	sys_clk	Link read interrupt (application is ready to receive data from the PCIe controller), active low.
Command Channel			
cmd_dx	In	cmd_clkx	Command channel input serial data.
cmd_clkx	In	cmd_clkx	Command channel input clock.
cmd_fsx	In	cmd_clkx	Command channel input frame sync.
cmd_dr	Out	cmd_clkx	Command channel output serial data.
cmd_fsr	Out	cmd_clkx	Command channel output clock.
cmd_clkr	Out	cmd_clkx	Command channel output frame sync.
Digital IO			
dig_io[43:0]	Inout	sys_clk	Digital IO bits. Many are differential pairs.
dig_io_clkp/n	In	sys_clk	Digital IO clock differential pair input.
Temperature Sensor and Power Supply Enables			
temp_sclk	Out	sys_clk	I2C bus clock to temperature sensor.
temp_sda	Inout	sys_clk	I2C data to/from the temperature sensor.
temp_alert	In	undefined	Temperature sensor alert input.
ps_enable	Out	sys_clk	Power supply enable.
ps_enable_n	Out	sys_clk	Power supply enable, active low.

Table 7. X3-SD16 Logic Ports

Application Logic Help Files

An hyper linked help file system is available for X3-SD16 logic developers in the *./RTL/Linux/docu* directory. This help system shows hierarchy, logic entities, important processes and variables for the application design as hyper-linked HTML documentation. This documentation is generated from the source code itself and shows all of the code used in the design.

To use the HTML documentation, open the *index.htm* file under the *docu* directory. A standard browser program such as Windows Internet Explorer or Mozilla Firefox can be used to view the design structure.

Memory Map

The X3-SD16 memory map for PCI Express bus accesses uses two regions in the host memory. Each region has its origin at the Base Address Register (BAR) assigned by the host as part of the plug-n-play process. The BAR0 region is used for module controls, PCI Express interface functions and application FPGA loading. The BAR1 region is used for command/status in the application FPGA. Both memory regions are accessed by the host computer as memory accesses to BAR + Offset.

The command channel is mapped to X3-SD16 PCI base address register 1 (BAR1) of the PCI bus. Each address show is relative to the BAR1 enumerated address as the Offset. All registers are 32-bit.

Offset	R/W	Function	Module
0x00	R	User FPGA code version	All
0x01	R/W	Control Reg 0	All
0x02	R/W	Test	All
0x03	R	Temperature	All
0x04	R/W	Temperature Warning (W) / Clear (R)	All
0x05	R/W	Temperature Fail (W) / Clear (R)	All
0x06	R/W	A/D trigger delay	All
0x07	R/W	Front Panel DIO Control Register	25M, Servo, A4D4, 2M, SD16
0x08	R/W	PLL Clock Control	All
0x09	R/W	Sample Clock Divider and Distribution	A4D4, Servo
0x0A	R/W	PLL interface	All
0x0B	R/W	A/D channel enables (DIO In)	All
0x0C	R/W	DAC channel enables (DIO Out)	A4D4, 25M, Servo, DIO (byte enables), SD16
0x0D	R/W	DAC trigger delay	A4D4, 25M, Servo, DIO (byte enables), SD16
0x0E	R/W	ZBT SRAM address	All
0x0F	R/W	ZBT SRAM data	All
0x10	R/W	A/D Amplifier Enables	SD
0x11	R/W	DAC Controls (DIO Output)	A4D4, Servo, DIO, 25M, SD16
0x12	R/W	ADC Controls	SD, SDF, 2M, SD16
0x13	R/W	P16 DIO 31..0 Register	All
0x14	R/W	P16 DIO Control Register	All
0x15	R/W	Front Panel DIO (Does not apply to the DIO module)	25M, Servo, A4D4, 2M, SD16
0x16	R/W	P16 DIO 43..32 Register	All
0x17	R/W	Trigger Control A/D (DIO In)	All
0x18	R/W	Trigger Control DAC (DIO Out)	A4D4, DIO, 25M, Servo, SD16
0x19	R/W	A/D Decimation (DIO In)	All
0x1A	R/W	DAC Decimation (DIO Out)	A4D4, DIO, 25M, Servo, SD16
0x1B	R/W	Packet Header DAC	Servo, A4D4, 25M, DIO, SD16

0x1C	R/W	Packet Header A/D	All
0x1D	R/W	Software Alert	All
0x1E	R/W	Alert Control Register	All
0x1F	R/W	Alert enables	All
0x20-2F	W	A/D Gain Error Coefficients	SD (16 channels), A4D4 (4 channels), 25M (2 channels) Servo(12 channels) 10M (8 channels), 2M (12 channels) SD16 (16 channels)
0x30-3F	W	A/D Offset Error Coefficients	SD (16 channels), A4D4 (4 channels), 25M (2 channels), Servo(12 channels), 10M (8 channels), 2M (12 channels) SD16 (16 channels)
0x40-4F	W	DAC Gain Error Coefficients	A4D4 (4 channels), 25M (2 channels), Servo(12 ch) SD16 (16 channels)
0x50-5F	W	DAC Offset Error Coefficients	A4D4 (4 channels), 25M (2 channels), Servo(12 ch) SD16 (16 channels)
0x60	W	A/D Device 0 Control Register DAC0 SPI Register (write only)	SDF SD16
0x61	W	A/D Device 1 Control Register DAC1 SPI Register (write only)	SDF SD16
0x62	W	A/D Device 2 Control Register	SDF
0x63	W	A/D Device 3 Control Register	SDF
0x70	W	A/D Gain 0	A4D4, 25M, Servo, 10M, SD16
0x71	W	A/D Gain 1	A4D4, 25M, Servo, 10M, SD16
0x72	W	A/D Gain 2	A4D4, Servo, 10M, SD16
0x73	W	A/D Gain 3	A4D4, Servo, 10M, SD16
0x74	W	A/D Gain 4	Servo, 10M, SD16
0x75	W	A/D Gain 5	Servo, 10M, SD16
0x76	W	A/D Gain 6	Servo, 10M, SD16
0x77	W	A/D Gain 7	Servo, 10M, SD16
0x78	W	A/D Gain 8	Servo, SD16
0x79	W	A/D Gain 9	Servo, SD16
0x7A	W	A/D Gain 10	Servo, SD16
0x7B	W	A/D Gain 11	Servo, SD16

0x7C	R/W	A/D Gain 12	SD16
0x7D	R/W	A/D Gain 13	SD16
0x7E	R/W	A/D Gain 14	SD16
0x7F	R/W	A/D Gain 15	SD16

Table 8. X3 FrameWork Logic Memory Map

Registers in the X3-SD16 FrameWork Logic

USER FPGA Logic Version – 0x00 (read)
This register returns the version number for the USER FPGA. This register allows verification that the proper logic is actually loaded into the FPGA. The values are hard coded in the logic to designate: which XMC board type, which hardware revision, and which hardware variant is appropriate for this logic.

Bits	Function
31..24	X3 module type (see type table in this document)
23..20	X3 hardware revision
19..16	X3 hardware variant
15..0	USER logic version

Table 9. ii_rev_code Register

Control Register 0 – 0x01 (write)
This register provides several controls for reseting the FPGA, allowing the logic to run, enabling test modes and software triggering.

Bits	Function	Module Notes
0	Software reset of User FPGA. ‘1’ = reset	
1	ADC Run. Allows data capture. ‘1’ = run (0= off- default)	
3..2	-	
4	Link Loopback; '0' = normal operation, '1' = loopback to PCI	
5	A/D test counter, ‘1’ = enabled (DIO In)	
6	DIO Out, ‘1’ = enabled	DIO only
7	-	
8	ADC MQ reset, '1' = reset	
9	DAC MQ reset, '1' = reset	
11..10	SW triggers 1..0	0 for A/D (DIO In) 1 for DAC (DIO Out)
15..12	-	
16	DAC Run, Allows data capture. ‘1’ = run (0= off- default) (DIO Out)	N/A for SD, SDF

17	DAC test mode (DIO Out)	DIO
(Others)	Not used	

Table 10. X3 Control Register 0

Test Register – 0x02
This register is reserved for test functions. Customers will not normally use this register.

Bits	Function
0	-
1	Force temp failure. '0' = normal (default).
31..2	Not used

Table 11. X3 Control Register 0

Temperature - 0x3 R
This register gives the current temperature when read. Scaling from the temperature sensor is a 2's complement number that is sign extended to 16-bits from the 12-bit value given in this table. Note that 0 degrees is 0 output and scale factor is 0.0625 C/bit.

Temperature(C) = reading * 0.0625

TEMPERATURE (°C)	DIGITAL OUTPUT (BINARY)	HEX
128	0111 1111 1111	7FF
127.9375	0111 1111 1111	7FF
100	0110 0100 0000	640
80	0101 0000 0000	500
75	0100 1011 0000	4B0
50	0011 0010 0000	320
25	0001 1001 0000	190
0.25	0000 0000 0100	004
0	0000 0000 0000	000
−0.25	1111 1111 1100	FFC
−25	1110 0111 0000	E70
−55	1100 1001 0000	C90

Table 5. Temperature Data Format

Bits	Function
15..0	Temperature (read only)
(Others)	Not used

Table 12. X3 Temperature Sensor Reading

Temperature Warning – 0x4 R/W
This register sets the temperature warning level for writes and clears temperature warning when read.

The warning level should be provided as a temperature reading set point (see table above).

Bits	Function
15..0	Temperature Warning Level (W), Clear on read (Default = X"0460" = 70 C)
(Others)	Not used

Table 13. X3 Temperature Warning Register

Temperature Failure – 0x5 R/W
This register sets the temperature failure level for writes and clears temperature failure when read.

The failure level should be provided as a temperature reading set point (see table above).

Bits	Function
15..0	Temperature Failure Level (W), Clear on read (Default = X"0460" = 70 C)
(Others)	Not used

Table 14. X3 Temperature Failure Register

PLL Clock Controls – 0x8 R/W
This register sets PLL CLKA input and REF clock inputs. An external clock mux is used to switch these clocks into the PLL.

Bits	Function
0	PLL CLKA select 0= EXT_CLK on front panel (default) 1= PXIE_DSTARA on P16
1	PLL REF CLK select 0= oscillator (default) SD,SDF = 24.576MHz 1= PXIE_CLK100 on P16
2	SDA serial data bit (Write only)
3	SCK bit for serial clock (Write only)
4	Readback for I2C data pin (Read only)
5	Readback for I2C clock pin(Read only)
6	VCXO Output Enable (1 = enabled, '0' = default)
(Others)	Not used

Table 15. X3 PLL Clock Controls

PLL Clock Divider and Distribution Control Interface – 0x9 R/W (A4D4, Servo only)

This register is the interface to an AD9510 used for sample clock division and distribution. The control registers are identical to the PLL used for sample clock generation at 0xA. This device however does NOT use the PLL functionality, only the clock division and output controls. The main purpose of this device is to divide the PLL output down so that the clock rate is useful for slower speed devices.

To configure this device, set the AD9510 to PLL off, CLK1 input, CLK2 disabled. Output 0 should be LVPECL. The other outputs are defined LVCMOS for X3-SD16. Outputs 1,2,3 are not used.

PLL Interface – 0x0A R/W
This register is the data interface to the AD9510 PLL used for sample clock generation.

Writes: write to this address A11..0 with a 24 bit word as is shown. Bit 23 is set to '0' for writes.

Reads: Write to the address A11..0 that is to be read. Set bit 23 to '1' indicating a read cycle. Then read from this address. A byte of data from the address A11..0 is returned

Bits	Function
7..0	Data byte (don't care when bit 15 = '1')
19..8	Address
22..20	-
23	Read/write access ('0' = write)
30..24	-
31	PLL status output (read only)

Table 16. X3 PLL SPI Port Interface

A/D channel enables – 0x0B
This register enables the channels for data flow. The X3-SD16 has 16 A/D channels, each has an enable bit that tells the logic whether that channel is active. Inactive channels are powered down. Active channels are part of the data stream.

Bits	Function
15..0	Enable channel A/D
31..16	unused

Table 17. X3-SD16 A/D channel Enables

DAC channel enables – 0x0C
This register enables the channels for data flow. For the DIO module this specifies the output unstack mode.

Bits	Function
15..0	Enable channel DAC
31..16	unused

Table 18. X3 DAC Channel Enables

Front Panel DIO output enables – 0x0D (25M, Servo, A4D4, SD16)
This register enables the Front Panel DIO outputs. One enable per byte.

Bits	Function
0	Byte 0 output enable (bits 7..0)
31..1	unused

Table 19. X3 Front Panel DIO Enables

ZBT SRAM Test Interface
The ZBT SRAM interface component is connected to the command channel for test purposes. The underlying logic component supports high speed SRAM use, but the command channel does not. Therefore these registers are used only for test.

To test the ZBT SRAM, write an address to the address register then read or write to that address in the ZBT SRAM.

ZBT SRAM Address – 0x0E

Bits	Function
19..0	Zbt address
31..20	unused

Table 20. X3 ZBT SRAM Address Register

ZBT SRAM Data – 0x0F

Bits	Function
31..0	Data

Table 21. X3 ZBT SRAM Data Register

DAC Control Register – 0x11 (Servo, 25M, A4D4 only)
This register controls the DAC FIFO reset and underflow reset.

Bits	Function	
2..0	DAC Mode "000" = 256 clocks per LR (default) others = 128 clocks per LR	SD16 only

3	-	
4	DAC FIFO reset, 1 = reset (0 = default)	
5	DAC FIFO underflow reset, 1= reset (0 = default)	
6	DAC outputs are loopback of A/D values	Servo only
(Others)	Not used	

Table 22. X3 DAC Control Register

ADC Control Register – 0x12

This register controls various module-specific controls.

Bits	Function	Module
2..0	ADC Mode – selects the sample mode of the A/D. SD: See PCM4204 data sheet. SD16: Mode ClkDiv => fs_clk/LR High Speed 00 x 256 High Res 01 x 512 Low Pwr 10 0 256 Low Pwr 10 1 512 Low Pwr 11 0 512 Low Pwr 11 1 2560 These bits are ADC_MODE <= clkdiv & mode(1 downto);	SD, SD16
3	-	
4	ADC FIFO reset, 1 = reset (0 = default). Also does hardware reset.	
5	ADC overflow reset, 1 = reset (0 = default)	
7..6	-	
11..8	Amplifier high pass disables (1 = disabled, 0 = default)	SD
12	ADC Mode : filtered or unfiltered. Modifies the behavior of the data interface to accept unfiltered data. (0= filtered, default).	SDF
13	A/D Sync : provides software synchronization to all A/D devices so that they sample simultaneously. (1= sync , 0= default)	SDF
14	-	
15	ADC HW Test (0= off, default)	SD16
31..14	Unused	

Table 23. X3 A/D Control Register

P16 DIO Data Low Register – 0x13

This register is the digital IO 31..0 on P16. Writing to this register updates the output bits, as enabled in the DIO control register. A read from this register gives the current state of all DIO bits.

Bits	Function
31..0	DIO bits 31..0

Table 24. X3 Digital IO Data Register

P16 DIO Control – 0x14

This register is the output enable control and sample enable control for the digital IO on P16. Writing to this register enables bytes as inputs or outputs. Bit 4 controls the sample enable selection.

Bits	Function
0	DIO bits 7..0 output enable. '0' = input, default
1	DIO bits 15..8 output enable. '0' = input, default
2	DIO bits 23..16 output enable. '0' = input, default
3	DIO bits 31..24 output enable. '0' = input, default
4	Sample DIO inputs when DIO_EXT_CLK is true, otherwise always sample (0=sample always, default)
Others	-

Table 25. X3 DIO bits 31..0 Control Register

P16 DIO Data High Register – 0x16

This register is the digital IO 43..32 on P16. Writing to this register updates the output bits, as enabled in the DIO control register. A read from this register gives the current state of all DIO bits.

Bits	Function
11..0	DIO bits 43..32
Others	-

Table 26. X3 DIO bits 43..32 Control Register

Front Panel DIO Data Register – 0x15 (25M, Servo, A4D4, SD16)

This register is the digital IO on the front panel. Writing to this register updates the output bits and sets the direction of the port. A read from this register gives the current state of all Front Panel DIO bits.

Bits	Function	Module
	Front panel DIO bits	Servo 11..0 25M 15..0 A4D4 11..0 2M 11..0 SD16 5..0
31	Output enable, 0= input(default)	

Table 27. X3 Front panel DIO Control Register

DAC Packet Headers – 0x1B (25M, Servo, A4D4)
This sets the PDN for the DAC packets.

Bits	Function
23..0	-
31..24	Peripheral ID number

Table 28. X3 DAC Packet Header

A/D Packet Headers – 0x1C
This sets the packet size for the A/D packets. For the DIO module this is the input stream.

Bits	Function
23..0	Number of dwords in packet, including 2 dword header
31..24	Peripheral ID number

Table 29. X3 A/D Packet Header

Gain and Offset Error Coefficient Registers
These registers store the correction coefficients that are used by the FPGA to correct gain and offset in real time. There is a gain coefficient and offset coefficient for each ADC. When the board is calibrated, these values are stored in the non-volatile Calibration ROM (serial Flash) which is controlled by the PCI FPGA. Each time the board is powered up, the PCI host reads the values from the calibration ROM and writes them into these registers in the USER FPGA.

Module	A/D Channels	DAC Channels
SD	16	-
SDF	4	-
A4D4	4	4
DIO	-	-
25M	2	2
Servo	12	12
10M	8	-
2M	12	-
SD16	16	16

A/D Gain Error Correction – 0x20 + channel number

Bits	Function
17..0	Gain correction: 0x10000 = 1
Others	unused

Table 30. X3 A/D Gain Correction Registers

A/D Offset Error Correction – 0x30 + channel number

Bits	Function
15..0	Offset correction: 0 = 0

Others	unused

Table 31. X3 A/D Offset Correction Registers

DAC Gain Error Correction – 0x40 + channel number

Bits	Function
17..0	Gain correction: 0x10000 = 1
Others	unused

Table 32. X3 DAC Gain Correction Registers

DAC Offset Error Correction – 0x50 + channel number

Bits	Function
15..0	Offset correction: 0 = 0
Others	unused

Table 33. X3 DAC Offset Correction Registers

A/D Analog Gain Control – 0x70..0x7B (Servo, 10M, A4D4)

A4D4, Servo, 25M, 10M , SD16: (A/D channel 0 = 0x70, ...)

This sets gain for and analog inputs. *This is the analog gain, not the gain error correction!* Writes to this register change the analog gain.

Bits	Function
1..0	Analog gain “00” = 1x gain “01” = 2x gain “10” = 5x gain “11” = 10x gain
31..2	-

Table 34. X3 Analog Gain Controls

Trigger Controls

Two trigger modes are supported for A/D data acquisition : unframed and framed. In unframed mode, the trigger output is true whenever the selected trigger source is true. In framed mode, the trigger output is true after a rising edge on the selected trigger source until the frame count number of sample clocks are counted. The frame will re-trigger only on another rising edge on the trigger source.

A point is counted on each rising edge of the sample clock input when sample enable is true. The trigger mode is selected to be either framed or unframed, software or external trigger. The trigger sources are external or software. The external trigger must be enabled to use it, however the software trigger is OR’d with the external trigger to allow it to be used anytime.

The samples can also be decimated by ratios up to 1/4095.

A/D Trigger Controls – 0x17; DAC Trigger Controls - 0x18
These registers set the trigger mode and enable the external trigger input. The external trigger functions are

Trigger 0 = A/D trigger

Trigger 1 = DAC trigger

Bits	Function
23..0	Frame count
26..24	Not used
27	External trigger select ('0' = JP1 fornt panel (default), '1' = P16 on DIO 34,) J4 on 80172 adapter)
28	DAC trigger uses A/D trigger (trigger0) (SD16)
29	-
30	Trigger mode: 0 = unframed, 1 = framed
31	External trigger enable '0' = disabled (default)

Table 35. X3 Trigger Controls

A/D Decimation – 0x19; DAC Decimation - 0x1A
This register sets the decimation rate. The minimum allowable value for this parameter is 1.

Bits	Function
11..0	Decimation factor. Sets of the A/D data for the enabled channels are kept for 1:N as set by this number.
31..12	-

Table 36. X3 DAC Decimation

Alerts

Up to 32 alerts may be monitored that indicate when important events happened in the data collection process. Each alert is individually enabled by setting its corresponding bit in the alert enable register. When an enabled alert occurs, a data packet is generated with the following format. A timestamp for the alert packet is created using the sample clock so that there is a one-to-one correspondence between the samples and alerts. The alert FIFO can be cleared by setting the FIFO reset to '1'.

Alert Control Register – 0x1E
This register provides controls associated with the alert mechanism.

Bits	Function
0	Timestamp run; '0' = disabled (default)
1	Alert FIFO reset; '1' = reset, '0' = normal (default)
23..2	-
31..24	Alert PDN (default = X"FF")

Table 37. X3 Alert Controls

Alert Log Enables – 0x1F
This register enables the alerts 31..0. A '1' in the corresponding bit enables the alert.

Alert	Description	Module Specific
0	Timestamp rollover	
1	Software Alert	
2	Over Temperature Alarm/ Sensor Failure	
3	Temperature Warning	
4	PLL Lost	
7..5	-	
8	ADC Queue Overflow	
9	ADC Trigger	
15..10		
16	ADC Overrange	Alert word gives the channel numbers having overranges
23..17	-	
24	DAC Underflow	
25	DAC Trigger	
31..26		

Table 38. X3 Alert Enables

Logic Clocks

In the X3-SD16 design, there are clocks for analog data sampling, a system clock and a command channel clock for communications. The FrameWork Logic is designed so that all hardware interface components and devices that use other clocks transition to the system clock domain. This makes the design process simpler since the majority of the logic is on the system clock domain and is synchronous. This makes the design more reliable and easier to modify.

There are several clocks available to the designer in the logic that are intended for different functions as shown in the following table.

The system clock (sys_clk_in) is a 67 MHz clock provided by the PCI Express bridge. This clock is a fixed rate and is present whenever the PCI Express interface is working.

The sample clock (fs_clk_p/n) is a phase-aligned copy of the A/D master clock. This is a LVPECL differential clock input that can range from 0 to 140 MHz. The sample clock is either an external clock or is generated in the PLL on the card. Sample clock is limited to 144 kHz when the A/D are used or 192 kHz when the DACs are only used.

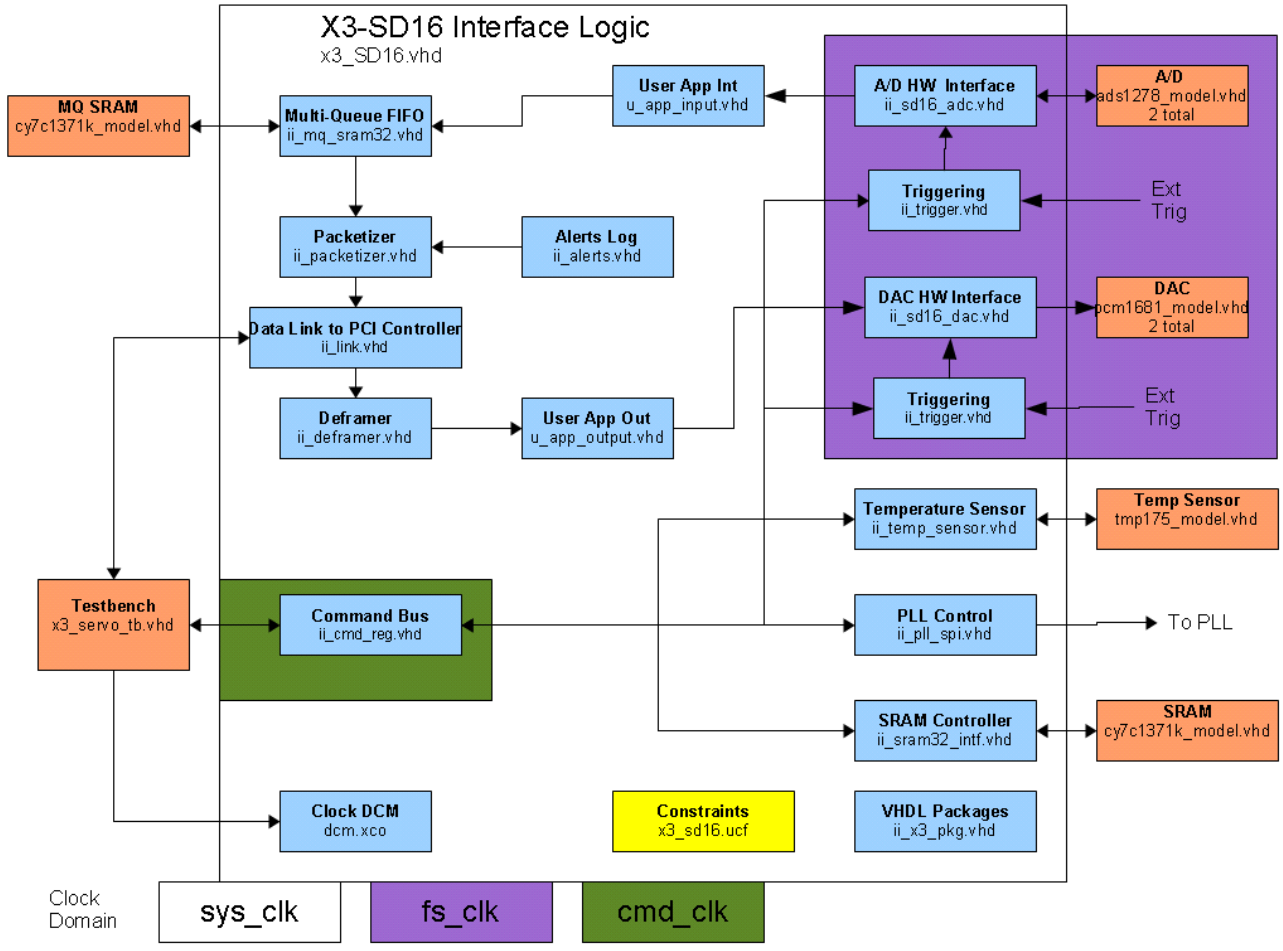
The command channel clock is used for the command channel only. The serial data and frame signals used by the command channel are synchronous to this clock. The clock is 16 MHz.

Clock into FPGA	*Function*	*Frequency*	Source
sys_clk_in	System clock	66.7 MHz	PCI Express bridge
fs_clk	Sample clock	0-140 MHz	External input or PLL
cmd_clkx	Command bus clock received from PCI controller FPGA	16 MHz max, depends on host PCI bus	PCI Express interface FPGA

Table 39. X3 Logic Clocks

It is possible to use the DCMs in the FPGA to generate many other clocks from the sources provided by programming them for their multiplier and divisor factors. The advantage of using the DCM over a logic-based division is that phase relationships are preserved so that the logic may still be synchronous. This tends to simplify the logic design by reducing the number of random phase clock domain transitions that must be managed.

The following diagram shows the clock domains in the X3-SD16 FrameWork Logic. The majority of the device runs on the system clock, which is a 67 MHz. Application logic programming built on top of the FrameWork logic is usually done in the system clock domain, allowing a simpler design process.



The following diagram shows the clock domains what logic resides on each domain.

Figure 32. X3-SD16 Clock Domains

A/D Interface Component

The hardware interface component providing the A/D interface function is ii_sd16_adc. This component provides the interface to the A/D devices, two Texas Instruments ADS1278, each of which has 8 channels of 24-bit 144 KSPS A/D channels. There are two devices to provide 16 simultaneous channels. The data interface to each device is a serial data port, plus several control signals for configuration and status.

The interface component also performs error correction and over-range detection. A FIFO interface to the system logic transitions from fs_clk domain to system clock domain and provides 1K sample buffering.

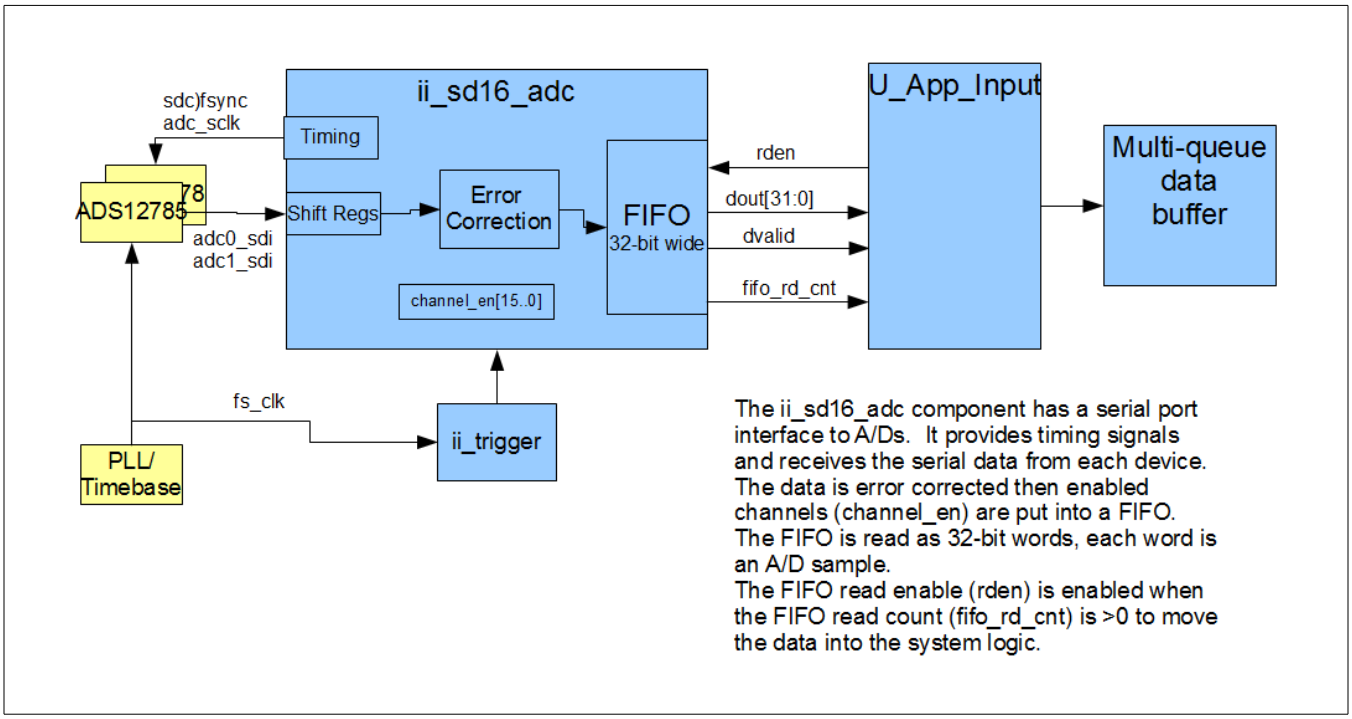


Figure 33. X3-SD16 A/D Interface Component

ADS1278 Device Interface

The A/D device requires a master clock that is a multiple of the sample rate. The multiple is a function of the sampling mode of the device and the desired sample rate.

Mode	Max fclk (MHz)	Mode	adc_clkdiv	SCLK	Max Sample Rate
High Speed	37	“00”	1	256	144531
High Resolution	27	“01”	1	512	52734
Low Power	13.5	“10”	0	256	52734
Low Power	27	“10”	1	512	52734
Low Speed	5.4	“11”	0	512	10547
Low Speed	27	“11”	1	2560	10547

Table 40. A/D Clock and Sample Rates for Sample Modes

The ii_sd16_adc generates the serial clock (adc_sclk) and frame (adc_fsync) signals based on the mode and adc_clkdiv settings. These signals are synchronized to the to either an internal sync or to an external sync when multiple cards are used. The sync ensures that the A/D devices sample in synchronization by controlling the frame sync signal.

The ii_sd16_adc component uses the frame-sync TDM serial interface to the ADS1278. The FPGA-generated serial clock and frame are used to receive the data on the serial data inputs (adc0_sdi, adc1_sdi) as shown the following diagram.

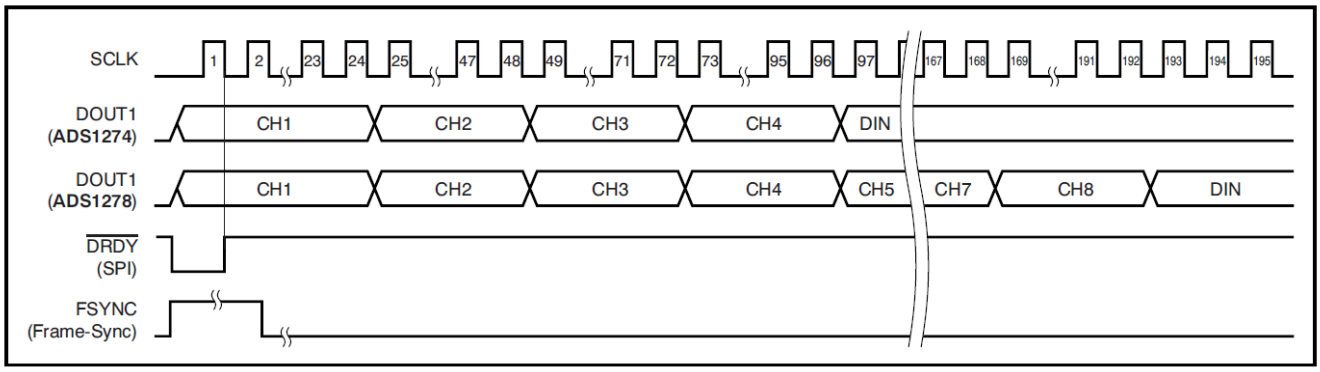


Figure 34. ADS1278 Serial Port Timing in TDM Mode

The data is received into the FPGA and captured in shift registers. A data word is taken from each shift register every 24 bits.

Error Correction

The A/D channels are corrected in the logic for bias and gain errors due to analog electronics. This is used for calibrating the A/D channels. Calibration coefficients are loaded into the system FLASH ROM during factory test and are then used for error correction. Each A/D channel has an offset and gain register mapped to system memory. The software is expected to read the coefficients from the ROM and write them to these registers as part of system initialization.

The error correction is a first-order error correction implemented in the logic as

$$y = m(x + b)$$, where m = gain correction, b = offset correction and x = input A/D sample

The offset coefficient is a 16-bit number, which is left shifted (multiplied by 32) before it is added to the sample. The gain coefficient is 16-bit.

Gain Coef = 0xC000 + Gain/2; when Gain = 0x8000, gain is 1.

Offset Coef = Offset * 32

The gain error compensation is ~ +/50% of the input range. All calculations use saturating math.

FIFO Data Buffer

The output from the component is a FIFO that buffers the A/D samples. The number of enabled channels determines the data format. If all channels are enabled, the samples are sequentially stored to the FIFO AD0..AD15. For a single channel, data is AD0(t), AD0(t+1). If a channel is turned off, such as channel 1, the ordering is AD0, AD2, AD3, AD4..AD15, AD0(t+1).

Data from the A/D FIFO is output as a 2's complement, 32bit. Negative numbers are sign-extended from the 24-bit sample data. Full scale is 0x007FFFFF, negative full scale is 0xFF80000.

		Bits
Data Port	Clock Domain	31..0
FIFO	sys_clk	A/D sample channel n

Table 41. X3-SD16 A/D Component Output Data Format

Where to Grab the A/D Data

The User Application Input component, u_app_input, is a good place to get at the A/D data stream. A/D data flows through this component after it is read from the system side of the FIFO. The FIFO performs the clock domain transition for the data from the sample clock to the system clock in this case. You can pace the data from the FIFO by controlling RDEN based on the number of points in the FIFO given by FIFO_RD_CNT. Remember that the FIFO count is for 32-bit words, not 16-bit samples. The RDEN does not allow data to flow when it is false, so the local FIFO buffers the data. Be careful not to overflow the FIFO when using FIFO_RD_CNT or points will be lost.

You should place your algorithm in the u_app_input component. It is nothing more than a register as delivered, so you can just modify it to add new signal processing to the data stream. Be sure to consider data pacing from the A/D FIFO and to the MQ data buffer.

DAC Interface Component

The hardware interface component for the DAC interface is ii_sd16_dac. This component provides the interface to the A/D devices, two Texas Instruments PCM1681, each of which has 8 simultaneous 24-bit, 192 KSPS DAC outputs. The data interface is a serial port to each of the two DAC devices.

The interface component also performs error correction underflow detection. A FIFO interface to the system logic transitions from system clock domain to fs_clk domain and provides 1K sample buffering.

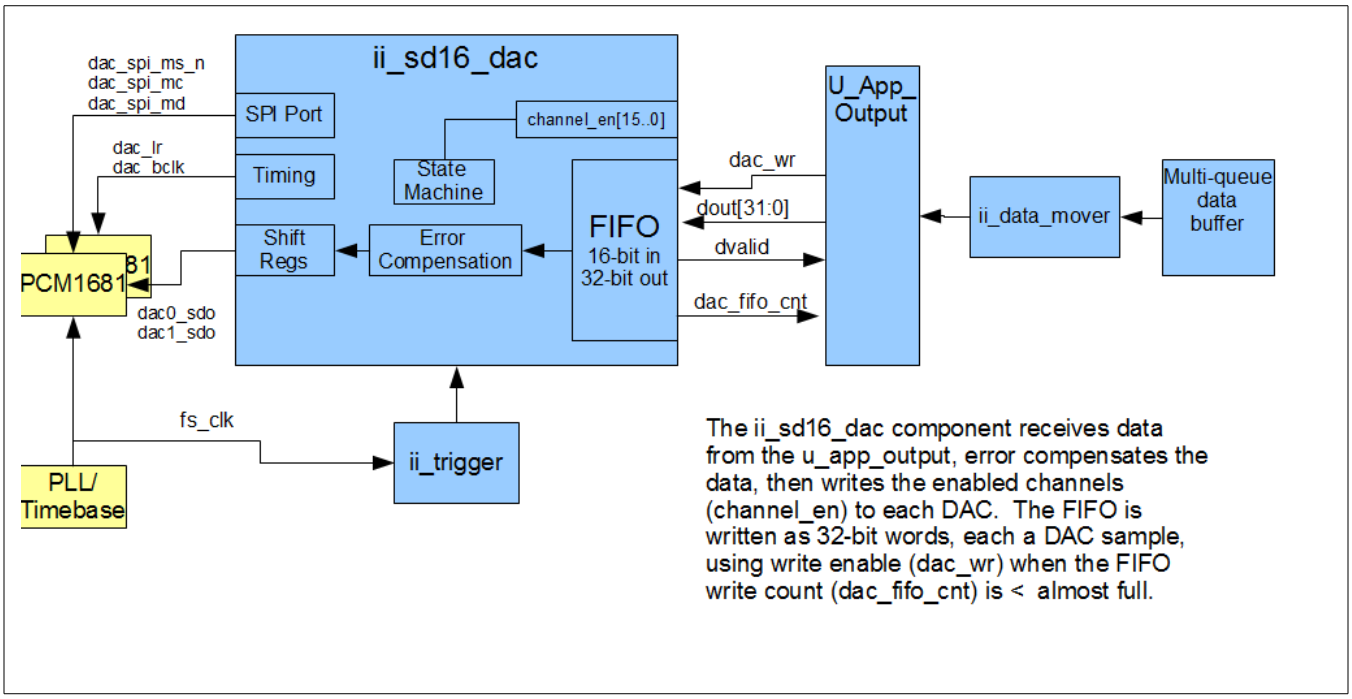


Table 42. X3-SD16 DAC Interface Component

PCM1681 Device Interface

The D/A devices require a master clock that is a multiple of the sample rate. The multiple is a function of the sampling mode of the device and the desired sample rate. The two supported clock rates are 128fs and 256fs modes.

Mode	Max fclk (MHz)	Mode	Clock Multiple	Max Sample Rate(ksps)
256fs	40	"00"	256	156.25
128fs	24.576	"01"	128	192

Table 43. D/A Clock and Sample Rates for Sample Modes

The ii_sd16_dac generates the serial bit clock (dac_bclk) and left-right frame (dac_lr) signals based on the mode setting. These signals are synchronized to the to either an internal sync or to an external sync when multiple cards are used. The sync ensures that the D/A devices sample in synchronization by controlling the frame sync signal.

The ii_sd16_dac component uses the left-justified audio format serial interface to the PCM1681. The FPGA-generated serial bit clock and left-right frame are used to transmit data on the serial data outputs (dac0_sdo, dac1_sdo) as shown the following diagram.

Left-Justified Data Format; L-Channel = HIGH, R-Channel = LOW (default)

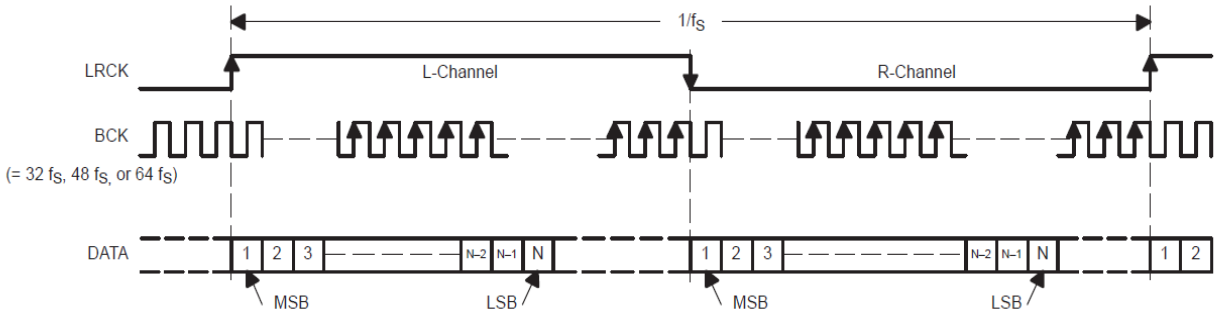


Figure 35. PCM1681 Serial Port Timing

Error Correction
The DAC channels are corrected in the logic for bias and gain errors due to analog electronics. This is used for calibrating the DAC channels. Calibration coefficients are loaded into the system FLASH ROM during factory test and are then used for error correction. Each D/A channel has an offset and gain register mapped to system memory. The software is expected to read the coefficients from the ROM and write them to these registers as part of system initialization.

The error correction is a first-order error correction implemented in the logic as

y = mx + b, where m = gain correction, b = offset correction and x = output D/A sample

The offset coefficient is a 16-bit number, which is left shifted (multiplied by 32) before it is added to the sample. The gain coefficient is 16-bit.

Gain Coef = 0xC000 + Gain/2; when Gain = 0x8000, gain is 1.

Offset Coef = Offset * 32

The gain error compensation is ~ +/50% of the input range. All calculations use saturating math.

FIFO Data Buffer
The output from the component is a FIFO that buffers the D/A samples. The number of enabled channels determines the data format. If all channels are enabled, the samples are sequentially stored to the FIFO DA0..DA15. For a single channel, data is DA0(t), DA0(t+1). If a channel is turned off, such as channel 1, the ordering is DA0, DA2, DA3, .DA15, DA0(t+1).

Data to the D/A FIFO must be 2's complement, 32bit. Negative numbers are sign-extended from the 24-bit sample data. Full scale is 0x007FFFFF, negative full scale is 0xFF80000.

		Bits
Data Port	Clock Domain	31..0
FIFO	sys_clk	DAC channel n

Table 44. X3-SD16 DAC Component Input Data

Where to Access the DAC Data Stream

The User Application Output component, u_app_output, is a good place to get at the DAC data stream. DAC data flows through this component after it is read from the multi-queue data buffer. You can pace the data flow from the multi-queue to the DAC interface input FIFO by controlling wren based on the number of points in the FIFO given by DAC_FIFO_CNT. Be careful not to overflow the FIFO or points will be lost.

You should place your algorithm in the u_app_output component. It is nothing more than a register as delivered, so you can just modify it to add new signal processing to the data stream. Be sure to consider data pacing from the multi-queue and to the DAC FIFO.

Triggering Component

The ii_trigger component controls the trigger inputs to the A/D and DAC components. There are two independent trigger components in the design – one for A/D and one for DAC. When the trigger is true, this tells the logic that a data point is acquired for A/D, or played out for the DACs. This allows the logic to control the A/D and DAC data flow for application-specific triggering requirements.

The ii_trigger component provides two methods for triggering: framed and unframed. In the framed mode, a programmable number of points are captured for each trigger rising edge. In unframed mode, data is captured as long as trigger is true. The framed mode allows the X3-SD16 to capture a snapshot of data for analysis or playback a waveform of fixed length.

The frame count is loaded as the number of points to be captured up to 2^24 points. A frame is captured or played out on each rising edge of the trigger input. If a frame capture is in progress, the trigger is ignored.

The trigger is either the external sync signal or the internal software controlled run signal, as selected by the trigger control register for each channel. The external trigger may be disabled to prevent false triggering. The software trigger is may always trigger the system even when the hardware trigger is true to allow the application to force a trigger condition.

External Trigger	Use	Notes
0	A/D	This external trigger may be used for DACs if ctl_reg(24)(28) = '1'
1	DAC	

Table 45. X3-SD16 External Triggers

For both the A/D and DACs, the trigger event is always for the full set of enabled channels. Once a trigger is started, all enabled channels will be acquired and played out. No partial sets are allowed.

Adding a Unique Trigger Method

To add a new trigger method to the design, you can either replace the trigger component ii_trigger, or modify it. The trigger signal to the A/D interface ii_sd16_adc must be true whenever a sample is to be collected. For the dac triggering, the trigger signal to ii_sd16_dac allows a point to played to the DACs. The trigger signal is synchronous to fs_clk, so the new signal you make must also be synchronous to fs_clk. This means that your new trigger logic must be on the fs_clk domain.

If you modify ii_trigger, just watch out for the clock domains. It is important to remember that the fs_clk can be quite slow compared to the system clock, so transitions should take this in to account. Signals coming from the system clock must usually be held until the sample clock has seen them. A one clock pulse from the system logic will not be seen since it is 67 MHz and the sample clock is no greater than 20 MHz.

The clocks are also asynchronous to one another, so no phase relationship can be used in your design. The sample clock, particularly from external sources, must be periodic and 50% duty cycle, but is not phase-aligned to the system clock.

Packetizing Component

The packetizing component forms data streams into packets by attaching a header to a bundle of data. The primary use of these packets is to transfer data to the host using the Velocia PCIe controller. Each data packet has a two word header, 32-bits each, preceding the data. The packets are programmable in size and for their other routing information.

The packetizer polls a defined number of data sources and creates packets when the source and destination of the packet are ready for the transaction. Setup information for the number of channels, size of the packets, sources enabled and headers is provided as part of the initialization of the packetizer. Sources and destinations for the packetizer are usually FIFOs in the logic, as in the FrameWork Logic. A read enable, synchronous to the input clock, is provided for each data source and a write enable for each destination. The packetizer pipelines the data and aligns it to the data stream flowing through during the packet assembly process.

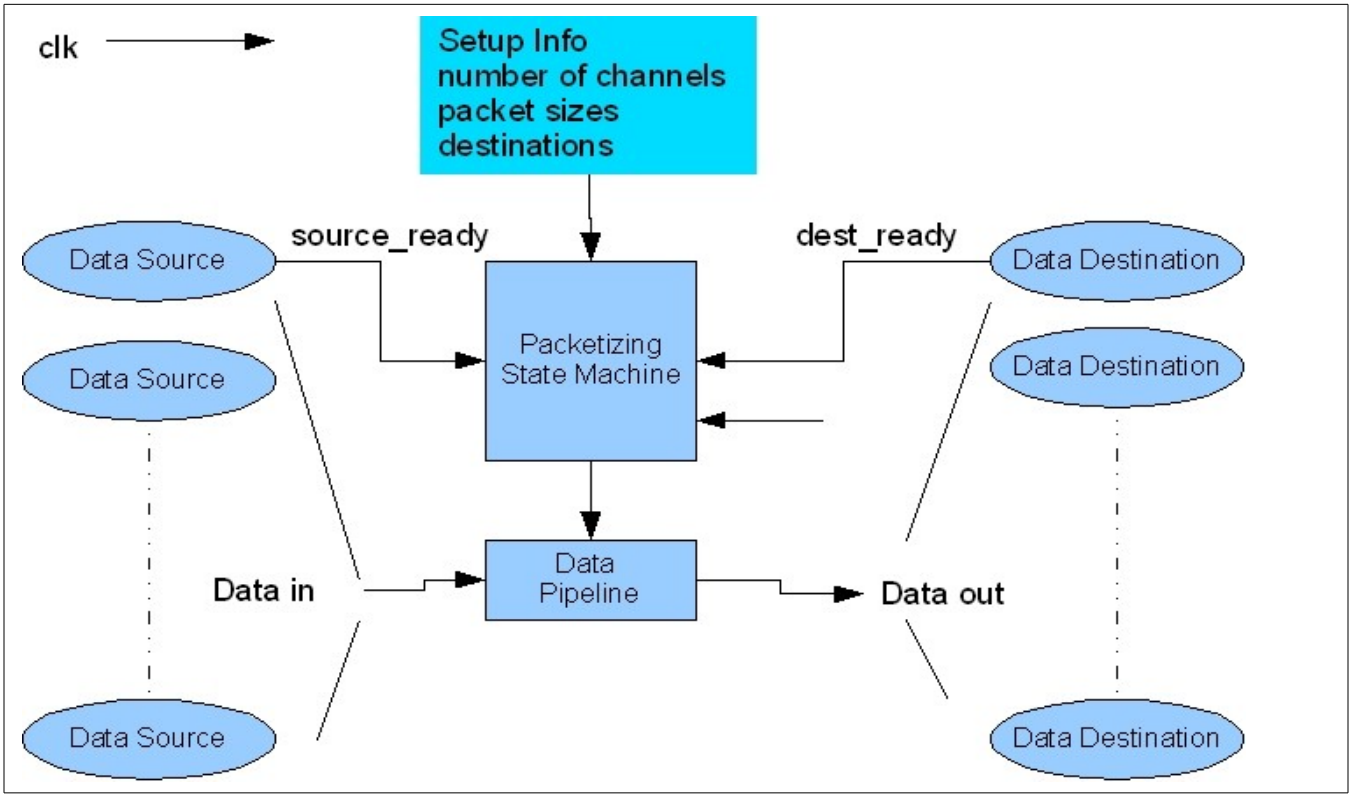


Figure 36. ii_packetizer Block Diagram

During operation, the packetizer scans the number of input channels and in a round robin and creates packets for the channels that are ready. Each channel has its packets built with the header information for that channel and the data payload attached to the header. The packet is transmitted as it is built to the destination, there is no data storage in the packetizing component.

In the FrameWork logic for the X3-SD16, the destination of all packets is the link to the PCI controller. The multi-queue data buffer is used to enqueue the data for the PCI controller for the A/D data stream.

Adding a New Packet Sources to the X3-SD16 FrameWork

To add a data stream to the X3-SD16 Framework, you will need to modify the generics in ii_x3_pkg to for the number of packet sources (num_channels) and the mux_addr_width. This tells the component how many data channels there are to packetize. The FrameWork logic for X3-SD16 has two channels : a/d data and the alerts. It is best just to add more after that so you can use the existing software with minimum changes.

```
Constant num_channels          : positive := 2;  -- the number channels to packetize (a/d , alerts)
constant data_width             : positive := 32; -- width of data path through packetizer mux in
ii_packetizer constant mux_addr_width  : positive := 1;  -- the number of mux address bits needed
for ii_packetizer
```

The mux address width is log2(num_channels) rounded up to the next integer. Do not change the data_width.

For the new channels, you must also specify the following in the top level source code. These signals connect the packetizer to your data source, provide pacing and packetizing information. Here are the changes to the top level to add a new channel:

```
(Don't Change These)
pack_en <= (others => '1'); --ctl_reg(11)(num_pd-1 downto 0); pack_dest <= (others => '0'); -- all go to PCI

(Add a 32-bit data source)
packet_din(0) <= mq_dout(0);   -- ADC data
packet_din(1) <= alert_dout;
packet_din(2) <= new_data;

(Assign a PDN to this channel for the packet header)
pd_addr(0) <= ctl_reg(28)(31 downto 24);  -- A/D channels
pd_addr(1) <= ctl_reg(30)(31 downto 24);  -- alerts
pd_addr(2) <= ctl_reg(x)(31 downto 24);  -- new channel

(Provide the new channel count for packetizing control)
channel_cnt(0) <= mq_queue_cnt(0);
channel_cnt(1) <= X"00" & "00" & alert_fifo_rdcnt;
channel_cnt(2) <= new_channel_cnt;

(Provide the new channel local count for data pacing, i.e.  The local FIFO count for the new
channel) channel_rdcnt(0)<= '0' & mq_fifo_rdcnt(0);
channel_rdcnt(1)<= alert_fifo_rdcnt;
channel_rdcnt(2)<= new_channel_fifo_rdcnt;

-- the packet sizes for each channel
(Assign how many points will be in each packet for the packet header)
channel_packet_size(0) <= ctl_reg(28)(23 downto 0);  -- A/D
channel_packet_size(1) <= X"000024";  -- Alerts (32 alerts plus status and timestamp)
channel_packet_size(2) <= new_packet_size;
```

The packetizer will now draw data from the new packet source paced on data availability, packet size, and local FIFO count. Software will receive packets with a header that has the new PDN, allowing the data stream to be parsed for the new source data.

Deframer Component

The deframer component receives packets from the PCIe Link component and routes them to the peripheral device number (PDN) embedded in the header. For the X3-SD16, the primary destination is the multi-queue data buffer, although additional peripherals can be added for applications. Data is pulled from the link FIFO, is stripped of its header, and written to the destination device. Each destination has a specific packet format that is required that must be followed.

The header for each packet gives the PDN and packet size for the deframer to use in data movement. The deframer has state machine that reads the packet header and then transfer the data payload to its destination. The state machine is idle until a minimum packet size is received (at least 4 words), then pulls off the two header words. The packet size, taken from the first header word, is used to move the data points, as available from the link FIFO to the destinations. This data moves are done by computing the maximum move that can be performed which is the minimum of the number of points in the link FIFO, how much space is available in the multi-queue input FIFO, and the number of points remaining in the packet. This process is repeated until the points are all moved for the packet.

The PDN is defined in the device mapping found in the X3 package (ii_x3_pkg.vhd). The four destinations defined for the X3-SD16 are the data buffers and data buffer control ports. The deframer can be remapped or expanded by adding additional PDNs to its mapping and providing data.

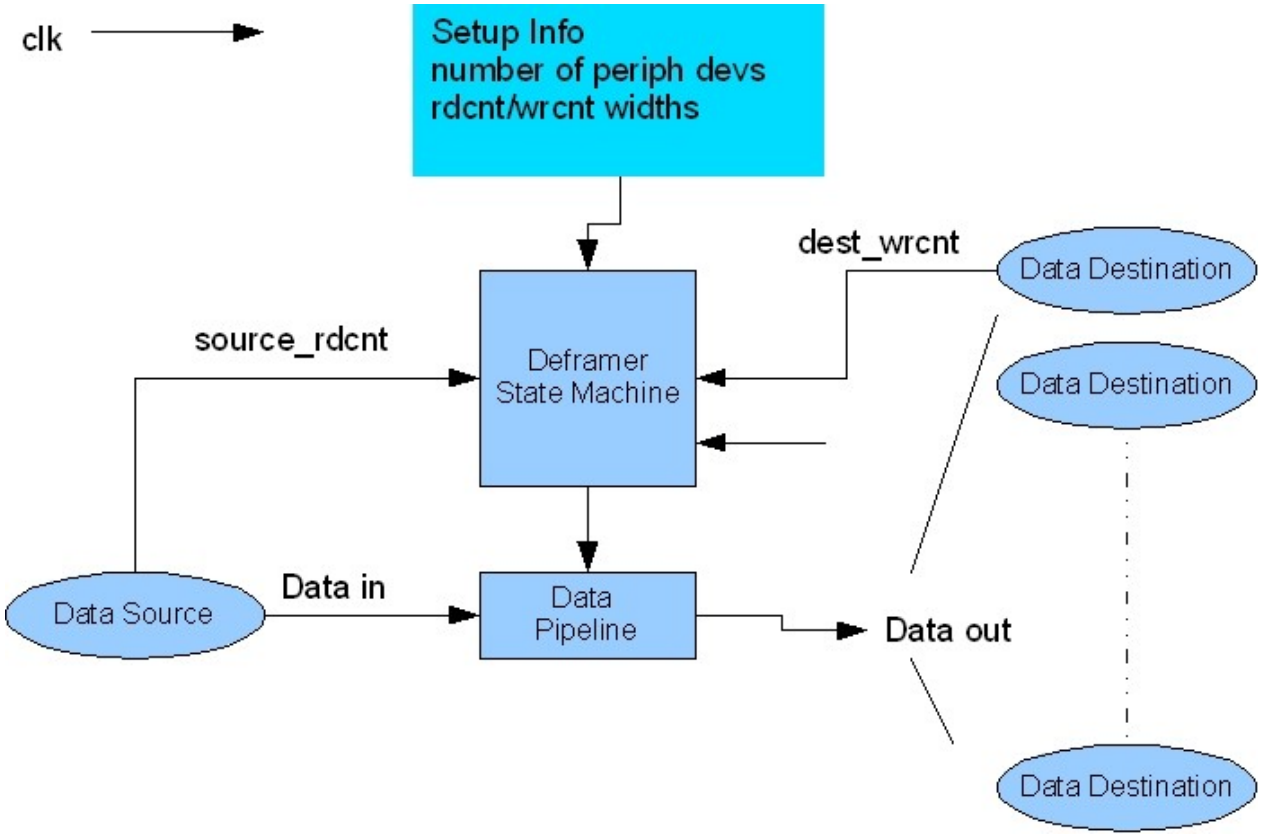


Figure 37. ii_deframer Block Diagram

In the FrameWork logic for the X3-SD16, the source is always the link to the PCI controller. The destinations are the data buffers and data buffer controls FIFOs.

Multi-Queue VFIFO Component

The X3-SD16 FrameWork Logic that implements two data buffers that are each 256Kx32 virtual FIFOs as described in the library components chapter. On the X3-SD16, the A/D data flows into a queue that is stored in the external SRAM buffer memory. This is managed just as a large circular buffer by the logic, providing local data storage for the module. The DAC also has a similar queue.

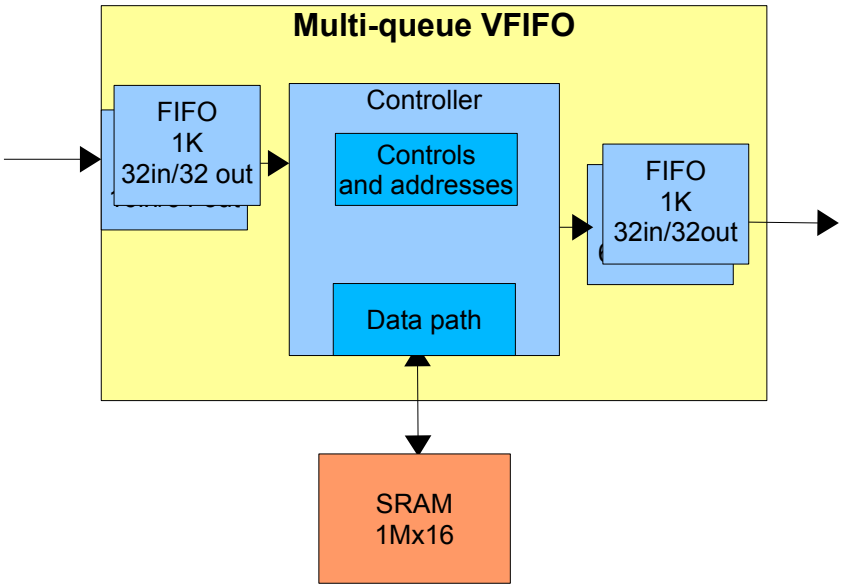


Figure 38. X3-SD16 Multi-Queue VFIFO Component Block Diagram

The multi-queue VFIFO on the X3 modules has one 512Kx32 SRAM device for its buffer memory, running at 83.3 MHz. This gives a raw bandwidth of 333 MB/s to the buffer memory. The rate at which data may be stored to memory is burdened by overhead for queue management and SRAM control, yielding an overall storage rate over 315 MB/s. This allows the X3-SD16 to acquire A/D directly to buffer memory and simultaneously playback to the DACs without data loss for the entire memory.

The queue for the X3-SD16 are sized as shown here.

Queue Number	Data	Queue Boundary Address (19..16)	Size (32-bit words)	Samples
0	A/Ds	X"00"	256K	512K
1	DACs	X"80"	256K	512K

Table 46. X3-SD16 Data Buffer Queue Sizes

Adding More Data Queues

The X3-SD16 has two queues implemented, but the component supports multiple independently managed queues. The component can be modified to support more additional queues by modifying the ii_mq_sram32 component.

See the library chapter describing the changes that must be made to change the number of queues and priority control.

Data Link to PCIe Controller

The FrameWork links the application logic to the PCIe controller via a 32-bit parallel data bus running at 67 MHz. The data link provides transfer rates of up to 264MB/s to the PCIe controller FPGA. Data flow is bidirectional, half duplex. This link is the primary data path for the data to flow to the host over the PCI bus. Pacing across the link is managed by the link.

The ii_link component has FIFOs for data interface to the application logic. Internal to the component is a FIFO in each direction and the link control logic. Data is simply inserted in the link FIFO as space permits. On the X3-SD16 logic, the packetizer performs this data move from the data buffer to the link. Data from the link goes to the deframer component for the DAC output stream.

The data sent to the PCI controller MUST be in the packet format as generated by the packetizer (ii_packetizer.vhd). The PCI controller parses the headers and interacts with the host system hardware and software to provide the automated PCI transfer mechanism used with the X3 XMC cards. Data received is also in the packet format, and is parsed by the deframer for routing to the peripheral device indicated by the packet PDN.

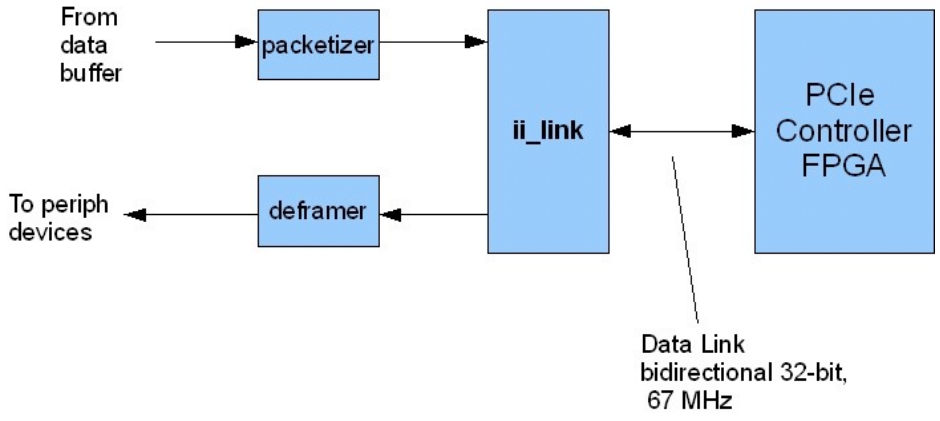


Figure 39. X3-SD16 data link to PCI Controller

Command Channel Component

The Command channel interface is a control interface allowing the host system to access the X3-SD16 application logic registers and status as memory-mapped devices. This control path is separate from the high speed data path implemented over the data link so that various controls and status needed to operate the module are simple to access and control.

The command channel is a simple serial bus interface from the PCI controller device that provides a 32-bit address and 32-bit data word path. A separate PCI BAR is used for the command channel that is presented to the application logic so that the PCI memory space for BAR1 is in the application logic. This means that the host computer, or any PCI device in the system, can access the application logic over the command channel.

The command channel component decodes the addresses sent from the PCIe controller and presents an array of decoded registers for use, and reads back from an address for status. This structure forms the basis of the control and status architecture for the X3-SD16. The memory map presented earlier in this document show the FrameWork logic assignments for the decoded registers and status words.

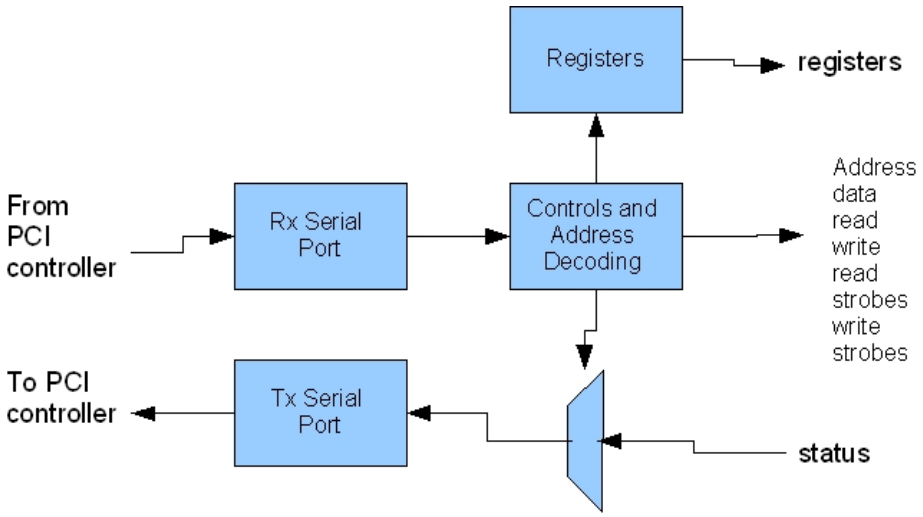


Figure 40. X3-SD16 Command Channel

The FrameWork logic for the X3-SD16 decodes 128 registers and status locations. Many of these are unused, allowing for custom implementations to simply connect to the array without modification. Additional registers or status words can be implemented by using the 128 read and write decode strobes, or by using the address and read/write strobes with external decoding.

PLL Control Component

The PLL control component allows the application logic to configure the external PLL/ clock distribution device used for analog sample rate clocking. The external device, Analog Devices AD9510, is configured over a three wire serial port interface. In the FrameWork logic, this is connected to the command channel interface so that the host computer can configure the PLL for the data acquisition process.

The X3-SD16 software application allows the host PC to configure the PLL reference clock and frequency as part of the module setup. The AD9510 has a large register set that the software configures as part of initialization. Its usually less work to just configure the PLL as part of the module initialization and not from the logic. The X3-SD16 must have this clock running to collect data, and cannot be changed during acquisition.

Custom implementations can control the PLL using the PLL control component from their logic design. Care should be taken to observe the physical limitations on the PLL frequency and lock time when directly controlling the PLL.

The generic logic chapter describes using the PLL interface component in more detail.

XMC P16 Interface

The X3-SD16 P16 interface in the FrameWork Logic implements a 32 bits of digital IO mapped to the command channel. These digital IO bits can be used for control and readback when a host card provides P16 access. The PCI Express carrier card from Innovative provides P16 access using an MDR68 cable for such applications.

More advanced applications can reprogram the digital IO bits for many other purposes since they directly connect to the application logic. The digital IO bits can be used for higher speed applications as LVDS differential pairs. The hardware manual gives the pairing information. The IO standard is LVTTL for single ended signaling, or LVDS for differential signaling.

SRAM Component

For computational memory, the X3-SD16 has one 512Kx32 ZBT SBSRAM (synchronous burst, Zero Bus Turn-around SRAM) memory comprised of one Cypress Technology CY7C1371D-133AXC (or equivalent) chip. This SRAM has a flow-through architecture and is two clocks latent for data. The SBSRAM operates at 83.3 MHz, supporting 333MB/s data rates. This SRAM is not used by the standard logic and is left available for application use. The interface to this SRAM is very basic, providing direct address control, allowing the application to use it in many ways.

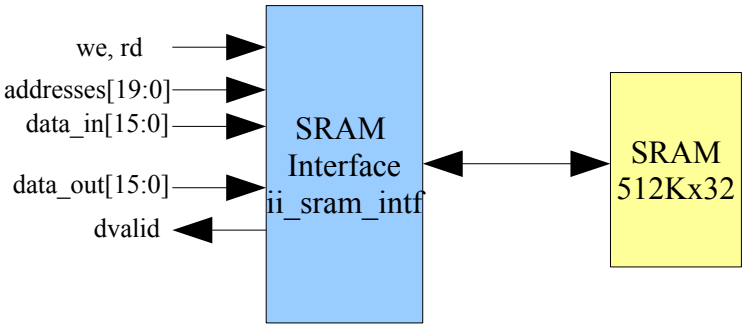


Figure 41. X3-SD16 SRAM Controller Block Diagram

For simulation purposes, a model of the SRAM is provided, *cy7c1371k.vhd*. The model has accurate timings and interface signaling that can be used for development of higher performance SRAM applications.

Adding Functionality to the X3-SD16 FrameWork Logic

The most common modifications to the logic are the addition of signal processing and data analysis to the data stream and the addition of registers/status to the design. These modifications can be done either in the VHDL or MATLAB tools built on top of the FrameWork logic.

Adding registers and status readback to the command channel

The command channel provides a method for the host to access the application logic via the PCI interface for control and status functions. In the FrameWork Logic, the memory map for the X3-SD16 shows the command channel decoded addresses. The command channel is NOT intended for high performance data streams, rather it is for low speed, out of band, asynchronous setup, initialization and status reporting. The data stream over PCIe link should be used for higher performance requirements above 2 MB/s.

Many of the registers and status words are either partially used, or not used at all. If you need a few registers or status words, this is the easiest approach. For instance, register location X"2" is not used in the design so this could be assigned to a new register by adding an assignment in the code

```
new_reg(31 downto 0) <= ctl_reg(2)(31 downto 0);
```

which makes a new 32-bit register, accessible by the host at address (PCI BAR1 +2).

Several of the status registers are also unused in the FrameWork Logic. These can be simply reassigned

```
cmd_status(2) <= new_status;
```

which makes a new 32-bit status, accessible by the host at address (PCI BAR1 + X"C2").

If more registers are needed, or special decoding is required, the command channel interface component provides signals for extending the address space up to 64K registers. Adding a new decoded register may use one of the decoded strobes, 64 total in the logic, or decode the addresses from the command channel. Adding a register is straightforward :

```
process (reset, cmd_clkx, cmd_wt)
begin
if (reset = '1') then
        new_reg <= (others => '0'); -- default value
elsif (rising_edge(cmd_clkx)) then
        if  (cmd_wt =  '1' and cmd_aout(15 downto 0) = X"FFFF")  then
                new_reg <= cmd_dout(31 downto 0);
        end if;
end if;
end process;
```

makes a 32-bit register at address X"FFFF".

Software Scripts for Interacting with Command Channel During Development

The command channel can be accessed by the host using the development software application provided by Innovative. This software provides a scripting feature that allows the developer to access command channel with out complex software using a text script during development. The script may be played from the application software and used to write to command channel mapped registers or status. This is convenient when the logic is being developed and no software is available to control the custom design. In fact, the scripts look similar to the testbench command channel accesses and can serve as a model for the script.

This scripting feature and the software is described fully in the software documentation. The script command set allows store, fetch and wait. The address a is the register number followed by the data and command. A wait command is used for delays.

Command	Syntax	Example
Store	a n l!	0x1 0xFF l! Store X"FF" to register 1
Fetch	a l@	0x5 l@ Fetch from register 5. Displays in console window on example application software.
Wait	n ms	10 ms Wait for 10 ms.

Table 47. X3 Script Commands

In the software examples provided with the X3-SD16, a script can be run before or after the data stream is started, referred to as “Before Stream Start” (BSS) and “After Stream Start” (ASS) scripts. The use of these scripts allows the logic developer to initialize and control the custom logic without having to write custom software for the device control. In many cases, these scripts can then be used in the final application software as well.

Adding Signal Processing to the Data Stream

Here is an example of changing the data flow in the Framework Logic to incorporate an signal processing function into the data stream. The data from the A/D interface component flows into the signal processing block, F(z), in the diagram which is in the u_app_input component. Performing an FFT, down-conversion or filter on the data would be an example of this sort of data flow. The output of F(z) then flows directly into the multi-queue data buffer, where it can continue to the PCI Express controller.

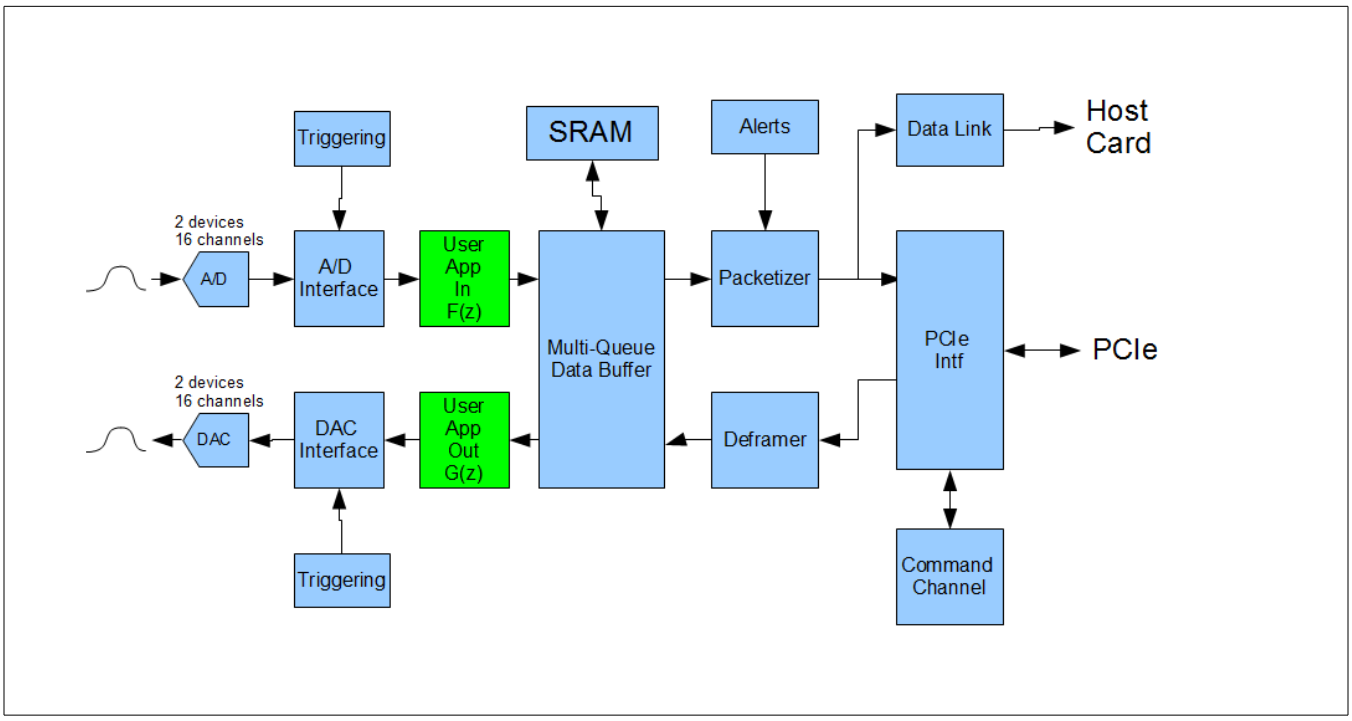


Figure 42. Adding Signal Processing Functions to X3-SD16 A/D Data Flow

If this implementation can be followed, then most of the existing software and logic can be re-used. If the data rate is changed by the signal processing, such as in a down conversion function, then the data packet size is usually reprogrammed to match that rate so that system response is acceptable. Usually, additional control functions are also added but with judicious use of the memory map those are routine.

The data source does not have to be a continuous stream; a single packet may be transmitted over the PCI to the host. The packet may be mixed with other data packets, from multiple sources, and the PCI controller will send the packet to the host, where the host software receive the packet in its data buffers. The host software should set the packet size and Peripheral ID number so that it can extract the data from the other packets in the PCI data buffer.

Design Considerations

Some of the common choices designers must make when using the X3-SD16 FrameWork logic are how to best integrate their signal processing functions, what method should be used for data buffering and to connect to their system. Let's take a look at some of the design problems that typically confront us:

- ***Where to add signal processing functions?***

  Usually the best place to access the A/D data stream is after the A/D interface component (ii_sd16_adc) in the u_app_input component. The A/D interface component has both a FIFO interface that delivers a stream of A/D to the user application. The FIFO output is on the sys_clk domain, so it is easier to use but does require that the stream be split to access individual channel data. The data points in the FIFO output stream have a data valid and channel number, so splitting the stream into channels is straightforward. Since most Xilinx signal processing functions have a ND (new data) enable at their input, the data, data_out( ) is connected to the data input and the dvalid( ) signal qualified for a channel number, is connected to ND. Connect the Xilinx RFD output to the A/D RFD input to pace the data if it is not always ready for data.

  The best to add the DAC signal processing is conversely in the u_app_output component. Samples for the DAC flow through the component enroute to the DAC interface from the data buffer. Data pacing is provided using the ii_sd16_dac FIFO write count and multi-queue data buffer output FIFO count.

- ***How should I buffer the data?***

  The FPGA blockRAMs are the most convenient to use for most applications. They can provide local data buffering for signal processing that is fast and flexible. Xilinx has many components for FIFOs, dual-port RAMs and look-up tables that use blockRAMs or distributed memory and are easy to use. The Spartan3A DSP 1.8M has 84 blockRAMs that are 1Kx16 each. Distributed RAM can be used for small buffers, just be aware that this consumes logic.

  For large data sets larger than 4K should consider using the SRAM or multi-queue. Otherwise you will use a large number of FPGA blockRAMs that may be needed for your signal processing and timing will become more difficult to meet.. The multi-queue provides data buffering like a large FIFO, so it is easy to use but not adaptable to other buffering requirements. Its primary use in the design is data buffering to the host PCIe. The SRAM is not used in the FrameWork Logic and is completely available. Its interface is simple and can be adapted to use the SRAM for any buffering scheme that is required with address control logic.

- Where can I get signal processing functions for logic?

  The Xilinx libraries are very good and offer most of the commonly used functions such as filters and FFTs. These are optimized for implementation in FPGAs and are well documented.

  For more complex signal processing designs, the X3 family has a board support package MATLAB. This BSP allows you to develop directly in MATLAB Simulink and implement the logic on the X3 module. This is a powerful development environment that integrates MATLAB with the Xilinx System Generator tools for the X3 modules. Once the algorithm is tested and working, it can be compiled into the FrameWork Logic and integrated with all the hardware features.

Adding New Components

New components can be developed in VHDL, from Xilinx Coregen blocks, or generated in MATLAB and included in the VHDL design. Each of these techniques involves including the signal processing block into the data flow to the packetizer and integration of the function into the VHDL code. I

The MATLAB technique has the distinct advantage that the MATLAB Simulink environment can be used to generate and observe signals. This allows the DSP engineer to work in a higher level above the bits and bytes of the VHDL and is a powerful way to develop DSP in the logic. Hardware-in -the-loop capabilities in the MATLAN environment can be used to fully test the signal processing design before it is added to the FrameWork Logic VHDL.

Xilinx offers a very good selection of cores that may be used in the X3-SD16 logic including filters, FFTs, correlations, signal generators and basic mathematical functions. These are relatively easy to design with in the VHDL environment and integrated in with the FrameWork logic by making them compatible with system logic. This means that the blocks are

- Components – all functions should be components and integrated into the FrameWork as such. This allows simulation at the component level and physical floorplanning.
- Synchronous – all functions should be on the system clock. This reduces the problems with clock domain transitions and simplifies timing analysis.
- FIFO IO – all data should flow through the component and have a FIFO at the inputs and outputs. This helps to decouple the component from the data flow of other devices.
- Providing data flow controls – an input ready and output ready are useful at the system level for moving the data around the system.
- 2's complement – for compatibility with other components

If these basis guidelines can be followed, the integration effort is reduced since it will match our design philosophy in the Framework Logic.

Terminating Unused IO Signals

When you delete unused components from the FrameWork logic, external devices must have their signals terminated properly. This prevents the external device from driving the FPGA and potentially causing a problem and consuming extra power. Here are some typical device signal terminations.

Device/Signal	*Function*	*Termination*
led	LED	'1' (off)
SBSRAM		
zbt_ce	Chip enable	'0'

Device/Signal	*Function*	*Termination*
zbt_oen	Data bus output enable	'1'
zbt_r_wn	Read/write	'1'
zbt_adv_ldn	Address advance/load	'1'
zbt_sclk	clock	'0'
zbt_bwn[3:0]	Byte writes	“11”
zbt_adr[19:0]	Addresses	All '0'
zbt_io[31:0]	Data bus	All 'Z'
Multi-queue SRAM		
mqram_ce	Chip enable	'0'
mqram_oen	Data bus output enable	'1'
mqram_r_wn	Read/write	'1'
mqram_sclk	clock	'0'
mqram_sa[18:0]	Addresses	All '0'
mqram_dq[31:0]	Data bus	All 'Z'
PLL and Timing Control		
pll_cs_n	PLL chip selects	'1'
pll_sclk	PLL serial clock	'0'
pll_sdio	PLL serial data	'0'
pll_sync	PLL sync control	'0'
pll_ref_sel	PLL reference selection	'0' (100 MHz)
pll_clka_sel	PLL clk A select	'0' (external clock)
DAC Interface Controls		
dac_spi_ms_n[1:0]	DAC SPI chip select, active low	'1'
dac_spi_mc	DAC SPI clock	'0'
dac_spi_md	DAC SPI data	'0'
dac_lr[1:0]	DAC left/right data frame	'1'
dac_bck[1:0]	DAC bit clocks	'0'
dac0_sdo[3:0]	DAC 0 serial data	'0'
dac1_sdo[3:0]	DAC 1 serial data	'0'
ADC Interface and Controls		
adc_sclk[1:0]	A/D serial data clock	“00”
adc_fsync[1:0]	A/D data frame syncs	“00”
adc_sync_n[1:0]	A/D sync control	“00”
adc_format[2:0]	A/D format	“000”
adc_mode[1:0]	A/D mode	“00”

Device/Signal	*Function*	*Termination*
adc_clkdiv	A/D clock divisor control	'0'
adc0_test	A/D 0 test (=1)	'0'
adc0_en	A/D 0 enable	'0'
adc1_test	A/D 1 test (=1)	'0'
adc1_en	A/D 1 enable	'0'
adc0_gain[15:0]	A/D 0 channel gain controls	'0'
adc1_gain[15:0]	A/D 1 channel gain controls	'0'
Command Channel		
cmd_dr	Cmd channel data	'0'
cmd_fsr	Cmd Channel frame	'0'
cmd_clkr	Cmd channel clock	'0'
Data Link		
link_d[31:0]	data to PCIe Controller	'Z'
link_rd_intn	Link read interrupt	'1'
link_wr_intn	Link write interrupt	'1'
P16 Digital IO		
dig_io[43:0]	P16 digital IO	'Z'
Temperature Sensor and Power Supply Enables		
temp_sclk	Temp sensor serial clock	'0'
temp_sda	Temp sensor serial data	'0'
ps_enable	Power supply enable	'1'
ps_enable_n	Power supply enable, active low	'0'

Table 48. X3-SD16 Unused Signal Terminations

I/O Signals From the FPGA

There are several connectors that connect directly to the FPGA that can be used for application interface system interface functions.

Connector Reference	Type	Use
P16	XMC connector P16	I/O and host card interfacing
JP1	Front panel connector	Primarily used for the analog inputs but does have 2 pins to the FPGA for trigger inputs.

Table 49. X3-SD16 Connectors Connected to the FPGA

The specific logic signal names associated with each connector and its pins are shown in the following tables.

P16, XMC IO Connector

On the X3 modules, P16 is used for digital IO and timing signals. These are direct connections to the application FPGA and can be used for many purposes providing the IO standards and timing restrictions are met.

The UCF file has constraints for the following signals. Preserve the pin designation and rename the signal as required for the modified functionality.

The connectors pinout in the X3 Hardware Manual gives the pinout, including power/ground connections.

Most of the signals can be used as differential pairs, LVDS_25 IO standard. Pairs are dig_io<0>/dig_io<1>, 2/3 and so forth, with the + on even numbers, - is odd numbers. The exception is dig_io<18> and dig_io<19> which may only be used single-ended.

Some of the signals are also routed as timing signals as noted.

Logic Signal Name	IO Standard	Connector Pin	FPGA Pin	Use
dig_io<0>	LVTTL	C1	P22	Digital IO
dig_io<1>	LVTTL	C2	N21	Digital IO
dig_io<2>	LVTTL	C3	L24	Digital IO
dig_io<3>	LVTTL	C4	M23	Digital IO
dig_io<4>	LVTTL	C5	N18	Digital IO
dig_io<5>	LVTTL	C6	N17	Digital IO
dig_io<6>	LVTTL	C7	J26	Digital IO
dig_io<7>	LVTTL	C8	J25	Digital IO
dig_io<8>	LVTTL	C9	N20	Digital IO
dig_io<9>	LVTTL	C10	M20	Digital IO

Logic Signal Name	IO Standard	Connector Pin	FPGA Pin	Use
dig_io<10>	LVTTL	C11	P18	Digital IO
dig_io<11>	LVTTL	C12	N19	Digital IO
dig_io<12>	LVTTL	C13	J23	Digital IO
dig_io<13>	LVTTL	C14	J22	Digital IO
dig_io<14>	LVTTL	C15	M21	Digital IO
dig_io<15>	LVTTL	C16	M22	Digital IO
dig_io<16>	LVTTL	C17	L18	Digital IO
dig_io<17>	LVTTL	C18	L17	Digital IO
dig_io<18>	LVTTL	C19	M19	Digital IO
dig_io<19>	LVTTL	F1	M18	Digital IO
dig_io<20>	LVTTL	F2	K25	Digital IO
dig_io<21>	LVTTL	F3	K26	Digital IO
dig_io<22>	LVTTL	F4	L22	Digital IO
dig_io<23>	LVTTL	F5	K21	Digital IO
dig_io<24>	LVTTL	F6	G23	Digital IO
dig_io<25>	LVTTL	F7	G24	Digital IO
dig_io<26>	LVTTL	F8	L20	Digital IO
dig_io<27>	LVTTL	F9	K20	Digital IO
dig_io<28>	LVTTL	F10	F25	Digital IO
dig_io<29>	LVTTL	F11	F24	Digital IO
dig_io<30>	LVTTL	F12	K23	Digital IO
dig_io<31>	LVTTL	F13	K22	Digital IO
dig_io<32>	LVTTL	F14	E24	Digital IO
dig_io<33>	LVTTL	F15	F23	Digital IO
dig_io<34>	LVTTL	F16	K19	Digital IO
dig_io<35>	LVTTL	F17	K18	Digital IO
dig_io<36>	LVTTL	F18	F22	Digital IO
dig_io<37>	LVTTL	F19	G22	Digital IO

Logic Signal Name	IO Standard	Connector Pin	FPGA Pin	Use
dig_io<38> (PXIE_DSTARA_P)	LVTTL	A9	P26	Digital IO; routed to clock mux and may be used as conversion clock.
dig_io<39> (PXIE_DSTARA_N)	LVTTL	B9	P25	Digital IO; routed to clock mux and may be used as conversion clock.
dig_io<40> (PXIE_CLK100_P)	LVTTL	D9	P21	Digital IO; routed to clock mux and may be used as a PLL reference clock.
dig_io<41> (PXIE_CLK100_N)	LVTTL	E9	P20	Digital IO; routed to clock mux and may be used as a PLL reference clock.
dig_io<42>	LVTTL	A19	P23	Digital IO
dig_io<43	LVTTL	B19	N24	Digital IO
dio_clkp/n	LVDS_25	D19/E19	B13/C13	Digital IO Clock

Table 50. X3-SD16 P16 Connections

Notes:

1. PXIE_DSTARA_P/N must be terminated with 100 ohms (R285) to use as differential input clock.
2. PXIE_CLK100_P/N must be terminated with 100 ohms (R287) to use as differential input clock.

JP1, Front Panel IO Connector

There are a total of 8 signals on the X3-SD16 from JP1 that connect directly to the FPGA. These are used as digital IO and as trigger signals in the standard FrameWork Logic.

The fp_dio signals are routed as differential pairs.

The external trigger signals are routed as single-ended with 100 ohm series resistors.

Logic Signal Name	IO Standard	Connector Pin	FPGA Pin	Use
ext_trigger<0>	LVTTL	68	F13	External trigger input 0. (A/D)
ext_trigger<1>	LVTTL	34	G13	External trigger input 1. (DAC)
fp_dio<0>/fp_dio<1>	LVTTL/LVDS	63/29	M5/M6	Digital IO
fp_dio<2>/fp_dio<3>	LVTTL/LVDS	64/30	M8/M7	Digital IO
fp_dio<4>/fp_dio<5>	LVTTL/LVDS	65/31	N5/N4	Digital IO

Table 51. X3-SD16 JP1 Connections to the FPGA

Synthesis and Fitting

A project file for the X3-SD16 is provided for Xilinx ISE tool that has all the project hierarchy, included files and options required.

ISE Project File
x3_sd16.xise

Table 52. X3-SD16 Xilinx ISE Project Filename

If you are using Xilinx ISE, you should load this project as a starting point and recompile the logic to verify that the project is ready to use. You should be able to successfully generate the BIT file for the logic and run the examples on the card.

If you are using another synthesis tool, you will need to reconstruct the logic hierarchy as shown in the HTML documentation in your toolset. The packages and libraries that support Xilinx parts must be used since there are Xilinx-specific logic elements used in the design. These libraries are provided by Xilinx for your simulation tool.

Constraints

There are several important classes of constraints used by the FrameWork Logic : timing, pin placement and IO standards. These constraints are shown in the .ucf (user constraint file) that is used during the fitting process. The constraints file is in the ./source directory.

The constraint file for the X3-SD16 is provided for the standard Spartan3A DSP 1.8M device. If you are using a higher speed grade, or different device then the file must be modified for the device you are using.

X3-SD16 FPGA	*Device Used*	*Constraint File*
	Xilinx XC3SD1800A-4FGG676C	x3_sd16.ucf

Table 53. X3-SD16 Constraint File

Adding constraints to the logic is done by editing new constraints into the UCF file, which is simple text, or by using one of the Xilinx tools to help create constraints. The Constraint Manager is useful for most timing related constraints, while Floorplanner or Pace are useful for logic physical constraints.

Warning! We have noticed some strange occurrences in Pace and Floorplanner that seem to delete constraints for GCLK inputs where the location (LOC) constraint is mysteriously removed. It's a good idea to keep a copy of the original file. Some designers like to put the Pace or Floorplanner constraints in an entirely new file, then append them by hand. Whatever your method, just be aware that this is a known bad behavior of the current tools. When your clock location constraints get deleted, nothing works unless the tools happen to put them back where they belong. A dice roll at best..

There are several important classes of constraints used by the FrameWork Logic: timing, pin placement and IO standards. These constraints are shown in the .ucf (user constraint file) that is used during the fitting process.

Timing Constraints

The timing constraints defined cover the clocks used in the design and the external device signal timing. Clock period period constraints are used to cover most of the logic since they define the clock rate for all flip-flops connected to that clock. These period constraints then cover most of the logic paths used in a synchronous design.

Here are the clock period constraints used by the FrameWork Logic:

```
NET "fs_clk_p" TNM_NET = "fs_clk_p";
TIMESPEC TS_fs_clk_p = PERIOD "fs_clk_p" 38 MHz HIGH 50 %;
NET "sys_clk_in" TNM_NET = "sys_clk_in";
TIMESPEC TS_sys_clk_in = PERIOD "sys_clk_in" 68 MHz HIGH 50 %;
NET "cmd_clkx" TNM_NET = "cmd_clkx"
```

These period constraints cover most of the design and should not be modified. The system clock rate, sys_clk_in, is generated by the PCIe interface. The actual clock rate is 66.67 MHz; 68 MHz gives a little margin.

External devices require an additional constraint to be sure that we get the signal on-chip and to its destination in time. Since the external chip, such as the multi-queue SRAM, may have a delay from the clock edge to when we get the signal, an additional constraint is defines the amount of time after the clock that the signal is given to the logic. This type of constraint is used on the multi-queue SRAM signals to guarantee that setup timings are met. A TIMESPEC is defined from the flip-flops (FFS) to the pin group as shown, then pin groups are associated with the this constraint.

```
# MQ SRAM constraints
NET "mqram_dq<*>"     TNM = "mq_data_io";

TIMESPEC "TS_mq_data_in"  = FROM "mq_data_io" to "FFS" 8 ns; # inputs
TIMESPEC "TS_mq_data_out" = FROM "FFS" to "mq_data_io" 8 ns; # outputs

NET "mqram_sa(*)"    TNM = "mq_ctrl";
NET "mqram_adv_ld" TNM = "mq_ctrl";
NET "mqram_oe_n"      TNM = "mq_ctrl";
NET "mqram_r_wn" TNM = "mq_ctrl";
```

```
NET "mqram_ce" TNM = "mq_ctrl";
```

These constraints require the multi-queue SRAM signals to get on and off-chip in 8 nS or less.

IOB Constraints

Warning! DO NOT CHANGE these assignments as damage may occur to the X3-SD16! They must be used on all compiles.

The only signal IO STANDARD that may be changed on X3-SD16 is on the P16 interface. In the example logic they signals are 2.5V LVCMOS with a constraint like this

```
"dig_io<28>"  LOC = "G18" | IOSTANDARD = LVTTL ;
```

In the Xilinx documentation, the compatible IO standards are listed. These include LVDS and others. You must use one of the standards compatible with pin in the UCF since the FPGA has this bank of pins configured for this voltage and may not be changed.

Logic Utilization

The results of the mapping process, as taken from the X3_SD16.mrp report, are as shown. Notice the memory consumed during the compile is 466 MB for this design!

The number of occupied slices is 35%, though the tools do not pack unrelated logic in slice if it is not necessary or allowed. This does not mean however that the design 35% full, since it is OK in many cases to pack the logic tighter, especially for logic that is not high performance. The tools tend to spread the logic sparsely when space is available, so the actual number of flip-flops and LUTs is more important which are 22% and 17% respectively. This design also has debug cores in the logic that consume about 10% of the logic.

```
Release 12.2 Map M.63c (nt)
Xilinx Mapping Report File for Design 'x3_sd16'

Design Information
------------------
Command Line   : map -intstyle ise -p xc3sd1800a-fg676-4 -cm area -ir all -pr
off -c 100 -o x3_sd16_map.ncd x3_sd16.ngd x3_sd16.pcf
Target Device  : xc3sd1800a
Target Package : fg676
Target Speed   : -4
Mapper Version : spartan3adsp -- $Revision: 1.52 $
Mapped Date    : Sun Jul 24 11:49:48 2011

Design Summary
--------------
Number of errors:      0
Number of warnings:   47
```

```
Logic Utilization:
  Number of Slice Flip Flops:         8,602 out of  33,280   25%
  Number of 4 input LUTs:             6,668 out of  33,280   20%
Logic Distribution:
  Number of occupied Slices:          6,689 out of  16,640   40%
    Number of Slices containing only related logic:   6,689 out of   6,689 100%
    Number of Slices containing unrelated logic:          0 out of   6,689   0%
      *See NOTES below for an explanation of the effects of unrelated logic.
  Total Number of 4 input LUTs:       7,109 out of  33,280   21%
    Number used as logic:             6,662
    Number used as a route-thru:        441
    Number used as Shift registers:       6

  The Slice Logic Distribution report is not meaningful if the design is
  over-mapped for a non-slice resource or if Placement fails.

  Number of bonded IOBs:                339 out of     519   65%
    IOB Flip Flops:                     127
    IOB Master Pads:                      2
    IOB Slave Pads:                       2
  Number of ODDR2s used:                  4
    Number of DDR_ALIGNMENT = NONE        4
    Number of DDR_ALIGNMENT = C0          0
    Number of DDR_ALIGNMENT = C1          0
  Number of BUFGMUXs:                     4 out of      24   16%
  Number of DCMs:                         1 out of       8   12%
  Number of DSP48As:                      8 out of      84    9%
  Number of RAMB16BWERs:                 18 out of      84   21%

Average Fanout of Non-Clock Nets:                3.56

Peak Memory Usage:  276 MB
Total REAL time to MAP completion:  55 secs
Total CPU time to MAP completion:   46 secs
```

Place and Route

The Place and Route Step results are taken from the X3_SD16.par report. Timing analysis is shown for the design. It is important to review this report to find timing errors. The PAR report shows how the design performed against each defined timing constraint. For further analysis, the timing analyzer tool can be used to pinpoint the source of each problem.

```
The NUMBER OF SIGNALS NOT COMPLETELY ROUTED for this design is: 0

   The AVERAGE CONNECTION DELAY for this design is:        1.685
   The MAXIMUM PIN DELAY IS:                             14.530
   The AVERAGE CONNECTION DELAY on the 10 WORST NETS is:  10.297

   Listing Pin Delays by value: (nsec)

    d < 3.00   < d < 6.00  < d < 9.00  < d < 12.00  < d < 15.00  d >= 15.00
   ---------   ---------   ---------   ---------   ---------   ---------
       33096        3531         553         110          44           0

Timing Score: 0

Number of Timing Constraints that were not applied: 4

Asterisk (*) preceding a constraint indicates it was not met.
   This may be due to a setup or hold violation.

------------------------------------------------------------------------------------------------------
  Constraint                              |  Check  | Worst Case |  Best Case | Timing |   Timing
                                          |         |    Slack   | Achievable | Errors |    Score
```

TS_fp_data_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "fp_data_bus" 7 ns	MAXDELAY	0.008ns	6.992ns	0	0
TS_clk_fx = PERIOD TIMEGRP "clk_fx" TS_sys_clk_in * 1.33333333 HIGH 50%	SETUP	0.063ns	10.966ns	0	0
	HOLD	0.573ns		0	0
COMP "pll2_cs_n" OFFSET = OUT 10 ns AFTER COMP "sys_clk_in"	MAXDELAY	0.083ns	9.917ns	0	0
COMP "pll1_cs_n" OFFSET = OUT 10 ns AFTER COMP "sys_clk_in"	MAXDELAY	0.092ns	9.908ns	0	0
TS_link_data_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "link_data_in" 8 ns	MAXDELAY	0.112ns	7.888ns	0	0
TS_mq_ctrl_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "mq_ctrl" 7 ns	MAXDELAY	0.312ns	6.688ns	0	0
TS_dac_data_bus = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "dac_data_bus" 7.5 ns	MAXDELAY	0.354ns	7.146ns	0	0
COMP "pll_sdio" OFFSET = OUT 10 ns AFTER COMP "sys_clk_in"	MAXDELAY	0.399ns	9.601ns	0	0
TS_dcm_clk0 = PERIOD TIMEGRP "dcm_clk0" TS_sys_clk_in HIGH 50%	SETUP	0.535ns	12.564ns	0	0
	HOLD	0.458ns		0	0
TS_cmd_clkx = PERIOD TIMEGRP "cmd_clkx" 68 MHz HIGH 50%	SETUP	0.546ns	6.807ns	0	0
	HOLD	0.651ns		0	0
TS_zbt_ctrl_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "zbt_ctrl" 7 ns	MAXDELAY	0.716ns	6.284ns	0	0
TS_mq_data_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "mq_data_io" 8 ns	MAXDELAY	0.798ns	7.202ns	0	0
TS_zbt_data_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "zbt_data_io" 9 ns	MAXDELAY	1.762ns	7.238ns	0	0
COMP "mqram_clk" OFFSET = OUT 15 ns AFTER COMP "sys_clk_in"	MAXDELAY	1.812ns	13.188ns	0	0
COMP "pll_sclk" OFFSET = OUT 10 ns AFTER COMP "sys_clk_in"	MAXDELAY	2.243ns	7.757ns	0	0
COMP "cmd_fsr" OFFSET = OUT 10 ns AFTER COMP "cmd_clkx"	MAXDELAY	2.363ns	7.637ns	0	0
TS_zbt_sclk_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "zbt_clk" 7 ns	MAXDELAY	2.723ns	4.277ns	0	0
COMP "cmd_dr" OFFSET = OUT 10 ns AFTER COMP "cmd_clkx"	MAXDELAY	2.747ns	7.253ns	0	0
TS_link_ctrl_out = MAXDELAY FROM TIMEGRP "FFS" TO TIMEGRP "link_ctrl" 8 ns	MAXDELAY	3.643ns	4.357ns	0	0
TS_mq_data_in = MAXDELAY FROM TIMEGRP "mq_data_io" TO TIMEGRP "FFS" 8 ns	SETUP	3.841ns	4.159ns	0	0
	HOLD	2.460ns		0	0
TS_zbt_data_in = MAXDELAY FROM TIMEGRP "zbt_data_io" TO TIMEGRP "FFS" 9 ns	SETUP	4.412ns	4.588ns	0	0
	HOLD	2.435ns		0	0
TS_link_ctrl_in = MAXDELAY FROM TIMEGRP "link_ctrl" TO TIMEGRP "FFS" 8 ns	SETUP	5.579ns	2.421ns	0	0
TS_link_data_in = MAXDELAY FROM TIMEGRP "link_data_in" TO TIMEGRP "FFS" 8 ns	SETUP	5.659ns	2.341ns	0	0
TS_U_TO_J = MAXDELAY FROM TIMEGRP "U_CLK" TO TIMEGRP "J_CLK" 15 ns	SETUP	10.202ns	4.798ns	0	0
	HOLD	1.532ns		0	0

```
  COMP "cmd_fsx" OFFSET = IN 10 ns BEFORE C | SETUP   |    13.281ns|    -3.281ns|       0|           0
  OMP "cmd_clkx"                            |         |            |            |        |
----------------------------------------------------------------------------------------------------------
  TS_U_TO_U = MAXDELAY FROM TIMEGRP "U_CLK" | SETUP   |    13.571ns|     1.429ns|       0|           0
   TO TIMEGRP "U_CLK" 15 ns                 | HOLD    |     1.032ns|            |       0|           0
----------------------------------------------------------------------------------------------------------
  TS_J_TO_J = MAXDELAY FROM TIMEGRP "J_CLK" | SETUP   |    17.397ns|    12.603ns|       0|           0
   TO TIMEGRP "J_CLK" 30 ns                 | HOLD    |     0.699ns|            |       0|           0
----------------------------------------------------------------------------------------------------------
  COMP "cmd_dx" OFFSET = IN 10 ns BEFORE CO | SETUP   |    17.514ns|    -7.514ns|       0|           0
  MP "cmd_clkx"                             |         |            |            |        |
----------------------------------------------------------------------------------------------------------
  TS_fs_clk_p = PERIOD TIMEGRP "fs_clk_p" 2 | SETUP   |    30.502ns|     7.959ns|       0|           0
  6 MHz HIGH 50%                            | HOLD    |     0.815ns|            |       0|           0
----------------------------------------------------------------------------------------------------------
  PATH "TS_J_TO_D_path" TIG                 | SETUP   |         N/A|     6.441ns|     N/A|           0
----------------------------------------------------------------------------------------------------------
  PATH "TS_D_TO_J_path" TIG                 | SETUP   |         N/A|     6.278ns|     N/A|           0
----------------------------------------------------------------------------------------------------------
  COMP "cmd_clkr" OFFSET = OUT 10 ns AFTER  | N/A     |         N/A|         N/A|     N/A|         N/A
  COMP "cmd_clkx"                           |         |            |            |        |
----------------------------------------------------------------------------------------------------------
  PATH "TS_U_TO_D_path" TIG                 | N/A     |         N/A|         N/A|     N/A|         N/A
----------------------------------------------------------------------------------------------------------
  TS_sys_clk_in = PERIOD TIMEGRP "sys_clk_i | N/A     |         N/A|         N/A|     N/A|         N/A
  n" 68 MHz HIGH 50%                        |         |            |            |        |
----------------------------------------------------------------------------------------------------------
  TS_fp_data_in = MAXDELAY FROM TIMEGRP "fp | N/A     |         N/A|         N/A|     N/A|         N/A
  _data_bus" TO TIMEGRP "FFS" 7 ns          |         |            |            |        |
----------------------------------------------------------------------------------------------------------


All constraints were met.
INFO:Timing:2761 - N/A entries in the Constraints list may indicate that the
   constraint does not cover any paths or that it has no requested value.


Generating Pad Report.

All signals are completely routed.

WARNING:Par:283 - There are 1 loadless signals in this design. This design will cause Bitgen to issue DRC
warnings.

Total REAL time to PAR completion: 6 mins 42 secs
Total CPU time to PAR completion: 6 mins 31 secs

Peak Memory Usage:  466 MB

Placement: Completed - No errors found.
Routing: Completed - No errors found.
Timing: Completed - No errors found.

Number of error messages: 0
Number of warning messages: 15
Number of info messages: 0

Writing design to file x3_sd16.ncd



PAR done!
```

This is not the complete PAR report, but it shows the sort of result that is reported. Each timing constraint is analyzed to see if it was met. If not, the report tags the result so that further investigation can be done.

MATLAB Simulink Board Support Package

A MATLAB board support package for X3-SD16 is available. This BSP allows logic to be developed in MATLAB Simulink using Xilinx System Generator for the X3-SD16. See the *X3 MATLAB Board Support Guide.*

Simulation

The test files are used in the simulation and testing of the FrameWork code. The testbench file is x3_sd16_tb.vhd and it uses several components for testing that are defined by the other model files for the external memory chips. The test bench files are in the ./simulation directory.

The testbench contains a set of simulation steps that exercise various functions on the FrameWork logic for basic interface testing. Behavioral procedures have been written to simulate the command channel interface that are useful in simulating accesses by the PCIe host to the X3-SD16. An end-to-end simulation of the data flow from the A/D out to the PCIe controller through the data link is provided. Simulation of the DAC outputs shows data packets from the PCIe link through to the DAC interface.

Setting Up the Simulation

The files unzipped from the FrameWork Logic archive contain all the source and macro files needed. You will normally need to make a ModelSim project reflecting your exact directory structure, although a ModelSim project (.mpf) file is provided. You will also need to compile the Xilinx unisim, simprim and Xilinx core libs and point to them in ModelSim. These libraries may be compiled from within Xilinx ISE by selecting the device on the Sources Window then selecting the Compile HDL Simulation Libraries on the Processes window. Set the project default to VHDL '93 if you intend to compile in ModelSim using the project manager.

Simulation Models for X3-SD16

The memory components for X3-SD16 have simulation models provided by the device vendor. These include accurate timing models so that real timings can be simulated. If a timing violation occurs during their use, it is real and should be considered.

Model	*Filename*	*Functional Behavior*
SRAM	CY7C1371k.vhd	SRAM model provided by Cypress Semiconductor.
A/D	ads1278_model.vhd	Simple A/D model behavioral model for TI ADS1278 that provides simple ramp data.
DAC	pcm1681_model.vhd	Simple DAC model behavioral model for TI PCM1681.

Table 54. X3-SD16 Simulation Models

Loading the Testbench

The simulator used from within ISE is Mentor Graphics ModelSim and support files for using the testbench from within environment are included. Selecting the testbench file in the Sources window then allows you to select the simulation in the Process Window. This should start ModelSim if your tools are set up correctly.

If you run ModelSim as a standalone tool outside the ModelSim environment, the DO files in the ./simulation directory can be used to set ups and run the simulation.

Testbench do files	Function
tb_x3_sd16.do	Compiles all files and loads design.
Wave.do	Loads a wave display format.

Table 55. X3-SD16 Simulation Macro Files

The DO files compile and load the design for simulation. Modify the do file to include new source files for your design to point at the files.

Running the Simulations

When you enter ModelSim from the ISE tool, a macro file is auto generated by the tools to compile up the design. You will also have to compile the memory files the first time in ModelSim to get them to run, since the auto generated macro does not do this.

If you use ModelSim in the standalone mode, just run the simulation from the ModelSim window after the project loads.

The simulation demonstrates the data flow and can be used as a starting point for modifying the design. In the wave window, the configuration accesses are followed data acquisition as shown here.

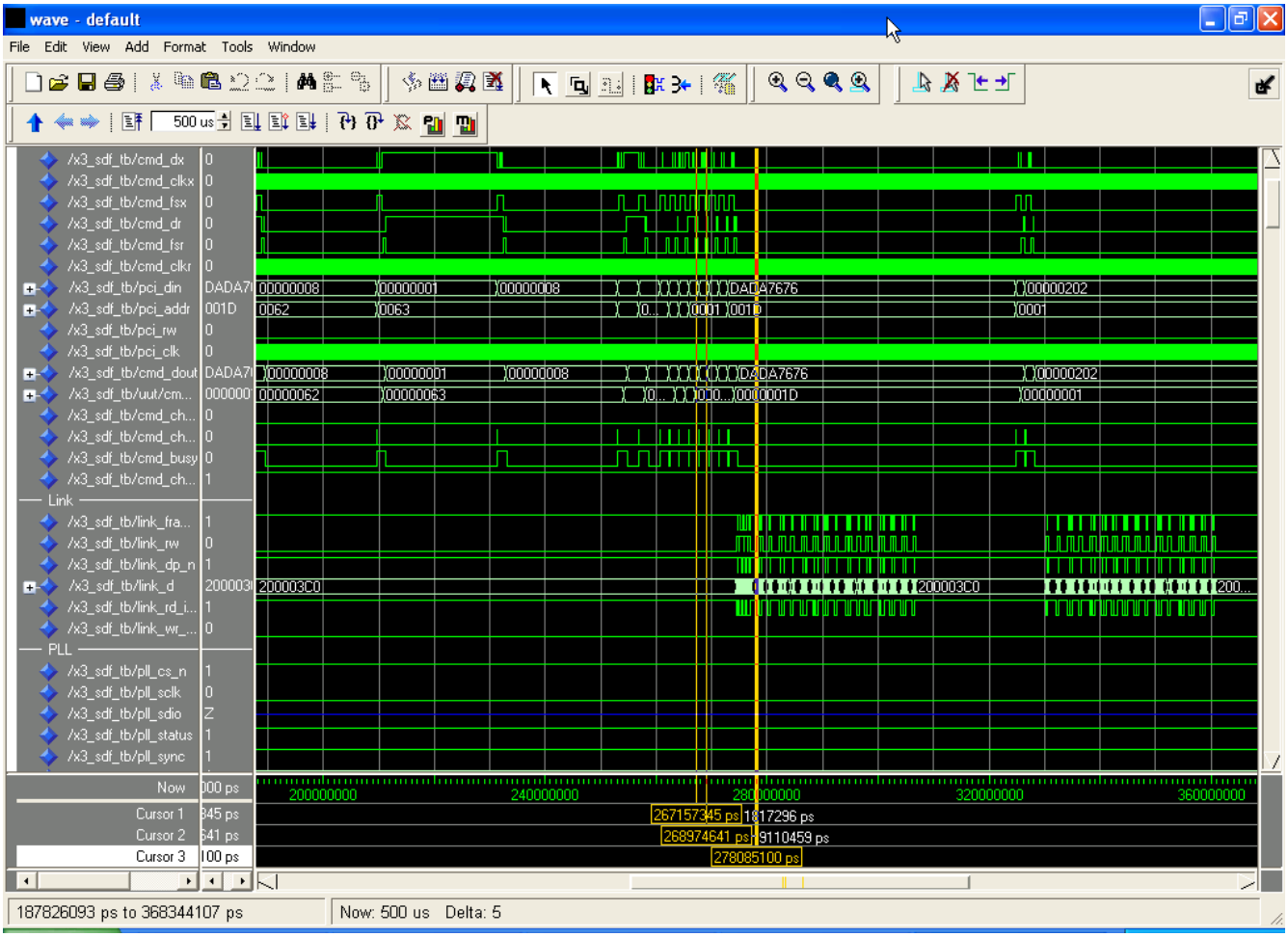


Figure 43. Simulation Results

Modifying the Simulations

The testbench file provides sample code to begin your simulations. Command channel accesses for initializing the X3-SD16 logic for data acquisition are shown in the testbed. Data flows from the A/D inputs, through the logic to the PCIe interface controller data link. The complete data flow can be observed in the simulation including operation of the multi-queue VFIFO.

In many cases, the testbed data flow can be modified by adding command channel reads and writes to configure the logic, then observe the data for the modified design. The command channel reads/writes are performed in the simulations using the procedure as in this example:

```
cmd_wr(X"000F", X"ABCDEF01"); -- write to ZBT data register
```

In this example, the command channel performed a write to address X"F" of X"ABCDEF01".

Some Things to Watch Out For in Simulation

Be sure to use a time resolution of ps for all simulations. The DCM simulation component will not work reliably unless ps resolution is used.

Many of the files use VHDL '93 and will issue errors if not compiled as '93 code. As a rule, always use '93 standard when compiling the FrameWork Logic.

Compile the packages with the project. These are frequently overlooked.

Making the logic images for downloading

The X3-SD16 logic image may be downloaded either over the PCIe bus or by using the Xilinx JTAG port. The image must be downloaded each time the board is powered up; there is no on-board ROM for the application logic.

Loading over PCIe

To download the logic using the PCIe bus, the Download.exe application is used, or one of the software methods discussed in the software manuals. The image must be either a BIT or EXO type (Motorola EXORMacs type).

BIT files are the most convenient since they are the native Xilinx format.

This EXO file is created using the Xilinx IMPACT tool from the BIT file that the tools generated after place and route was completed. See section entitled "Making the Device Image".

Loading over JTAG

The application logic may also be downloaded over the JTAG port to the device at any time. Connect the Xilinx cable to JP3 (see hardware manual for location and pin out).

Xilinx IMPACT software is used to load the image. When you open the IMPACT tool, you can select the download to device in the wizard and this will lead you through the process. The JTAG chain will identify only the application logic device and you will assign the BIT file created by the ISE tool to load the image.

There are three devices in the Xilinx scan path of the X3-SD16: the Xilinx configuration FLASH for the PCI FPGA (XCF02S) and two FPGAs (the XC3S250E and XC3SD1800A). The configuration FLASH and XC3S250E should be put into bypass mode during downloading.

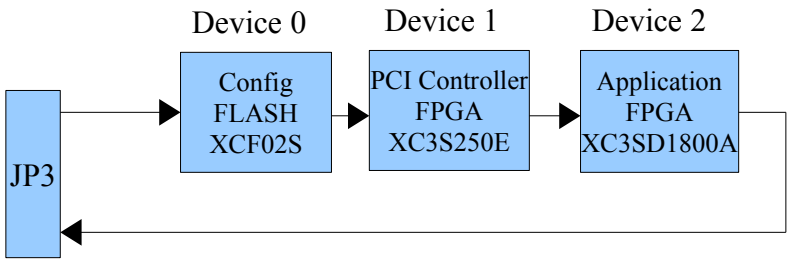


Figure 44. X3-SD16 Xilinx JTAG Scan Path

The image may also be loaded inside of ChipScope by right-clicking the target device and specifying the bit file.

Module-Specific Logic Components

The module-specific components are primarily hardware interface logic for each module. Each component is described, ports defined and a block diagram presented. In many cases the component has constraints defined in the particular module logic design that must be used during implementation.

X3-SDF Logic Components

These components are only used in X3-SDF module logic.

Component: ii_sdf_adc

Supported Platforms: X3-SDF

Source File: ii_sdf_adc.vhd

Description:
This component is the interface to the A/D converters on the X3-SDF XMC module. The A/D device, Analog Devices AD7760, is a 24-bit device with programmable decimation and filters. The component provides a configuration and control interface to the A/D devices and a data interface.

This A/D has two data modes that have different data timings: normal and modulator data. Data rate in the normal mode is up to 5 MSPS, 24-bits from each channel. Data rate in modulator data mode is 20 MSPS, 16-bits from each channel. Modulator data mode is only supported for logic development and is not used in the FrameWork Logic implementation.

Data to the system logic is output on individual channel outputs for both data modes synchronous to the sample clock fs_clk. The normal data mode can also use a FIFO to the system logic, providing a transition to the system clock domain and data buffering. Flow control to the FIFO allows the system to pace the data output to match processing requirements.

Overflow error bits indicate to the system when an A/D channel saw an out of range analog input.

Data Modes
In the normal mode, the logic fetches the data from the A/D devices after each DRDY as two 16-bit words, assembles the words into 32-bits and latches the incoming data. The data is output to the individual channel outputs and is also stored into a FIFO. Data from the FIFO are 24-bit A/D values, sign extended to 32-bits, delivered with a data valid for each enabled channel and a channel number. Data always appears in the ascending channel order (0, 1, 2, 3). The **dvalid** signal indicates when data is valid on the **dout** bus. The channel_out port indicates the channel number for the **dout**.

In the modulator data mode, the logic latches data every sample clock edge when **DRDY** is true. The data is 16-bits from each channel and is output on logic ports for the system. The output data in this mode is only available on the individual channel outputs and is synchronous to the sample clock (**fs_clk**).

The mode bit controls which data mode is used. This bit must not change while run is true and CANNOT be dynamically changed. The A/D devices must be reconfigured to change the data mode.

Triggering
The **trigger** input controls when samples are saved to the FIFO. This allows the other logic to control the data acquisition process by only keeping the data of interest. The trigger is forced to align with the data period so that no partial data sets are collected.

The trigger input is required to be synchronous with the sample clock (**fs_clk**). When run is true, the component stores data to the FIFO each time a data set is collected and trigger is true. When trigger is false, data is simply discarded.

The channel data outputs are active only when **trigger** is true.

A/D Overflow Indicators
The A/D embeds an overflow flag in the status byte for each data sample when in normal mode. A overflow condition in the A/D indicates that the input was over-voltage. Data is corrupted and inaccurate when an overflow occurs. On X3-SDF, the overflow bits are connected to an alert. When enabled, the module will generate an alert message to the system indicating that an overflow occurred.

In modulator data mode, these bits are not active.

Flow Control
The data from the FIFO has is enabled to flow when **RFD** is true. The downstream logic should use this input to the control the flow of data from the FIFO. If RFD is false and the FIFO is full, data is lost.

A/D Configuration and Status
The A/D must be configured prior to use for its operating mode. The A/D interface allows the logic to access A/D registers as a memory mapped device using a chip select, address and data bus. For specific information on the AD7760 register set, refer to its data sheet.

Access to the A/D registers are done after reset but before run is asserted. The component performs accesses to the A/D using the control inputs to read and write the registers. The sample clock must be running to access the A/D. Accesses are a function of the sample clock and are self-timed by the logic component. The **ctl_rdy** signal indicates when the access is completed by returning to '1' at the end of the access.

Test Mode
For test, a ramp generator may be substituted for the data. This is used to verify data flow instead of real A/D data. The data ramps from 0 to 2^23 by 1 each data period. The A/D DRDY must be active for the ramp generator to run, so the A/D's and sample clock must be configured to operate.

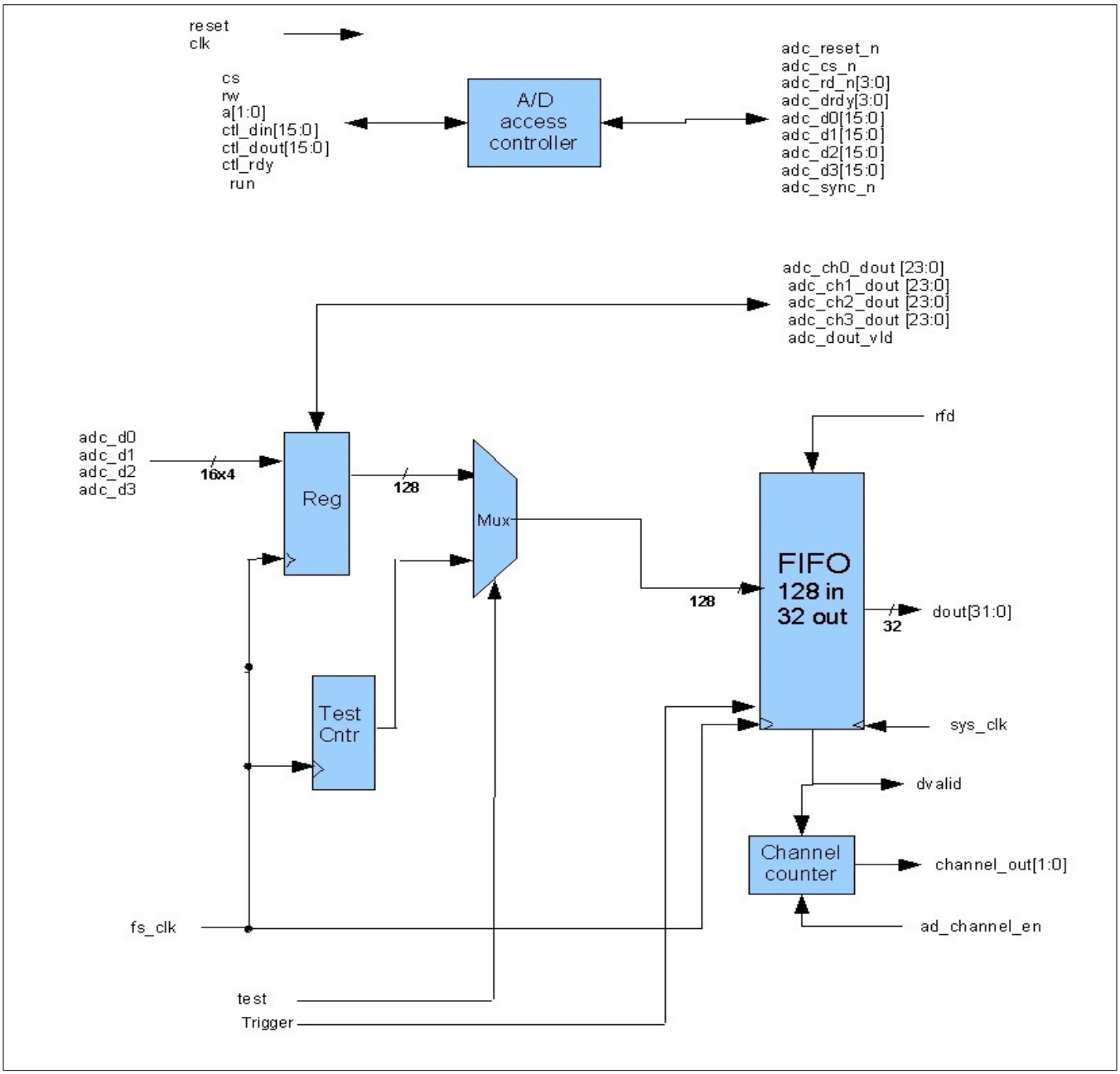


Figure 45. ii_sdf_adc Interface Component

Table 56. ii_sdf_adc Component Ports

Port	Direction	Function
reset	In	reset
clk	In	system clock (67 MHz)
fs_clk	In	sample clock (20 MHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true.
adc_reset_in	In	Reset to the A/D devices from the system logic.
sync	In	Sync control to the A/D devices from the system logic.
mode	In	Normal or Modulator data mode control, '0' = normal.
channel_en[3:0]	In	Channel enables for A/Ds.
cs	In	Chip select for configuration accesses
rw	In	Read/write(low) for configuration accesses
a[1:0]	In	Control address for which A/D is being accessed
ctl_din[15:0]	In	Data bus in for control accesses
ctl_dout[15:0]	Out	Data bus out for control accesses
trig_en	In	The trigger is enabled when true (sync to fs_clk)
adc_reset_n	Out	Reset output to the A/D devices
adc_cs_n[3:0]	Out	A/D chip select outputs, active low.
adc_rd_n[3:0]	Out	A/D chip read enable outputs, active low.
adc_drdy[3:0]	In	A/D data ready inputs.
adc_d0[15:0]	Inout	A/D 0 data bus
adc_d1[15:0]	Inout	A/D 1 data bus
adc_d2[15:0]	Inout	A/D 2 data bus
adc_d3[15:0]	Inout	A/D 3 data bus
adc_sync_n	Out	A/D sync output, active low
rfd	In	Allows data to flow from the FIFO
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
channel_out[1:0]	Out	Channel number for **dout**
overrange[3:0]	Out	A/D overrange indicator
adc_ch0_dout[23:0]	Out	A/D 0 data out (sync to fs_clk)
adc_ch1_dout[23:0]	Out	A/D 1 data out (sync to fs_clk)
adc_ch2_dout[23:0]	Out	A/D 2 data out (sync to fs_clk)
adc_ch3_dout[23:0]	Out	A/D 3 data out (sync to fs_clk)
adc_dout_vld	Out	A/D data outputs are valid
test	In	Enable test mode (0= disabled)

X3-Servo Logic Components

These components are only used in X3-Servo module logic.

Component: ii_servo_adc

Supported Platforms: X3-Servo

Source File: ii_servo_adc.vhd

Description:
This component is the interface to the A/D converters on the X3-Servo XMC module. The A/D devices, Texas Instruments ADS8365, are 16-bit devices with six channels per device. The component provides an interface to the A/D devices for reading data, error compensation, channel selection and data buffering. All channels are assumed to be synchronous (using the same sample clock), although a periodic clock is not required.

A/D Interface
The A/D interface collects data from two device each of which has 6 channels. A common data bus between the two devices is used. Data is read from all channels, whether enabled or not, each sample period. When the A/D device gives and End-Of-Conversion signal, the logic performs twelve successive reads for the channels in ascending order from 0 to 11. Unused channels are discarded in the component according to the channel enable input vector. The component also assumes that the A/D is used in incrementing mode for sequential channel reads which is set by the **adc_a** bits.

Error Compensation
The A/D component provides first order error compensation for each A/D channel using the ii_offgain component (see generic library description). This compensates for offset and first-order gain errors. Individual gain and offset error coefficients are provided via the **gain_array** and **offset_array** input arrays.

Data Stacking and Buffering
Data to the system logic is output as a stream of 32-bit numbers, composed of two 16-bit samples. The data is stacked in ascending channel order, for the channels that are enabled in the **channel_en** input vector.

A 16-bit input, 32-bit output FIFO that is 1Kx32 in size is used for data buffering. The data if read when RDEN is true. Flow control to the FIFO should be implemented in the system logic using the **fifo_rd_cnt** output allows the system to pace the data output to match processing requirements and avoid overflow.

Error Reporting
Overrange error bits indicate to the system when an A/D channel saw an out of range analog input. An overrange is reported whenever the error corrected sample is at full scale, X”8000” or X”7FFF”. These bits are reset when run is false ('0').

Test Features

The test input substitutes a ramp in place of the A/D data. This is used for data path testing during logic design. It can be used to verify that no data loss occurs due to flow control problems by checking the output for the monotonically increasing ramp data. Real A/D data can be much more difficult to check since a missing point is not easily detected.

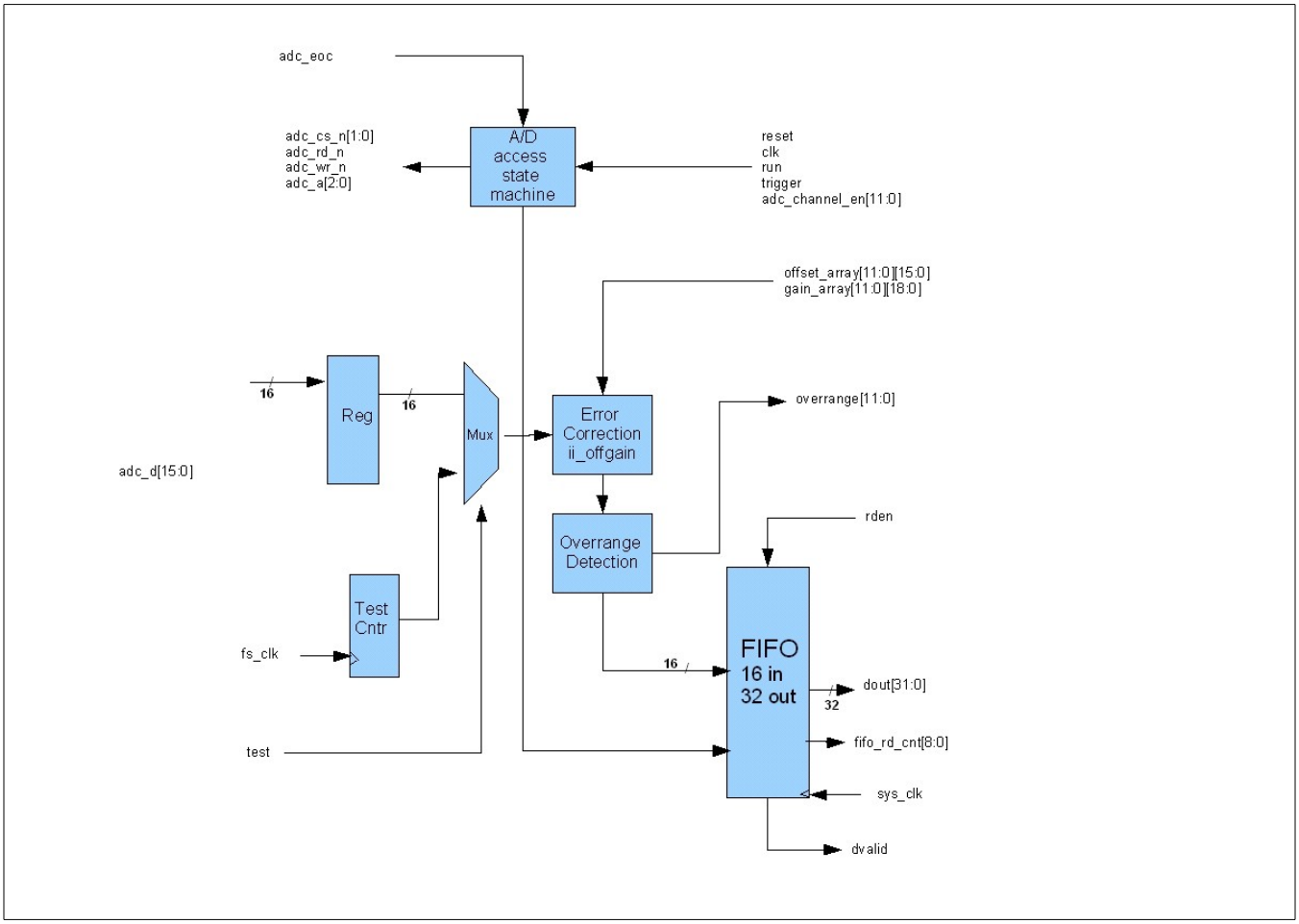


Figure 46. ii_servo_adc Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (67 MHz)
fs_clk	In	sample clock (250 kHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true (synchronous to fs_clk).
channel_en[11:0]	In	Channel enables for A/Ds.
offset_array	In	Array of offsets for error correction, defined in ii_x3_pkg.vhd.
gain_array	In	Array of gains for error correction, defined in ii_x3_pkg.vhd.
adc_cs_n[1:0]	Out	A/D chip selects, active low. One per device.
adc_rd_n	Out	A/D read enable, active low.
adc_wr_n	Out	A/D write enable, active low.
adc_a[2:0]	Out	A/D address inputs, used to control channel selection. Set to “110” in the component for incrementing mode.
adc_eoc	In	A/D end-of-convert signal. A falling edge on this signal indicates conversion is completed and data is ready to read.
adc_d[15:0]	Inout	A/D data bus. Shared between two devices.
rden	In	Allows data to flow from the FIFO
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
overrange[11:0]	Out	A/D overrange indicators
test	In	Enable test mode (0= disabled)

Table 57. ii_servo_adc Component Ports

Component: ii_servo_dac

Supported Platforms: X3-Servo

Source File: ii_servo_dac.vhd

Description:
This component is the interface to the D/A converters on the X3-Servo XMC module. The DAC devices, Texas Instruments DAC8822, are six 16-bit devices with two channels per device supporting 12 channels total. The component provides an interface to the DAC devices for writing data, error compensation, channel selection and data buffering. All channels are assumed to be synchronous (using the same sample clock), although a periodic clock is not required.

Data Unstacking and Buffering
Data from the system logic is provided as a stream of 32-bit numbers, composed of two 16-bit samples. The data is unstacked in ascending channel order, for the channels that are enabled in the channel_en input vector.

A 32-bit input, 16-bit output FIFO that is 1Kx32 in size is used for data buffering. The data is written when **dac_wr** is true. Flow control to the FIFO should be implemented in the system logic using the **dac_fifo_cnt** output allows the system to pace the data output to match processing requirements and avoid underflow or overflow conditions.

DAC Interface
The DAC interface writes data to each of the two device each of which has 6 channels. A common data bus between the two devices is used. Data is written to all channels, whether enabled or not, each sample period. When the A/D device gives and End-Of-Conversion signal, the logic performs twelve successive reads for the channels in ascending order from 0 to 11. Unused channels are discarded in the component according to the channel enable input vector. The component also assumes that the A/D is used in incrementing mode for sequential channel reads which is set by the **adc_a** bits.

Error Compensation
The DAC component provides first order error compensation for each DAC channel using the ii_offgain component (see generic library description). This compensates for offset and first-order gain errors. Individual gain and offset error coefficients are provided via the **gain** and **offset** input arrays.

Error Reporting
Underflow error bit indicates to the system that the DAC FIFO was empty when a point was needed. Data was not provided in time. This bit is reset when run is false ('0').

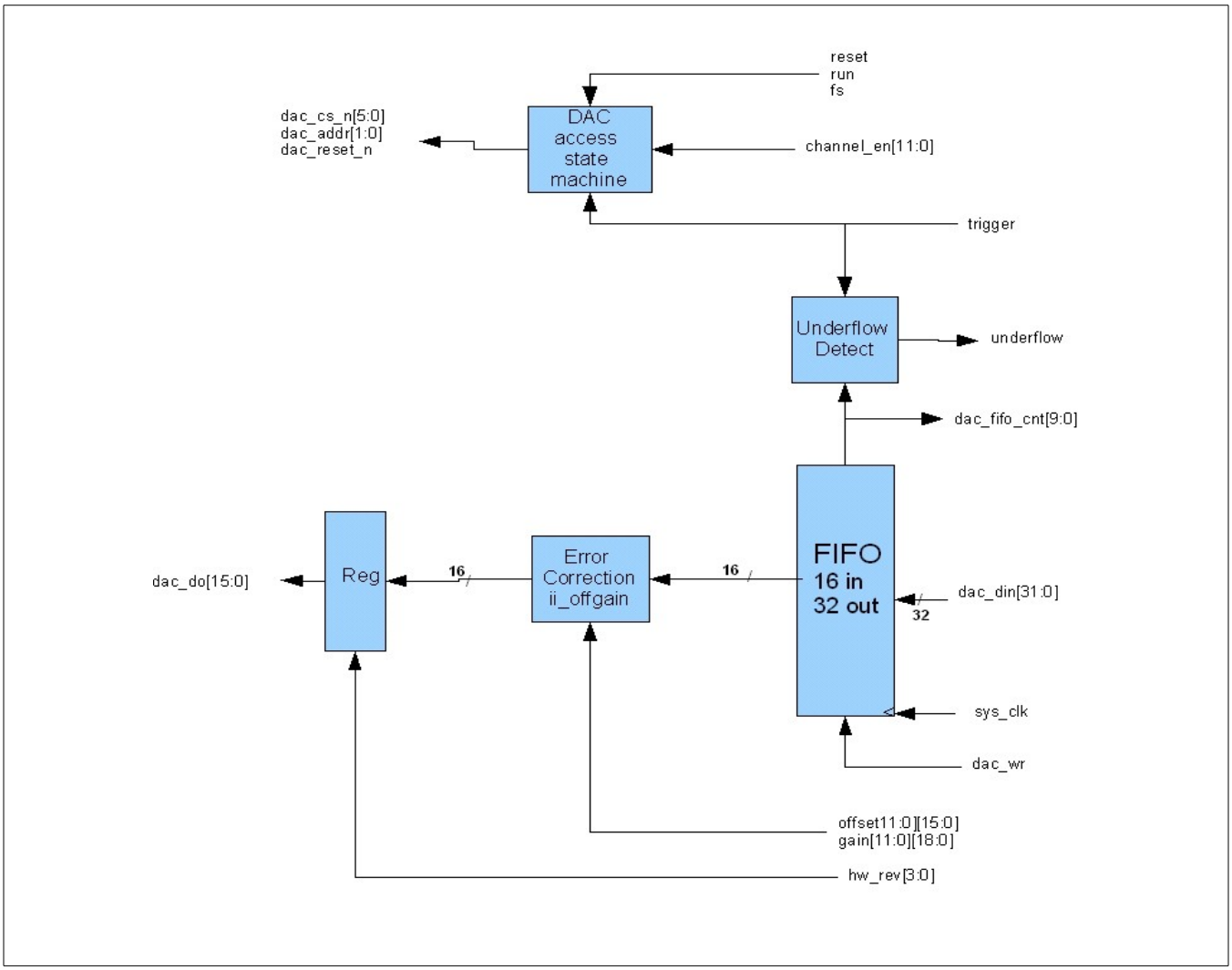


Figure 47. ii_servo_dac Component Block Diagram

Port	Direction	Function
hw_rev[3:0]	In	Hardware revision. Used by component to correct DAC data for hardware error on revision 0.
reset	In	reset
clk	In	system clock (67 MHz)
fs	In	sample clock (2 MHz or less)
run	In	Enable the component to operate.
dac_wr	In	Write enable to FIFO from system logic.
dac_din[31:0]	In	DAC sample pairs from the system.
channel_en[11:0]	In	Channel enables for DACs.
offset	In	Array of offsets for error correction, defined in ii_x3_pkg.vhd.
gain	In	Array of gains for error correction, defined in ii_x3_pkg.vhd.
dac_reset_n	Out	DAC reset, active low.
dac_cs_n[5:0]	Out	DAC chip selects, active low. One per device.
dac_addr[1:0]	Out	DAC address outputs, used to control channel selection.
dac_do[15:0]	Out	DAC data bus. Shared to 6 devices.
dac_fifo_cnt[9:0]	Out	DAC FIFO count
fifo_status[15:0]	Out	DAC FIFO status. (not used)
underflow	Out	DAC FIFO underflow error. Reset on RUN false.
trigger	In	Acquire data when true (synchronous to fs_clk).

Table 58. ii_servo_dac Component Ports

X3-SD16 Logic Components

These components are only used in X3-SD16 module logic.

Component: ii_sd16_adc

Supported Platforms: X3-SD16

Source File: ii_sd16_adc.vhd

Description:
This component is the interface to the A/D converters on the X3-SD16 XMC module. The A/D devices, Texas Instruments ADS1278, are 24-bit devices with eight channels per device. The component provides an interface to the A/D devices for reading data, error compensation, channel selection and data buffering.

ADS1278 Device Interface
The A/D device requires a master clock that is a multiple of the sample rate. The multiple is a function of the sampling mode of the device and the desired sample rate.

Mode	Max fclk (MHz)	Mode	adc_clkdiv	SCLK	Max Sample Rate
High Speed	37	"00"	1	256	144531
High Resolution	27	"01"	1	512	52734
Low Power	13.5	"10"	0	256	52734
Low Power	27	"10"	1	512	52734
Low Speed	5.4	"11"	0	512	10547
Low Speed	27	"11"	1	2560	10547

Table 59. A/D Clock and Sample Rates for Sample Modes

The ii_sd16_adc generates the serial clock (adc_sclk) and frame (adc_fsync) signals based on the mode and adc_clkdiv settings. These signals are synchronized to the to either an internal sync or to an external sync when multiple cards are used. The sync ensures that the A/D devices sample in synchronization by controlling the frame sync signal.

The ii_sd16_adc component uses the frame-sync TDM serial interface to the ADS1278. The FPGA-generated serial clock and frame are used to receive the data on the serial data inputs (adc0_sdi, adc1_sdi) as shown the following diagram.

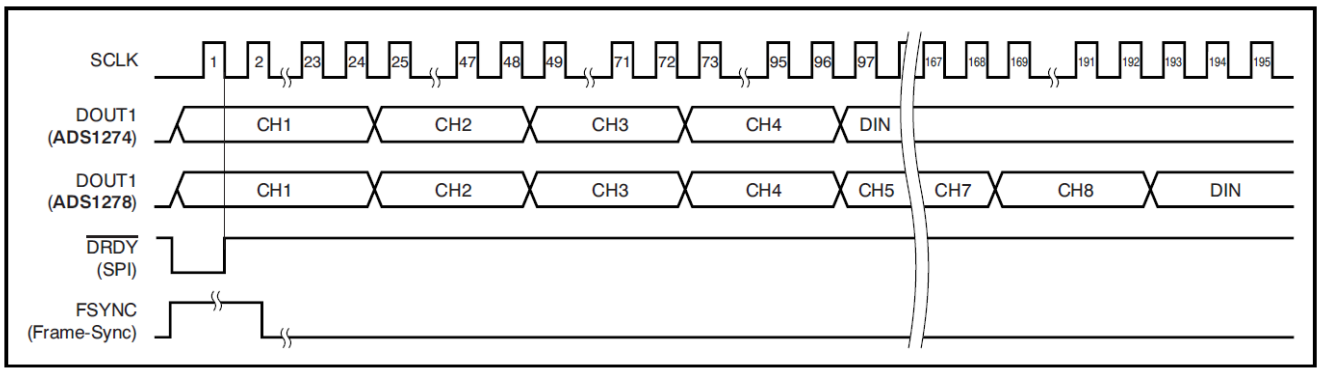


Figure 48. ADS1278 Serial Port Timing in TDM Mode

The data is received into the FPGA and captured in shift registers. A data word is taken from each shift register every 24 bits.

A/D Synchronization

The A/D has a synchronization pin, adc_sync_n, that holds the A/D until released. Multiple devices can run synchronously by deasserting sync at the same time. The sync_in signal from the system logic controls the sync to the A/D device. If the sync_master input is true, then the sync timing is modified to match pipeline delays with non-master cards.

All synchronization assumes that the fs_clk on all cards is synchronous, and that the sync_in is synchronous to fs_clk.

Error Correction

The A/D channels are corrected in the logic for bias and gain errors due to analog electronics. This is used for calibrating the A/D channels. Calibration coefficients are loaded into the system FLASH ROM during factory test and are then used for error correction. Each A/D channel has an offset and gain register mapped to system memory. The software is expected to read the coefficients from the ROM and write them to these registers as part of system initialization.

The error correction is a first-order error correction implemented in the logic as

$y = m(x + b)$, where m = gain correction, b = offset correction and x = input A/D sample

The offset coefficient is a 16-bit number, which is left shifted (multiplied by 32) before it is added to the sample. The gain coefficient is 18-bit.

Gain Coef = 0x1000000 + Gain; Gain is 18-bit number

Offset Coef = Offset * 32

The gain error compensation is ~ +/- 0.78% of the input range. All calculations use saturating math.

FIFO Data Buffer

The output from the component is a FIFO that buffers the A/D samples. The number of enabled channels determines the data format. If all channels are enabled, the samples are sequentially stored to the FIFO AD0..AD15. For a single channel, data is AD0(t), AD0(t+1). If a channel is turned off, such as channel 1, the ordering is AD0, AD2, AD3, AD4..AD15, AD0(t+1).

Data from the A/D FIFO is output as a 2's complement, 32bit. Negative numbers are sign-extended from the 24-bit sample data. Full scale is 0x007FFFFF, negative full scale is 0xFF80000.

		Bits
Data Port	Clock Domain	31..0
FIFO	sys_clk	A/D sample channel n

Table 60. X3-SD16 A/D Component Output Data Format

Error Reporting

Overrange error bits indicate to the system when an A/D channel saw an out of range analog input. An overrange is reported whenever the error corrected sample is at full scale, X"800000" or X"7FFFFF". These bits are reset when run is false ('0').

Overflow errors are reported when the output FIFO is full when a point must be stored. When this error flag asserts, one or more sets points in lost. A set of points is all channels for one sample period.

Test Features

The test input substitutes a ramp in place of the A/D data. This is used for data path testing during logic design. It can be used to verify that no data loss occurs due to flow control problems by checking the output for the monotonically increasing ramp data. Real A/D data can be much more difficult to check since a missing point is not easily detected.

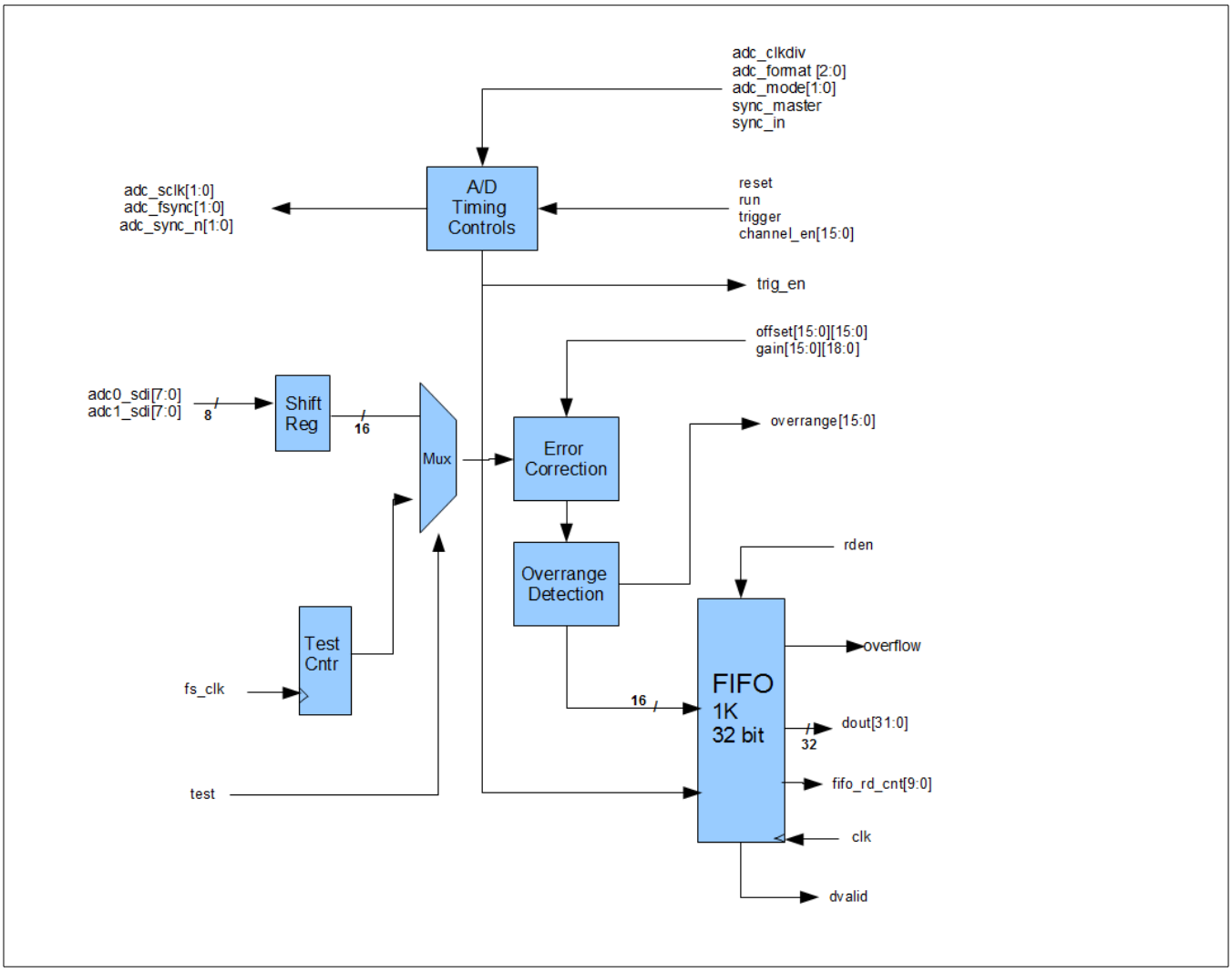


Figure 49. ii_sd16_adc Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (67 MHz)
fs_clk	In	sample clock (250 kHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true (synchronous to fs_clk).
sync_master	In	Indicates to the logic this is the master sync card
sync_in	In	Sync input from system logic
channel_en[15:0]	In	Channel enables for A/Ds.
adc_mode	In	A/D operation mode
adc_clkdiv	In	A/D master clock divisor setting
trig_en	Out	A/D Trigger enabled; sync'd to fs_clk
adc_sclk[1:-0]	Out	A/D serial clocks
adc0_sdi[7:0]	In	A/D 0 serial data inputs
adc1_sdi[7:0]	In	A/D 1 serial data inputs
adc_fsync	Out	A/D frame sync (L/R)
adc_sync_n	Out	A/D sync control, active low
adc_format	Out	A/D data format control
gain	In	Array of gains for error correction, defined in ii_x3_pkg.vhd.
offset	In	Array of offsets for error correction, defined in ii_x3_pkg.vhd.
fifo_rd_cnt[9:0]	Out	Output FIFO read count
rdy	In	Allows data to flow from the FIFO
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
overflow	Out	A/D FIFO overflow error indicator
overrange[15:0]	Out	A/D overrange indicators
test	In	Enable test mode (0= disabled)

Table 61. ii_sd16_adc Component Ports

Component: ii_sd_dac

Supported Platforms: X3-SD16

Source File: ii_sd16_dac.vhd

Description:
This component is the interface to the D/A converters on the X3-SD16 XMC module. The DAC devices, Texas Instruments PCM1681, are two 24-bit devices with eight channels per device supporting 16 channels total. The component provides an interface to the DAC devices for sample data, error compensation, channel selection and data buffering. All channels are assumed to be synchronous (using the same sample clock.

PCM1681 Device Interface
The D/A devices require a master clock that is a multiple of the sample rate. The multiple is a function of the sampling mode of the device and the desired sample rate. The two supported clock rates are 128fs and 256fs modes.

Mode	Max fclk (MHz)	Mode	Clock Multiple	Max Sample Rate(ksps)
256fs	40	"00"	256	156.25
128fs	24.576	"01"	128	192

Table 62. D/A Clock and Sample Rates for Sample Modes

The ii_sd16_dac generates the serial bit clock (dac_bclk) and left-right frame (dac_lr) signals based on the mode setting. These signals are synchronized to the to either an internal sync or to an external sync when multiple cards are used. The sync ensures that the D/A devices sample in synchronization by controlling the frame sync signal.

The ii_sd16_dac component uses the left-justified audio format serial interface to the PCM1681. The FPGA-generated serial bit clock and left-right frame are used to transmit data on the serial data outputs (dac0_sdo, dac1_sdo) as shown the following diagram.

Left-Justified Data Format; L-Channel = HIGH, R-Channel = LOW (default)

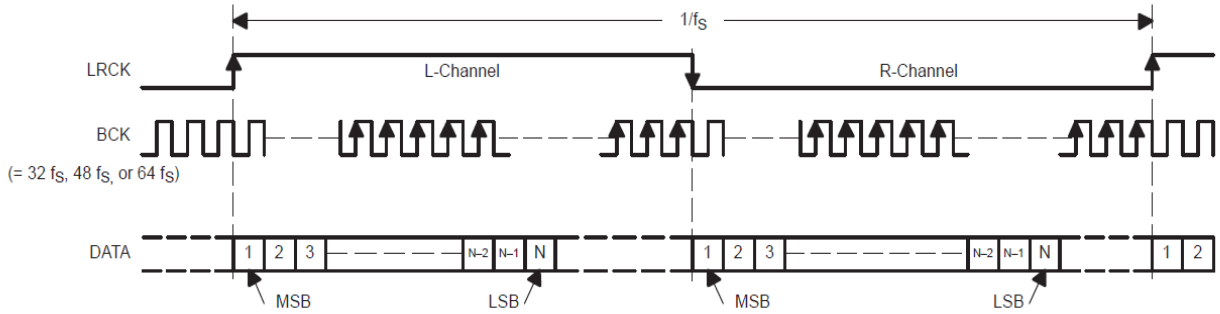


Figure 50. PCM1681 Serial Port Timing

D/A Synchronization
The D/A is synchronized to the system by controlling the timing of the output data and timing signals to the DAC device. Multiple devices can run synchronously by using the sync_in from the system logic to time the data output to the DAC devices.

All synchronization assumes that the fs_clk on all cards is synchronous, and that the sync_in is synchronous to fs_clk.

Error Correction
The DAC channels are corrected in the logic for bias and gain errors due to analog electronics. This is used for calibrating the DAC channels. Calibration coefficients are loaded into the system FLASH ROM during factory test and are then used for error correction. Each D/A channel has an offset and gain register mapped to system memory. The software is expected to read the coefficients from the ROM and write them to these registers as part of system initialization.

The error correction is a first-order error correction implemented in the logic as

y = mx + b, where m = gain correction, b = offset correction and x = output D/A sample

The offset coefficient is a 16-bit number, which is left shifted (multiplied by 32) before it is added to the sample. The gain coefficient is 18-bit.

Gain Coef = 0x1000000 + Gain; Gain is 18-bit number

Offset Coef = Offset * 32

The gain error compensation is ~ +/- 0.78% of the input range. All calculations use saturating math.

FIFO Data Buffer
The output from the component is a FIFO that buffers the D/A samples. The number of enabled channels determines the data format. If all channels are enabled, the samples are sequentially stored to the FIFO DA0..DA15. For a single channel, data is DA0(t), DA0(t+1). If a channel is turned off, such as channel 1, the ordering is DA0, DA2, DA3, .DA15, DA0(t+1).

Data to the D/A FIFO must be 2's complement, 32bit. Negative numbers are sign-extended from the 24-bit sample data. Full scale is 0x007FFFFF, negative full scale is 0xFF80000.

		Bits
Data Port	Clock Domain	31..0
FIFO	sys_clk	DAC channel n

Table 63. X3-SD16 DAC Component Input Data

Error Reporting
Underflow error bit indicates to the system that the DAC FIFO was empty when a point was needed. Data was not provided in time. This bit is reset when run is false ('0').

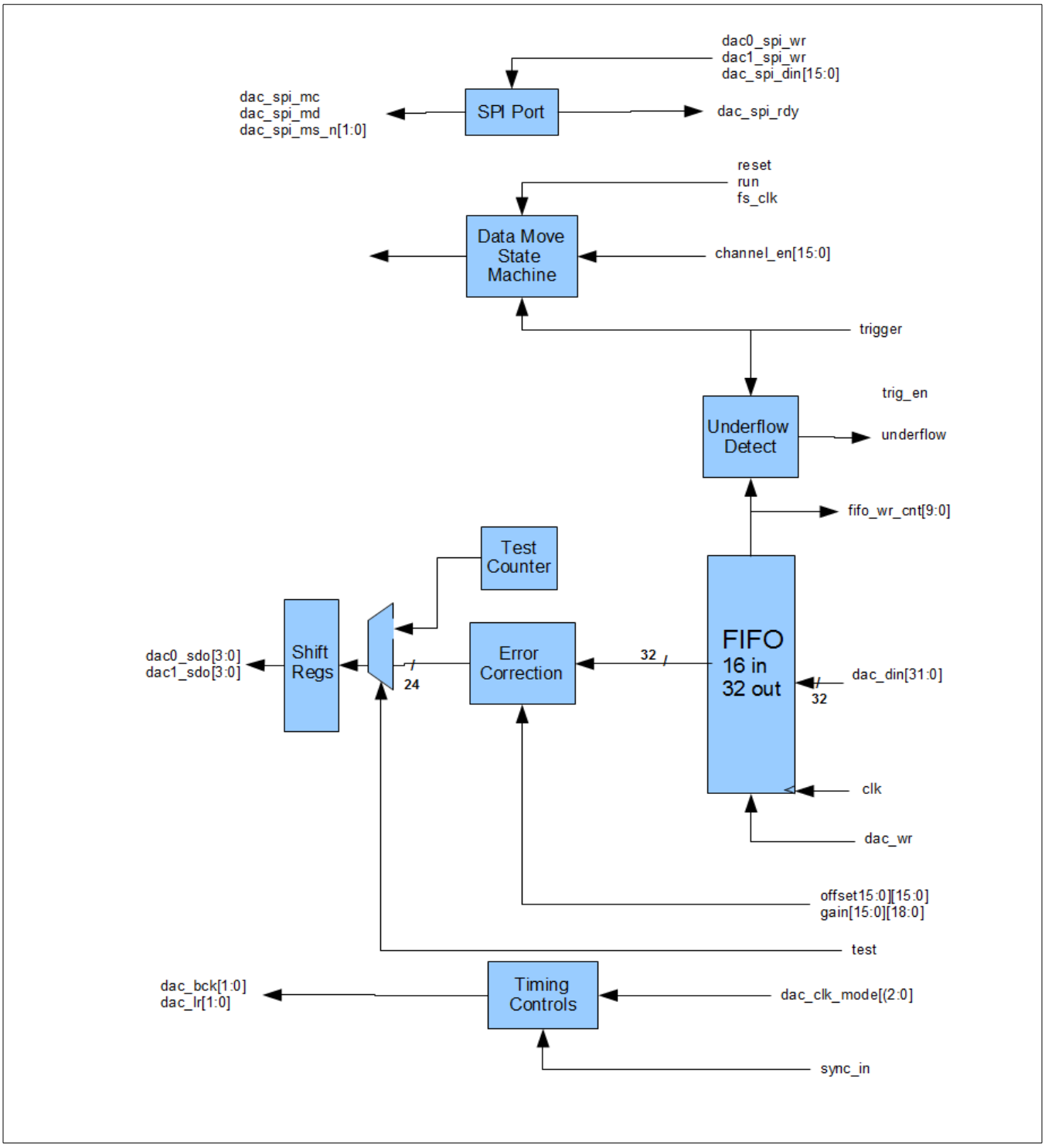


Figure 51. ii_sd16_dac Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (67 MHz)
fs_clk	In	sample clock (a multiple of the sample rate)
run	In	Enable the component to operate.
dac_clk_mode[2:0]	In	DAC sampling mode controls 2..0
dac0_spi_wr	In	DAC 0 SPI port write
dac1_spi_wr	In	DAC 1 SPI port write
dac_spi_din[15:0]	In	DAC SPI data 15..0
dac_spi_rdy	Out	DAC SPI port is ready to use
sync_in	In	Sync from system logic for multicard synchronization
dac_wr	In	Write enable to FIFO from system logic.
dac_din[31:0]	In	DAC sample pairs from the system.
channel_en[15:0]	In	Channel enables for DACs.
offset	In	Array of offsets for error correction, defined in ii_x3_pkg.vhd.
gain	In	Array of gains for error correction, defined in ii_x3_pkg.vhd.
dac_bck	Out	DAC bit clocks
dac0_sdo[3:0]	Out	DAC0 output bit streams, 1 per L/R pair
dac1_sdo[3:0]	Out	DAC1 output bit streams, 1 per L/R pair
dac_lr[1:0]	Out	DAC L/R data frame controls
dac_spi_mc	Out	DAC SPI clock
dac_spi_md	Out	DAC SPI data
dac_spi_ms_n[1:0]	Out	DAC SPI enables, active low
fifo_wr_cnt[9:0]	Out	DAC FIFO count
underflow	Out	DAC FIFO underflow error. Reset on RUN false.
trigger	In	Acquire data when true (synchronous to fs_clk).
trig_en	Out	Trigger is enabled on this clock (sync'd to fs_clk)
test	In	Test mode enable; inserts ramp for data when true.

Table 64. ii_sd16_dac Component Ports

X3-10M Logic Components

These components are only used in X3-10M module logic.

Component: ii_10m_adc

Supported Platforms: X3-10M

Source File: ii_10m_adc.vhd

Description:
This component is the interface to the A/D converters on the X3-10M XMC module. The A/D devices, Linear Technology LTC2203, are 16-bit devices capable of 25 MSPS. The component provides an interface to the A/D devices for reading data, error compensation, channel selection and data buffering. All channels are assumed to be synchronous (using the same sample clock). A periodic clock is required.

A/D Interface
The A/D interface collects data from eight A/D devices, all running synchronously. Each pair of A/D devices shares a 16-bit data bus into the logic. In the logic, the **adc_cs_n[]** chip select enables the devices on the bus so that data is captured from one device on the falling edge of sample clock (**fs_clk**) and the other on the rising edge. In the logic, an IDDR (input dual data rate flip-flop) is used to capture the data.

Data is read from all channels, whether enabled or not, each sample period. The A/D uses a bit scrambling technique to reduce electrical noise so the logic unscrambles the data samples by XOR bit 0 with the other bits to recover the data word. All 8 samples are assembled into a 64-bit word that stored into the FIFO.

Error Compensation
As words are read from the FIFO, the A/D component provides first order error compensation. The 32-bit words from the FIFO, containing two data samples, are run through two ii_offgain components (see generic library description). This compensates for offset and first-order gain errors. Individual gain and offset error coefficients are provided via the **gain** and **offset** input arrays.

Data Stacking and Buffering
A 128-bit input, 32-bit output FIFO that is 1Kx32 in size is used for data buffering. The data if read when **RFD** is true. Flow control to the FIFO should be implemented in the system logic using this signal.

As the data is read from the FIFO it then flows into the error correction component. The dvalid signal from the FIFO is used to enable the error correction and then the logic provides a **dvalid** for each data pair of enabled channels as they emerge from the error correction. The other channels that are not enabled to give a **dvalid** signal and are discarded.

Data to the system logic is output as a stream of 32-bit numbers, composed of two 16-bit samples. The data is stacked in ascending channel pair order, for the channel pairs that are enabled in the **channel_en** input vector.

Error Reporting

Overrange error bits indicate to the system when an A/D channel saw an out of range analog input. An overrange is reported by the A/D whenever the analog signal is out of range. The A/D provides these signals each sample, which are read by the logic using a DDR flip-flop when the A/D samples are read. These bits are reset when run is false ('0').

Test Features

The test input substitutes a ramp in place of the A/D data. This is used for data path testing during logic design. It can be used to verify that no data loss occurs due to flow control problems by checking the output for the monotonically increasing ramp data. Real A/D data can be much more difficult to check since a missing point is not easily detected.

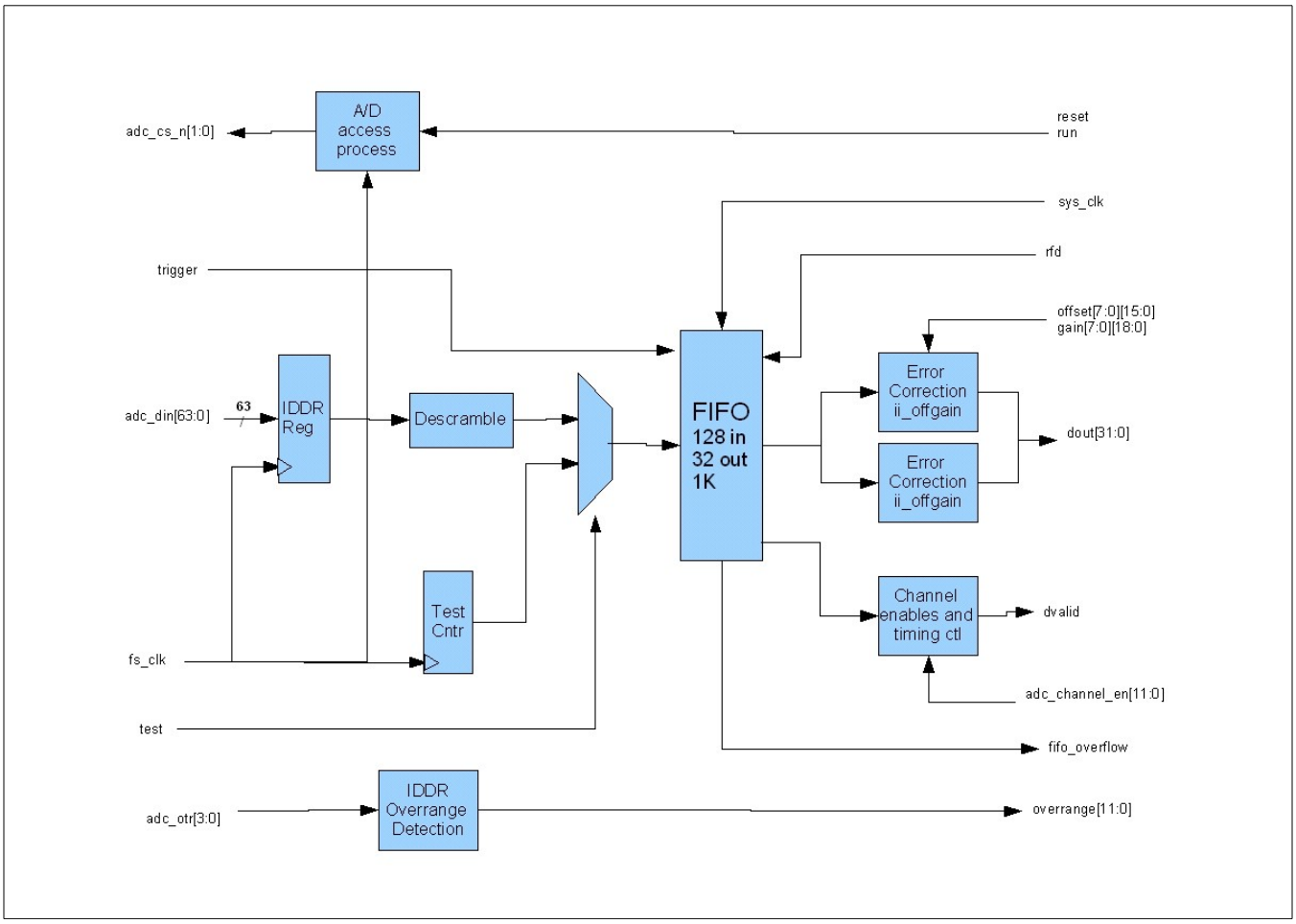


Figure 52. ii_10m_adc Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (67 MHz)
fs_clk	In	sample clock (25 MHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true (synchronous to fs_clk).
rfd	In	System logic input indicating that it is ready to receive data. A true on this signal causes the FIFO to be read.
channel_en[7:0]	In	Channel enables for A/Ds.
offset	In	Array of offsets for error correction, defined in ii_x3_pkg.vhd.
gain	In	Array of gains for error correction, defined in ii_x3_pkg.vhd.
adc_cs_n[1:0]	Out	A/D chip selects, active low. One for devices read on fs_clk rising edges, the other for all falling edge devices.
adc_otr[3:0]	In	A/D overrange inputs. Each input is shared between two devices.
adc_din[63:0]	In	A/D data bus. Shared between two devices.
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
overflow	Out	A/D FIFO overflow.
test	In	Enable test mode (0= disabled)

Table 65. ii_10m_adc Component Ports

X3-SD Logic Components

These components are only used in X3-SD module logic.

Component: ii_sd_adc

Supported Platforms: X3-SD

Source File: ii_sd_adc.vhd

Description:
This component is the interface to the A/D converters on the X3-SD XMC module. The A/D devices, Texas Instruments PCM4204, are 4 channels of 24-bit, 216 ksps. The component provides an interface to the four A/D devices (16 channels total) for reading data, error compensation, channel selection and data buffering. All channels are assumed to be synchronous (using the same sample clock). A periodic clock is required.

A/D Interface
The A/D interface collects data from four A/D devices, all running synchronously. The A/D are configured on the module to run in slave mode so that the logic is the source of the data timing signals **bclk** and **lr**. The A/D provides data on the **sdo** input to the logic that is collected in an input shift register to assemble the data words from the serial bit stream. There are 24 bits per sample, 32 clocks per lr phase. The left channel data is collected when **lr** is '1'.

Data is read from all channels, whether enabled or not, each sample period. After each **lr** phase, the 8 channels of data are error corrected in the logic and the enabled channels are stored into the FIFO.

Error Compensation
At the end of each **lr** phase, the 8 channels are error corrected for first-order gain and offset errors. Individual gain and offset error coefficients are provided via the **gain** and **offset** input arrays. The offset is first corrected using a 24-bit adder, then the gain is corrected using a 24-bit multiplier. The logic has saturation checking for the calculation.

Data Buffering
A 24-bit input, 32 deep FIFO is used for data buffering. The input to this FIFO is on fs_clk domain, and output is on system clock. This clock domain transition is the sole purpose of this FIFO. As data is written to the FIFO, it immediately flows out to the system with a data valid (**dvalid)** indicating when the data is valid on the output data bus.

Error Reporting

Overrange error bits indicate when an A/D channel saw an out of range analog input. The A/D provides these signals each sample. They are not used on the component.

Test Features

The test input substitutes a ramp in place of the A/D data. This is used for data path testing during logic design. It can be used to verify that no data loss occurs due to flow control problems by checking the output for the monotonically increasing ramp data. Real A/D data can be much more difficult to check since a missing point is not easily detected.

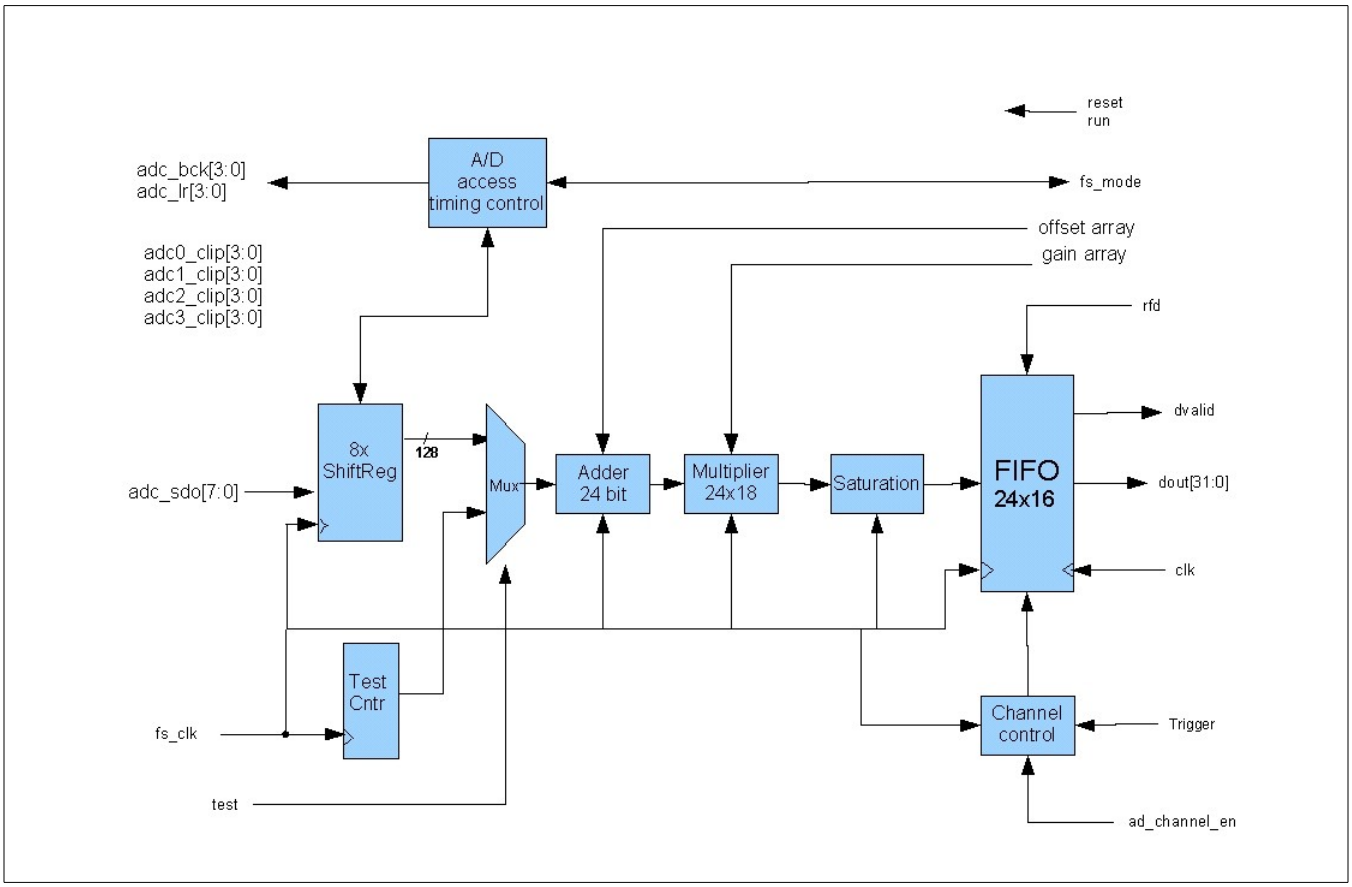


Figure 53. ii_sd_adc Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (67 MHz)
fs_clk	In	sample clock (25 MHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true (synchronous to fs_clk).
fs_mode[2:0]	In	A/D sample mode control bits. Specifies the A/D operating mode for rate and filter shaping. See PCM4204 data sheet for details.
channel_en[15:0]	In	Channel enables for A/Ds.
offset	In	Array of offsets for error correction, defined in ii_x3_pkg.vhd.
gain	In	Array of gains for error correction, defined in ii_x3_pkg.vhd.
adc_bck[3:0]	Out	A/D bit clock. This is a division of fs_clk that is set by fs_mode.
adc_sdo[7:0]	In	A/D data output. Serial data from the A/D, two channels per device.
adc_lr[3:0]	Out	A/D left/right control. Frames data for left/right channel from the A/D.,
adc0_clip[3:0]	In	A/D overrange bits. One per channel on device 0.
adc1_clip[3:0]	In	A/D overrange bits. One per channel on device 1.
adc2_clip[3:0]	In	A/D overrange bits. One per channel on device 2.
adc3_clip[3:0]	In	A/D overrange bits. One per channel on device 3.
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
test	In	Enable test mode (0= disabled)

Table 66. ii_sd_adc Component Ports

X3-25M Logic Components

These components are only used in X3-25M module logic.

Component: ii_x3_25m_adc_intf

Supported Platforms: X3-25M

Source File: ii_x3_25m_adc_intf, fifo_1kx32_async_vld.vhd, ii_offgain.vhd

Description:
This component is the interface to the A/D converters on the X3-25M XMC module. The A/D devices, Linear Technology LTC2207, are 16-bit, 105 MSPS devices. The component provides an interface to the two A/D devices for receiving data, error compensation, channel selection and data buffering. All channels are assumed to be synchronous (using the same sample clock). A periodic clock is required.

A/D Interface
The A/D interface collects data from two A/D devices, both running synchronously, using two 16-bit data buses to the FPGA. The data bits are time-delayed using an IBUF_DLY_ADJ element per bit to match the data timing to the sample clock (fs_clk) then are captured in an input register. The IBUF_DLY_ADJ is set to 3 ns in the standard logic. Data is read from both channels, whether enabled or not, each sample period.

The A/D uses a bit scrambling technique to reduce electrical noise. The logic unscrambles the data samples by XORing bit 0 with the other bits to recover the data word. Data is then stacked into 32-bit words, formatted according to the enabled channels.

Enabled channels	Bits 31..16	Bits 15..0
1 and 0	AD1[15..0]	AD0[15..0]
0 only	AD0[15..0](t-1)	AD0[15..0](t)
1 only	AD1[15..0](t-1)	AD1[15..0](t)

Data Buffering
A 32-bit input, 1K deep FIFO is used for data buffering. The input to this FIFO is on fs_clk domain, and output is on system clock. Data can be read by the system logic using RDEN, with FIFO_RD_CNT. Data is valid for the system when DVALID is true, synchronous to sys_clk.

Error Compensation
As each point is read from the FIFO it is error corrected for first-order gain and offset errors. Individual gain and offset error coefficients are provided via the **gain0, gain1, offset0,** and **offset1** input vectors. The *ii_offgain* component is used for the error correction, as detailed in the generic library chapter.

Test Features
The test input substitutes a ramp in place of the A/D data. This is used for data path testing during logic design. It can be used to verify that no data loss occurs due to flow control problems by checking the output for the monotonically increasing ramp data. Real A/D data can be much more difficult to check since a missing point is not easily detected.

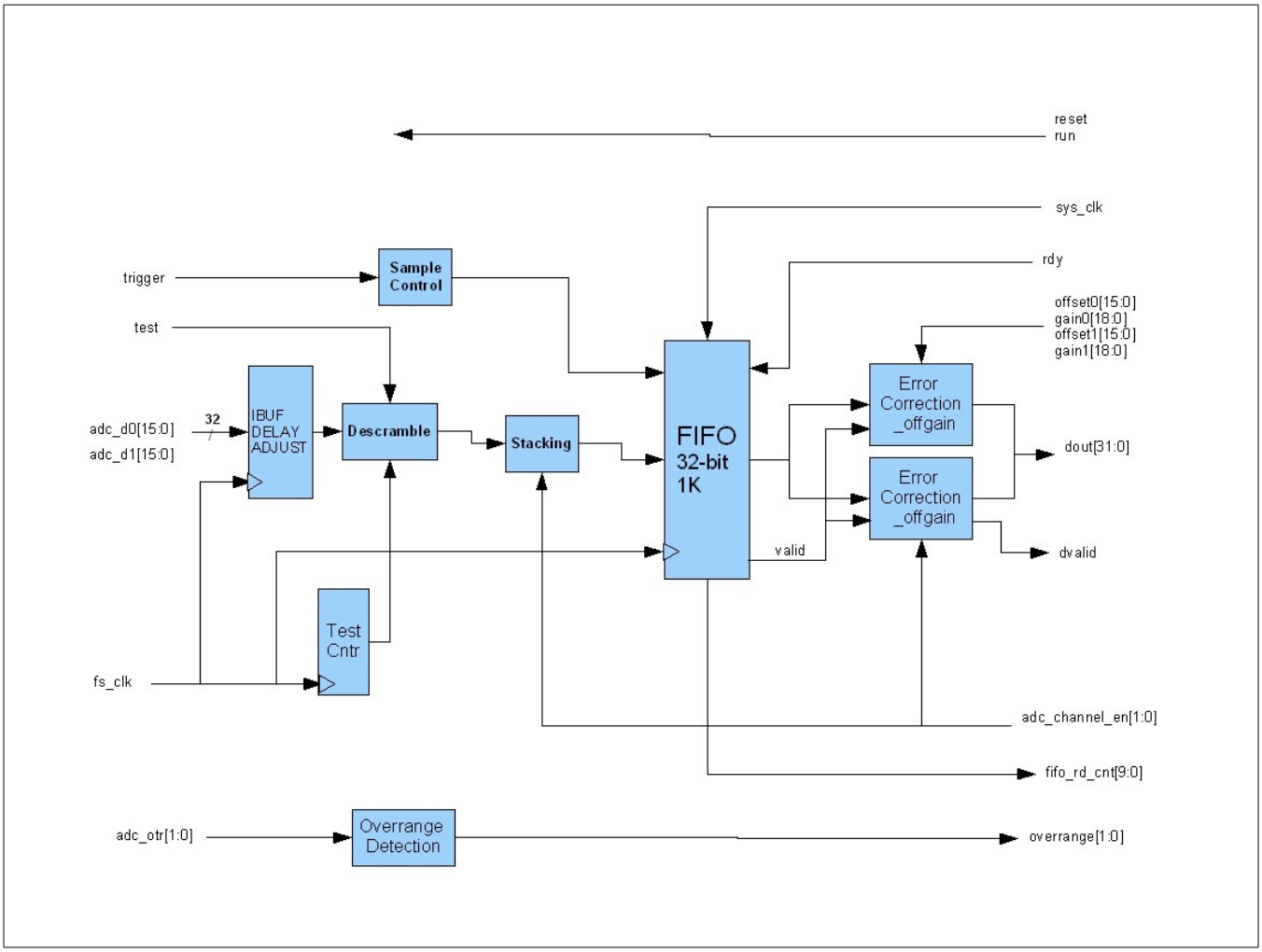


Figure 54. ii_x3_25m_adc_intf Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (107 MHz)
fs_clk	In	sample clock (105 MHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true (synchronous to fs_clk).
channel_en[1:0]	In	Channel enables for A/Ds.
rdy	In	System is ready to receive data, essentially FIFO read enable
offset0[15:0]	In	A/D 0 offset for error correction
gain0[17:0]	In	A/D 0 gain for error correction
offset1[15:0]	In	A/D 1 offset for error correction
gain1[17:0]	In	A/D 1 gain for error correction
adc_d0[15:0]	In	A/D 0 data bus (synchronous to fs_clk).
adc_d1[15:0]	In	A/D 1 data bus (synchronous to fs_clk).
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
fifo_rd_cnt[9:0]	Out	FIFO read count, synchronous to clk
test	In	Enable test mode (0= disabled)

Table 67. ii_x3_25m_adc_intf Component Ports

Component: ii_x3_25m_dac_intf

Supported Platforms: X3-25M

Source File: ii_x3_25m_dac_intf, fifo_1kx32_async_vld.vhd, ii_offgain.vhd

Description:
This component is the interface to the two D/A converters on the X3-25M XMC module. The DAC devices, Linear Technology LTC1668, are 16-bit, 50 MSPS devices. The component provides an interface to the two DAC devices for data buffering, error compensation, channel selection and providing output data. All channels are assumed to be synchronous (using the same sample clock). A periodic clock is not required.

Data Buffering
A 32-bit input, 1K deep FIFO is used for data buffering from the system logic. Data can be written by the system logic using DAC_WR, with DAC_FIFO_CNT used to pace the data according to the available space in the FIFO.

DAC Interface
As data is received from the system logic into the input FIFO, the 32-bit words are unstacked into 16-bit samples. The data is expected to be formatted according to the enabled channels.

Enabled channels	Bits 31..16	Bits 15..0
1 and 0	DAC1[15..0]	DAC0[15..0]
0 only	DAC0[15..0](t-1)	DAC0[15..0](t)
1 only	DAC1[15..0](t-1)	DAC1[15..0](t)

The data samples are then moved into the error correction logic.

Error Compensation
As each point is error corrected for first-order gain and offset errors. Individual gain and offset error coefficients are provided via the **gain0, gain1, offset0,** and **offset1** input vectors. The *ii_offgain* component is used for the error correction, as detailed in the generic library chapter.

DAC Outputs
There is a 1Kx16 FIFO for each of the two DAC devices that hold the outbound samples. The output FIFO is read when trigger is true on each rising edge of sample clock.

The DAC data has an output register on the FPGA pins to improve timing. This register is reset to mid-scale (0x8000) so that the DAC output is zero on reset.

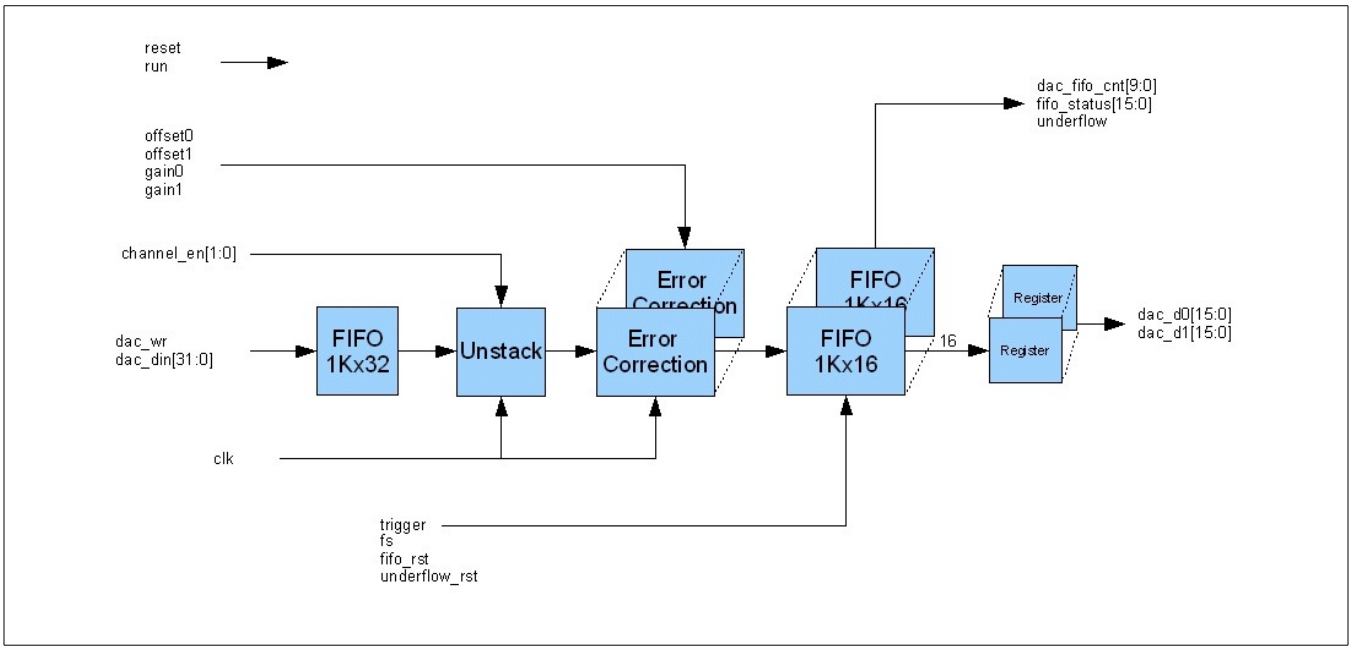


Figure 55. ii_x3_25m_dac_intf Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (107 MHz)
fs_clk	In	sample clock (50 MHz or less)
run	In	Enable the component to operate.
underflow_rst	In	Reset underflow error indicator bit.
fifo_rst	In	Reset fifo.
dac_wr	In	Write enable to DAC FIFO.
dac_din[31:0]	In	DAC data bus input from system logic.
channel_en[1:0]	In	Channel enables for A/Ds.
offset0[15:0]	In	DAC 0 offset for error correction
gain0[17:0]	In	DAC 0 gain for error correction
offset1[15:0]	In	DAC 1 offset for error correction
gain1[17:0]	In	DAC 1 gain for error correction
dac_d0[15:0]	Out	DAC 0 data bus (synchronous to fs_clk).
dac_d1[15:0]	Out	DAC 1 data bus (synchronous to fs_clk).
dac_fifo_cnt[9:0]	Out	FIFO write count, synchronous to clk
fifo_status[15:0]	Out	FIFO status (not used)
underflow	Out	DAC FIFO underflow. (1 = error).
trigger	In	Acquire data when true (synchronous to fs_clk).

Table 68. ii_x3_25m_dac_intf Component Ports

X3-A4D4 Logic Components

These components are only used in X3-A4D4 module logic.

These components are only used in X3-25M module logic.

Component: ii_x3_a4d4_adc

Supported Platforms: X3-A4D4

Source File: ii_x3_a4d4_adc, fifo_1k_16i_32o.vhd, ii_offgain.vhd

Description:
This component is the interface to the A/D converters on the X3-A4D4 XMC module. The A/D devices, Texas Instruments ADS8422, are 16-bit, 4 MSPS devices. The component provides an interface to the four A/D devices for receiving data, error compensation, channel selection and data buffering. All channels are assumed to be synchronous (using the same sample clock). A periodic clock is NOT required.

The entire component runs on the system clock domain. For the standard logic, the system clock is 107 MHz.

The RUN signal is used as a reset. When RUN is false, the component is reset. This allows the logic to reset the A/D without a complete system reset.

A/D Interface
The A/D devices convert every falling edge of the sample clock. The A/D BUSY signal goes true during the conversion then goes false when the data is ready. The logic monitors the BUSY input from A/D 0 and looks for the falling edge to initiate the data reading process. Each A/D device has a data latch the holds the data when BUSY is true and updates when BUSY is false. The A/D is strapped to always drive its data to the latch (CS_N = '0', RD_N = '0').

Once the falling edge of BUSY is detected, a state machine reads the four A/D data latches. The latches are read in sequence 0 to 3, for all devices whether or not the A/D is enabled. The latches are enabled on the share data bus using the ADC_OE_N[3..0] signals. Data reads are 3 cycles per device.

Error Compensation
After the data is read, samples are corrected for first-order gain and offset errors. Individual gain and offset error coefficients are provided via the **gain0.. gain3, offset0..offset3** input vectors. The *ii_offgain* component is used for the error correction, as detailed in the generic library chapter. The channel enable vector is used to mask the writes for the enabled channel samples into the *ii_offgain* component.

Data Buffering
The FIFO takes in the 16-bit error corrected samples and outputs stacked 32-bit sample pairs. The FIFO is 1K samples deep for data buffering. Data can be read by the system logic using the RDY input. FIFO level is monitored using

FIFO_RD_CNT to provide flow control information. Data is valid for the system when DVALID is true, synchronous to sys_clk.

Data is stacked in the output data word with two 16-bit samples per 32-bit word. If a single channel is used, the lower 16 bits is the older sample. The samples are always stacked from the lowest channel number enabled to the highest when more than one channel is enabled.

Number of Enabled channels	Bits 31..16	Bits 15..0
1 (e.g. channel 0)	ADC0[15..0](t)	ADC0[15..0](t-1)
2 (e.g. channel s 1, 0)	ADC1[15..0](t)	ADC0[15..0](t)
3 (e.g. channel 2,1,0)	ADC1[15..0](t) ADC0[15..0](t+1)	ADC0[15..0](t) ADC3[15..0](t)
3 (e.g. channels 3,2,1,0)	ADC1[15..0](t) ADC3[15..0](t)	ADC0[15..0](t) ADC2[15..0](t)

Test Features

The test input substitutes a ramp in place of the A/D data. This is used for data path testing during logic design. It can be used to verify that no data loss occurs due to flow control problems by checking the output for the monotonically increasing ramp data. Real A/D data can be much more difficult to check since a missing point is not easily detected.

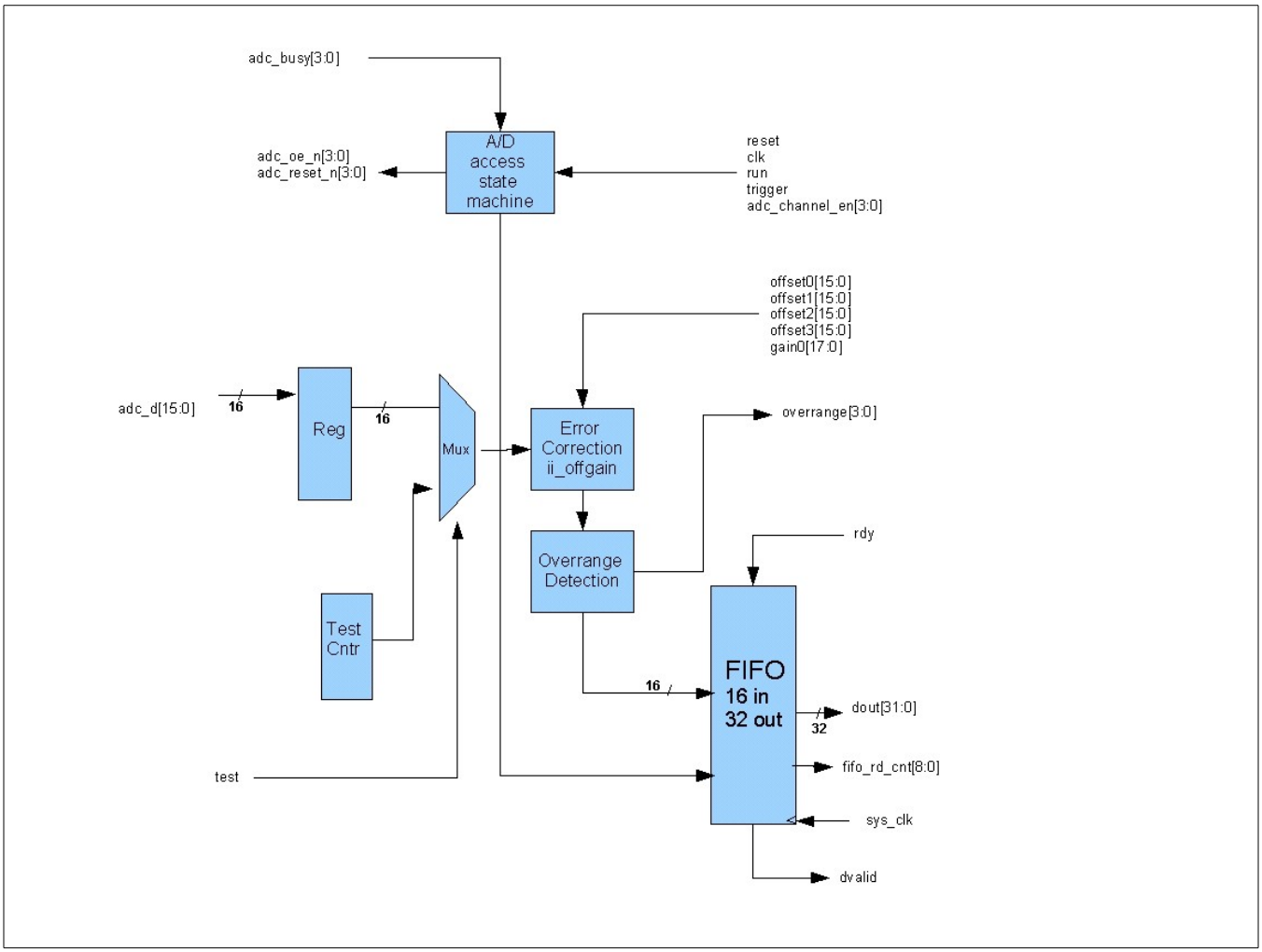


Figure 56. ii_x3_A4D4_adc Component Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	system clock (107 MHz)
fs_clk	In	sample clock (4 MHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true (synchronous to fs_clk).
channel_en[3:0]	In	Channel enables for A/Ds.
rdy	In	System is ready to receive data (FIFO read enable)
offset0[15:0]	In	A/D 0 offset for error correction
gain0[17:0]	In	A/D 0 gain for error correction
offset1[15:0]	In	A/D 1 offset for error correction
gain1[17:0]	In	A/D 1 gain for error correction
offset2[15:0]	In	A/D 2 offset for error correction
gain2[17:0]	In	A/D 2 gain for error correction
offset3[15:0]	In	A/D 3 offset for error correction
gain3[17:0]	In	A/D 3 gain for error correction
adc_reset_n[3:0]	Out	A/D resets, active low
adc_oe_n[3:0]	Out	Output enables for data latches, active low
adc_busy[3:0]	In	A/D busy. Conversion is in process when '1';
adc_d[15:0]	In	A/D data bus
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
fifo_rd_cnt[8:0]	Out	FIFO read count, synchronous to clk
test	In	Enable test mode (0= disabled)

Table 69. ii_x3_a4d4_adc Component Ports

Component: ii_x3_a4d4_dac

Supported Platforms: X3-A4D4

Source File: ii_x3_a4d4_dac, fifo_1kx32_vld.vhd, ii_offgain.vhd

Description:
This component is the interface to the four D/A converters on the X3-A4D4 XMC module. The DAC devices, Linear Technology LTC1668, are 16-bit, 50 MSPS devices. The component provides an interface to the four DAC devices for data buffering, error compensation, channel selection and providing output data. All channels are assumed to be synchronous (using the same sample clock). A periodic clock is not required.

The RUN signal is used as a reset to the component.

Data Buffering
A 32-bit input, 1K deep FIFO is used for data buffering from the system logic. Data can be written by the system logic using DAC_WR, with DAC_FIFO_CNT used to pace the data according to the available space in the FIFO.

DAC Interface
As data is received from the system logic into the input FIFO, the 32-bit words are unstacked into 16-bit samples. The data is expected to be formatted according to the enabled channels.

Number of Enabled channels	Bits 31..16	Bits 15..0
1 (e.g. channel 0)	DAC0[15..0](t)	DAC0[15..0](t-1)
2 (e.g. channel s 1, 0)	DAC1[15..0](t)	DAC0[15..0](t)
3 (e.g. channel 2,1,0)	DAC1[15..0](t) DAC0[15..0](t+1)	DAC0[15..0](t) DAC2[15..0](t)
3 (e.g. channels 3,2,1,0)	DAC1[15..0](t) DAC3[15..0](t)	DAC0[15..0](t) DAC2[15..0](t)

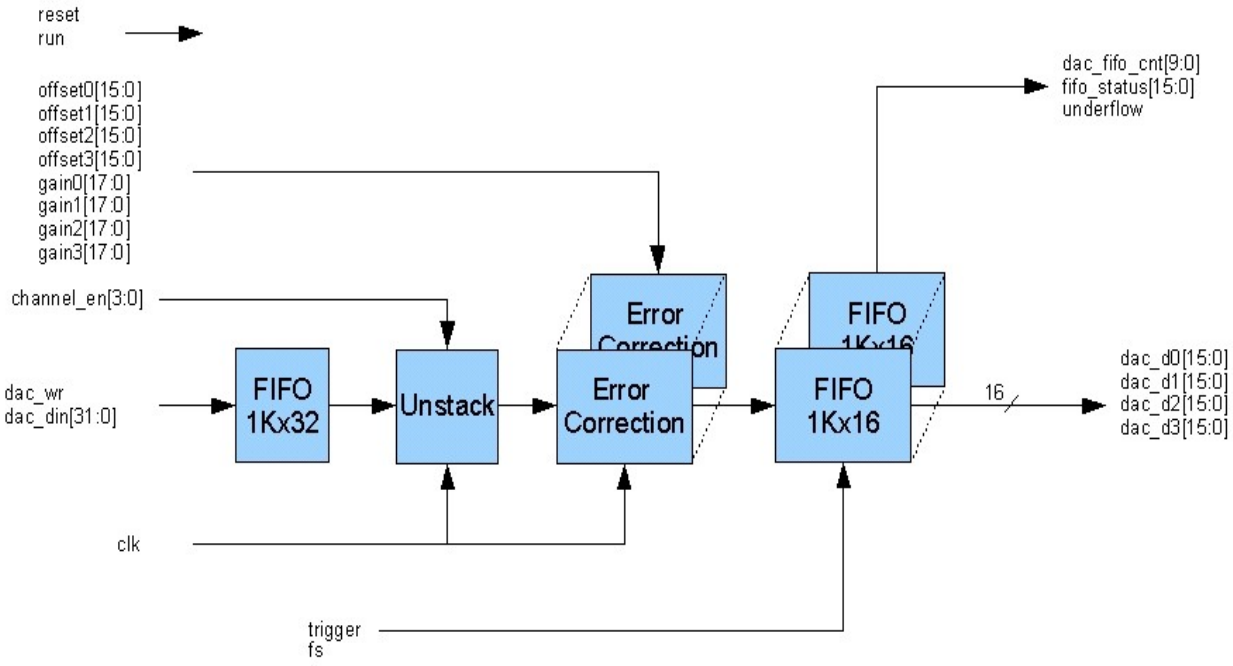
The data is unstacked and routed to error correction for that channel as space permits in the output FIFO.

Error Compensation
As each point is error corrected for first-order gain and offset errors. Individual gain and offset error coefficients are provided via the **gain0..gain3, offset0..offset3** input vectors. The *ii_offgain* component is used for the error correction, as detailed in the generic library chapter.

DAC Outputs

There is a 1Kx16 FIFO for each of the four DAC devices that hold the outbound samples. The output FIFO is read when trigger is true on each rising edge of sample clock.

This FIFO is reset to mid-scale (0x8000) so that the DAC output is zero on reset.



Port	Direction	Function
reset	In	reset
clk	In	system clock (107 MHz)
fs_clk	In	sample clock (50 MHz or less)
run	In	Enable the component to operate.
dac_wr	In	Write enable to DAC FIFO.
dac_din[31:0]	In	DAC data bus input from system logic.
channel_en[3:0]	In	Channel enables for A/Ds.
offset0[15:0]	In	DAC 0 offset for error correction
gain0[17:0]	In	DAC 0 gain for error correction
offset1[15:0]	In	DAC 1 offset for error correction
gain1[17:0]	In	DAC 1 gain for error correction
offset2[15:0]	In	DAC 2 offset for error correction
gain2[17:0]	In	DAC 2 gain for error correction
offset3[15:0]	In	DAC 3 offset for error correction
gain3[17:0]	In	DAC 3 gain for error correction
dac_d0[15:0]	Out	DAC 0 data bus (synchronous to fs_clk).
dac_d1[15:0]	Out	DAC 1 data bus (synchronous to fs_clk).
dac_d2[15:0]	Out	DAC 2 data bus (synchronous to fs_clk).
dac_d3[15:0]	Out	DAC 3 data bus (synchronous to fs_clk).
dac_fifo_cnt[9:0]	Out	FIFO write count, synchronous to clk
fifo_status[15:0]	Out	FIFO status (not used)
underflow	Out	DAC FIFO underflow. (1 = error).
trigger	In	Acquire data when true (synchronous to fs_clk).

Table 70. ii_x3_a4d4_dac Component Ports

X3-DIO Logic Components

These components are only used in X3-DIO module logic.

Component: ii_dio_out

Supported Platforms: X3-DIO

Source File: ii_dio_out.vhd, fifo_1kx32_async_vld.vhd, ii_dio_out_buffer.vhd

Description:
This component is the output interface for the front panel DIO on the X3-DIO XMC module. There are 64-bits for the front panel IO, organized as 8 bytes. Each byte is treated as a "channel"that may be individually enabled and configured as output. The component takes in data from the system, unstacks the enabled channels from the data words and enqueues them into an output FIFO for each byte. This is done so that the data is optimally packed into the incoming data words for highest data rates.

All channels are assumed to be synchronous (using the same sample clock). A periodic clock is not required.

The **RUN** signal is used as a reset to the component.

Data Buffering
A 32-bit input, 1K deep FIFO is used for data buffering from the system logic. Data can be written by the system logic using **WREN**, with **RDY** or **FIFO_WR_CNT** used to pace the data according to the available space in the FIFO.

DIO Out Interface
As data is received from the system logic into the input FIFO, the 32-bit words are unstacked into bytes for the enabled channels. A state machine is used to compute the number of enabled channels, a channel list consisting of the enabled channel numbers, and finally a channel pointer vector that shows the order of the enabled channels in the incoming words as will occur in a stream of 32-bit words from the input. Since the incoming words are 32-bit (4 bytes per word), the channel pointer (**ch_ptr**) length is the least common multiple between the number of enabled channels (**ch_lcm**+1) and 8 bytes for output. The channel pointer vector is then used as a byte enable list for each of the output bytes. The byte buffer preserves the time ordering of the data.

Number of Enabled channels	Data Word	Bits 31..24	Bits 23..16	Bits 15..8	Bits 7..0
1 (e.g. channel 0)	0	Ch0[7..0](t+3)	Ch0[7..0](t+2)	Ch0[7..0](t+1)	Ch0[7..0](t+0)
	1	Ch0[7..0](t+7)	Ch0[7..0](t+6)	Ch0[7..0](t+5)	Ch0[7..0](t+4)
2 (e.g. channels 6, 0)	0	Ch6[7..0](t+1)	Ch0[7..0](t+1)	Ch6[7..0](t+0)	Ch0[7..0](t+0)
	1	Ch6[7..0](t+3)	Ch0[7..0](t+3)	Ch6[7..0](t+2)	Ch0[7..0](t+2)

3 (e.g. channels 7,5,4)	0	Ch4[7..0](t+1)	Ch7[7..0](t+0)	Ch5[7..0](t+0)	Ch4[7..0](t+0)
	1	Ch5[7..0](t+2)	Ch4[7..0](t+2)	Ch7[7..0](t+1)	Ch5[7..0](t+1)
	2	Ch7[7..0](t+3)	Ch5[7..0](t+3)	Ch4[7..0](t+3)	Ch7[7..0](t+2)

Data flow through the component is paced according to the space available in the output FIFOs in each of the *ii_dio_buffer* component. Each byte buffer has a **byte_rfd** output that indicates when it can receive data. The state machine controlling data flow requires that all byte buffer **byte_rfd** are true (all are ready) and the input FIFO must have data to initiate data flow through the component.

DIO Out Outputs

There is a 1Kx8 FIFO for each of the eight DIO Out channels that hold the outbound samples. The output FIFO is read when trigger is true on each rising edge of sample clock.

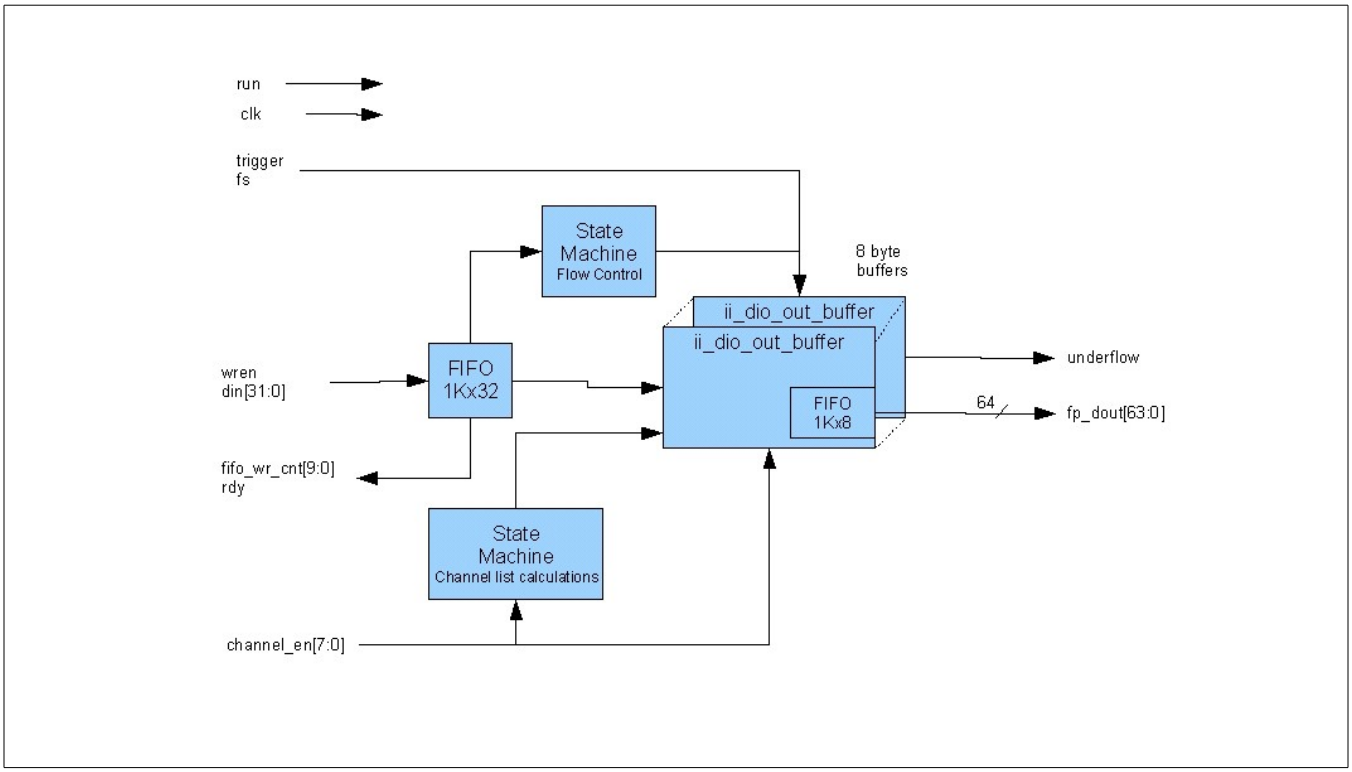


Figure 58. ii_x3_dio_out Component Block Diagram

Port	Direction	Function
clk	In	System clock (107 MHz)
fs_clk	In	Sample clock (50 MHz or less)
run	In	Enable the component to operate.
trigger	In	Trigger enables data flow from the output when true.
channel_en[7:0]	In	Channels for each data byte. Output is enabled for that byte when the channel_en bit is true.
rdy	Out	When RDY is true, the input FIFO has space for at least 256 points.
fp_dout[63:0]	Out	Output bits
wren	Out	Write enable to input FIFO.
din[31:0]	In	Input data bus
fifo_wr_cnt[9:0]	Out	Input FIFO write count
underflow	Out	DAC FIFO underflow. (1 = error).
test	In	Not used.
underflow	Out	Output FIFO underflow. (1 = error).

Table 71. ii_x3_dio_out Component Ports

Component: ii_dio_out_buffer

Supported Platforms: X3-DIO

Source File: ii_dio_out_buffer.vhd, fifo_1k_32i_8o.vhd, fifo_16x8.vhd

Description:
This component receives 32-bit data words, extracts the bytes for this channel and creates a data stream. The logic parses the input 32-bit words according to the byte enables and then packs them into a 32-bit word through successive shifting steps. The resulting 32-bit word is a packed stream of data bytes, ordered in time, that is held in an output FIFO. The output FIFO is 8-bit output that is read on each sample clock when trigger is true.

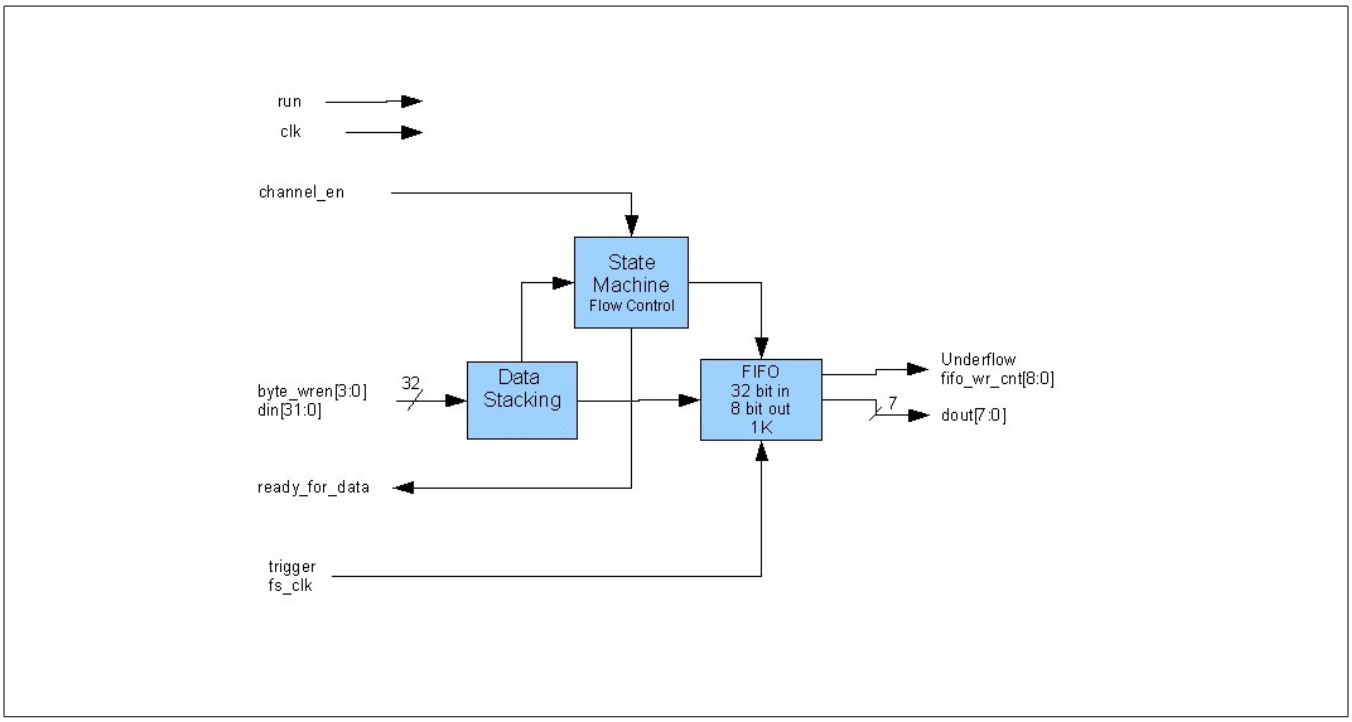


Figure 59. ii_dio_out_buffer Component Block Diagram

Port	Direction	Function
run	In	Enable the component to operate.
clk	In	System clock (107 MHz)
channel_en	In	Channel enable
fs_clk	In	Sample clock (50 MHz or less)
run	In	Enable the component to operate.
byte_wren[3:0]	In	Byte write enables. Indicates which bytes of the 32-bit input word are for this buffer.
din[31:0]	In	Input data bus
dout[7:0]	Out	Output bits
ready_for_data	Out	Component is ready to receive data when true.
trigger	In	Trigger enables data flow from the output when true.
ready_for_data	Out	Component is ready to receive data when true.
fifo_wr_cnt[8:0]	Out	Input FIFO write count
underflow	Out	Output FIFO underflow. (1 = error).

Table 72. ii_dio_out_buffer Component Ports

Component: ii_x3_dio_in

Supported Platforms: X3-DIO

Source File: ii_x3_dio_in, fifo_1k_64i_32o.vhd, fifo_16x8.vhd, fifo_1kx64_vld

Description:
This component is the interface to the front panel DIO inputs on the X3-DIO XMC module. This component acquires the 64-bit digital input port on the front panel on each sample clock rising edge when trigger is true. All bits are sampled simultaneously. Data is captured into a 64-bit FIFO from the input pins. The 64-bits are treated as 8 channels, each of the bytes is considered a "channel". The channel enables (channel_en) correspond to the 8 bytes of input data. A state machine calculates the channel list and channel order according to the channel enable list. The channel list is then used to pack the bytes into 32-bits words consisting of only the enabled byte data. These bytes are packed sequentially by channel number. For example, if bytes 0,1,6 are enabled as inputs, then the data is packed in this order:

word0 = [B0 B6 B1 B0]

word1 = [B1 B0 B6 B1]

The input to the component uses the sample clock to capture the data. The remaining data processing uses system clock. For the standard logic, the system clock is 107 MHz.

The RUN signal is used as a reset. When **RUN** is false, the component is reset. This allows the logic to reset the DIO In without a complete system reset.

Data Buffering
There is a 1Kx64 FIFO on the input used to capture the data.

The component also has a 1kx32 FIFO that supplies the data to system logic. Data is enabled to the system when **rdy** is true. Each data word has a **dvalid** for when the data is valid. The data path is reset when run is false ('0').

Data for the enabled channels is stacked in the output data word with four channels per 32-bit word. The channels are always stacked from the lowest channel number enabled to the highest when more than one channel is enabled.

Number of Enabled channels	Data Word	Bits 31..24	Bits 23..16	Bits 15..8	Bits 7..0
1 (e.g. channel 0)	0	Ch0[7..0](t+3)	Ch0[7..0](t+2)	Ch0[7..0](t+1)	Ch0[7..0](t+0)
	1	Ch0[7..0](t+7)	Ch0[7..0](t+6)	Ch0[7..0](t+5)	Ch0[7..0](t+4)
2 (e.g. channels 6, 0)	0	Ch6[7..0](t+1)	Ch0[7..0](t+1)	Ch6[7..0](t+0)	Ch0[7..0](t+0)
	1	Ch6[7..0](t+3)	Ch0[7..0](t+3)	Ch6[7..0](t+2)	Ch0[7..0](t+2)
3 (e.g. channels 7,5,4)	0	Ch4[7..0](t+1)	Ch7[7..0](t+0)	Ch5[7..0](t+0)	Ch4[7..0](t+0)

	1	Ch5[7..0](t+2)	Ch4[7..0](t+2)	Ch7[7..0](t+1)	Ch5[7..0](t+1)
	2	Ch7[7..0](t+3)	Ch5[7..0](t+3)	Ch4[7..0](t+3)	Ch7[7..0](t+2)

Test Features

The test input substitutes a ramp in place of the input data. This is used for data path testing during logic design. It can be used to verify that no data loss occurs due to flow control problems by checking the output for the monotonically increasing ramp data.

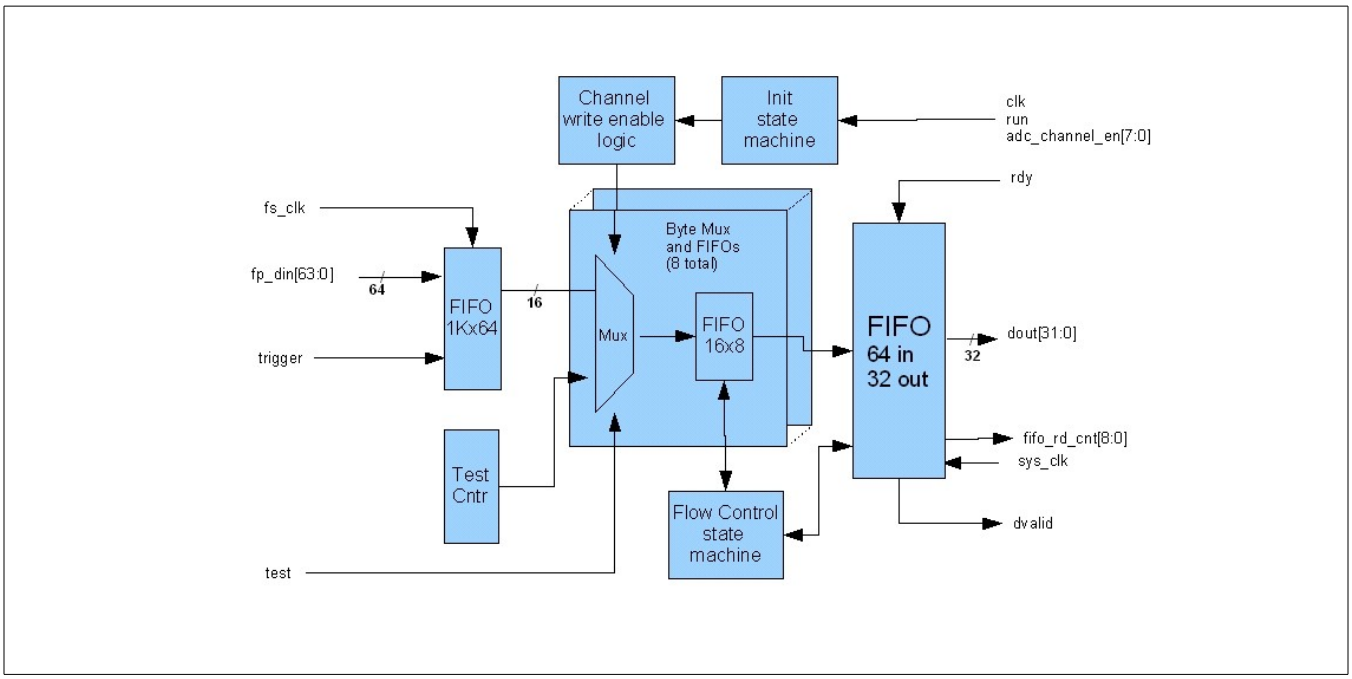


Figure 60. ii_x3_dio_in Component Block Diagram

Port	Direction	Function
clk	In	system clock (107 MHz)
fs_clk	In	sample clock (100 MHz or less)
run	In	Enable the component to operate.
trigger	In	Acquire data when true (synchronous to fs_clk).
channel_en[7:0]	In	Channel enables for input channels (bytes)
rdy	In	System is ready to receive data (FIFO read enable)
fp_dio[63:0]	In	Front panel DIO input bits
adv_reset_n[3:0]	Out	A/D resets, active low
dvalid	Out	Data is valid when true
dout[31:0]	Out	Data output from the FIFO
fifo_rd_cnt[8:0]	Out	FIFO read count, synchronous to clk
overflow	Out	FIFO overflow error bit
test	In	Enable test mode (0= disabled)

Table 73. ii_x3_dio_in Component Ports

Generic Logic Library

These components are used in all X3 modules for basic functionality. In many cases, the component has been designed to use Spartan3/3A DSP logic features used in the X3 family.

ii_cmd_reg

Source Files: ii_cmd_reg.vhd, ii_command_bus.vhd

Description:

This component is the command channel interface to the PCIe interface controller. The command channels provides PCIe bus accesses to the application logic over a dedicated link, outside of the higher speed primary data path. The ii_cmd_reg component receives a access word from the PCI bus controller containing the read/write, address and data for the transaction. The component decodes this word and provides the address, data and a read or write strobe to the application logic.

Command Channel Control Signals
The command channel provides an address, data, read and write strobes to decode an access. This is a pure reflection of the PCI bus so that accesses from the host to BAR1+address. The **rd_stb** and **wt_stb** are the read and write enables synchronous to the data(dout) and address (aout).

For convenience, the ii_cmd_reg component also decodes 128 write addresses and 128 read addresses for expansion. These are provided in the **rd_stb_dec** and **wt_stb_dec** output vectors.

Decoding a Command Channel Write Access
A command channel write access can be decoded using either the address and write strobe, or the pre-decoded write enables. An example of this decoding is

```
process (reset, sys_clk, aout, wt_stb)
begin
        if (reset = '1') then
                cmd_register <= (others => '0');
        elsif (rising_edge(sys_clk)) then
                if (aout(15 downto 0) = X"1000" and wt_stb = '1') then
                        cmd_register <= dout;
                end if;
        end if;
end process;
```

creates a 32-bit register at BAR1+ X"1000".

If you want to use the predecoded strobes, then here is an example

```
process (reset, sys_clk, wt_stb_dec)
begin
        if (reset = '1') then
                cmd_register <= (others => '0');
        elsif (rising_edge(sys_clk)) then
                if ( wt_stb_dec(33)  = '1') then
                        cmd_register <= dout;
                end if;
        end if;
```

```
    end process;
```

creates a 32-bit register at BAR1+X”21” .

Decoding a Command Channel Read Access

To read data from the command channel, it is necessary to drive the status input with the data at the command channel address when read is true. A command channel read access can be decoded using either the address and read strobes, or the pre-decoded read enables. An example of this decoding is

```
-- data mux for command channel reads
gen_cmd_rdback: for i in 0 to 127 generate
    cmd_din <= cmd_status(i) WHEN ( CONV_INTEGER(cmd_aout(6 downto 0)) = i ) else (others => 'Z');
 end generate;
```

read a 32-bit register at PCIe address BAR1+ aout.

For devices requiring a long readback time, greater than 2 **cmd_clkx** periods, can use the **cmd_rdy** input to hold the readback until the data is ready. The command channel will hold the access until the data is ready. The application logic should not assert ready until the data is valid. The wait states should be limited to no more than 16 system clocks.

Command Channel Registers

A 128 element array of 32-bit output registers is provided with the number of registers specified by the ii_x3_pkg.vhd file. These are decoded at BAR1+0..127 . These 32-bit registers are reset to X”00000000” by a system reset. All output registers are re-clocked to the sys_clk domain.

The array must be defined in the ii_x3_pkg and included in the project for this component.

Summary of Command Channel Decodes

Decode	PCI Address	Read Decode	Write Decode	Output Register
0	BAR1 + X”0”	rd_stb(0)	wt_stb(0)	ctl_reg(0)
1	BAR1 + X”4”	rd_stb(1)	wt_stb(1)	ctl_reg(1)
2	BAR1 + X”8”	rd_stb(2)	wt_stb(2)	ctl_reg(2)
3..31	BAR1 + X”C”.. BAR1 + X”1F”	rd_stb(3).. rd_stb(31)	wt_stb(3)... wt_stb(31)	ctl_reg(3)... ctl_reg(31)
32..127	BAR1 + X”20”.. BAR1 + X”7F”..	rd_stb(32).. rd_stb(127)	wt_stb(32)... wt_stb(127)	ctl_reg(32)... ctl_reg(127)

Table 1: ii_cmd_reg Memory Map

The memory decoding for the command channel is relative to BAR1 of the PCI mapping for the XMC. Since the registers are 32-bit, the decodings are spaced 4 apart in the memory map.

Important Implementation Note:
It is recommended that the compile tools be allowed to ignore hierarchy on this component so that unused registers are optimized out. Most registers are sparsely populated, so this results less logic.

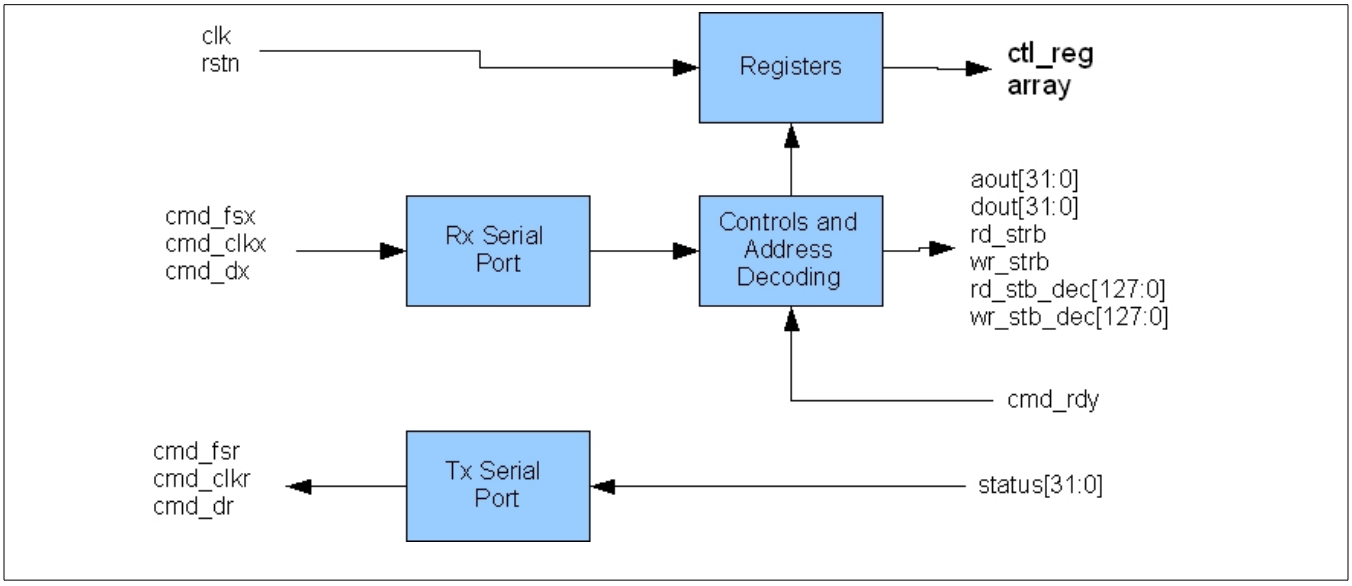


Figure 61. ii_cmd_reg Component

Port	*Direction*	*Function*
Rstn	in	System reset, active low
clk	In	System clock
cmd_clkx	In	command transmit clock from PCI interface controller
cmd_dx	In	command transmit data from PCI interface controller
cmd_fsx	In	command transmit frame from PCI interface controller
cmd_clkr	Out	command receive clock to PCI interface controller
cmd_dr	Out	command receive data to PCI interface controller
cmd_fsr	Out	command receive frame to PCI interface controller
aout[31:0]	Out	command address
dout[31:0]	Out	command data
wt_stb	Out	command write strobe
rd_stb	Out	command read strobe
wt_stb_dec[127:0]	Out	write strobe decodes
rd_stb_dec[127:0]	Out	read strobe decodes
ctl_reg	Out	array of control registers, see 'ii_x3_pkg.vhd'
status[31:0]	In	Read data input
cmd_rd_rdy	In	Read ready. Read completes when this is true.

Table 74. ii_cmd_reg Component Ports

ii_data_mover

Source files: ii_data_mover.vhd

Description:

This component is used to move data in a system, usually between two FIFOs. The component calculates the amount of data to move by comparing the amount of data in the source FIFO to the space available in the destination FIFO (**TO_FIFO_THRESH** – **to_fifo_count**) and moves the LESSER of the two.

When the from FIFO has enough data, as indicated by the **from_FIFO_count**, the data flow state machine generates the control signals to read from the from FIFO and write to the to FIFO. The data width is specified by the data_width generic and must be equal on both data paths.

Data mover moves data between two devices, such as FIFOs, using flow control.

The data mover component has two clocks of overhead per transfer. For example, a move of 8 points would require 10 clocks, or is 80% efficient.

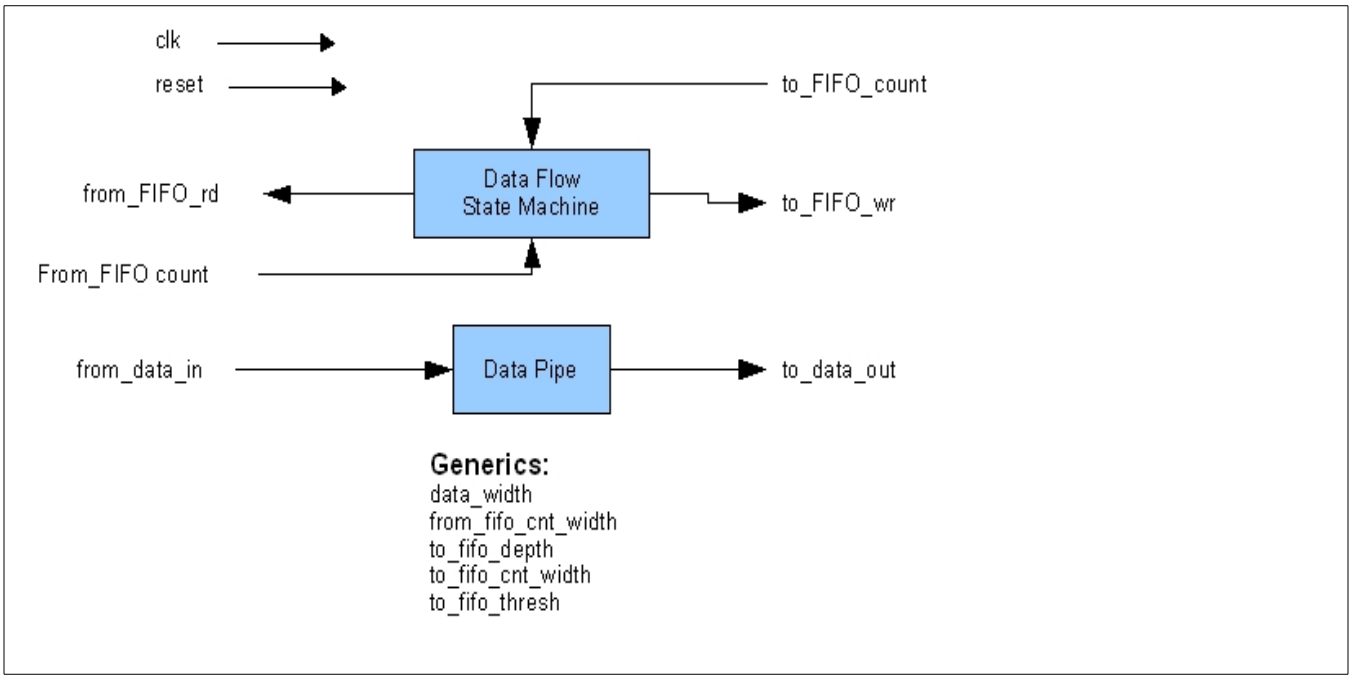


Figure 62. ii_data_mover Component

Generic	*Type*	*Function*
DATA_WIDTH	Integer	Width of the data path in bits, default = 16
FROM_FIFO_CNT_WIDTH	Integer	Width of the source FIFO count, count = 10
TO_FIFO_DEPTH	Integer	Size of the destination FIFO, default = 1024
TO_FIFO_CNT_WIDTH	Integer	Width of the destination FIFO count, count = 10
TO_FIFO_THRESH	Integer	Full threshold of the destination FIFO, default = 1000

Table 75. ii_data_mover Component Generics

Port	*Direction*	*Function*
clk	In	Clock input
reset	In	reset
from_fifo_count(FROM_F IFO_CNT_WIDTH-1 downto 0)	In	Source FIFO data count
to_fifo_count(to_FIFO_C NT_WIDTH-1 downto 0)	In	Destination FIFO data count
from_fifo_rd	Out	Source FIFO read enable
to_fifo_wr	Out	Destination FIFO write enable
from_data_in(data_width -1 downto 0)	In	Source data input
to_data_out(data_width-1downto 0)	Out	Destination data output

Table 76. ii_data_mover Component Ports

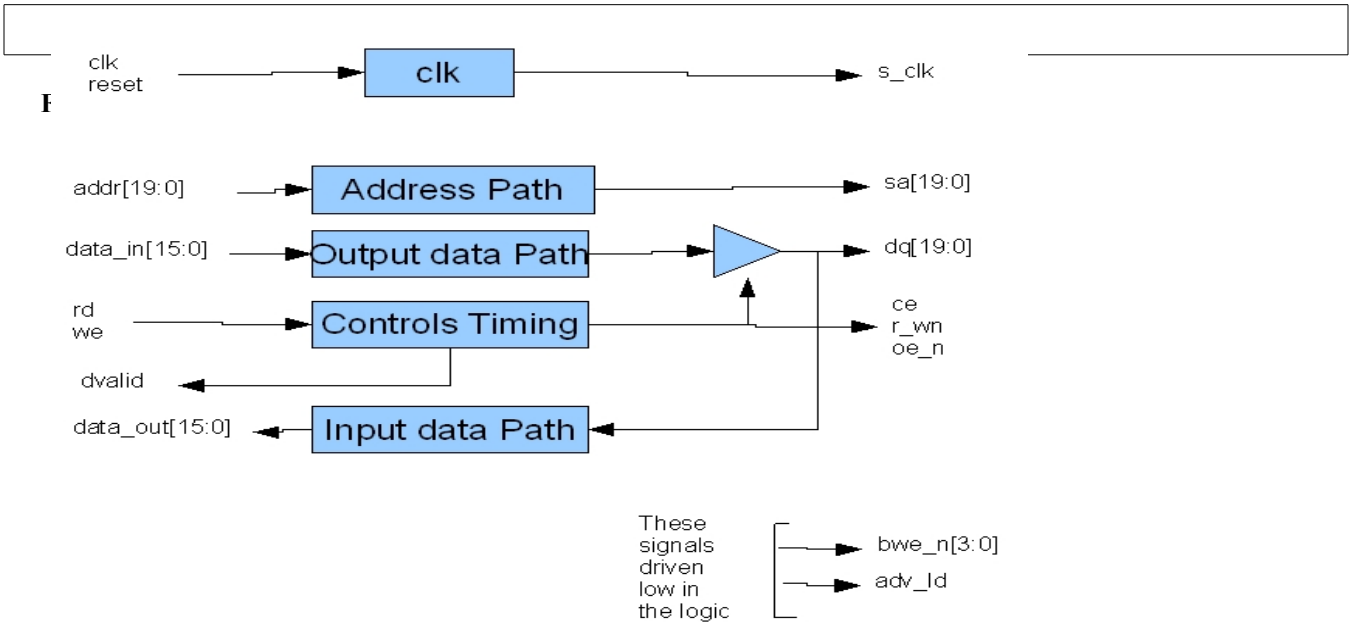
ii_sram_intf

Source file: ii_sram_intf.vhd

Description:

This component provides an interface from the application logic to synchronous burst SRAM memories. These memory devices are frequently used as buffer memory for logic applications.

The SBSRAM controller interface component supports a 20-bit address bus and 32-bit data path. The user interface is requires a clock, 20-bit address, read/write control for the device, and data if writing. The component does the transaction with the device. Continuous accesses to the RAM are allowed. The address should be loaded each cycle with the data.



Port	*Direction*	*Function*
reset	In	Asynchronous reset.
addr[19:0]	In	SBSRAM address
clk	In	Clock. All signals are synchronous to this clock. The SRAM runs at this clock rate.
din[data_bits-1 downto 0]	In	Input data bus.
rd	In	Read enable input
we	In	Write enable input
dvalid	Out	Data is valid on the output bus when true.
dout[data_bits-1 downto 0]	Out	Output data bus.
sa[19:0]	Out	SBSRAM address bits
ce	Out	SBSRAM chip enable
s_clk	Out	clock to SBSRAM
r_wn	Out	SBSRAM read/write(low) control
adv_ld	Out	SBSRAM synchronous address advance/load(low)
bwe_n[3:0]	Out	SBSRAM Byte enables. Bit 0 controls byte 0 (bits 7..0) and so forth.
oe_n	Out	SBSRAM output enable, active low.
dq[15:0]	Inout	SBSRAM data bus

Figure 64. ii_sram_intf Component Ports

The SRAM device is Cypress CY7C1383 (or equivalent) device simulation model is cy7c1383d.vhd. Each X3 SD or SDF module has 2 of these SRAMs that both use this component for control. The SRAM used for data buffering uses this component for its low-level interface to the SRAM under its ii_mq_sram multi-queue data buffer control.

The maximum clock rate is 80 MHz for X3 modules. The clock must be free-running.

The data path to the X3 SBSRAM is 16-bits. The generic data_bits is specified in the ii_x3_pkg package.

Using ii_sram_intf

Read and write accesses to the SRAM are shown below. Both reads and writes may be bursts of addresses, non-sequential in memory. The accesses may be back-to-back since the SRAM is ZBT (Zero Bus Turnaround) capable. Data read from the SRAM has a 4 clock latency. When the data is valid from a read, the **dvalid** signal is true. All signals are synchronous to the clock.

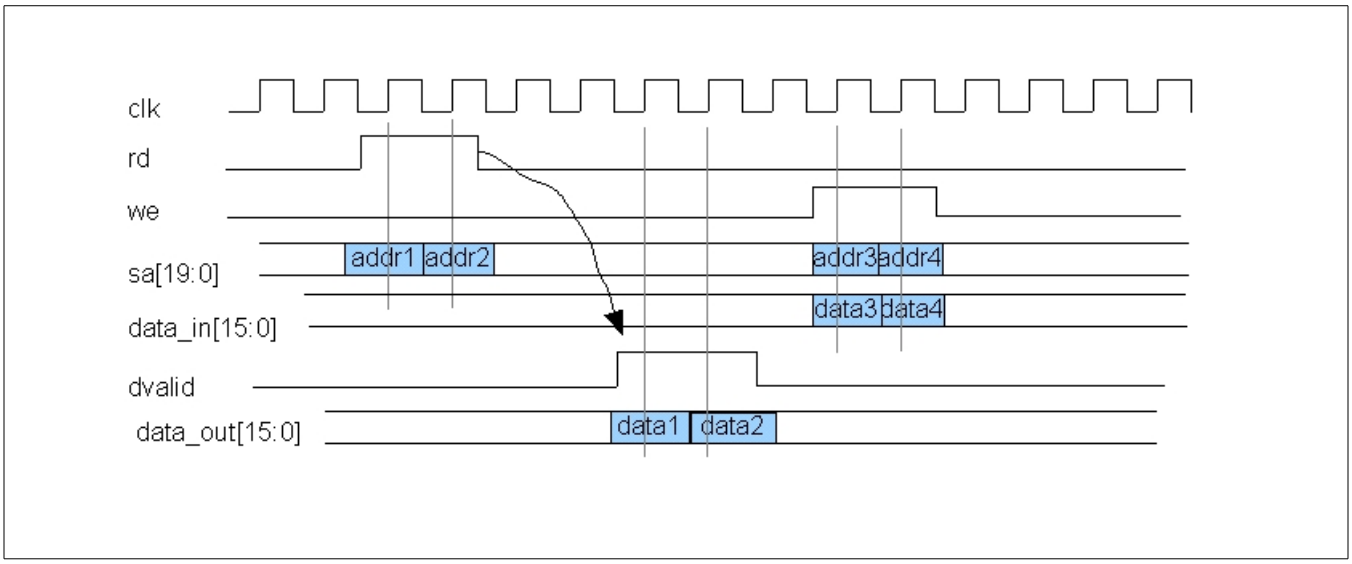


Figure 65. ii_sram_intf Access Timing

ii_sram32_intf

Source file: ii_sram32_intf.vhd

Description:

This component provides an interface from the application logic to synchronous burst SRAM memories. These memory devices are frequently used as buffer memory for logic applications.

The SBSRAM controller interface component supports a 19-bit address bus and 32-bit data path. The user interface is requires a clock, 20-bit address, read/write control for the device, and data if writing. The component does the transaction with the device. Continuous accesses to the RAM are allowed and the address should be loaded each cycle with the data. Reads from the SRAM are 4 cycles latent and data is returned with a valid (**dvalid**) for each read.

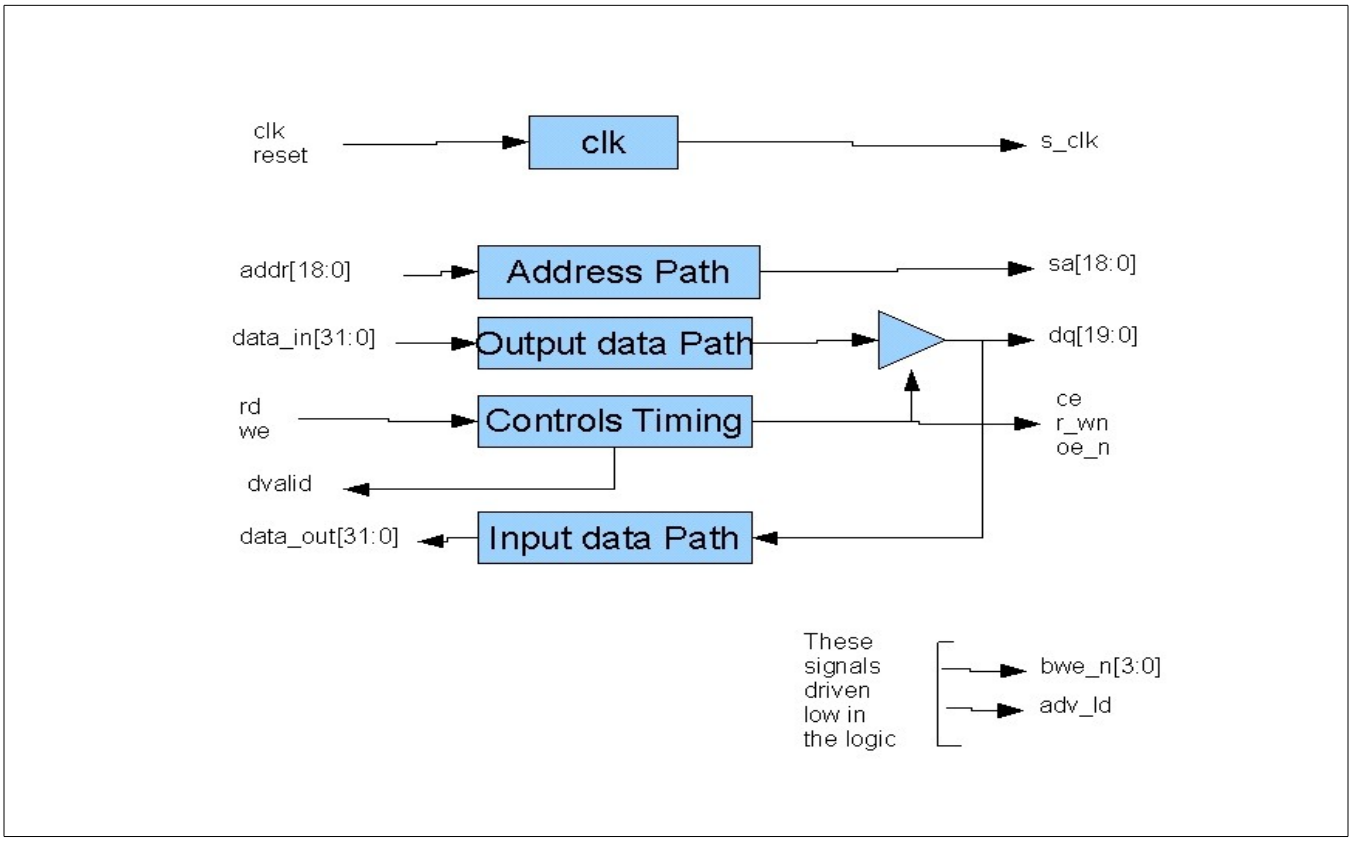


Figure 66. ii_sram32_intf Component

Port	*Direction*	*Function*
reset	In	Asynchronous reset.
addr[18:0]	In	SBSRAM address
clk	In	Clock. All signals are synchronous to this clock. The SRAM runs at this clock rate.
din[31:0]	In	Input data bus.
rd	In	Read enable input
we	In	Write enable input
dvalid	Out	Data is valid on the output bus when true.
dout[31:0]	Out	Output data bus.
sa[18:0]	Out	SBSRAM address bits
ce	Out	SBSRAM chip enable
s_clk	Out	clock to SBSRAM
r_wn	Out	SBSRAM read/write(low) control
adv_ld	Out	SBSRAM synchronous address advance/load(low)
bwe_n[3:0]	Out	SBSRAM Byte enables. Bit 0 controls byte 0 (bits 7..0) and so forth.
oe_n	Out	SBSRAM output enable, active low.
dq[31:0]	Inout	SBSRAM data bus

Table 77. ii_sram32_intf Component Ports

The SRAM device is Cypress CY7C1371 (or equivalent) device simulation model is cy7c1371kd.vhd. Each X3 module, except X3-SD and X3-SDF, has 2 of these SRAMs that both use this component for control. The SRAM used for data buffering uses this component for its low-level interface to the SRAM under its ii_mq_sram32 multi-queue data buffer control.

The maximum clock rate is 100 MHz for X3 modules. The clock must be free-running.

The data path to the X3 SBSRAM is 32-bits. The generic data_bits is specified in the ii_x3_pkg package.

Using ii_sram32_intf

Read and write accesses to the SRAM are shown below. Both reads and writes may be bursts of addresses, non-sequential in memory. The accesses may be back-to-back since the SRAM is ZBT (Zero Bus Turnaround) capable. Data read from the SRAM has a 4 clock latency. When the data is valid from a read, the **dvalid** signal is true. All signals are synchronous to the clock.

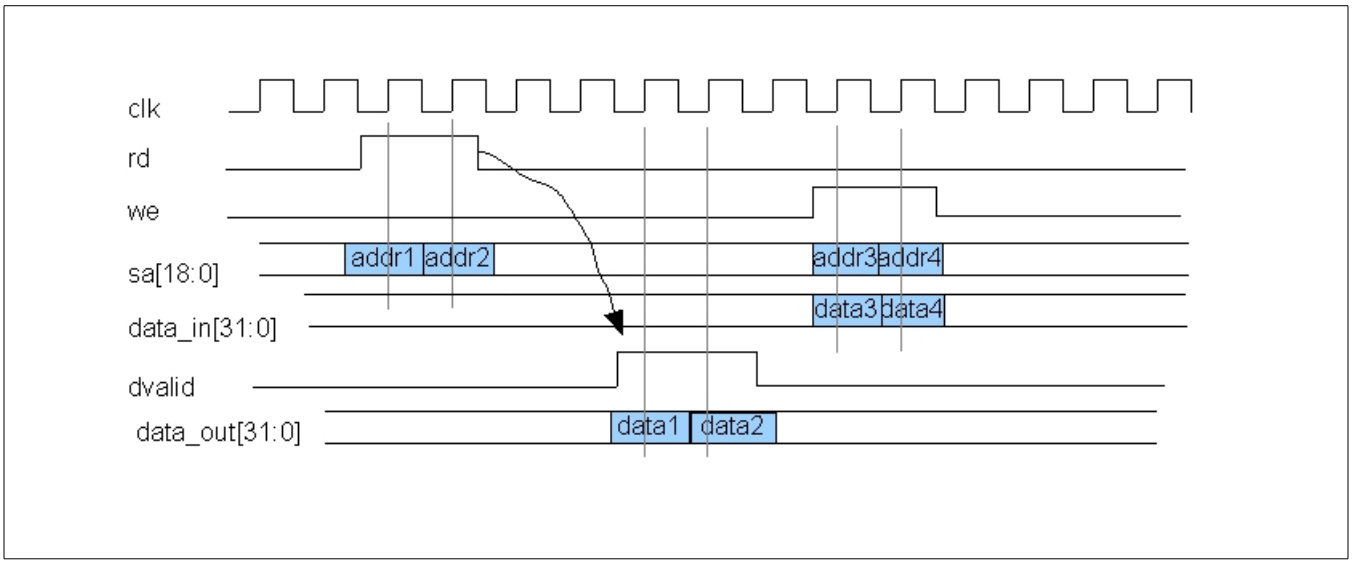


Figure 67. ii_sram32_intf Access Timing

ii_packetizer

Source file: ii_packetizer.vhd

Description:

The packetizing component forms data streams into packets by attaching a header to a data payload. The primary use of these packets is to transfer data to the host using the Velocia PCI controller. Each data packet has a two word header, 32-bits each, preceding the data. The packets are programmable in size and for their other routing information.

During operation, the packetizer scans the number of input channels and in a round robin and creates packets for the channels that are ready. Each channel has its packets built with the header information for that channel and the data payload attached to the header. The packet is transmitted as it is built to the destination, there is no data storage in the packetizing component.

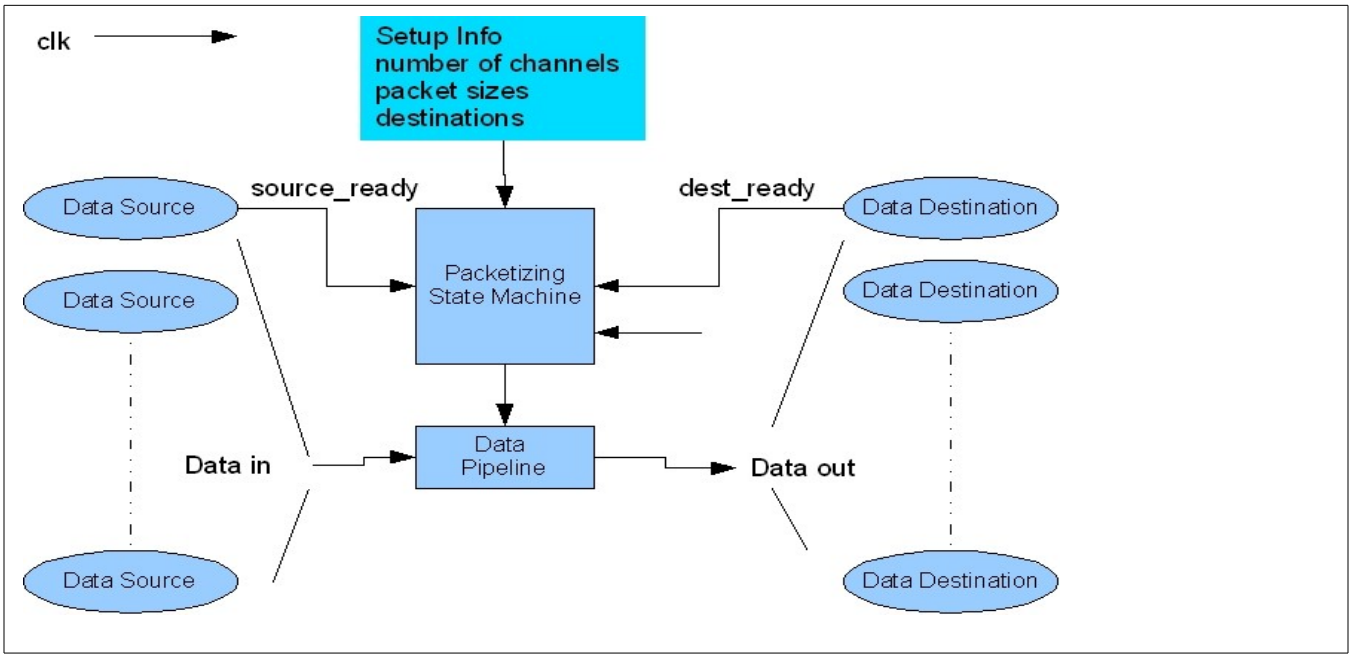


Figure 68. ii_packetizer Block Diagram

The component reads data from **num_channels** of data sources for the packet_size given and gives out a packet with a header. The data width is specified by data_width; input and output are identical in size. The data sources must provide data continuously when channel_rd() is true; the data destinations must sink data continuously when the channel_wr() is true. The status of the source and destination devices is required by the **src_rdcnt()** and **dest_wrcnt()** to allow the data movement to occur. No movement occurs if adequate room for the packet is not available. Sources and destinations for the packetizer are usually FIFOs in the logic, as in the FrameWork Logic.

Format of the packet is a two dword header, followed by a data payload. The header format is

bits[31:24] = peripheral device number

bits[23:0] = packet size including header in dwords

dword = 32 bit word

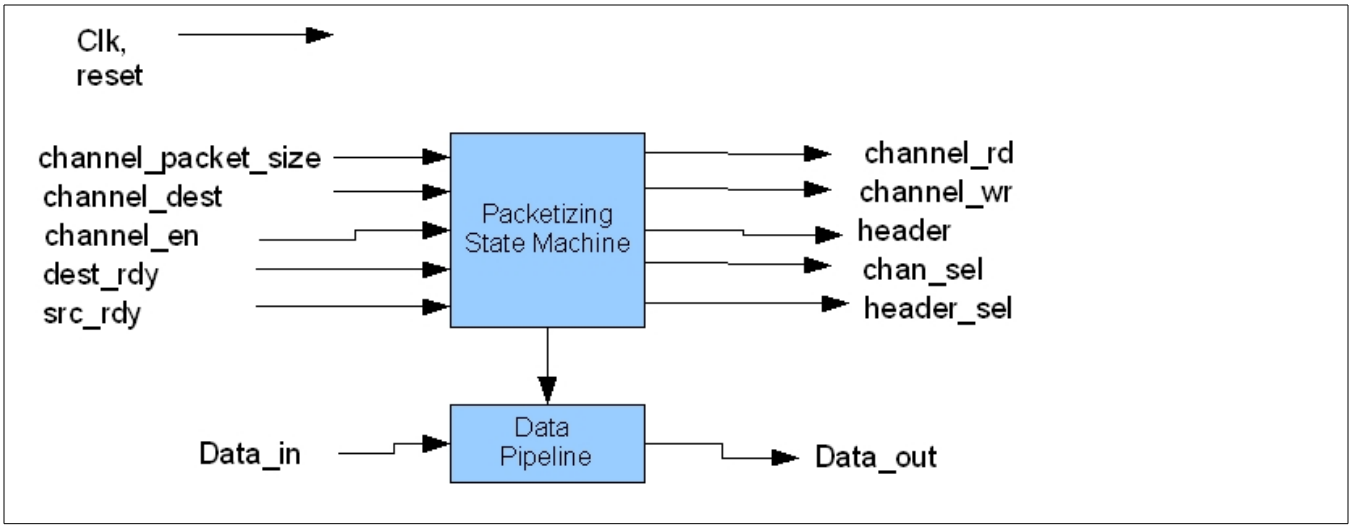


Figure 69. ii_packetizer Component

Generic	*Type*	*Function*
max_packet_size	Integer	The maximum size of a packet, default = 8

Table 78. ii_packetizer Generic Ports

Port	*Direction*	*Function*
reset	In	Asynchronous reset
clk	In	System clock
data_in	In	Channel data input bus; width is defined in ii_x3_pkg.vhd
data_out	Out	Channel data output bus; width is defined in ii_x3_pkg.vhd
channel_en	In	Channel enables vector (num_channels-1 downto 0)
channel_dest	In	Channel destinations vector (num_channels-1 downto 0)
channel_packet_size	In	The array of packet sizes for each channel
src_rdy	In	source FIFO ready vector (num_channels-1 downto 0)
dest_rdy	In	destination FIFO ready vector (num_channels-1 downto 0)
channel_rd	Out	channel source FIFO read vector(num_channels-1 downto 0)
channel_wr	Out	channel destination FIFO write vector (num_channels-1 downto 0)
header	Out	packet header out (data_width-1 downto 0)
chan_sel	Out	channel mux selects (mux_addr_width-1 downto 0)
header_sel	Out	channel mux header select

Table 79. ii_packetizer Component Ports

ii_deframer

Source file: ii_deframer.vhd

Description:

This component is used to receive packets route them to destination in the logic. The number of devices in the system is defined in the ii_x3_pkg package file as **num_pd**. Packets can be up to 2^{23} in size, including the 2 word header.

The deframer component parses incoming packets and routes them to the peripheral device number (PDN) embedded in the header. Data is pulled from the source FIFO, is stripped of its header, and written to a destination device. Packets must have a two-word header followed by the data payload.

Dword #	Description
0	Header 1: PDN (bits 31..24) & Packet Size N (bits 23..0)
1	Header 2: 0x00000000
2..N	data

The deframer uses the PDN to route the packet data payload to the specified system device. The 8-bit PDN is specified in the logic design to equate to a physical device by defining the mapping of the PDN to the destination devices. The **pd_addr** array defines the PDN of each physical device so that for example **pd_addr(0)** defines the PDN used by physical device 0.

The header also gives the packet size for the deframer to use in data movement. The deframer state machine read the packet header and then transfers the data payload to its destination as defined by the PDN mapping. The state machine is idle until a minimum packet size is received (at least 6 words), then pulls off the two header words. The packet size, taken from the first header word, is then used to move the data points, as available from the source to the destinations. This data moves are done by computing the maximum move that can be performed which is the minimum of the number of points in the source, how much space is available in the destination, and the number of points remaining in the packet. This process is repeated until the points are all moved for the packet.

Most implementations use FIFOs for the source and destination of the data. This allows the deframer to efficiently service multiple devices and decouple their data flow requirements.

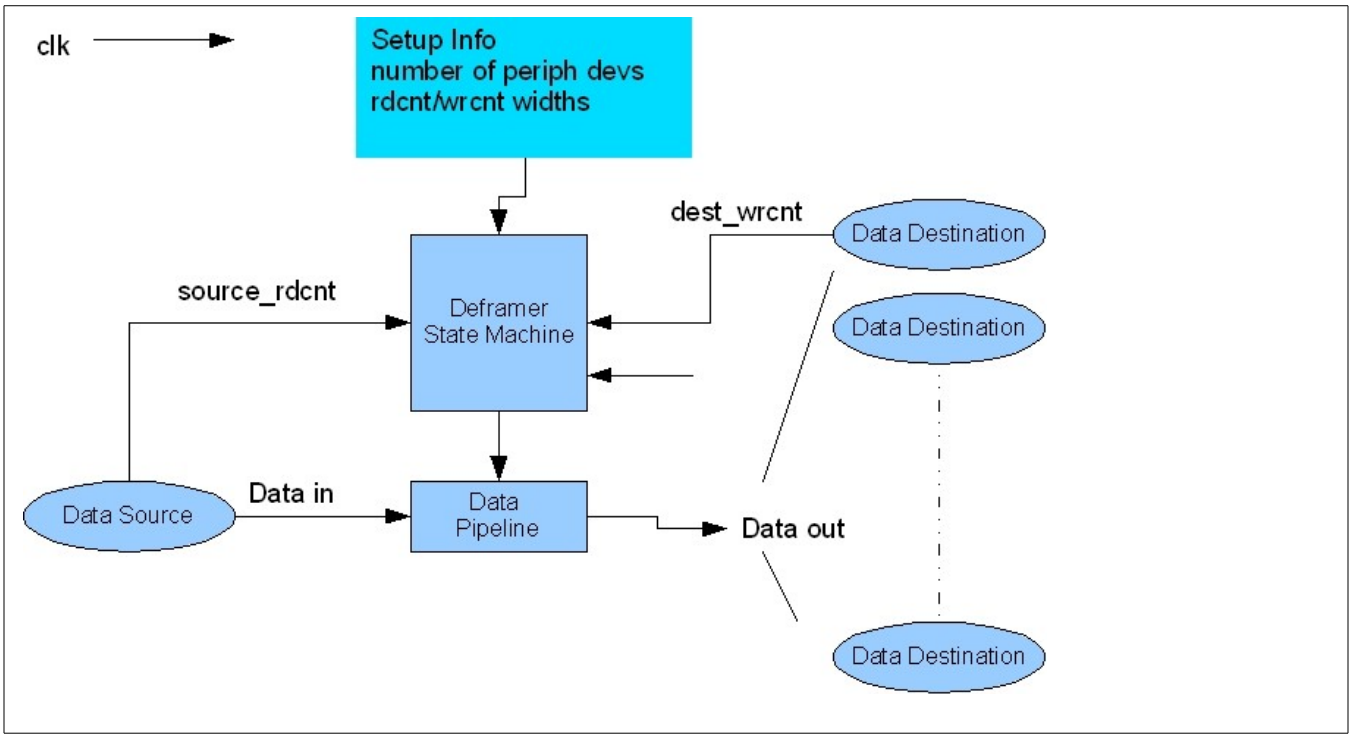


Figure 70. ii_deframer Component

Port	*Direction*	*Function*
reset	In	Asynchronous reset
sys_clk	In	System clock
pd_addr	In	peripheral device numbers for decoding
src_rd	Out	Source read enable
src_rdcnt	In	Points available at the source
data_in	In	Data input bus (32 bits)
dest_wr	Out	Destination write enable
dest_wrcnt	In	Destination write count
data_out	Out	Data bus output (32-bits)
new_packet	Out	A new packet header is being parsed (used for debug)

Table 80. ii_deframer Component Ports

ii_alerts

Source files: ii_alerts.vhd, fifo_1k_x32_vld.vhd

Description:

The alert component is used to monitor critical system events, such as triggering events, and report the time these occurrences to the system. This is done by monitoring error indicators (alert signals) in the logic and generating a packet to the host for each alert. In most cases, these packets are rare and are used for data acquisition process management.

The ii_alert component monitors input alerts and looks for rising edges on the enabled alerts. An enable for each alert is provided on the alert_enable inputs which correspond to the alerts on a bit-by-bit basis. When enabled, a rising edge indicates an alert is signaled and the logic then generates a packet indicating which alerts were triggered, the system time it was triggered and a status word for each alert. The status words can be anything of interest, the logic just puts these into the packet.

The system time is from a 32-bit counter clocked by the sample rate (**fs_clk**) clock. This time stamp is included in each packet indicating when an alert occurred. A time stamp rollover can generate an alert allowing software extension of the system time counter.

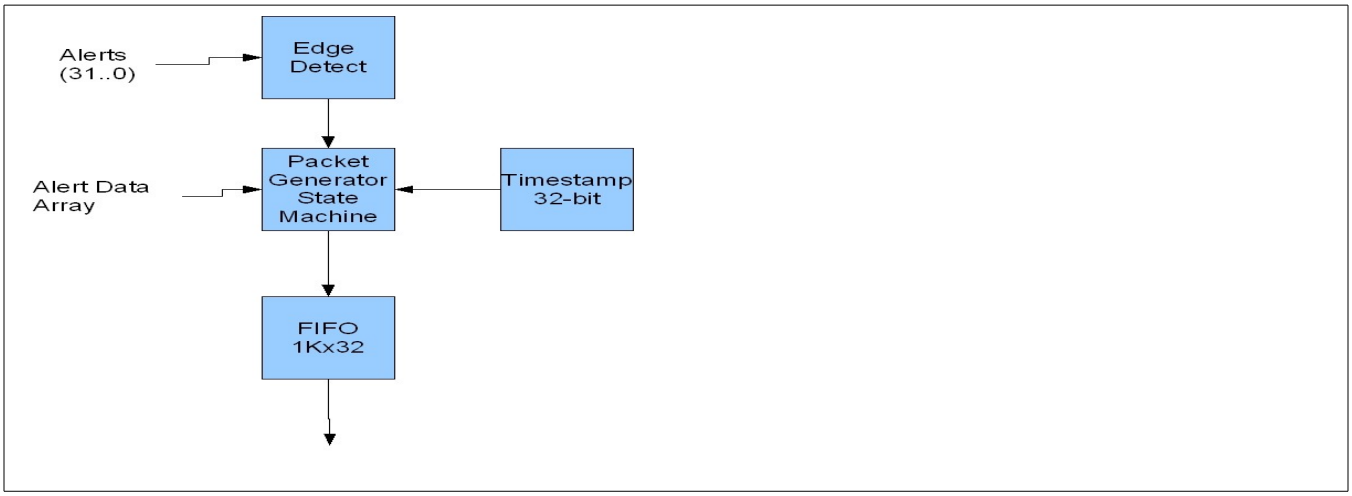


Figure 71. ii_alerts Component

Alert Data Format

The alert data messages are timestamped using a 32-bit counter running off the sample clock, showing the time that the alert occurred. Multiple alerts can be active for each alert packet as reported in the alerts signaled field of the packet.

Dword #	Description
0	Alerts Signaled
1	Timestamp
2	0
3	Software Word
4	temp_sensor_error & temp_error & "00" & X"000" & temp_data;
5	temp_warning & "000" & X"000" & temp_data;
6	X"1303000" & "000" & pll_status_q(2); -- PLL lock status
7	0
8	0
9	0
10	X"1303000" & "000" & mq_overflow(0);
15..11	0
16	X"1303000" & adc_overrange
31..17	0

Table 81. ii_alerts Packet Format

The array of alert data is the status word that is included in the alert packet for each enabled alert. This 32-bit word can be anything of interest and is included in the timestamped alert packet when any alert is triggered.

Adding New Alerts

To add new alert, it is recommended that one of the unused alerts be redefined. This reduces the amount of work required to change the software support system. The new alert should have a rising edge trigger and an alert message. The alert message should be added to the alert_data array at the same index as the alert. This message is a 32-bit word of any data that is pertinent to the alert.

Heres a little code snippet:

```
Alert(7) <= new_alert;
alert_data(7) <= new_alert_message;
```

The new alert will now appear in the alert data packet at alert 7.

The alert component is usually followed by a packetizing component, such as ii_packetizer, since the alert information is sent to the host. The PCIe controller on the X3 XMCs requires a specific header format that is added by the packetizer.

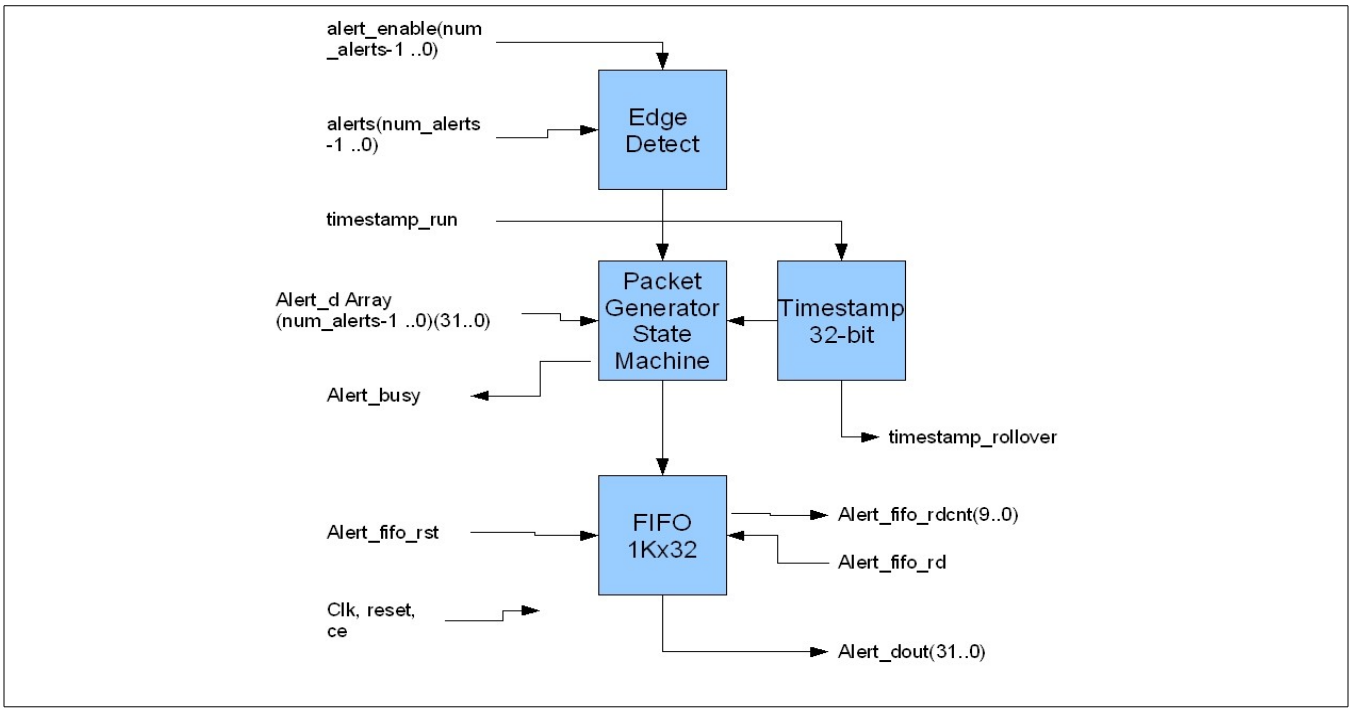


Figure 72. ii_alerts Component

The alert log is intended for occasional use in the system. There is an output FIFO to allow easy integration with the system logic. If an alert signaling at a high rate however, this can overwhelm the system. Alerts are not missed in this case unless the FIFO fills. If the FIFO fills the alert will remain pending until there is room in the FIFO for another alert packet. If the active alerts signal again when the FIFO is full however, only the first occurrence is signaled. A busy output from the and fifo read count are provided to monitor the status as necessary.

Port	*Direction*	*Function*
clk	In	clock
ce	In	Clock enable
reset	In	reset
alert_d(num_alerts -1 ..0)(31..0)	In	Array of status words for the alert packet. The dimension is defined in ii_x3_pkg for num_alerts.
alert(num_alerts -1..0)	In	Alert signal inputs. The component monitor these signals for a rising edge.
alert_enable((num_alerts -1 ..0)	In	Enable mask for the alerts. Each bit corresponds to the alert inputs.
timestamp_run	In	Time stamp run enable. The time stamp is reset to 0 when false.
timestamp_rollover	Out	The timestamp counter rolled over. Used for software extending the counter.
alert_dout[31..0]	Out	Alert data output
alert_fifo_rd	In	FIFO read enable
alert_fifo_rdcnt[9..0]	Out	FIFO read count
alert_fifo_rst	In	Reset the FIFO
alert_busy	Out	The alert state machine is busy when true.

Table 82. ii_alerts Component Ports

ii_mq_sram

Source files: ii_mq_sram.vhd, ii_sram_intf.vhd

Description:

**** NOTE: This component is only used on X3-SD and X3-SDF. For all other X3 modules, see ii_mq_sram32.****

This component creates multiple data queues using an SRAM buffer memory. Each queue is independently managed to give the functional equivalent of multiple FIFOs. The number of queues, depth of each queue and performance can be optimized for various application requirements. In case of a single queue, the data buffer behaves like a single FIFO with depth of 1Mx16.

The ii_mq_sram component implements a multi-queue data buffer as shown in the following block diagram. Each queue has a 1K input and output FIFO implemented in using an FPGA blockRAM that is connected to the buffer memory controller. The input and output FIFOs provide the immediate data buffering between the

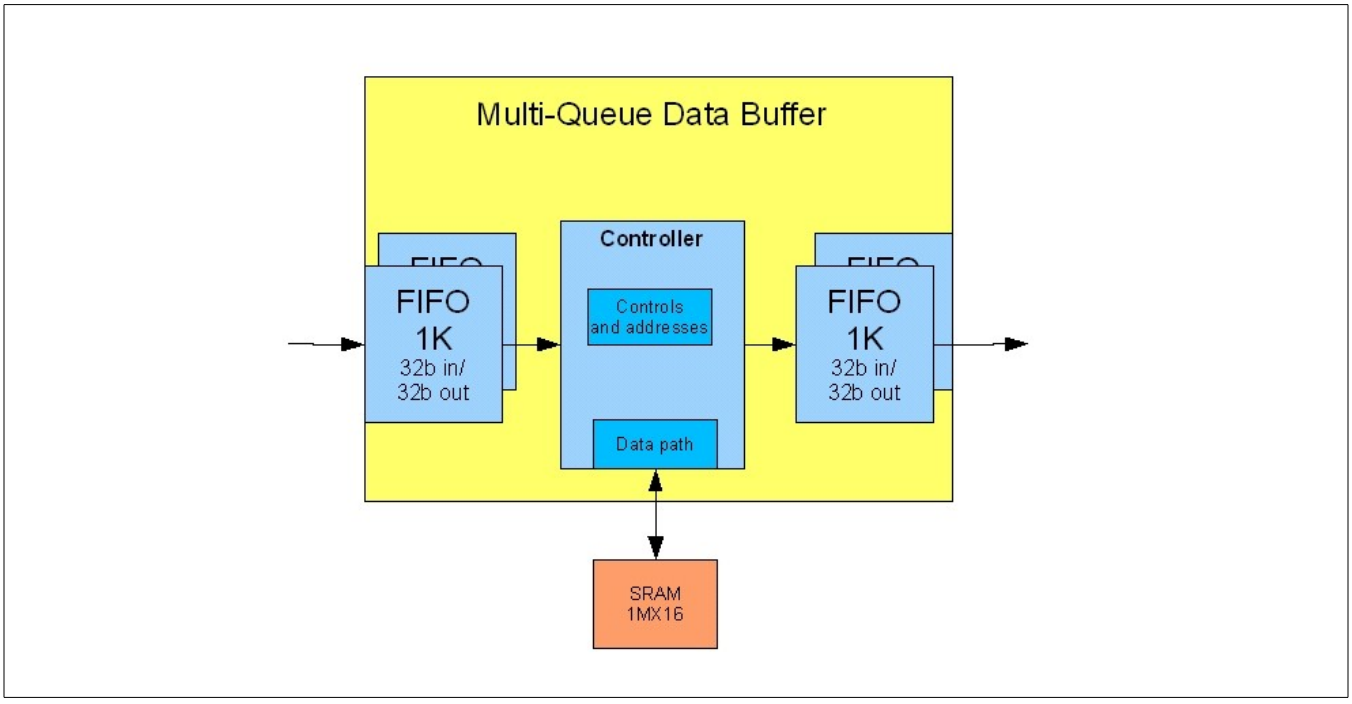


Figure 73. Multi-queue SRAM Component Simplified Block Diagram

controller and the system and are the interface to the system logic. Data flow to and from the multi-queue is paced on the levels of the input and output FIFOs.

The controller is responsible for moving data from the input FIFOs to the queue in buffer memory and from the buffer memory to the output FIFO. The data movement controller polls each input FIFO and moves data from an input FIFO to the buffer memory, and retrieves data to the queue output FIFO as space permits. Data movement is paced by the availability of data in the input FIFO and space in the output FIFO for each queue, as indicated by the FIFO write count for input and read count for output.

Controls for queue priority and flow control allow the controller to manage multiple queues at high data rates with efficiency of about 95%. An error flag to the system indicates when a queue overflows.

Memory Interface

The multi-queue data buffer uses an SBSRAM device, 1Mx16, for memory. This is a ZBT device and supports data rates of up to 120 MB/s. The memory interface component is ii_sram_intf which provides the low level address, data and control signal timing to communicate with the SBSRAM. The controller provides the read/write addresses and data to the ii_sram_intf component, which performs the accesses to the external SRAM device.

Changes to the multi-queue buffers, such as the number or size of the queues, do not require changes to the SRAM interface component. These changes are made in the ii_mq_sram controller code.

Buffer Status

The number of points (32-bit words) in each queue is the sum of the number of points in the SRAM plus the output FIFO counts. The queue counts are provided in the **mq_cnt** output array as 21 bit numbers.

The overflow flags indicate that a queue has potentially overflowed. The overflow flag indicates that either the input FIFO was full when written to, or that the queue is full. If the **mq_ififo_rdy[]** flow control signal for each queue is used, overflow should not occur unless the data rate exceeds the system capability. Overflow flags are also used as inputs to the alert log so that system messages are generated when buffer overflows occur.

Data Rates and Pacing

Data pacing control signals for flow control are **mq_ififo_rdy[]**, **mq_wr_cnt[]** and **mq_rdcnt[]**. The **mq_ififo_rdy[]** signal indicates that the input FIFO is less than **FIFO_AF** (default value is X"1E0" 32-bit words). The **mq_rdcnt[]** and **mq_wrcnt[]** arrays from each FIFO indicate the number of points in the output FIFO available for immediate consumption and number of points in the input FIFO.

Data rates for reads and writes to any queue are limited by several factors including the clock rate, data path width, and priority configuration. The controller has some overhead, amounting to about 10% of the available time used for queue management. Here is an approximate equation, followed by some actual measured rates.

*storage rate = sys_clk * (data width in bytes)/point * .90 efficiency*

Throughput Rate = storage rate / 2

Platform	SRAM data path width	SRAM Clock Rate (MHz)	Max Rate on any Queue (MB/s)	Number of Queues in FrameWork Logic	Priority Direction	Max Throughput Rate (MB/s)
X3-SDF	16 bits	67	134	1	Store (Write to SRAM)	67
X3-SD	16 bits	67	134	1	Store (Write to SRAM)	67

Table 83. ii_mq_sram Data Rates Summary

The rates quoted for the various platforms are for the FrameWork logic as delivered. The clock rates may be modified in custom applications, but cannot exceed 100 MHz because of the SRAM access speeds.

Queue Priority Control

The controller services the queues in a round-robin so that all queues have equal performance under most loading conditions. The FIFOs for each queue are inspected and data is moved if possible.

It is desirable in many systems to give priority to one direction, either storage to the queue or fetching from the queue, to improve real-time performance of the system. For example, data sourced from an A/D to be stored in the queue must be stored in time or lost forever, so a write priority would be preferred for this queue. The priority_select_wr0_rd1 vector allows the application to specify the priority direction for each queue. For write direction ('0'), the controller stores data from the input FIFO to SRAM if any is available. If no data is in the input FIFO, then it can perform a fetch from SRAM if space is available in the output FIFO. The read direction works similarly, forcing the controller to service the output FIFO first before input.

As the input FIFOs near overflow condition, the multi-queue controller will begin to service the queues on an crisis basis to prevent data loss if at all possible. If any FIFO exceeds its priority_threshold for the priority direction, then the controller immediately services that FIFO. If the queue space permits, then the controller will move data for that queue immediately. If no space is available, the controller will move to the next queue. Once the crisis is averted, the controller resumes equal priority queue servicing.

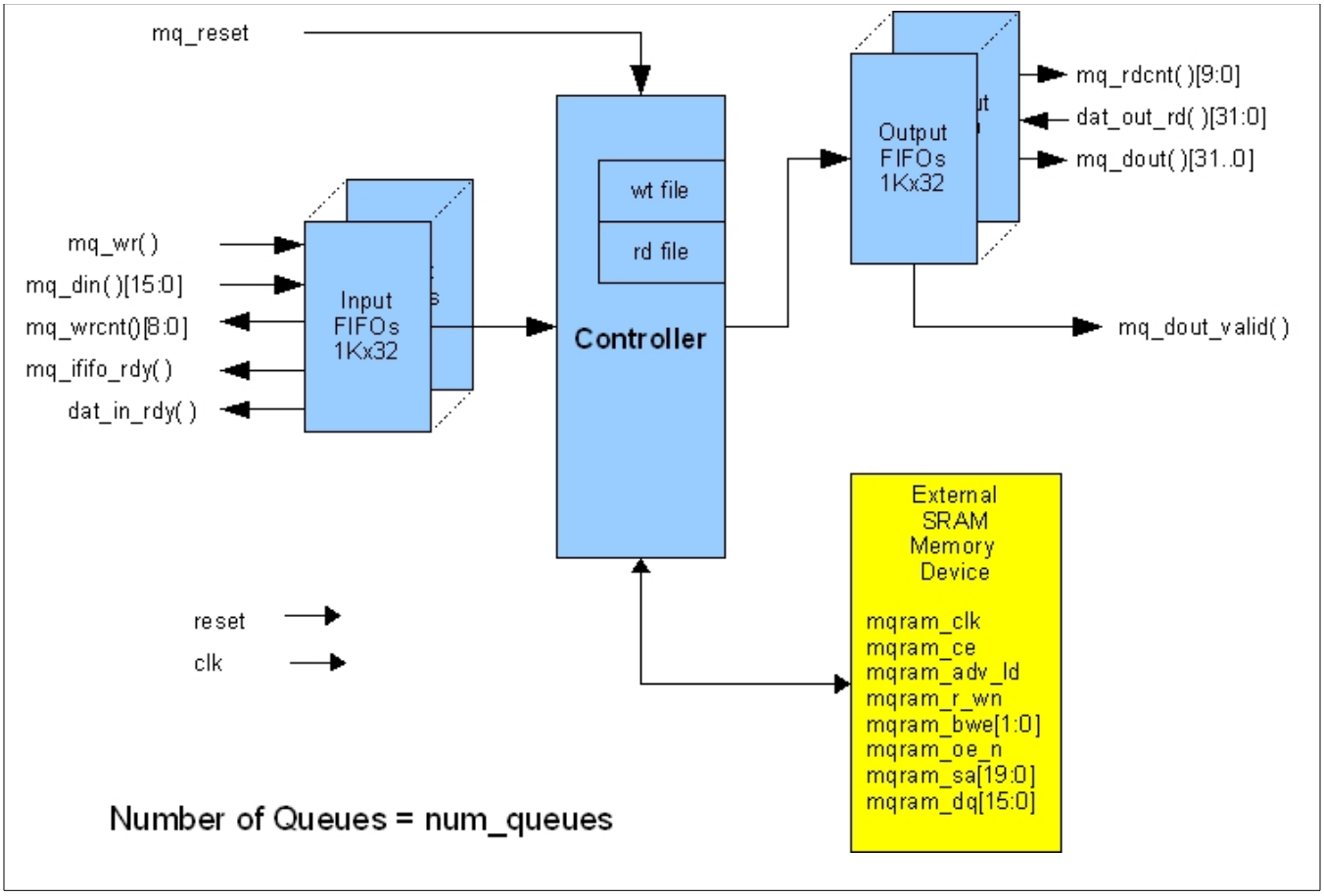


Figure 74. ii_mq_sram Component

Changing the Queue Size

To change the queue size, the controller code must be modified. In the code, the queue size is defined by the arrays queue_start and queue_size. The queue_start is where in buffer memory the each queue begins. This is the address bits 19..16, so a 1 means that the queue starts at X"10000" in SRAM. The queue_size defines how big the queue is in 16-bit words. The maximum queue_size is X"FFFFF" for the 1Mx16 SRAM on X3 SD and SDF modules.

```
queue_start_generate : for i in 0 to num_queues generate
  queue_start(i) <= x"0" when (i = 0) else
                    x"8" when (i = 1) else
                    x"f";
end generate;

queue_size_generate : for i in 0 to num_queues generate
  queue_size(i) <= X"80000";
end generate;
```

The queues can be different sizes but must be at least 0x10000 (64KW).

Changing the Number of Queues
The number of queues is set by the **num_queues** generic. A maximum of 16 queues is supported. You must also change the code to specify the queue_start and queue_size constants to describe each queue size.

Four blockrams are required for each queue that is created.

Generic	Function	
num_queues	Sets the number of queues generated by the component. Default is 1.	
Port	**Direction**	**Function**
reset	In	reset
clk	In	System clock.
mq_reset(num_queues-1 downto 0)	In	Queue resets
mq_wr(num_queues-1 downto 0)	In	Queue write enables
mq_din	In	Data input array, 16-bit path, defined in ii_x3_pkg
mq_rd(num_queues-1 downto 0)	In	Queue read enables
mq_dout	In	Data output array, 16-bit path, defined in ii_x3_pkg
mq_dout_valid[num_queues-1 :0]	Out	Output data valid for each queue
mq_wrcnt	Out	Input FIFO array counts, 8..0 defined in ii_x3_pkg
mq_rdcnt	Out	output FIFO array counts, 9..0 defined in ii_x3_pkg
mq_cnt	Out	Array of number of points in each queue, defined in ii_x3_pkg
mq_overflow(num_queues-1 downto 0)	Out	Queue overflow indicators.
mq_ififo_rdy(num_queues-1 downto 0)	Out	Indicates that the input FIFO for that queue has less than FIFO_AF points . (default is X”1E0”)
mqram_clk	Out	SRAM clock output
mqram_ce	Out	SRAM chip enable.
mqram_adv_ld	Out	SRAM address advance/load control. Set to '0' so addresses are always loaded.
mqram_r_wn	Out	SRAM read/write.
mqram_bwe(1 downto 0)	Out	SRAM byte write enables, set to “00”
mqram_oe_n	Out	SRAM output enable, active low.
mqram_sa(19 downto 0)	Out	SRAM address bus.
mqram_dq(15 downto 0)	InOut	SRAM data bus.
priority_select_wr0_rd1(num_queues-1 downto 0)	In	Select read or write priority for each queue. 0 = write priority.
priority_threshold	Out	Array of FIFO thresholds for priority determination. Array defined in ii_x3_pkg

Table 84. ii_mq_sram Component Ports

ii_mq_sram32

Source files: ii_mq_sram32.vhd, ii_sram32_intf.vhd

Description:

**** NOTE: This component is used on on all X3 modules except X3-SD and X3-SDF. For X3-SD and X3-SDF modules, see ii_mq_sram.****

This component creates multiple data queues using an SRAM buffer memory. Each queue is independently managed to give the functional equivalent of multiple FIFOs. The number of queues, depth of each queue and performance can be optimized for various application requirements. In case of a single queue, the data buffer behaves like a single FIFO with depth of 512Kx32.

The ii_mq_sram32 component implements a multi-queue data buffer as shown in the following block diagram. Each queue has a 1K input and output FIFO implemented in using an FPGA blockRAM that is connected to the buffer memory controller. The input and output FIFOs provide the immediate data buffering between the

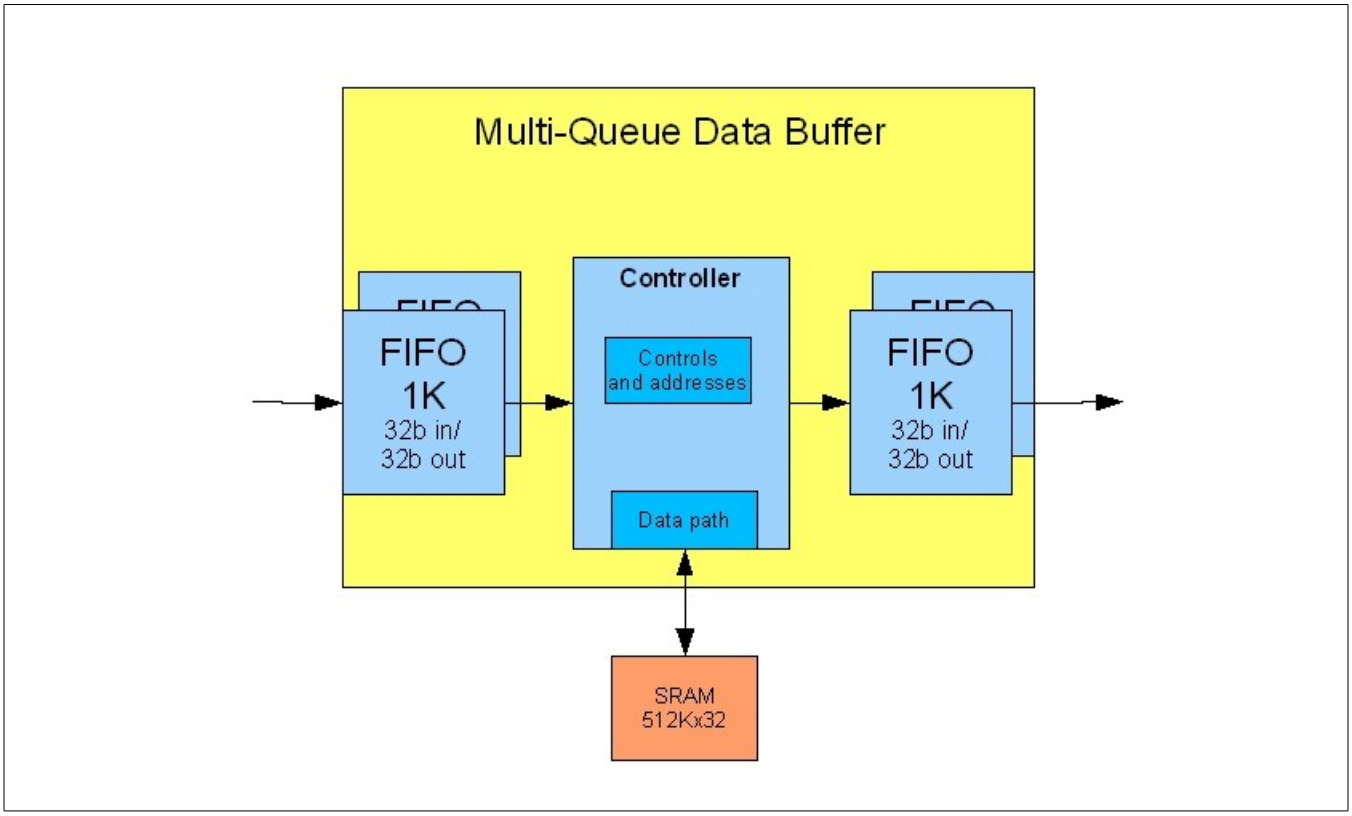


Figure 75. Multi-queue SRAM 32-bit Component Simplified Block Diagram

controller and the system and are the interface to the system logic. Data flow to and from the multi-queue is paced on the levels of the input and output FIFOs.

The controller is responsible for moving data from the input FIFOs to the queue in buffer memory and from the buffer memory to the output FIFO. The data movement controller polls each input FIFO and moves data from an input FIFO to the buffer memory, and retrieves data to the queue output FIFO as space permits. Data movement is paced by the availability of data in the input FIFO and space in the output FIFO for each queue, as indicated by the FIFO write count for input and read count for output.

Controls for queue priority and flow control allow the controller to manage multiple queues at high data rates with efficiency of about 95%. An error flag to the system indicates when a queue overflows.

Memory Interface

The multi-queue data buffer uses an SBSRAM device, 512Kx32, for memory. This is a ZBT device and supports data rates of up to 333 MB/s. The memory interface component is ii_sram32_intf which provides the low level address, data and control signal timing to communicate with the SBSRAM. The controller provides the read/write addresses and data to the ii_sram32_intf component, which performs the accesses to the external SRAM device.

Changes to the multi-queue buffers, such as the number or size of the queues, do not require changes to the SRAM interface component. These changes are made in the ii_mq_sram32 controller code.

Buffer Status

The number of points (32-bit words) in each queue is the sum of the number of points in the SRAM plus the output FIFO counts. The queue counts are provided in the **mq_cnt** output array as 20 bit numbers (size is defined in ii_x3_pkg.vhd).

The overflow flags indicate that a queue has potentially overflowed. The overflow flag indicates that either the input FIFO was full when written to, or that the queue is full. If the **mq_ififo_rdy[]** flow control signal for each queue is used, overflow should not occur unless the data rate exceeds the system capability. Overflow flags are also used as inputs to the alert log so that system messages are generated when buffer overflows occur.

Data Rates and Pacing

Data pacing control signals for flow control are **mq_ififo_rdy[]**, **mq_wr_cnt[]** and **mq_rdcnt[]**. The **mq_ififo_rdy[]** signal indicates that the input FIFO is less than **FIFO_AF** (default value is X"1E0" 32-bit words). The **mq_rdcnt[]** and **mq_wrcnt[]** arrays from each FIFO indicate the number of points in the output FIFO available for immediate consumption and number of points in the input FIFO.

Data rates for reads and writes to any queue are limited by several factors including the clock rate, data path width, and priority configuration. The controller has some overhead, amounting to about 10% of the available time used for queue management. Here is an approximate equation, followed by some actual measured rates.

*storage rate = sys_clk * (data width in bytes)/point * .90 efficiency*

Throughput Rate = storage rate / 2

Platform	SRAM data path width	SRAM Clock Rate (MHz)	Max Rate on any Queue (MB/s)	Number of Queues in FrameWork Logic	Queue Sizes	Priority Direction	Max Throughput Rate (MB/s)
X3-Servo	32 bits	107	428	2	1MB each	Store (Write to SRAM for A/D data)	428
X3-10M	32 bits	83.33	333	1	2MB	Store (Write to SRAM for A/D data)	167
X3-A4D4	32 bits	107	428	2	1MB each	Store (Write to SRAM for A/D data)	214
X3-DIO	32 bits	107	428	2	1MB each	Store (Write to SRAM for A/D data)	214
X3-25M	32 bits	107	428	2	1MB each	Store (Write to SRAM for A/D data)	214

Table 85. ii_mq_sram32 Data Rates Summary

The rates quoted for the various platforms are for the FrameWork logic as delivered. The clock rates may be modified in custom applications, but cannot exceed 133 MHz because of the SRAM access speeds.

Queue Priority Control

The controller services the queues in a round-robin so that all queues have equal performance under most loading conditions. The FIFOs for each queue are inspected and data is moved if possible.

It is desirable in many systems to give priority to one direction, either storage to the queue or fetching from the queue, to improve real-time performance of the system. For example, data sourced from an A/D to be stored in the queue must be stored in time or lost forever, so a write priority would be preferred for this queue. The **priority_select_wr0_rd1** vector allows the application to specify the priority direction for each queue. For write direction ('0'), the controller stores data from the input FIFO to SRAM if any is available. If no data is in the input FIFO, then it can perform a fetch from SRAM if space is available in the output FIFO. The read direction works similarly, forcing the controller to service the output FIFO first before input.

As the input FIFOs near overflow condition, the multi-queue controller will begin to service the queues on an crisis basis to prevent data loss if at all possible. If any FIFO exceeds its priority_threshold for the priority direction, then the controller immediately services that FIFO. If the queue space permits, then the controller will move data for that queue immediately. If no space is available, the controller will move to the next queue. Once the crisis is averted, the controller resumes equal priority queue servicing.

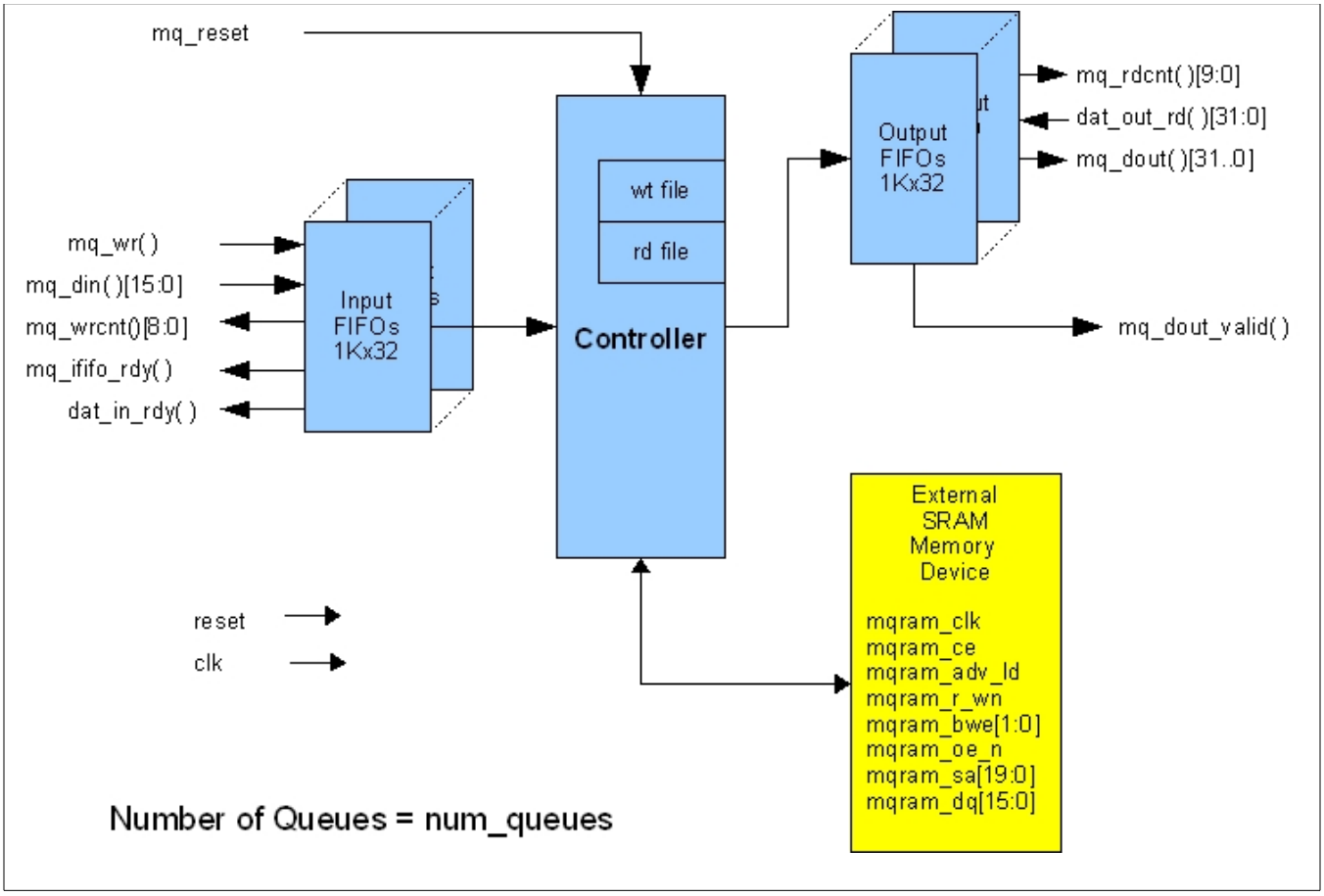


Figure 76. ii_mq_sram32 Component

Changing the Queue Size

To change the queue size, the controller code must be modified. In the code, the queue size is defined by the arrays queue_start and queue_size. The queue_start is where in buffer memory the each queue begins. This is the address bits 19..16, so a 1 means that the queue starts at X"10000" in SRAM. The queue_size defines how big the queue is in 16-bit words. The maximum queue_size is X"FFFFF" for the 512Kx32 SRAM on X3 modules.

```
queue_start_generate : for i in 0 to num_queues generate
queue_start(i) <= x"0" when (i = 0) else
                  x"4" when (i = 1) else
                  x"7";
end generate;

queue_size_generate : for i in 0 to num_queues generate
queue_size(i) <= "111" & X"FFFF" when (num_queues = 1) else  -- all of sram
                                 "011" & X"FFFF" when (num_queues = 2) else  -- half of sram
                                          "001" & X"FFFF" ;

end generate;
```

The queues can be different sizes but must be at least 0x10000 (64KW).

Changing the Number of Queues
The number of queues is set by the num_queues generic. A maximum of 16 queues is supported. You must also change the code to specify the queue_start and queue_size constants to describe each queue size.

Four blockrams are required for each queue that is created.

Generic	**Function**	
num_queues	Sets the number of queues generated by the component. Default is 1.	
Port	***Direction***	***Function***
reset	In	Reset
mq_reset(num_queues-1 downto 0)	In	Reset for individual queues
clk	In	System clock.
mq_wr(num_queues-1 downto 0)	In	Queue write enables
mq_din[31:0]	In	Data input array, 16-bit path, defined in ii_x3_pkg
mq_rd(num_queues-1 downto 0)	In	Queue read enables
mq_dout[31:0]	In	Data output array, 16-bit path, defined in ii_x3_pkg
mq_dout_valid[num_queues-1 :0]	Out	Output data valid for each queue
mq_wrcnt	Out	Input FIFO array counts, 8..0 defined in ii_x3_pkg
mq_rdcnt	Out	output FIFO array counts, 9..0 defined in ii_x3_pkg
mq_cnt	Out	Array of number of points in each queue, defined in ii_x3_pkg
mq_overflow(num_queues-1 downto 0)	Out	Queue overflow indicators.
mq_ififo_rdy(num_queues-1 downto 0)	Out	Indicates that the input FIFO for that queue has less than FIFO_AF points . (default is X"1E0")
mq_sram_delay	In	Timing delay adjust for SRAM data path IOBs. Do NOT modify.
mqram_clk	Out	SRAM clock output
mqram_ce	Out	SRAM chip enable.
mqram_adv_ld	Out	SRAM address advance/load control. Set to '0' so addresses are always loaded.
mqram_r_wn	Out	SRAM read/write.
mqram_bwe(1 downto 0)	Out	SRAM byte write enables, set to "00"
mqram_oe_n	Out	SRAM output enable, active low.
mqram_sa(18 downto 0)	Out	SRAM address bus.
mqram_dq(31 downto 0)	InOut	SRAM data bus.
priority_select_wr0_rd1(num_queues-1 downto 0)	In	Select read or write priority for each queue. 0 = write priority.
priority_threshold	Out	Array of FIFO thresholds for priority determination. Array defined in ii_x3_pkg

Table 86. ii_mq32_sram Component Ports

ii_trigger

Source file: ii_trigger.vhd

Description:

The ii_trigger component provides triggering control for data acquisition. The output of the component is a trigger signal that is used to control data capture and storage and is usually driven into the trigger port of an A/D converter interface or other IO component. This allows the logic to acquire data as is needed by the specific application for processing and logging.

The ii_trigger component provides two methods for triggering: framed and unframed. In the framed mode, the **trigger** signal goes false after programmable number of points, assuming that a data point is captured for each rising edge of the sample clock. In unframed mode, the trigger output is true as controlled by the input triggers. The trigger is started by either the **sw_trig** input or **ext_sync** input. These two signals are OR'ed together so that either one can begin a trigger. In the framed mode, a rising edge on the **trigger** begins the frame. In unframed mode, the **trigger** is true simply whenever the OR'ed signal is true.

The frame count is loaded as the number of points to be captured. The frame count has a maximum size of 2^{23}.

The trigger is either the external sync signal or the internal software controlled run signal, as selected by the trigger control register for each channel. The external trigger may be disabled to prevent false triggering. The software trigger is may always trigger the system even when the hardware trigger is true to allow the application to force a trigger condition.

This component should use the SAMPLE CLOCK for all logic so that the trigger is synchronous to the sampling. This is required so that the number of samples in frame is counted correctly. The control signals to the component are usually NOT on this clock domain, but rather on system clock domain. It is best to double-register all inputs crossing this domain so that metastability is avoided.

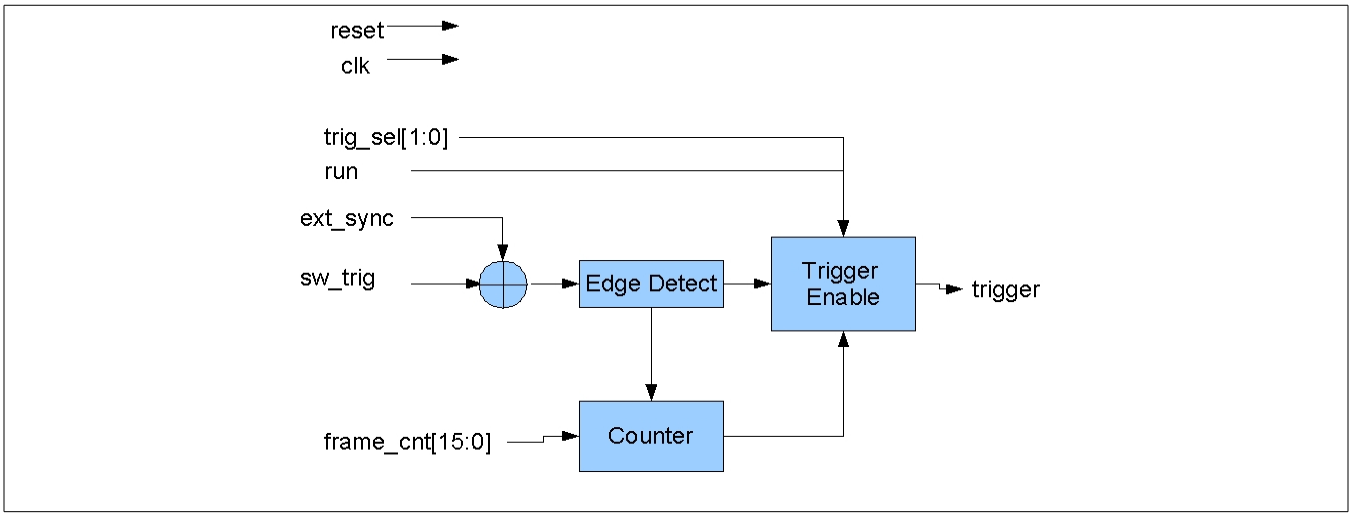


Table 87. ii_trigger Block Diagram

Port	Direction	Function
reset	In	reset
clk	In	Sample clock
sample_en	In	Enable this trigger component for sampling. Synchronous to clk.
ext_sync	In	Trigger input, usually from an external signal.
sw_trig	In	“Software trigger” input. This signal is OR'ed with ext_sync to allow software or logic to start/stop an acquisition.
run	In	Enable the trigger component to run.
trig_sel[1:0]	In	Trigger selections. Bit 0 = trigger mode; '0' = unframed, '1' = framed Bit 1 = ext trigger enable; '0' = disabled, '1' = enabled
frame_cnt[23:0]	In	Size of the frame to be captured.
Trigger	Out	Trigger output. This will fire each time a sample should be collected.
Trigger_en	Out	The trigger is enabled. This signal is valid whenever the trigger is true. When decimation is used, this signal is true when the decimation is counting and when points are true.

Table 88. ii_trigger Component Ports

ii_link

Source files: ii_link.vhd, fifo_1kx32_async_vld.vhd

Description:

This component, when used with ii_link_master, creates a 32-bit bidirectional data bus with flow control. On the X3 modules, this data link is used to move data from the PCI Express interface FPGA to the application logic. The link itself does not require any data formating or specific packet structure.

The link provides an interrupt to the link master indicating that points are ready to be read or if any points can be transmitted into the local FIFO. The link interrupt **link_rd_intn** indicates that data can be read by the master from the link read FIFO, while the **link_wr_intn** indicates that space is available in the write FIFO to accept more data. Depending on the **link_rw** direction control , the link drives the **link_d** data lines with the number of points that can be transferred.

After an interrupt from the link to the master, the link master then reads the number of points that can be transferred and generates the control signals to the link allowing the points to move. The number of points transferred is the determined by the number points the link can transfer and the number of points available in master FIFO. The link_frame_n indicates that at transfer is in progress in the direction of as signaled by the **link_rw**.

The data phase of the transfer is indicated by the **link_dp_n** control signal. Data points are moved as a continuous burst by the master. The maximum data rate of the link is about 220 MB/s for a 67 MHz clock. Since the link data path is bidirectional, the link attempts to give equal time to each direction by balancing reads and writes.

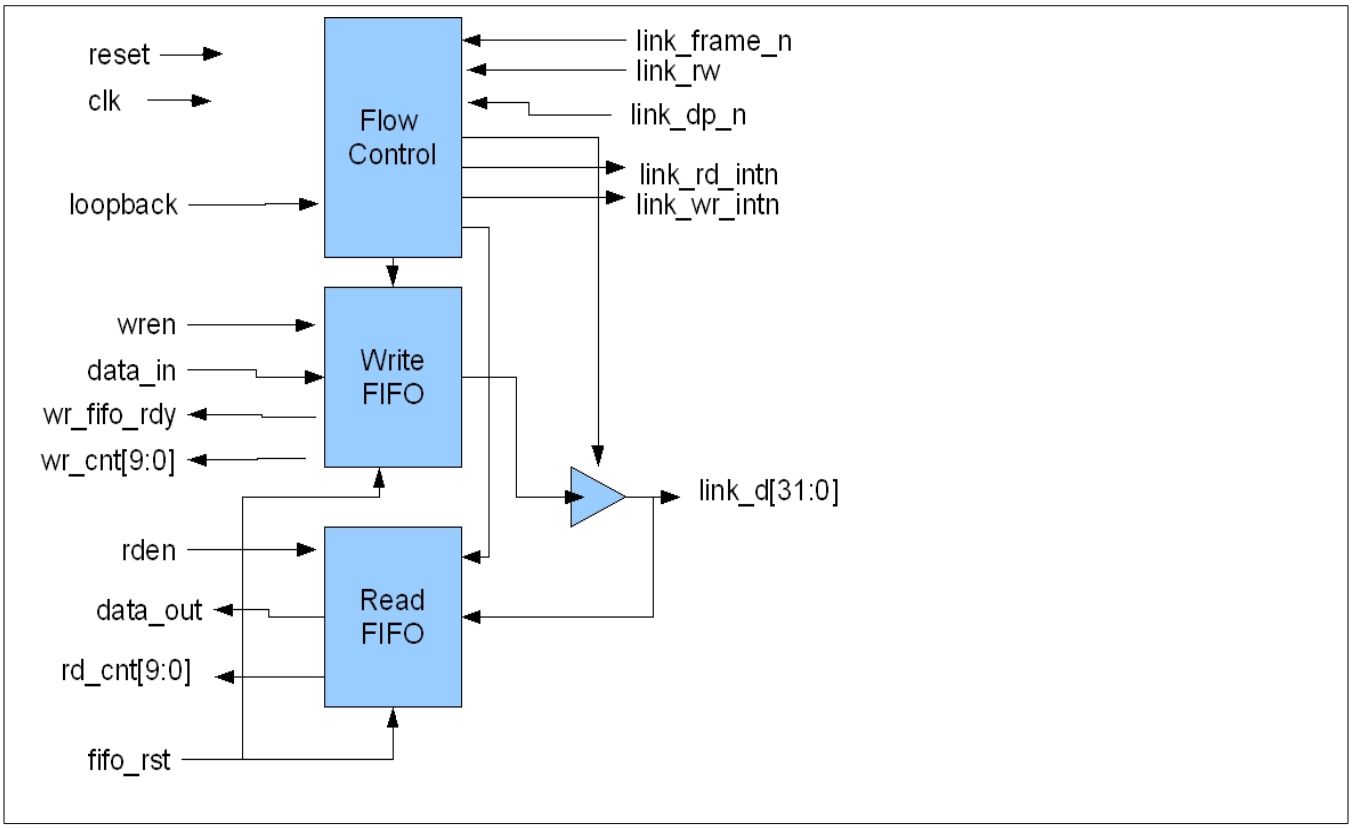


Figure 77. ii_link Block Diagram

Port	*Direction*	*Function*
reset	In	System reset.
clk	In	Clock input
loopback	In	Loopback test for the link. Data is automatically written from the read FIFO to the write FIFO when true.
wren	In	Write enable to the write FIFO. Data is stored to the FIFO on rising edges of clk when wren is true.
data_in[31:0]	In	Input data bus from the logic.
wr_fifo_rdy	Out	Write FIFO has room for at least 16 points when true.
wr_cnt[9:0]	Out	Write FIFO count.
rden	In	Read enable to the read FIFO. Data is provided on rising edges of clk when rden is true.
data_out[31:0]	Out	Data from the link to the logic.
rd_cnt[9:0]	Out	Read FIFO count.
fifo_rst	In	Reset the FIFOs.
link_d[31:0]	Inout	Link data bus. This is the data bus on the external connector.
link_frame_n	In	Link frame indicates that the master is accessing the link. Active low.
link_rw	In	Link read(high)/write(low) direction control from the master.
link_dp_n	In	Link data phase indicator. Active low.
link_rd_intn	Out	Interrupt from the link to the master indicating the data is available for a read. Active low.
link_wr_intn	Out	Interrupt from the link to the master indicating the data is data is needed by the link and can be written into its FIFO. Active low.

Table 89. ii_link Component Ports

ii_link2

Source files: ii_link2.vhd, fifo_1kx32_async_vld.vhd

Description:

This component, when used with ii_link_master, creates a 32-bit bidirectional data bus with flow control. On the X3 modules, this data link is used to move data from the PCI Express interface FPGA to the application logic. The link itself does not require any data formating or specific packet structure.

The link provides an interrupt to the link master indicating that points are ready to be read or if any points can be transmitted into the local FIFO. The link interrupt **link_rd_intn** indicates that data can be read by the master from the link read FIFO, while the **link_wr_intn** indicates that space is available in the write FIFO to accept more data. Depending on the **link_rw** direction control , the link drives the **link_d** data lines with the number of points that can be transferred.

After an interrupt from the link to the master, the link master then reads the number of points that can be transferred and generates the control signals to the link allowing the points to move. The number of points transferred is the determined by the number points the link can transfer and the number of points available in master FIFO. The link_frame_n indicates that at transfer is in progress in the direction of as signaled by the **link_rw**.

The data phase of the transfer is indicated by the **link_dp_n** control signal. Data points are moved as a continuous burst by the master. The maximum data rate of the link is about 220 MB/s for a 67 MHz clock. Since the link data path is bidirectional, the link attempts to give equal time to each direction by balancing reads and writes.

The component provides separate clock domains for the system clock and link clock. For X3 modules, the link clock is 67 MHz, while system clock can be higher. (Clocks up to 112 MHz have been used.)

The separation of the link and system clocks is the difference between ii_link and ii_link2 components.

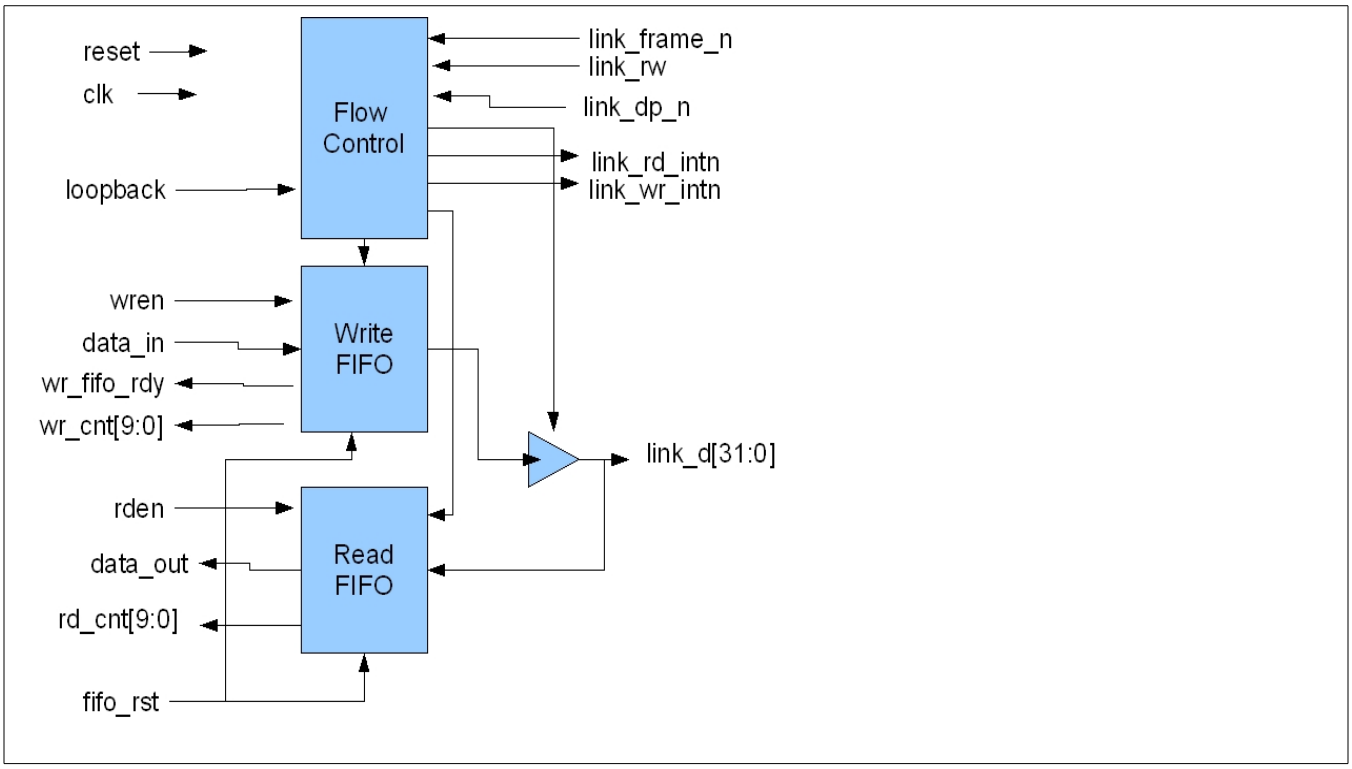


Figure 78. ii_link2 Block Diagram

Port	*Direction*	*Clock*	*Function*
reset	In	-	System reset.
link_clk	In		Clock input for link (67 MHz)
sys_clk	In		Clock input for system
loopback	In	sys_clk	Loopback test for the link. Data is automatically written from the read FIFO to the write FIFO when true.
wren	In	sys_clk	Write enable to the write FIFO. Data is stored to the FIFO on rising edges of clk when wren is true.
data_in[31:0]	In	sys_clk	Input data bus from the logic.
wr_fifo_rdy	Out	sys_clk	Write FIFO has room for at least 16 points when true.
wr_cnt[9:0]	Out	sys_clk	Write FIFO count.
rden	In	sys_clk	Read enable to the read FIFO. Data is provided on rising edges of clk when rden is true.
data_out[31:0]	Out	sys_clk	Data from the link to the logic.
rd_cnt[9:0]	Out	sys_clk	Read FIFO count.
fifo_rst	In	sys_clk	Reset the FIFOs.
link_d[31:0]	Inout	link_clk	Link data bus. This is the data bus on the external connector.
link_frame_n	In	link_clk	Link frame indicates that the master is accessing the link. Active low.
link_rw	In	link_clk	Link read(high)/write(low) direction control from the master.
link_dp_n	In	link_clk	Link data phase indicator. Active low.
link_rd_intn	Out	link_clk	Interrupt from the link to the master indicating the data is available for a read. Active low.
link_wr_intn	Out	link_clk	Interrupt from the link to the master indicating the data is data is needed by the link and can be written into its FIFO. Active low.

Table 90. ii_link2 Component Ports

ii_pll_spi

Source files: ii_pll_spi.vhd

Description:

This component is the SPI serial port interface to the AD9510 PLL and clock distribution device. This SPI port provides the control and status information to the AD9510. All initialization for the AD9510 is performed through this port.

The PLL communicates only in bytes for this logic component. Bit 23 specifies a read or write cycle. The address field specifies the PLL register address. The first two bytes of the SPI word are transmitted to the PLL followed by the data byte for writes. For reads, the PLL receives a byte after the address is sent. During the transmission, the PLL ready is false. The serial clock is slow (67 MHz/4 = 16.75 MHz), the host can outrun the transactions so the PLL ready signal can be used to pace transactions.

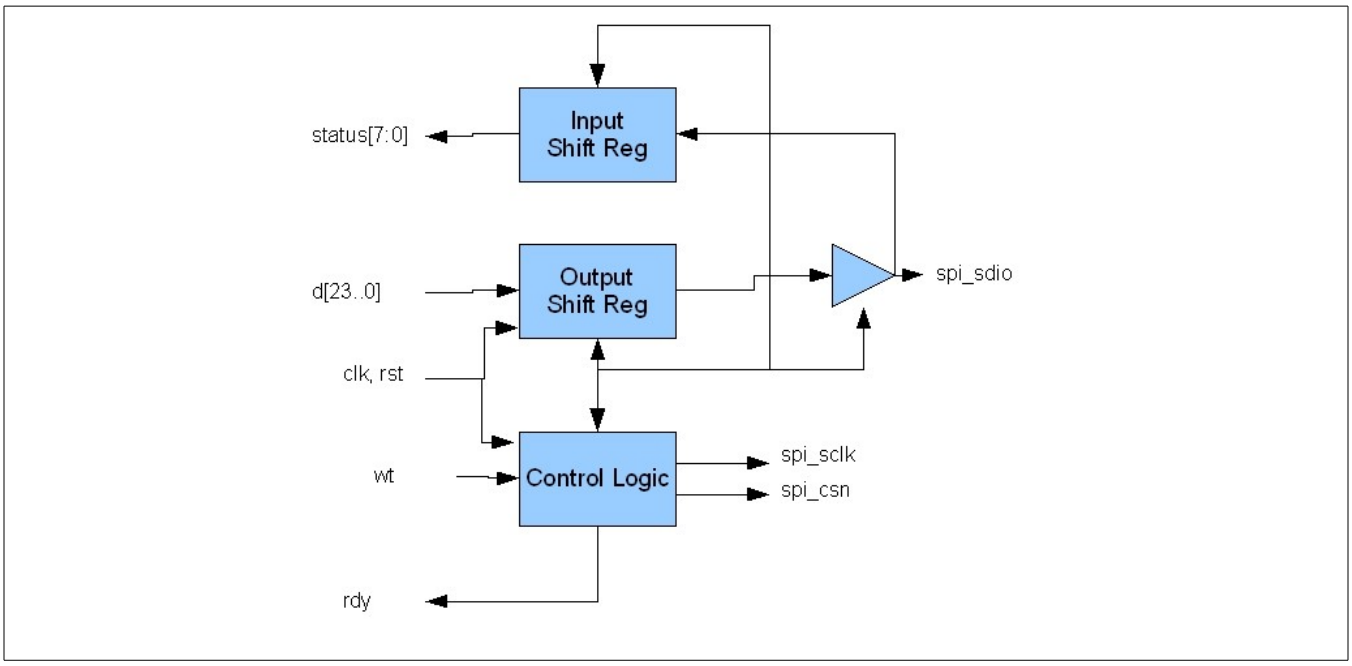


Figure 79. ii_pll_spi Component

Port	Direction	Function
rst	In	System reset.
clk	In	Clock input
d[23:0]	In	Data bus input, 23..0
wt	In	Write enable, synchronous to clk
rdy	Out	True when component is ready to use. False during transactions.
spi_sdio	Inout	PLL serial data bit.
spi_cs_n	Out	Chip select to PLL, active low
spi_sclk	Out	SPI clock

Table 91. ii_pll_spi Component Ports

ii_temp_sensor

Source Files: ii_temp_sensor.vhd, i2c_bus.vhd, i2c_master_top.vhd, i2c.vhd, i2c_master_bit_ctrl.vhd, i2c_master_byte_ctrl.vhd

Description:

This component provides a thermal monitoring function and an interface to a Texas Instruments TMP175 temperature sensor. The interface configures the temperature sensor and then polls the temperature sensor every 100 mS (approx) for its current reading. This temperature is compared to a warning and failure temperature to generate warning and powerdown controls to the system for a thermal failure.

The interface to the temperature sensor is an IIC serial bus. This is a two-wire serial protocol that is an industry standard for low speed communications and control. The ii_temp_sensor component implements at its low level an IIC controller core and wraps it at a higher level with the controls to configure the thermal warning and failure conditions.

Using ii_temp_sensor

This component operates autonomously after setup and is used to monitor the system temperature for thermal protection. For autonomous operation, the component requires a continuous clock, must be out of reset and the clear bits should be false.

The component initializes the temperature sensor and then polls the sensor for its current temperature. If the sensor cannot communicate because of a malfunction, a sensor_error is output. A sensor error results in a powerdown output also since the system temperature is unknown.

Temperature Reading

The current temperature is read at the temperature port. This output is updated approximately every 100 mS (for 67 MHz system clock). The conversion is

*deg C = 0.0625 C/bit * temperature*

0C = 0, highest temperature is X"07FF" = 128 deg C.

where temperature is a 16-bit, 2's complement number.

Setting the Warning and Failure Temperatures

To change the warning and failure temperature settings from their default values, write to the logic component using the write_warning and write_powerdown inputs. The value latched into the comparison register is

X"0" & config_data & X"0"

Note that only temperatures > 0 can be used for comparison.

Level	X3 Family Default	Output
Warning	70C (0x460)	temp_warning
Failure	70C (0x460)	temp_powerdown

Error Flag Outputs

The ii_temp_sensor has several outputs indicating the system status. These can be used to control power supplies or as system alerts. The outputs latch true once fired and must be cleared by the system.

Signal	Fires When...	Indicates	Clear With...
temp_warning	Temperature > C_RESET_THRESHOLD ii_x3_pkg = 70C (0x460)	Temperature is above warning level	temp_warning_clear
Powerdown	Temperature > C_RESET_THRESHOLD ii_x3_pkg = 70C (0x460)	Temperature is above powerdown level	Powerdown_clear
temp_error	Sensor temperature > 127C or sensor alert output fired	Bad sensor reading or sensor alert	rst
sensor_error	IIC controller can not communicate with the temperature sensor	Sensor or logic malfunction	rst

On the X3 modules, the powerdown output is connected to one or more of the analog power supplies. If an over-temperature condition occurs, the analog is powered down but not the FPGA. This allows the module to continue to communicate if possible so that it can be restarted.

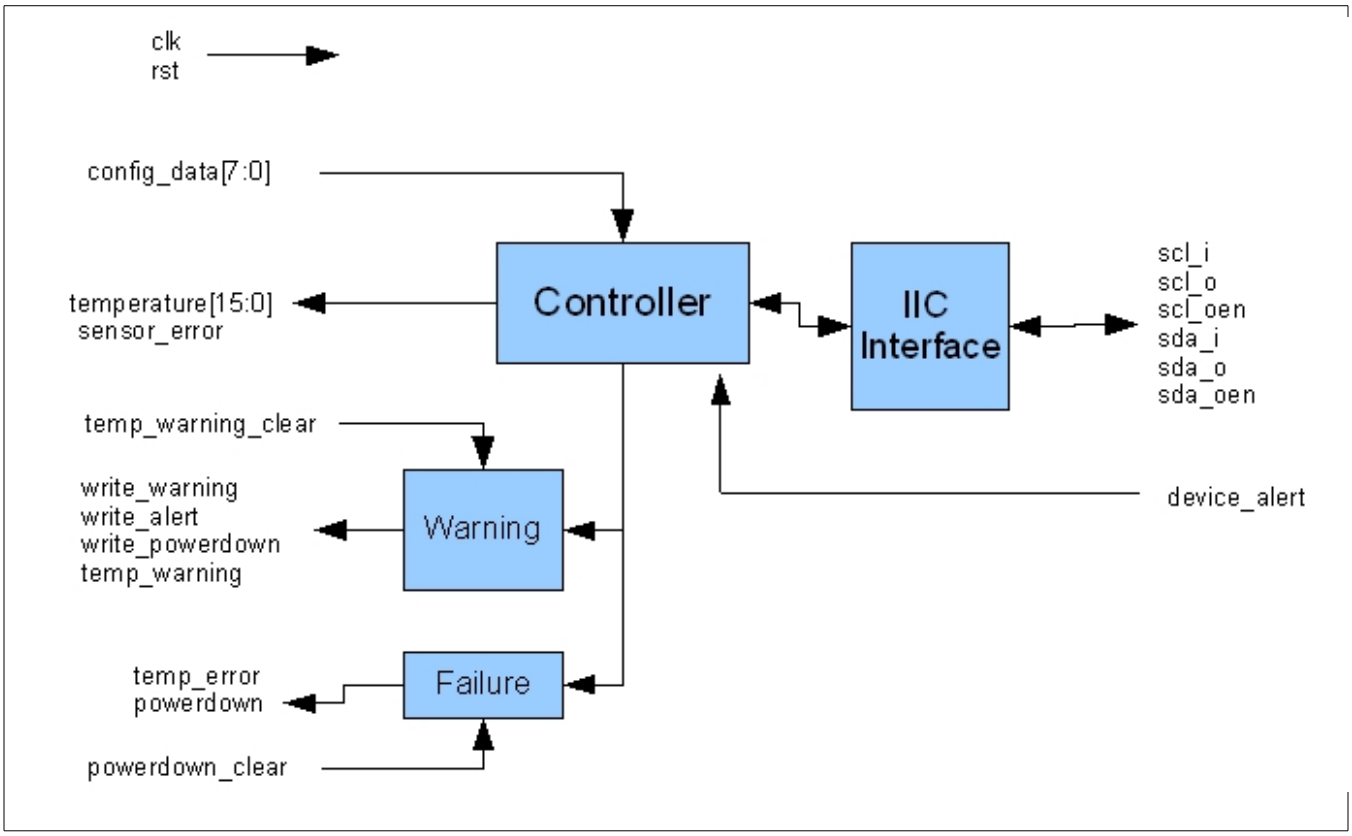


Figure 80. ii_temp_sensor Component

Generic	Function
sensor_address[6:0]	IIC address for the sensor
scl_rate	IIC clock rate. Options are normal (400 kbaud), high (1Mbaud), fast (5 Mbaud) (this component uses normal only).
wishbone_clk_rate	Input clock rate in Hz. Default is 66000000.

Port	*Direction*	*Function*
rst	In	System reset.
clk	In	Clock input
write_warning	In	Write the warning level.
write_alert	In	Write the alert level.
write_powerdown	In	Write the powerdown level.
powerdown_clear	In	Clears the powerdown output.
temp_warning_clear	In	Clears the temperature warning output.
config_data[7:0]	In	Data bus input
Temperature[15:0]	Out	Current temperature reading.
powerdown	Out	powerdown from overtemp or sensor failure
temp_warning	Out	temperature exceeded warning level
temp_error	Out	temperature exceeded failure level
sensor_error	Out	the sensor would not communicate
device_alert	In	TMP175 alert input.
scl_i	In	IIC clock input.
scl_o	Out	IIC clock output.
scl_oen	Out	IIC clock output enable, active low.
sda_i	In	IIC data input.
sda_o	Out	IIC data output.
sda_oen	Out	IIC data output enable, active low.

Table 92. ii_temp_sensor Component Ports

ii_offgain

Source file: ii_offgain.vhd

Description:

This component is performs error correction for data. Data is multiplied by a gain and has an offset added to it to compensate for analog errors. The gain factor is a 2's complement, 16-bit number ranging from +2 to -2 that allows for precise gain correction to the input data. The offset value is a 15-bit, 2's complement number that compensates for bias errors.

The error compensated output is

$$y = Gx + O$$

where x = input data, G = gain, O = offset

A gain of 1 represented by 0x10000 and offset of 0 equal to 0.

The component uses a hardware multiplier in the FPGA followed by an adder for the offset correction. The data is inspected after error correction to perform saturation and prevent numeric overflow.

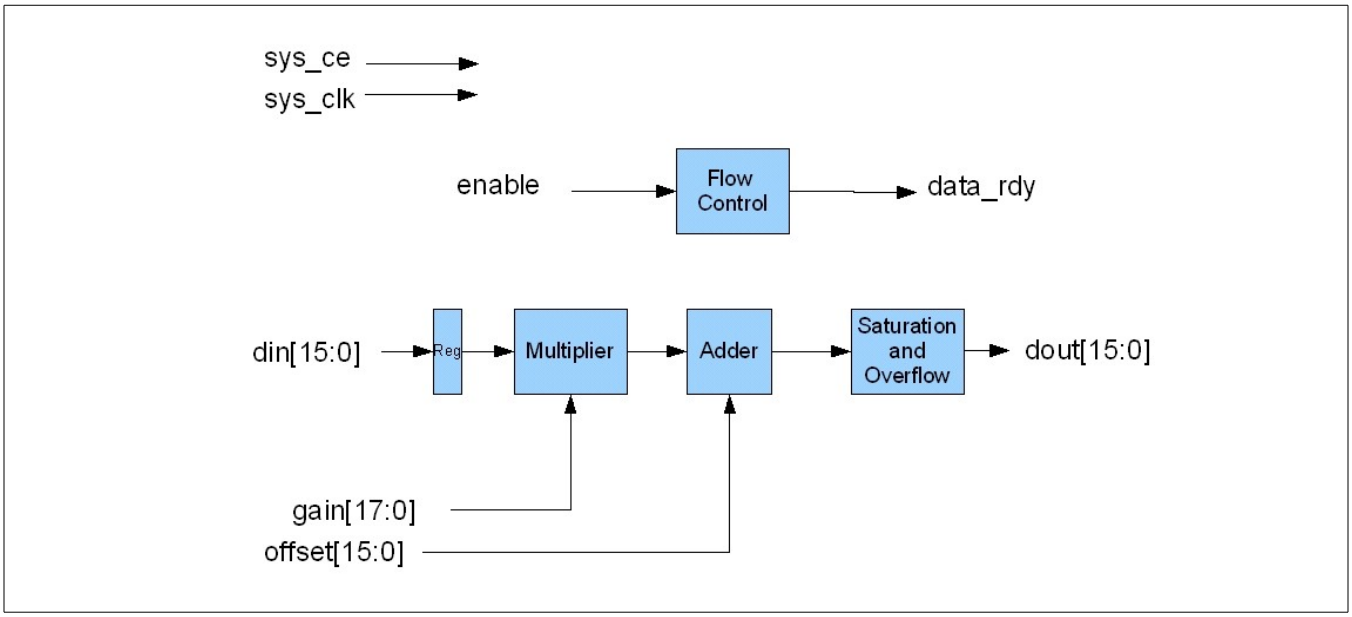


Figure 81. ii_offgain Component Block Diagram

Port	*Direction*	*Function*
reset	In	System reset.
sys_clk	In	Clock input
sys_ce	In	Enable
din[15:0]	In	Input data
dout[15:0]	Out	Output data
gain[17:0]	In	Gain factor, X"10000" = 1
offset[15:0]	Out	Offset factor, 0 is offset of 0
correction[3:0]	Out	The saturation logic mode: 0 = no saturation, 1 = positive saturation, 2 = negative saturation
dvalid	Out	Data is valid when true
tp[31:0]	Out	Test points for debug

Table 93. ii_offgain Component Ports

End of Manual