Video Starter Kit

User Guide

XILINX ®

Revision History

Video Starter Kit
UG217 (v1.5) October 26, 2006

The following table shows the revision history for this document.

Date	Version	Revision
12/22/05	1.0	Initial Xilinx release.
02/13/06	1.1	Edits throughout the document.
03/14/06	1.2	Minor edits in Chapter 6. Replaced Figure 6-5.
06/27/06	1.3	Minor edits.
10/10/06	1.4	Edits and additions to Chapter 5, "VSK Diagnostics and Support Tool Kit" and Chapter 7, "Compiling the VIODC FPGA Design."
10/26/06	1.5	Updated Figure 5-6 and Figure 5-9. Added new Table 5-5.

Contents

Preface: About This Guide

Chapter 1: Video Starter Kit Overview

Chapter 2: Developing Video Applications In System Generator

Chapter 5: VSK Diagnostics and Support Tool Kit

Chapter 6: VSK Tutorial

Chapter 7: Compiling the VIODC FPGA Design

Appendix A: VSK I/O Connector Location Pictures

Schedule of Figures

Chapter 1: Video Starter Kit Overview

Chapter 2: Developing Video Applications
In System Generator

Chapter 3: EDK Integration

Chapter 4: Hardware Co-Simulation

Chapter 5: VSK Diagnostics and Support Tool Kit

Chapter 6: VSK Tutorial

Chapter 7: Compiling the VIODC FPGA Design

Appendix A: VSK I/O Connector Location Pictures

Schedule of Tables

About This Guide

This user guide provides a description of the Video Starter Kit (VSK) contents, features, hardware, and software. The Video Starter Kit hardware consists of a ML402 FPGA development platform with a Video Input and Output Daughter Card (VIODC) and an LVDS video camera. The Video Starter Kit can be used with System Generator to develop EDK processing cores that process live video streams.

Guide Contents

This user guide contains the following chapters:

- Chapter 1, "Video Starter Kit Overview" – provides a kit overview with a brief description of the ML402 development platform, the VIODC, and the LVDS video camera.

- Chapter 2, "Developing Video Applications In System Generator" – The Video Starter Kit provides for both simulation and real-time operation for each of the components in a video system.

- Chapter 3, "EDK Integration" – details the design-flow for incorporating a MicroBlaze™ processor into MVI framework. In particular, it describes using the EDK processor block in System Generator and the automatically generated software drivers to read and write data to the System Generator design.

- Chapter 4, "Hardware Co-Simulation"– provides a description of the hardware co-simulation interfaces that make it possible to compile a System Generator diagram into an FPGA bitstream and associate this bitstream with a new run-time hardware co-simulation block.

- Chapter 5, "VSK Diagnostics and Support Tool Kit"– describes how the VSK diagnostics program serves to tie together the components of the VSK development toolkit into a program for configuring the ML402 and VIODC boards for video processing applications and for providing simple loopback and video processing functions. The VSK support toolkit consists of both hardware and software modules.

- Chapter 6, "VSK Tutorial"– illustrates the process of creating a video *processing core* or pcore which is compatible with systems constructed with the Xilinx Embedded Development Kit (EDK). EDK pcores are reusable peripherals which can be imported into any EDK project.

- Chapter 7, "Compiling the VIODC FPGA Design"– describes how to compile the System Generator `vsk_viodc_xxx.mdl` design to a bitstream (*xxx* is the version number).

- Appendix A, "VSK I/O Connector Location Pictures" – contains pictures showing connection locations on the VIODC, LVDS video camera, the ML402 board, and the ML402 Evaluation Platform.

Additional Resources

To find additional documentation, see the Xilinx website at:

http://www.xilinx.com/literature/index.htm.

To search the Answer Database of silicon, software, and IP questions and answers, or to create a technical support Web Case, see the Xilinx website at:

http://www.xilinx.com/support.

Conventions

This document uses the following conventions. An example illustrates each convention.

Typographical

The following typographical conventions are used in this document:

Convention	Meaning or Use	Example
Courier font	Messages, prompts, and program files that the system displays	speed grade: - 100
Courier bold	Literal commands that you enter in a syntactical statement	**ngdbuild** *design_name*
Helvetica bold	Commands that you select from a menu	**File → Open**
	Keyboard shortcuts	**Ctrl+C**
Italic font	Variables in a syntax statement for which you must supply values	**ngdbuild** *design_name*
	References to other manuals	See the *Development System Reference Guide* for more information.
	Emphasis in text	If a wire is drawn so that it overlaps the pin of a symbol, the two nets are *not* connected.
Square brackets [ ]	An optional entry or parameter. However, in bus specifications, such as **bus[7:0]**, they are required.	**ngdbuild** [*option_name*] *design_name*
Braces { }	A list of items from which you must choose one or more	**lowpwr ={on|off}**
Vertical bar		Separates items in a list of choices	**lowpwr ={on|off}**

Convention	Meaning or Use	Example
Vertical ellipsis . . .	Repetitive material that has been omitted	IOB #1: Name = QOUT' IOB #2: Name = CLKIN' . . . .
Horizontal ellipsis . . .	Repetitive material that has been omitted	**allow block** *block_name* *loc1 loc2 ... locn;*

Online Document

The following conventions are used in this document:

Convention	Meaning or Use	Example
Blue text	Cross-reference link to a location in the current document	See the section "Additional Resources" for details. Refer to "Title Formats" in Chapter 1 for details.
Red text	Cross-reference link to a location in another document	See Figure 2-5 in the *Handbook*.
Blue, underlined text	Hyperlink to a website (URL)	Go to http://www.xilinx.com for the latest speed files.

Video Starter Kit Overview

Key Features

- Standard Video Development Platform for Xilinx FPGAs
- Real Time HD video simulation using Xilinx System Generator's *Hardware in the Loop*
- Video Starter Kit (VSK) includes:
    - Video I/O Daughter Card (VIODC) supports common video interfaces and standards
    - ML402 board with FPGA development platform (Xilinx XC4VSX35 FPGA)
    - LVDS Camera featuring Micron MTV022 automotive CMOS image sensor
    - Xilinx System Generator Software (8.2) for VSK
    - Xilinx ISE Software (v8.2) for VSK
    - XIlinx EDK Software (v8.2) for VSK
    - Application demos
    - Video cables and power supply
- VIODC features:
    - High Definition Component video input and output including 1080I, 720P, and 525P
    - Standard Definition S-video and Composite video input and output
    - Digital Video Interface (DVI) input and output up to 165 MHz
    - VGA analog input and output up to UXGA
    - SDI Serial Digital video interface input with cable equalizer and output cable driver. (The VSK is a demonstration platform only. For HD-SDI verification and compliance, Xilinx recommends using the Cook Technologies SDV board).
    - LVDS camera input
- Software development features:
    - System Generator Blockset for Mathwork's Simulink
    - High-Speed Ethernet Hardware-in-the-Loop co-simulation provides near real-time video simulation
    - High performance Multi-Port DDR memory controller
    - Automatically create MicroBlaze™ video peripherals with memory mapped I/O
    - Import MicroBlaze projects into System Generator models

VSK Hardware Overview

The Video Starter Kit hardware consists of a ML402 FPGA development platform with a VIODC and an LVDS video camera.

ML402 Development Platform

The VSK is based on the Virtex™-4 ML402 XtremeDSP Evaluation Platform. The ML402 board contains a programmable XC4VSX35 FPGA and a number of standard peripheral interfaces, such as Ethernet, RS232, and DDR memory.
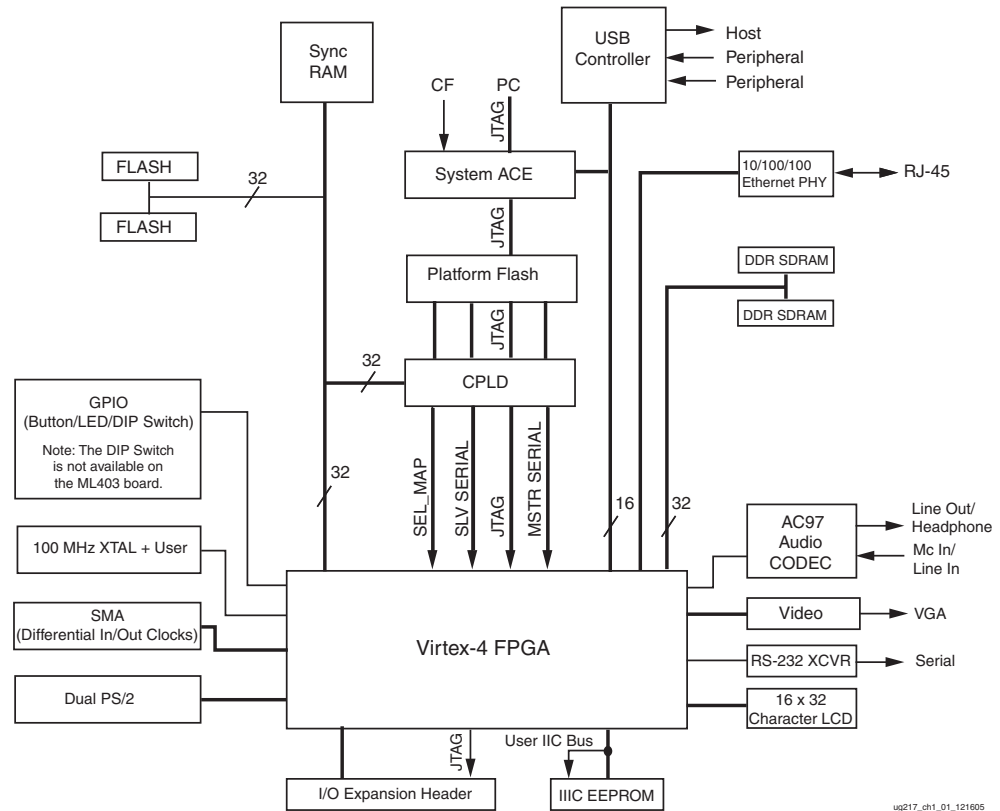
Figure 1-1: **ML402 Block Diagram**

XC4VSX35 FPGA

At the heart of the ML402 board is the XC4VSX35 FPGA, which contains both substantial logic resources (15,360 logic slices), dual port memory (192 x 18-Kbit block RAMs) and very high performance DSP blocks (192 DSP48 slices). In addition to high performance processing capability, the XC4VSX35 FPGA provides access to the VIODC card and the various external interfaces on the ML402 board.

Gigabit Ethernet

The 10/100/1 Gigabit Ethernet port provides a link between the VSK and a PC for high speed video rate simulation. This high-speed simulation capability is known as Hardware-in-the-Loop Co-Simulation. Simulation rates up to 600 Mb/s are achievable.

RS-232 Port

The RS-232 port provides a link to a PC terminal program, such as HyperTerminal. Used for debugging and controlling a MicroBlaze™ embedded processor. It must be connected to the PC using a NULL modem cable.

DDR Memory

A 267 MHz 32-bit wide DDR memory is used to store video frames.

System Ace Controller

The System Ace controller provides access to Compact Flash memory cards which are used to hold demos and bootable FPGA configurations.

I/O Expansion Header

The 64-signal pin expansion header is used to connect to the VIODC. For more information on the ML402 board, refer to the ML402 webpage on the Xilinx website.

Video Input and Output Daughter Card

The VIODC is a standard video interface card for Xilinx development platforms. It is compatible with the ML401, ML402, ML403 boards and other future Xilinx development platforms.

The VIODC is shown in Figure 1-2 with the video ports labeled. The VIODC provides access to high definition and standard definition video streams as well as computer graphics video interfaces, such as VGA over DVI and SDI interfaces. The following interfaces are supported.
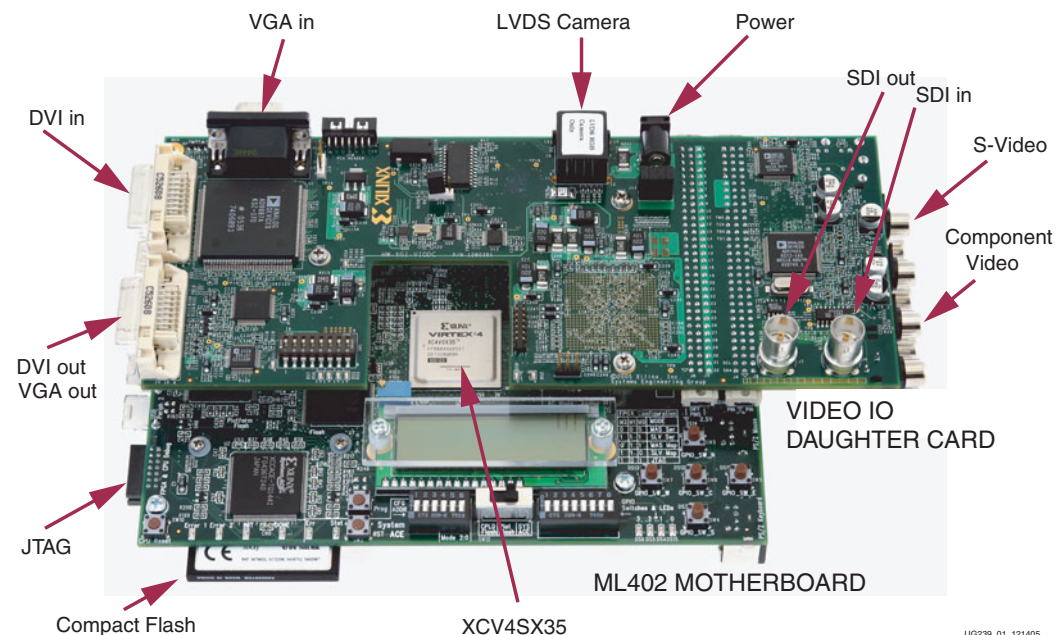
Figure 1-2: **VIODC and ML402 Board with Video Interface Ports Labeled**

LVDS Camera Input

The LVDS camera input port supports the <u>Irvine Sensors</u> LVDS RGB Camera with a <u>Micron</u> MT9V022 1/3 inch CMOS image sensor. The camera provides 752 x 480 pixels at 60 Hz progressive scan. It features low noise and very high dynamic range. The interface is implemented using LVDS signaling over standard Cat-6 Ethernet cables. Note that the LVDS camera interface is not compatible with Ethernet.

Component Video I/O

The Component Video I/O uses standard RCA connectors to provide High Definition (HD) video to the VIODC. Component Video is encoded as YPbPr video channels. The Component Video input on the VIODC supports 1080I, 720P, and 525P video standards. The Component Video interface devices on the VIODC support 10-bit digital video.

DVI Digital Video I/O

The VIODC supports DVI video input and output. DVI is commonly used to interface to flat panel displays and computer graphics cards. The VIODC DVI interfaces support up to 165 MHz pixel clocks. In addition to computer graphics, DVI is also used to carry HD video and is commonly found in high-end consumer video equipment, such as plasma displays, and can be found on some DVD players. The DVI ports can also be connected to HDMI interfaces by using a simple adapter.

S-Video and Composite Video

The VIODC supports S-Video inputs and outputs. These interfaces can be configured to support NTSC, PAL, and virtually any other Standard Definition (SD) video format.

SDI Video Interface

A complete SDI video interface capable of supporting both SD and HD SDI is included with the VIODC. The SDI standard is a high-speed serial interface used to carry video over coax cable. It is generally used in a studio environment. The SDI system includes cable equalizers and genlock circuitry. (The VSK is a demonstration platform only. For HD-SDI verification and compliance, Xilinx recommends using the <u>Cook Technologies SDV board</u>).

XCV2P7 FPGA

The VSK also includes a Xilinx XCV2P7 FPGA, which is used to interface to the various video interfaces, as well as the ML402 main board. It features Multi-Gigabit Transceivers (MGTs), which are used to support the SDI interface. It also enables the VIODC to be used in a stand-alone fashion.

VSK Demo Application

Several demo applications are included with the VSK. One demo (Figure 1-3) shows how the VSK can be configured as a video processor. This demo application is included in the VSK_diagnostics design included in the VSK examples directory. It can be used to apply some common video filters to the video from the RGB LVDS camera or other video sources.
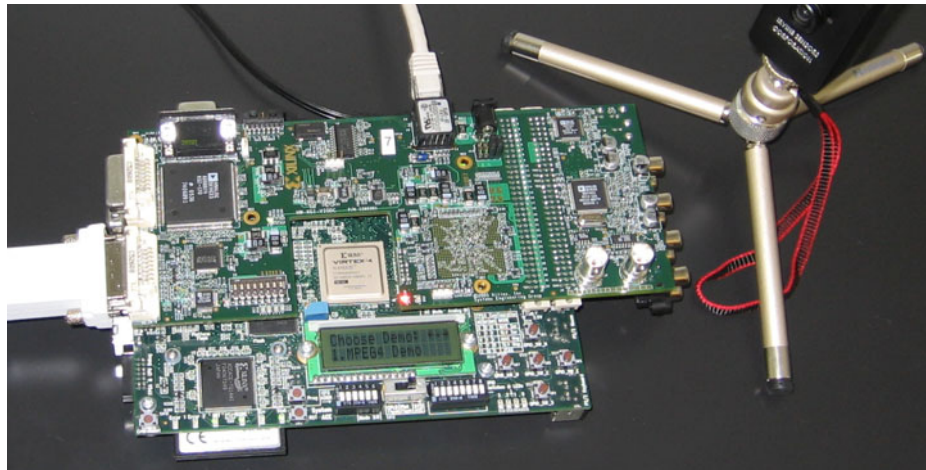
Figure 1-3:   **RGB Camera Demo Setup**

Figure 1-4 illustrates a common video processing pipeline. The design implements a sequence of video operations including gamma correction, Bayer filtering, and color space correction. It is implemented using the System Generator blockset for MATLAB Simulink. System Generator is used to export this design as an EDK pcore, which is a standard Xilinx peripheral for embedded processors. After a pcore is created, it can be used in any Xilinx EDK project. Any EDK project can use a pcore, even if they are targeted to other development boards or FPGAs.
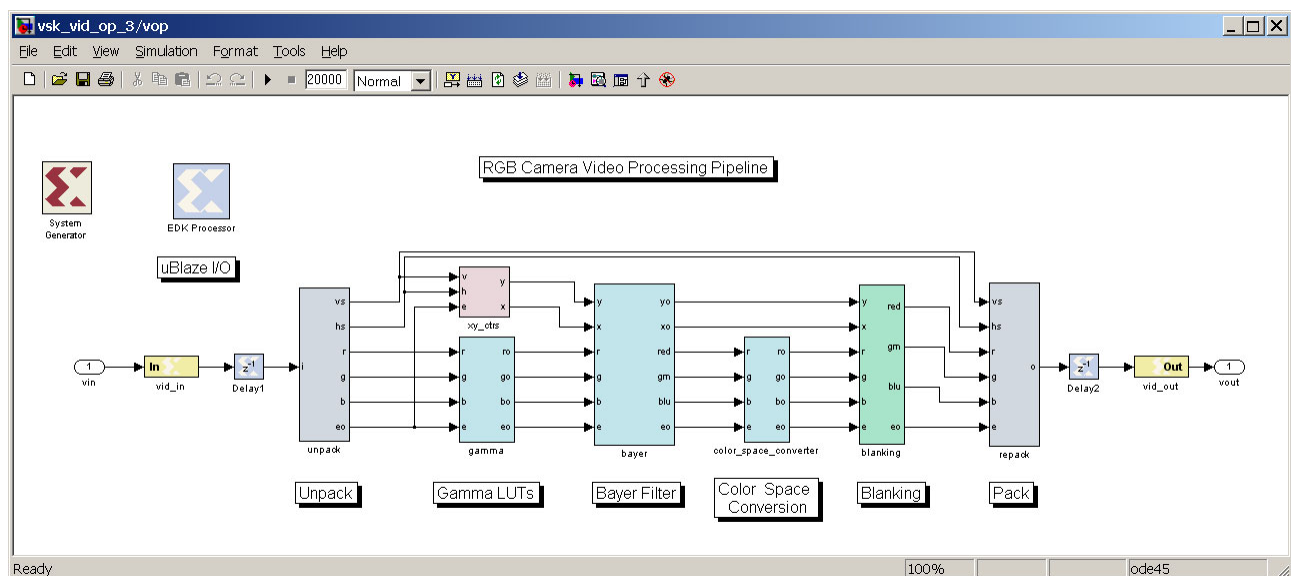
Figure 1-4:   **RGB Camera Video Processing Pipeline**

The RGB camera processing pipeline design has a video input and a video output port. In the complete application, these video buses are connected to other pcores implementing

video processing, memory or I/O to the VIODC. The block diagram of the VSK diagnostics program is shown in Figure 1-5.
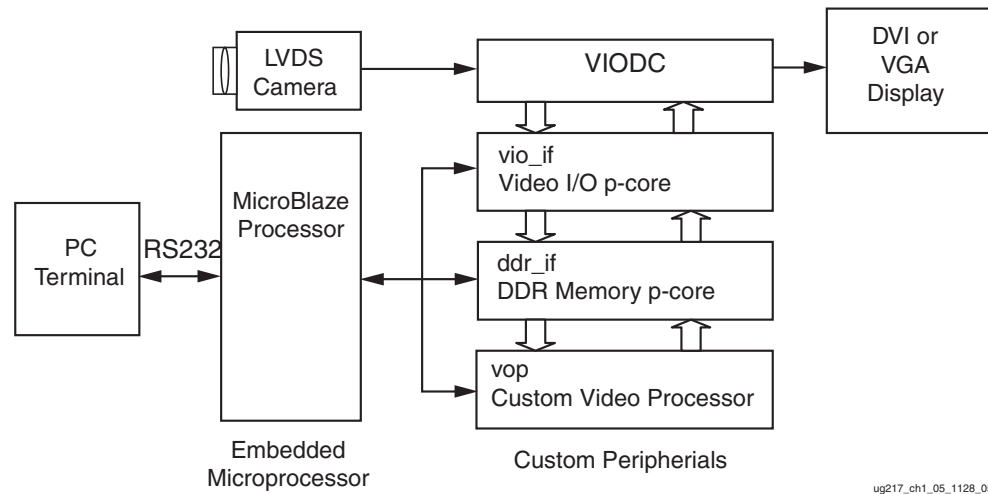
Figure 1-5: **Block Diagram of VSK RGB Camera Demo Included in the VSK**

For this application, the embedded MicroBlaze processor is used to configure the video processing pipeline. It communicates with memories and registers in the video pcores via System Generator shared memory primitives. The hardware logic and software drivers required by MicroBlaze to communicate with the shared memories in the pcore are automatically generated by System Generator during compilation.

The video processing demo is included in a pcore named vop for Video Op. The Sysgen design is named vsk_vid_op_6.mdl. This demo is part of the VSK diagnostic program and can be found in the VSK examples directory. For more information, refer to Chapter 5, "VSK Diagnostics and Support Tool Kit."

Software and Application Updates Available Online

VSK software and applications are supported by a VSK web page and the latest software and applications versions can be found there. The software support is integrated into System Generator and will be upgraded as new System Generator versions are released. New applications for the VSK will be posted on the VSK web page as they are developed.

Software Support Package Overview

The VSK includes hardware, software, and applications. Xilinx software is used to create applications which run on the VSK and Xilinx FPGAs. Three basic software flows are supported. These are illustrated in Figure 1-6, Figure 1-7, and Figure 1-8.
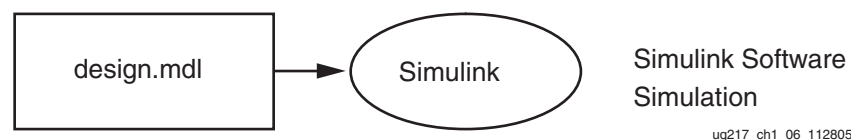
Figure 1-6: **Software Simulation Flow**

Software Simulation

In the first flow, called software simulation, Simulink designs that are constructed from the System Generator blocks are compiled and run in MATLAB Simulink. Figure 1-6 shows the software flow for software simulation. This flow is quick and easy to develop and offers good performance using the built-in MATLAB floating-point matrix operations. Larger systems, however, slow down significantly, and it is difficult to use this approach with live video streams. Additionally, problems occur when the design is implemented in fixed-point blocks for FPGA implementation. This can result in very low pixel rates due to poor simulation performance, often requiring hours per frame of video.

Hardware Implementation

The second software flow (Figure 1-7) compiles the user's design to hardware and runs the hardware on the VSK video development platform. This allows the video IP to be tested using live video streams.
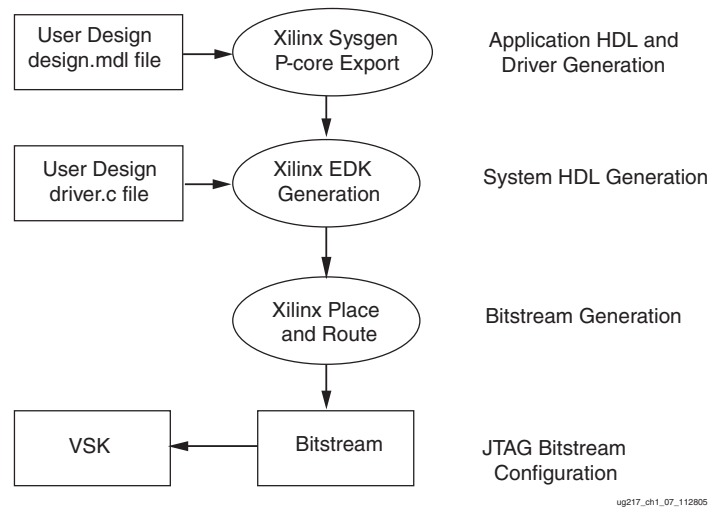
Figure 1-7: **Real-Time Deployment Flow**

Hardware Co-Simulation

The third type of software flow (Figure 1-8) is a hybrid of the software simulation and hardware deployment called Hardware-in-the-Loop co-simulation. In it, the bulk of the design is generated as in the real-time deployment flow. However, hardware data streams can be routed to the Simulink software simulation using the Xilinx System Generator Hardware co-sim engine.

The advantage of this mode is that small video filters which are part of a larger video processing system can be run in simulation, while the bulk of the video system is implemented in real time hardware. If buffering is employed, the software simulation can operate on stored video frames from the video stream at a reduced frame rate. Using the Ethernet Co-Simulation, near real-time video rates can be achieved. This translates to a few frames per second using 640 x 480 video.
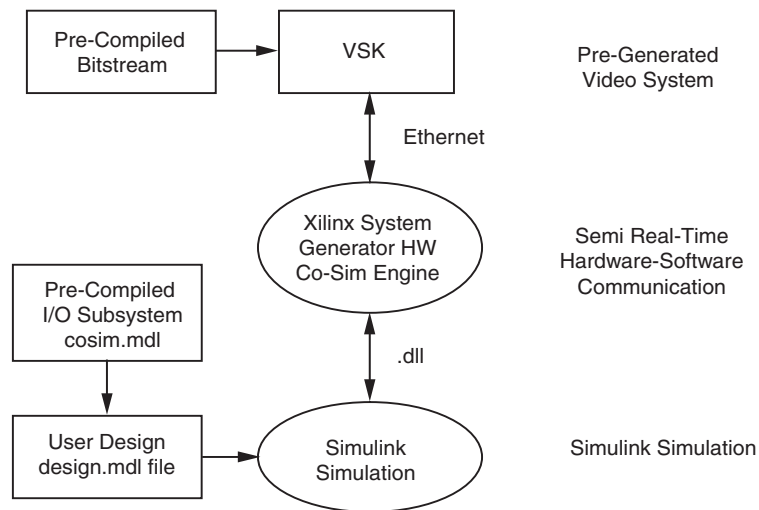
Figure 1-8: **Hardware-in-the-Loop Flow**

The pre-generated bitstream and co-sim subsystem can be generated from another system generator diagram or it can be an existing EDK project. The user can use the EDK import feature in System Generator to import an EDK project as a co-sim block.

VIODC HDL Support Package

While the above software flows leverage the advantages of developing video IP in System Generator/MATLAB/Simulink, users may prefer to use traditional HDL design flows.The Video Starter Kit also includes two demonstrations written in Verilog. These examples exercise all the video functions on the VIODC and are generally self-explanatory. In addition, these demos can run on a stand-alone VIODC board.

Refer to the *DVI, VGA, and Component Video Demonstration User Guide* and the *SDI Video Demonstration User Guide* for further information.

System Generator Support

System Generator includes several features that are useful for developing video applications with the Video Starter Kit. These are outlined below and additional documentation for each of these features can be found in the VSK document package. In addition, tutorials are included in the examples directory to assist in learning to use these powerful features.

DDR Memory Controller

The VSK includes a capable multi-port memory controller. The controller supports the DDR memory on the ML402 board. It can also be targeted to other boards and is fully configurable for port size and number of ports. The controller contains a simulation model and can be run in HDL co-simulation mode, or compiled to run in real time and as part of hardware co-simulation block.

Pcore Export and EDK Import

Pcore export is a new method of generation from Simulink diagrams that allows the user diagram to be imported into any EDK project as a Pcore. Conversely, EDK import allows EDK projects to be imported into System Generator diagrams. Pcore export is used to generate the three pcores in the VSK diagnostic demo from System Generator diagrams.

Multiple Subsystem Generator

The Multiple Subsystem Generator flow allows System Generator models to include multiple clock domains. This flow is used to generate the design for the VIODC FPGA, which is included in the VSK diagnostics program.

Ethernet Co-Sim

High-speed Ethernet co-sim and point-to-point Ethernet co-sim are supported by the System Generator for the VSK. To achieve near real-time video rates, reusable buffer blocks are included in the Ethernet demos that are included in the VSK.

Diagnostics

A diagnostics program called the vsk_diagnostics is included with the VSK. Refer to Chapter 5, "VSK Diagnostics and Support Tool Kit" for more information. The diagnostics include System Generator designs for the VIODC FPGA, and three pcores integrated into an EDK project for the ML402 FPGAs. They also include software routines to configure the video interface chips on the VIODC and to control the video processing pcore, video interface pcore, and DDR memory pcore.

Demonstrations

Several demonstration designs are included with the VSK.

MPEG Decoding Demo

The VSK diagnostics include an MPEG-4 demo, which can be run from the Compact Flash. Refer to the *Video Starter Kit Quick Start Guide (UG239)* and the *MPEG-4 Demonstration User Guide (UG234)* for more details.

VSK Diagnostics Camera Demo

The VSK diagnostics also include a camera demo, which can be run from the Compact Flash. Refer to *Video Starter Kit Quick Start Guide (UG239)* and Chapter 5, "VSK Diagnostics and Support Tool Kit" for more details.

SDI Demo

A demo featuring the SDI interface and written in Verilog is available for the VIODC. Refer to the documents *SDI Video Demonstration User Guide* in the VSK document package for further information.

Video Demo in Verilog

A demo which exercises all the video interfaces on the VIODC is available. Refer to the documents *DVI, VGA, and Component Video Demonstration User Guide* in the VSK document package for further information

Developing Video Applications In System Generator

Overview

Figure 2-1 shows a typical video processing system. In this system, a microprocessor is used to control a video pipeline consisting of a video source and sink, a large memory for storage of video data, and a video processing system.
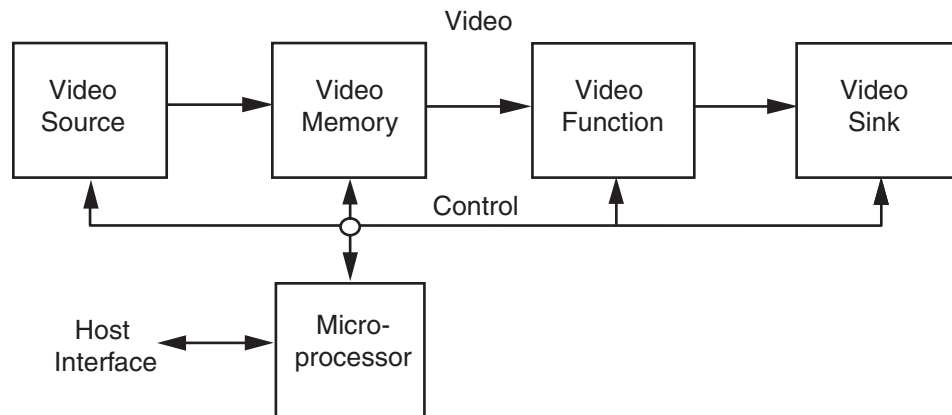
Figure 2-1: **Video System Diagram**

As the video system is being developed, these functions can be implemented in real hardware or in simulation. Simulation of video processing applications creates special challenges for simulation due to both the real-time nature of video streams and the enormous amount of video data required per frame.

Typically, video development requires real-time hardware to prove the video operation on real-time data streams, as well as a simulation environment to develop and test the video processing components. The Video Starter Kit (VSK) provides both for simulation and real-time operation for each of the components in a video system.

Real-Time Operation

Real-time operation (Figure 2-2) is provided by the combination of the VIODC and ML402 development boards. The VIODC provides video sources and sinks to the ML402 and the ML402 is used for real-time processing of the video streams. The ML402 board includes a state of the art Xilinx Virtex-4 FPGA for real-time video processing, memory, and communications peripherals.   Video memory is provided on the ML402 board in the form of DDR SDRAM memory. The ML402 board supports a MicroBlaze processor with a

standard set of peripherals. Host communications can be supported over RS-232, USB, or Ethernet ports.
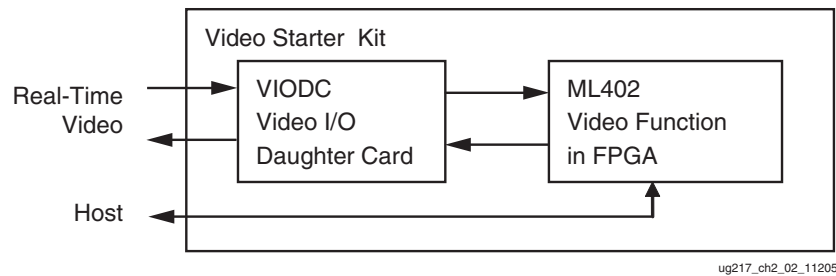
Figure 2-2: **Real-Time Video Processing**

Hardware-in-the-Loop Video Simulation

FPGAs are software programmable, yet they have the unique ability to operate on real-time video streams at clock rates well over 300 MHz.   Using FPGAs to process real-time data during algorithm development is known as Hardware-in-the-Loop Co-Simulation. Hardware-in-the-Loop Co-Simulation (also known simply as hardware co-sim) can be used to quickly demonstrate video processing functions at real-time rates.
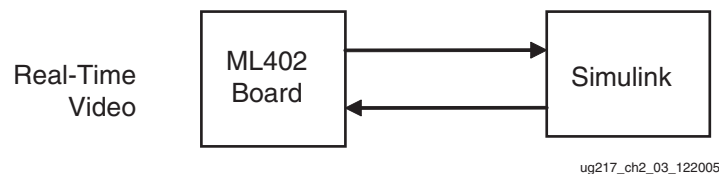
Figure 2-3: **Hardware-in-the-Loop Video Processing**

Hardware-in the Loop Co-Simulation

Xilinx System Generator provides the ability to replace Simulink subsystems with a *hardware co-sim* token. For example, the Figure 2-4 shows a video processing block which implements a gain and offset function on a single video channel. In Figure 2-5 the video processing block has been compiled into a hardware co-sim block.

When the Simulink Play button is pressed to start simulation, the FPGA design containing the gain and offset function is loaded to a development board such as the ML402. Then data is passed from the Simulink source to the function in the FPGA, the clock is stepped automatically, and output data is passed back to the Simulink scope.

Hardware co-sim can also be used with buffer blocks to substantially increase data bandwidth and throughput. When used with Mathworks Video and Image Acquisition Blockset, hardware co-sim can be used to provide non-real-time video simulation.
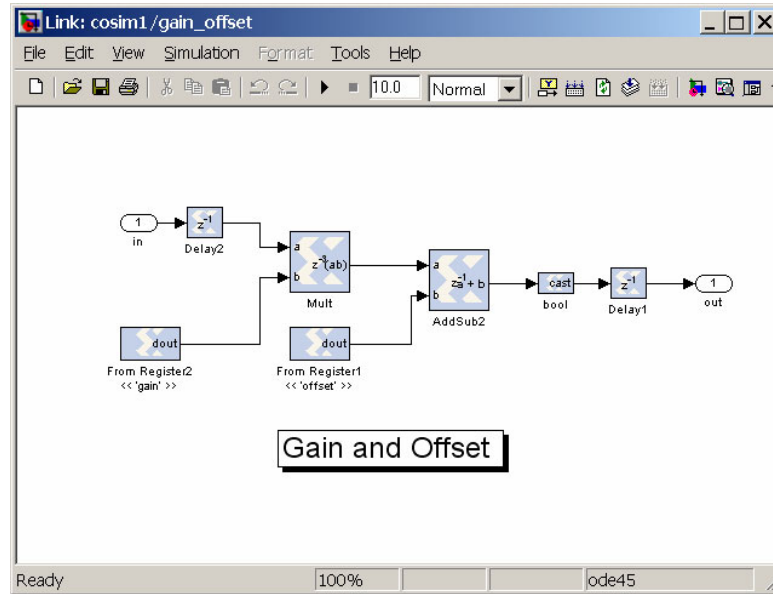
Figure 2-4: **Simulink Diagram Implementing a Gain and Offset Function Using Xilinx System Generator Blocks**
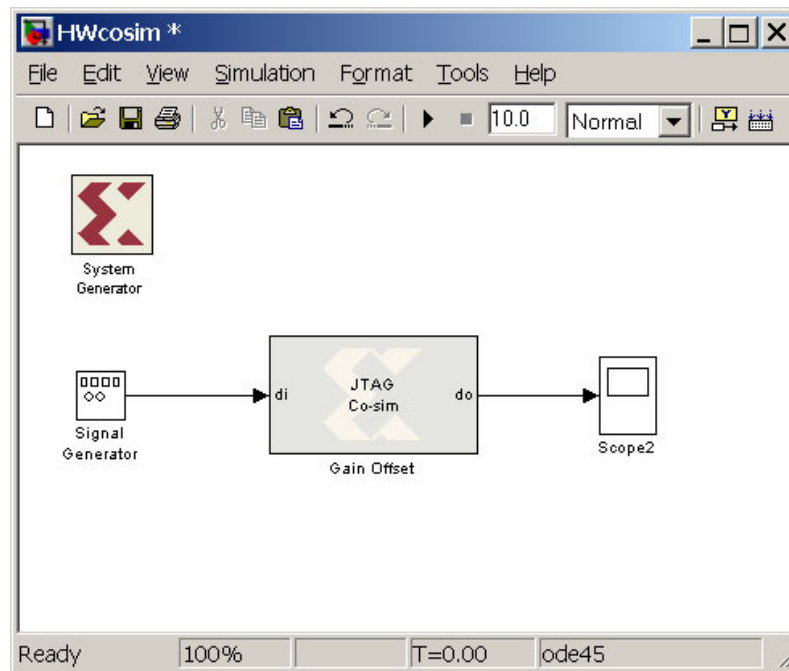
Figure 2-5: **Gain and Offset Function Compiled to a Hardware Co-Sim Token**

Software Simulation Modes

During development, it is very useful to be able to simulate video processing blocks using real video data streams. Software simulation (shown in Figure 2-6) is supported for the VSK using Mathworks Simulink software in conjunction with the Xilinx System Generator. During simulation, real-time operation is simulated by operating on captured video in the

video memory at reduced video clock rates. Although software simulation is not as fast as real-time hardware operation, Simulink provides a quick and facile method of developing and testing video processing functions. In addition, standard video processing functions available for Simulink, such as the Image Processing Toolbox, allow video processing functions to be quickly prototyped and tested.
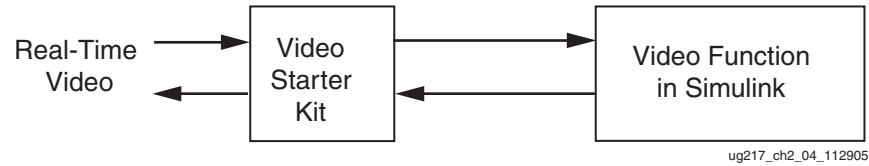
Figure 2-6: **Software Simulation Using Live Video Signals with Simulink**

The infrastructure required to source and sink video streams from the VSK to the Simulink simulation is provided by the Xilinx System Generator co-simulation feature, which allows simulation diagrams to communicate directly with hardware. System Generator also provides the ability to simulate flowgraphs constructed from Xilinx Blockset components, as well as VHDL and Verilog *black boxes* using HDL simulators such as ModelSim.

Hardware-Software Systems

Video systems often require a control processor. The processor typically is used to communicate with a host system, set up video processing operations, compute coefficients and generally operate as a low rate data processor. Xilinx System Generator and the Embedded Development Kit (EDK) software tools can be used together to implement and simulate a system with a processor and FPGA video processor functions operating on live video streams.

Generating a Video Processor as an EDK Pcore

The Xilinx EDK can be used to construct processor systems with integrated memory and peripherals. Processor peripherals, such as memory and Ethernet interfaces, can be included into EDK projects and are known as processor cores or *pcores*. Xilinx System Generator can be used to generate custom pcores for functions such as video processing. Figure 2-7 illustrates a processor system with standard and custom peripherals. The processor is used to configure the video processing pipeline, and video data is passed directly between stages of the video pipeline. This particular configuration with three custom pcores is used for the VSK camera processing demo, VSK, and the VSK diagnostics.
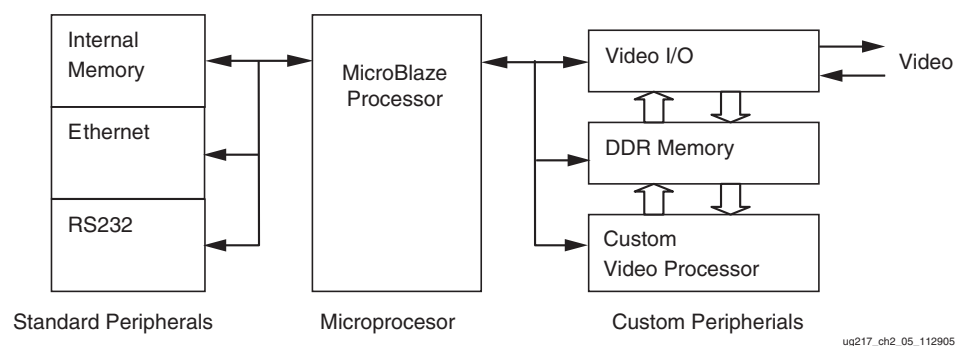
Figure 2-7: **MicroBlaze Processor with Peripherals and Three Custom Video Peripherals**

The above system can be used to implement and test video peripherals. After developed and validated, a pcore peripheral is able to be incorporated into any EDK system. System Generator can be used to construct and test EDK pcores.   This facility is available as the EDK Export compilation target in System Generator.

Hardware-Software Communication

Memory Mapped Hardware

Hardware is often controlled by software programs. This requires communication channels between the hardware peripheral and the software program. System Generator includes special register, FIFO, and memory blocks which can be automatically mapped into the processor's function space. These blocks are termed *shared memories* because they are shared between the processor and the System Generator model diagram. These blocks make it simple to use a processor to read and write hardware memories. See Figure 2-8.
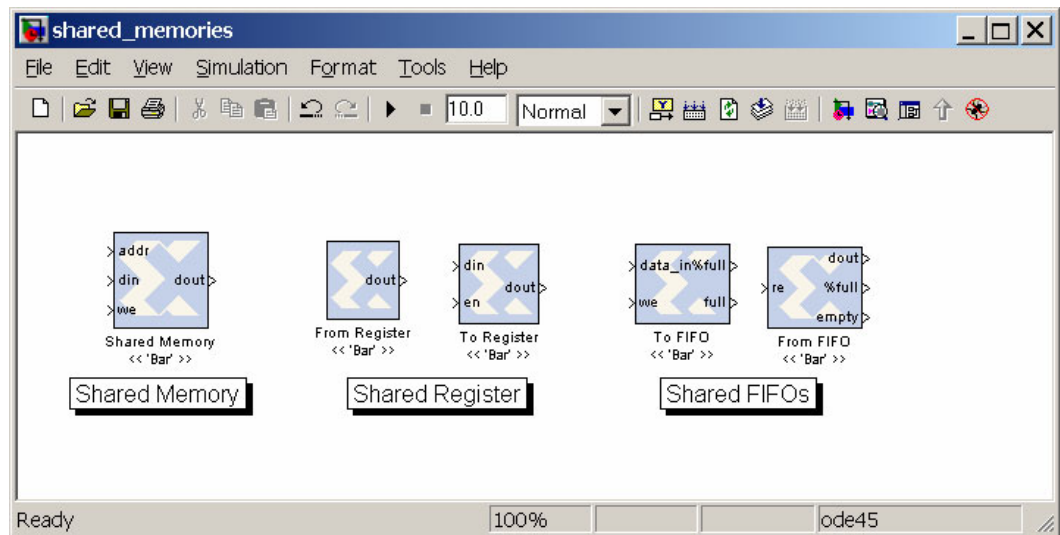
Figure 2-8:   **System Generator Shared Memory Blocks**

MicroBlaze Processor Communicating with a Shared Memory

When a diagram (shown in Figure 2-9) containing shared memory blocks is compiled as a EDK pcore, the shared memory objects are mapped into the function space of the C program, using an FSL communications link.
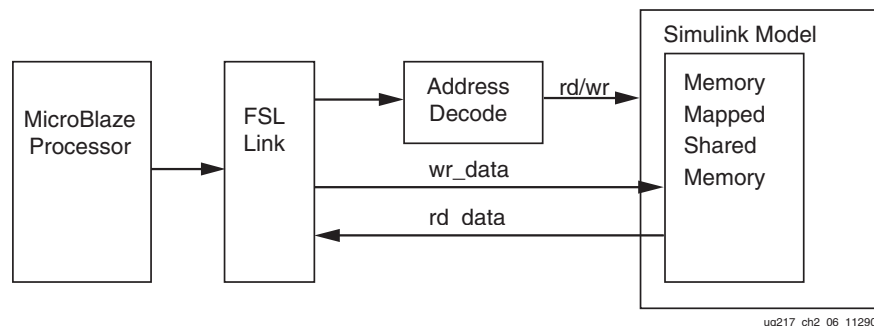
Figure 2-9:   **MicroBlaze Processor Communicating with a Shared Memory**

Hardware-Software Co-Simulation

Often both hardware and software are developed concurrently. The development of video processing peripherals generally require the development of both hardware and software drivers for the hardware. This task is eased if software co-simulation is available as hardware is simulated. Hardware-software co-simulation is available as part of System Generator hardware co-simulation. This facility is named EDK co-simulation.

EDK Co-Simulation

System Generator allows for the incorporation of a MicroBlaze block into a Simulink diagram. When a System Generator diagram that includes a MicroBlaze block is compiled and run under hardware co-simulation, the processor is able to communicate with any shared memory objects which are also in the diagram. For example in Figure 2-10, the MicroBlaze processor is able to read from register X and write to register Y. In addition, a C header file containing function calls is generated to allow for easy communication with shared memory objects.
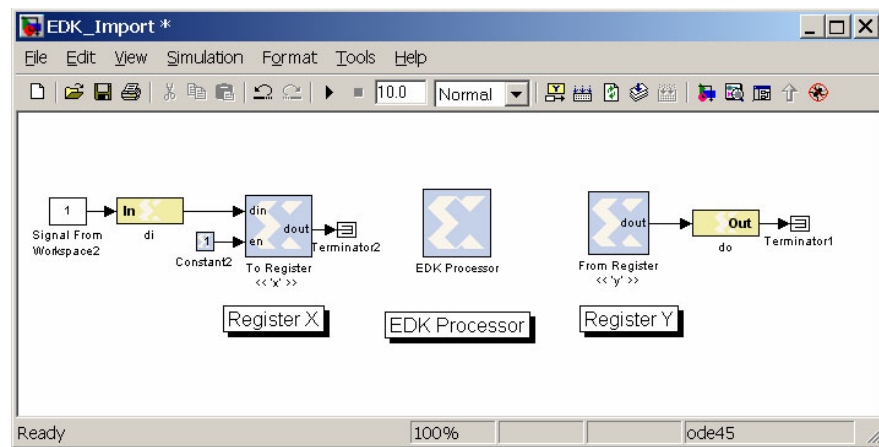
Figure 2-10: **EDK Import with Registered IO**

If the EDK project includes a Serial Port, the user can interact with the processor via the RS-232 interface.

VSK Video Processor Development System

The VSK can be used to develop video processing peripherals as EDK pcores. To abstract the low level details of the video processing, a pre-built system called VSK1 has been developed to provide video streams directly to a user pcore. It includes a processor and pcores to interface to the VIODC and DDR memories and process video. These pcore designs can be used as is or modified to create new pcores.

The ML402 board and the VIODC are configurable hardware systems which both contain FPGAs. FPGA designs written in System Generator have been developed for each of these FPGAs, and MicroBlaze software drivers have been developed to provide access to the various functions incorporated into the FPGAs. These FPGA designs and software drivers are used in the VSK diagnostics and demos. As a set, they provide a video pcore development environment for Video processors and are described in the following sections.

ML402 FPGA

The ML402 software and hardware are partitioned into a MicroBlaze and three pcores.
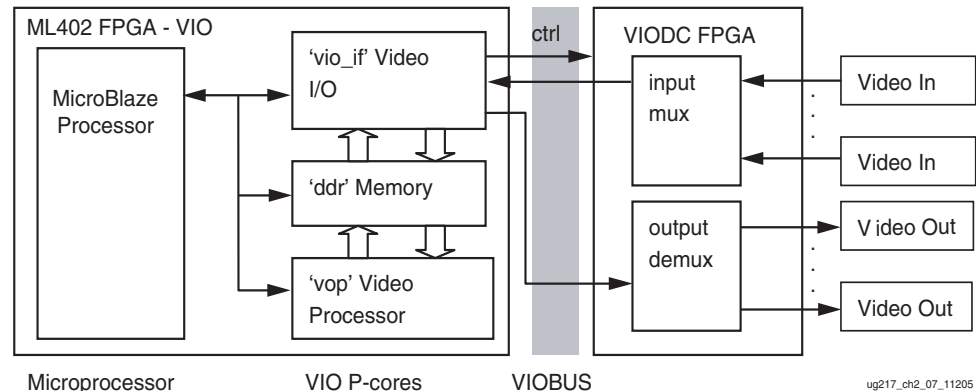
Figure 2-11: **ML402 FPGA**

MicroBlaze Subsystem

The MicroBlaze subsystem consists of a simple MicroBlaze processor, a 64-KB internal memory and an RS-232 system. It can be extended to include access to external SRAM and the various peripherals on the ML402 board.

vio_if Pcore

The vio_if pcore is used to communicate with the VIODC over the vio bus. It supports one video input and one video output bus. Each video bus is 26 bits plus a pixel enable. This vio_if pcore also provides IIC communication to the devices supported by the VIODC FPGA, as well as a small serial bus to communicate with internal VIODC registers.

ddr_if Pcore

The ddr_if pcore provides access to DDR memory. It is designed to store and playback video sequences and provide access to the MicroBlaze. The DDR pcore is built around the Multiport System Generator DDR Memory controller.

vop Pcore

The pcore labeled vop is the pcore which contains a basic video processing pipeline. The MicroBlaze project provides video input and output streams to the core. The existing core can be replaced if modified to develop a new video processor.

VIODC FPGA

The Video Starter Kit contains an XCV2P7 FPGA on the VIODC. An FPGA design has been developed to allow the MicroBlaze processor to configure the various video interface ICs and select from among the various video sources. The VIODC connects to the ML402 board over a 64-pin interface bus. This bus is named the VIOBUS.

The VIODC provides an interface to the various video interface ICs and routes video to and from the ML402 FPGA.

Video Sources

The VSK allows for various video source to be used to source a video stream. Live video streams are piped down from the VIODC card. The VIODC supports VGA, DVI, SDI, HD component video, SD S-video, and composite video. In addition, test patterns can be used to create live video streams. When the viodc.bit FPGA program is loaded to the VIODC, video sources available from the VIODC can be selected by configuring the VIODC using the MicroBlaze. (SDI video input is not available in version 1 of the VIODC program). During simulation, video sources can include MATLAB matrices and MATLAB Simulink Image Processing Blockset video sources.

Video Sinks

The Video Starter Kit allows video streams to be driven to any of the video output ports. The VIODC supports VGA, DVI, SDI, HD component video, SD S-video, and composite video outputs. When the viodc.bit FPGA program is loaded to the VIODC, video sources available from the VIODC can be selected by configuring the VIODC using the MicroBlaze. During simulation, video can also be routed to MATLAB matrices and MATLAB Simulink Image Processing Blockset video displays.

An additional VGA output suitable for VGA or 525P video output is provided on the ML402 FPGA. It can also be used as a diagnostic display.

EDK Integration

Overview

Embedded processors are important components inside any MVI system. The Xilinx Embedded Development Kit (EDK) is an integrated software solution for designing embedded processing systems, and the MicroBlaze™ 32-bit soft-processor core is supported by the EDK. This chapter details the design-flow for incorporating a MicroBlaze processor into MVI framework. In particular, it describes using the EDK processor block in System Generator and the automatically generated software drivers to read and write data to the System Generator design.

Two methods are described:

- System Generator design exported into an EDK system
- EDK project imported into a System Generator design for hardware co-simulation.

Communication between an EDK processor and the System Generator design is accomplished via shared-memories—registers, FIFOs, and RAMs.

MicroBlaze Processor Interface

The MicroBlaze processor interface is exposed through the EDK processor block provided by System Generator. Figure 3-1 shows the communication between user-defined logic and the MicroBlaze processor.
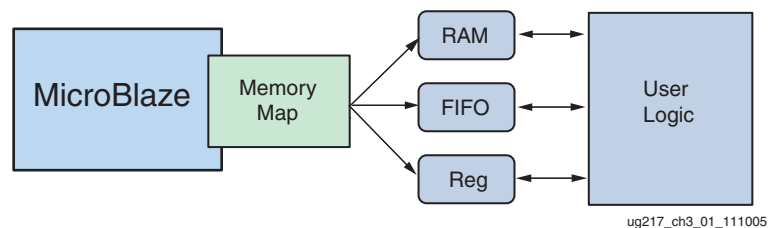
Figure 3-1: Memory-Mapped User Logic

Memories used in a user's design form the processor interface into the user-logic. These memories are *associated* with a MicroBlaze processor through the EDK processor block's GUI interface. After an association is made, System Generator automatically creates a memory-map that marshals data to and from the processor. Memories that are associated to the processor can be accessed using C-code device drivers automatically generated by System Generator.

EDK Pcore Export Mode

When used in pcore export mode, the memory map block shown in Figure 3-1 and all the blocks to its right are packaged into a pcore peripheral. Software drivers and documentation for the memory-map interface are also generated and delivered with the peripheral.

EDK Import Mode

When used in EDK import mode, an EDK project file is imported into System Generator by running the EDK Import Wizard. When the import wizard completes, the EDK system is pulled into the System Generator design as a black-box. During the import process, the EDK system is augmented with Fast Simplex Link (FSL) interfaces that communicate with the memory-map shown in Figure 3-1.

Adding a Processor to a System Generator Design

The EDK Processor Block

A processor can be added to a System Generator design by using the EDK processor block, which can be found in the System Generator Index and Control logic libraries. Figure 3-2 shows an image of the EDK Processor block.

Figure 3-2: **EDK Processor Block**

Double clicking the block opens the block's GUI, shown in Figure 3-3. Refer to the *System Generator User Guide* for a more detailed explanation of the GUI.

Interfacing the EDK Processor to User Logic

Shared-memories instanced in a user's design can be associated to an EDK Processor by adding that memory into the processor's memory map. Figure 3-3 shows an EDK Processor block with four shared memories added to the memory-map: a RAM, a register, and two FIFOs. Memories visible to the EDK processor are listed in the *Available Memories* pull-down menu. Selecting a memory and pressing the Add button adds that memory into the processor's memory-map. Right-clicking on the *Memory Maps* tree-view reveals a pop-up menu that provides services, such as allowing memories to be removed from the memory map and re-syncing the memory map. Re-syncing a memory map regenerates the decoder logic used to marshal data to their respective memories.
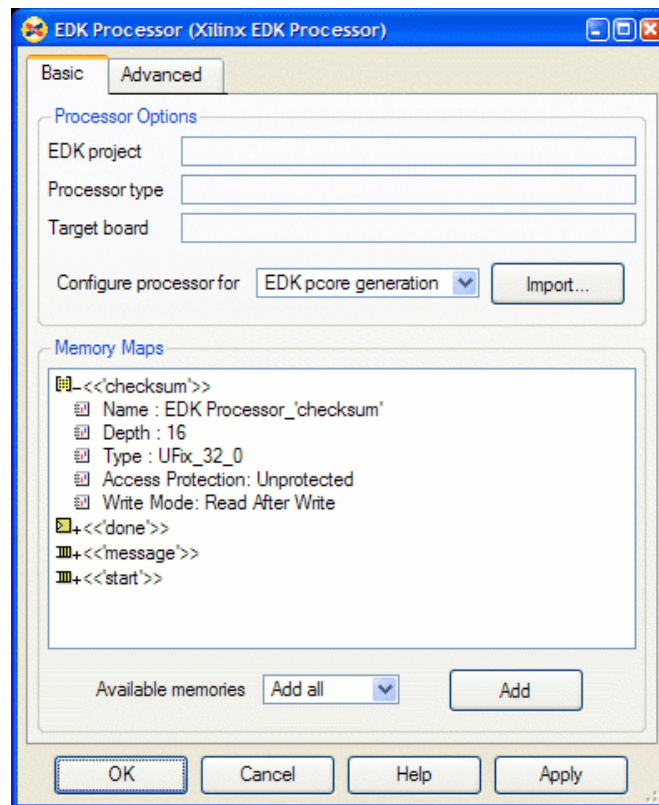
Figure 3-3:   **EDK Processor GUI**

Exporting the Design as a Pcore

The Xilinx Platform Studio (XPS) is a tool that is shipped with the EDK. XPS is an integrated development environment that allows a processor to be created, configured with IP, and for software to be written and compiled into the processor's bitstream. When configuring a processor in XPS, a user is presented with a list of available IP that can be connected to a processor. Each IP that can be added on as a peripheral to a processor must be packaged as a pcore. System Generator provides a compilation target that compiles a user design into a pcore.

To export a user design as a pcore, the EDK processor block must be used in conjunction with the *Export as a pcore to EDK* compilation target found in the System Generator block. The EDK processor must to be configured for EDK export mode. This is done by setting *Configure Processor For* to *EDK Pcore Generation* as shown in Figure 3-3.

After configuring the EDK processor, an EDK pcore can be generated by using the System Generator compilation GUI. Refer to Figure 3-4. Double click on the System Generator block and set *Compilation* to *Export as a pcore to EDK*. Press the *Settings…* button to bring up the EDK Export Settings dialog box. This controls where the pcore is exported to and assigns a version number to the pcore.

Refer to *System Generator Compilation Types* chapter in the *System Generator User Guide* for more detailed information on the EDK export compilation target.
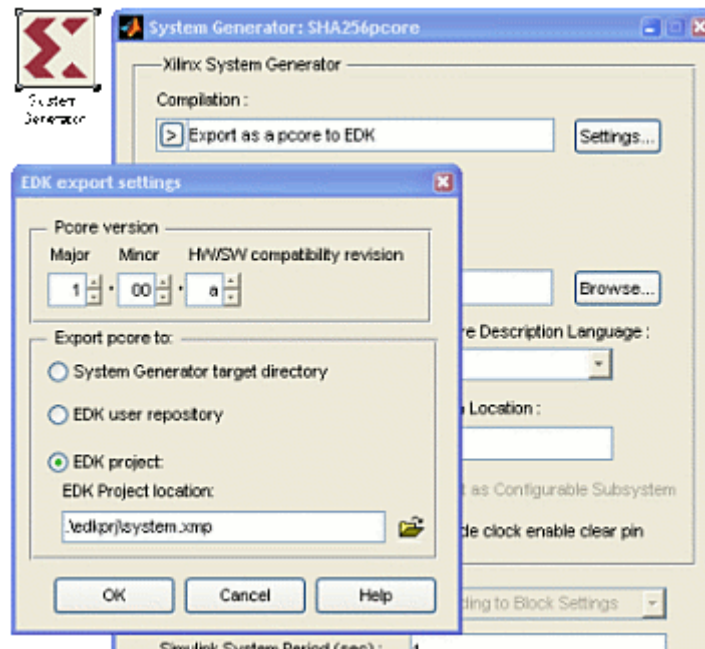
Figure 3-4: **Export as a Pcore to EDK**

The pcore compilation process produces a collection of files organized in a specific directory structure. The files and directory structure are listed in Table 3-1.

Table 3-1: **Pcore Directory Structure**

Directory Name	File Type/ Extension	Description
data	mdd, mpd pao, tcl	Data files used by the EDK to configure a pcore. The mdd and tcl files are used by LibGen to generate device drivers for the pcore. The mpd and pao files describe the interface of the pcore.
doc	pdf	An Adobe PDF file documenting the memory-map interface and also providing information and examples on how to read and write to the pcore.
doc/html/api	html	An html version of the PDF file.
hdl vhdl verilog	vhd v	hdl directory contains two other directories: VHDL files used by the pcore. Verilog files used by the pcore.
netlist	edn, ngc	Precompiled netlists used in the pcore.
src	Makefile, c, h	Template C header and source files used during device driver generation.

A tutorial showing this design-flow can be found in the Hardware Software Co-Design in System Generator chapter in the *System Generator User Guide*.

Importing an EDK Project into System Generator

A project created in the EDK can be imported into a System Generator project by using the EDK Import Wizard. Projects imported using the wizard can only contain one MicroBlaze processor. The EDK Import Wizard can be launched through the EDK processor block GUI

Figure 3-5 shows one way to launch the EDK Import Wizard. If the EDK project string is empty, selecting *HDL netlisting* in the *Configure processor for* parameter launches the EDK Import Wizard.
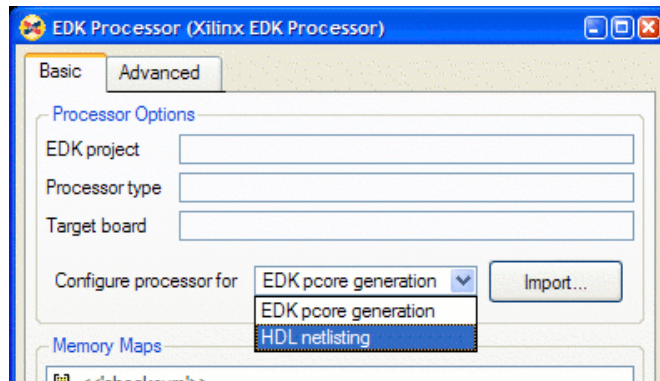
Figure 3-5:   Launching Import Wizard

If an EDK project has been imported previously, the EDK project field contains the path to the project that has been imported. If the processor configuration for that project has changed, the import wizard should be rerun. This can be achieved by clicking on the Import button.

The EDK Import Wizard (Figure 3-6)modifies the given EDK system by adding in a pair of FSL FIFOs that will be used by the memory map created by the EDK processor block when shared-memories are associated with the processor. The EDK is next called upon to create the processor netlist. Finally, the EDK Import Wizard creates the wrapper and associated configuration files required for the netlists to be imported into System Generator. Following this, when the EDK processor block is configured for *HDL netlisting*, System Generator is capable of generating netlists containing the MicroBlaze processor. This allows designs containing a MicroBlaze processor to be compiled for hardware co-simulation.
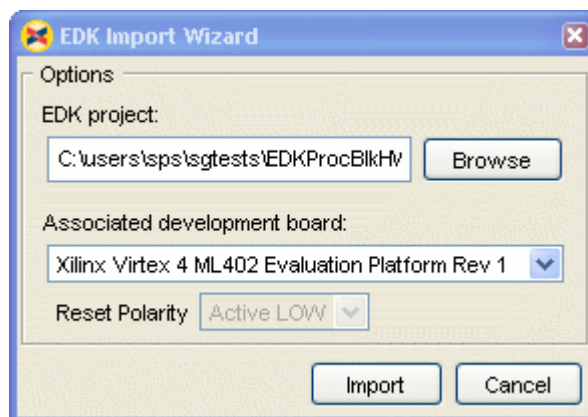
Figure 3-6:   **EDK Import Wizard**

An EDK Processor block can be compiled into a co-simulation block by selecting one of the hardware compilation targets available, as shown in Figure 3-7. For the ML402 board, three options are available: JTAG, network-based, and point-to-point. For information on hardware co-simulation, refer to Chapter 4, "Hardware Co-Simulation."
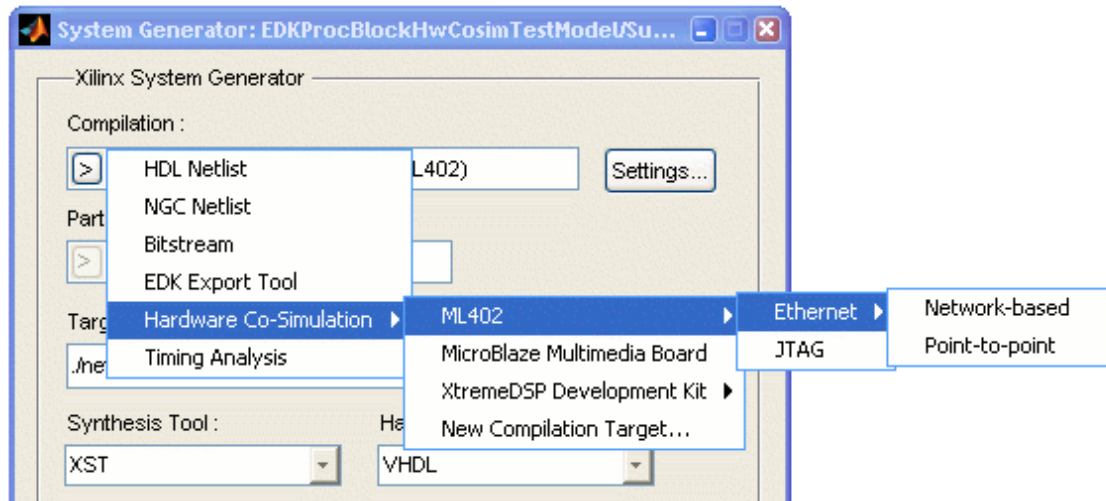
Figure 3-7: **Hardware Co-Simulation Options**

If an EDK processor block is detected during generation of a hardware co-simulation block, the *Software* tab of the co-simulation block's GUI becomes active (Figure 3-8). The Software tab contains path locations to EDK Project and also to the Block Memory Map (BMM) file that was generated with the co-simulation block. The BMM file is used by the EDK to update the processor with compiled C code.
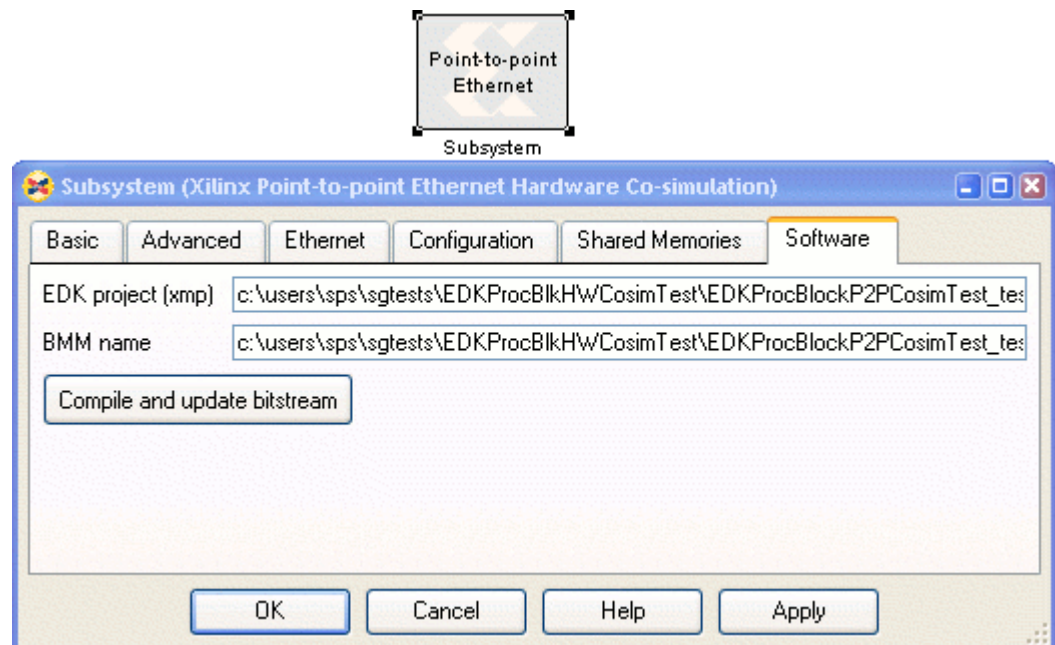
Figure 3-8: **Software Tab**

Code written in the EDK Project can be compiled and updated into the co-simulation block by using the *Compile and update bitstream* button in the Software tab.

A tutorial showing this design-flow can be found in the "Hardware Software Co-Design in System Generator" chapter in the *System Generator User Guide*.

Writing Software Code

Shared-memory blocks associated to an EDK Processor can be accessed by name inside C code. When System Generator creates the memory map, template C-code software drivers and their corresponding documentation are created. This C-code driver is inflated by the EDK during the LibGen phase of compilation in the EDK software compilation flow.

The documentation generated for a pcore can be found in the `doc` directory in a pcore. A PDF and an HTML version of the documentation are provided, and these can be accessed from the EDK.

Figure 3-9 shows a screen capture of a Xilinx Platform Studio (XPS) instance. The peripheral `sha256pcore_sm` is a pcore generated with System Generator. In this EDK project, that pcore has been added to the system and renamed `sha`. Right click on the peripheral in the XPS assembly view to call up the popup menu shown in Figure 3-9. Selecting *View API Documentation* shows the documentation for the peripheral. The documentation contains information on which header files to include, the naming convention of the memory-map identifiers, the software driver function prototypes, and example code snippets.
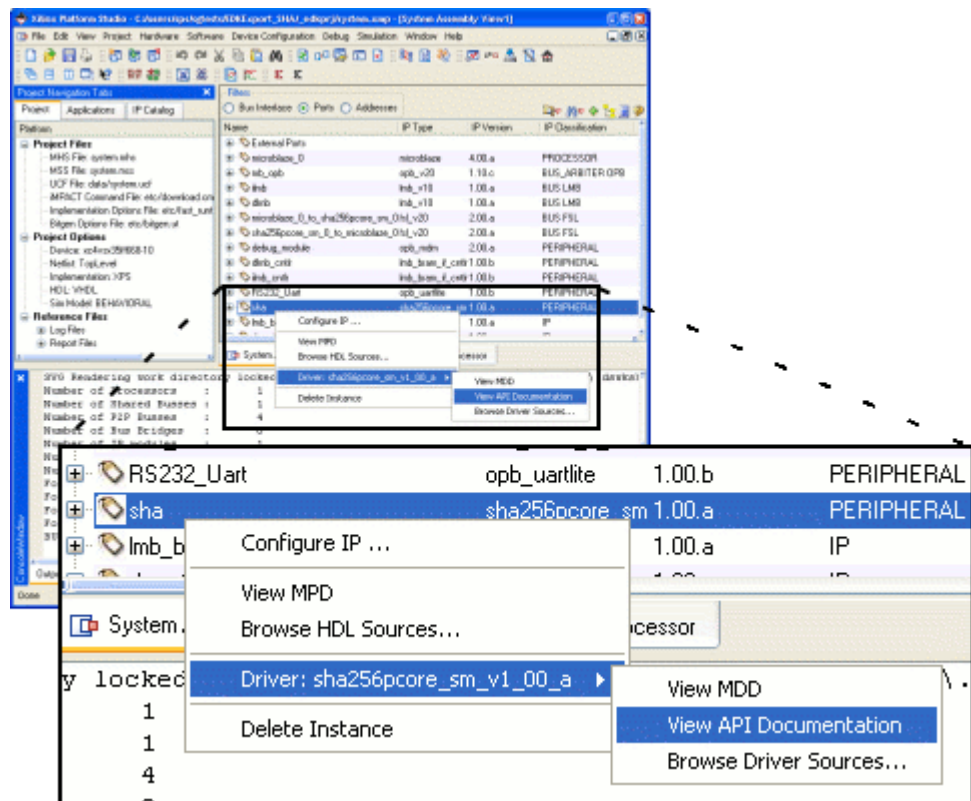
Figure 3-9:   **Xilinx Platform Studio - Assembly View**

The generated software drivers contain six basic functions for accessing shared memories. The `<inst>` label in the function list shown below refers to the instance name of the pcore. When a user-defined pcore is added into an EDK project, a unique instance name

can be given to that pcore. In Figure 3-9 shows a pcore that has been named `sha`. As such, when reading data from that peripheral, the read function is `sha_Read()`.

When an EDK project is imported into System Generator, the EDK Import Wizard automatically configures the system with a *stub* peripheral representing the communications between the EDK and the System Generator peripheral. The peripheral is named `xlsg_iface`. As such, when reading data from that peripheral, the read function is `xlsg_iface_Read()`.

```
int <inst>_Read       (unsigned int memName,
                       unsigned int addr,
                       unsigned int* val)
int <inst>_ArrayRead  (unsigned int memName,
                       unsigned int startAddr,
                       unsigned int transferLength,
                       unsigned int** valBuf);
int <inst>_Write      (unsigned int memName,
                       unsigned int addr,
                       unsigned int val);
int <inst>_ArrayWrite (unsigned int memName,
                       unsigned int startAddr,
                       unsigned int transferLength,
                       const unsigned int* valBuf);
unsigned int <inst>_GetFifoDataCount (unsigned int memName);
unsigned int <inst>_GetFifoEmptyCount (unsigned int memName);
```

Figure 3-10 shows a snippet of the automatically generated documentation. This shows that the pcore has four shared-memories that can be accessed: a register, two FIFOs, and a RAM. The first column shows the name of the shared-memory, as defined in the System Generator diagram. Column 3 shows the identifiers that should be used in the C-code drivers.

Shared Memory Name	Native Precision*	Identifier**	Notes
done		<INST>_DONE	FromRegister
	UFix_1_0	<INST>_DONE_DOUT	Read only
message		<INST>_MESSAGE	ToFIFO [31]
	UFix_32_0	<INST>_MESSAGE_DIN	Write only
	UFix_5_0	<INST>_MESSAGE_PERCENTFULL	Read only
	UFix_1_0	<INST>_MESSAGE_FULL	Read only
	Boolean	<INST>_MESSAGE_RST	Write only
start		<INST>_START	ToFIFO [15]
	UFix_1_0	<INST>_START_DIN	Write only
	UFix_4_0	<INST>_START_PERCENTFULL	Read only
	UFix_1_0	<INST>_START_FULL	Read only
	Boolean	<INST>_START_RST	Write only
checksum	UFix_32_0	<INST>_CHECKSUM	RAM [0:15]

Figure 3-10: **Memory Map Documentation**

For instance, reading from the memory called *checksum* should be done as follows:

```
sha_Read(SHA_CHECKSUM, 0, &val);
```

The memory name to read from is `<INST>_CHECKSUM`. In this case, the instance of the pcore is `sha`. Reading should be done from address 0 of that shared memory and the result placed into the register called `val`.

When reading and writing to RAMs, the address is specified as an index into that memory. When reading and writing to registers and FIFOs, address refers to their address in the memory map. For instance, reading the percent full port of the memory named `message` is performed with the following code:

```
sha_Read(SHA_MESSAGE, SHA_MESSAGE_PERCENTFULL, &val);
```

The memory name to read from is `SHA_MESSAGE` and the address to read from is `SHA_MESSAGE_PERCENTFULL`.

Hardware Co-Simulation

Hardware Co-Simulation Overview

System Generator provides hardware co-simulation interfaces that make it possible to compile a System Generator diagram into an FPGA bitstream and associate this bitstream with a new run-time hardware co-simulation block. A run-time block is a System Generator block that serves as a proxy between the simulation environment and the underlying FPGA hardware. When the design is simulated in Simulink, results for the compiled portion are calculated in hardware instead of software.

Co-Simulation Communication Primitives

System Generator provides different interfaces that allow a Simulink design to communicate with an FPGA platform during hardware co-simulation. The types of interfaces that should be used depend on the particulars of the design. Each interface is briefly described below.

Ports

Ports are the most common interface for communicating with FPGA co-simulation hardware. Here the term *port* is used to include both System Generator Gateway blocks and Simulink Inport/Outport blocks. See Figure 4-1
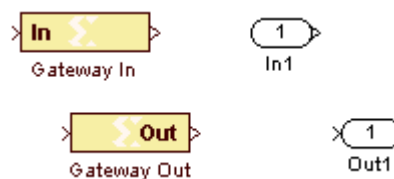
Figure 4-1: **Ports**

When a design that includes a port is compiled for hardware co-simulation, a corresponding port is created in a co-simulation memory map inside the FPGA that can be written to or read from by the PC during co-simulation. A port with an equivalent name and data type is also included on the run-time co-simulation block. This means that each port in the design translates into a corresponding port on the run-time co-simulation block. In this manner, the external interface of a run-time co-simulation block is created to match the interface of the subsystem for which it was compiled. The ordering of the ports on the run-time co-simulation block follows these rules:

- Simulink Inport and Outport blocks are added to a run-time co-simulation block in order of port indexes. For example, if a Simulink Inport has an index of 1, it will be the

first input port on the co-simulation block. An Outport with an index of 2 will be the second output port on the co-simulation block.

- Because Gateway blocks lack an explicit index or ordering in Simulink, these ports are added to a run-time co-simulation block in alphabetical order. If a design contains both Gateways and Inport/Outport blocks, the Gateway ports appear last (i.e., they have a higher index).

Shared Register

A To Register, From Register or *shared register pair* can be generated and co-simulated in FPGA hardware. Here a shared register pair is defined as a To Register block and From Register block that specify the same name (e.g., Bar). See Figure 4-2. In hardware, a shared register is implemented using a synthesizable register component (for VHDL) or a module (for Verilog).
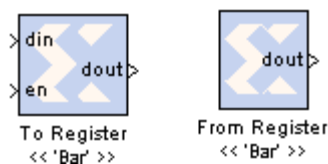
Figure 4-2: **Shared Register Pair**

When a design that includes a shared register pair is compiled for hardware co-simulation, the pair is replaced by a single register instance. Both sides of the register are attached to *user design logic*; that is, logic that originated from the original System Generator model. In this case, control of the register is not shared between the PC and FPGA hardware since all register ports are attached to user design logic. Compiling a shared register pair to hardware is equivalent to compiling a System Generator Register or Delay block.

Compiling a single To Register or From Register block for hardware co-simulation results in a different type of implementation. A single register is still created to replace the To or From Register block. Only in this case, the register connects to both the PC and FPGA memory map logic. The side of the register in the original model remains connected to user design logic. The other side of the register attaches to memory map interface logic.

For designs that use hardware co-simulation, shared register pairs are typically distributed between software and FPGA hardware. In other words, one half of the pair is implemented in the FPGA, while the other half is simulated in software using a To or From Register block. When data is written to a software To Register block, the hardware register is updated to with the same data. Similarly, when data is written into the hardware register, the same data is read by the From Register software block. A software shared register can connect to a hardware shared register simply by specifying the name of the shared register as it was compiled for hardware co-simulation.

Shared Memory

Compiling a shared memory block for hardware co-simulation adds an addressable block of memory into the co-simulation memory map (Figure 4-3). Shared memory blocks can be configured with a *lockable* or *unprotected* access protection mode. This section touches on both protection modes since they are involved during co-simulation semantics.
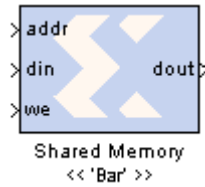
Figure 4-3: **Shared Memory**

In lockable access mode, the System Generator co-simulation hardware must acquire lock over the shared memory object before it can access its contents. When the hardware acquires (releases) lock of the shared memory, the memory contents are transferred to (from) the FPGA using a high-speed data transfer.

Two images of the shared memory data are used when a lockable shared memory is co-simulated. One memory image is stored using dual port memory in the FPGA. This image is accessed by the System Generator hardware co-simulation design and co-simulation memory map logic. The other image is implemented as a shared memory object on the host PC. This software shared memory image is accessed by any software shared memory objects used in a design.

A software process or hardware circuit that wishes to access the shared memory must first obtain the lock. If the hardware has lock of the memory, no software objects can access the memory contents. Likewise, if a software object controls the memory, the hardware cannot read or write to the memory. Note that lockable hardware shared memories include additional logic to handle the mutual exclusion.

Having two shared memory images requires synchronization between software and hardware to ensure the images are coherent. This synchronization is accomplished by transferring the memory image between software and hardware upon lock transfer. System Generator performs high-speed data transfers between the host PC and FPGA.

Unprotected shared memory blocks can be written to or read from at any time during co-simulation—the memory has no notion of mutually exclusive access. To ensure data coherency between software and hardware, a single image of the shared memory data is shared between hardware and software. This image is stored in the FPGA using dual port memory that is accessible as part of the co-simulation memory map. System Generator allows both hardware design logic and other software-based shared memory objects on the host PC to access the shared memory data concurrently. When software shared memory objects read or write data to the shared memory, System Generator seamlessly handles communication with the hardware memory resource.

FIFO

A To FIFO, From FIFO, or *shared FIFO pair* can be generated and co-simulated in hardware. A shared FIFO pair is defined as a To FIFO block and From FIFO block which specify the same name (e.g., Bar) (Figure 4-4).
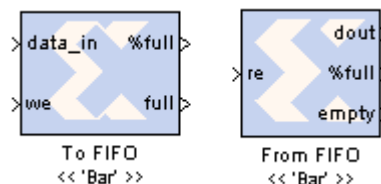
Figure 4-4: **Shared FIFO Pair**

In hardware, a shared FIFO is implemented using the FIFO Generator core. The core is configured to use independent (asynchronous) clocks and block memory for data storage. Shared FIFOs allow the user to safely transfer data to and from the FPGA platform in co-simulation designs that use a free-running clock mode. Shared FIFOs can also be used to support burst transfers during co-simulation for applications with high throughput requirements.

When a shared FIFO pair is generated for co-simulation, a single asynchronous FIFO core replaces the two software shared FIFO blocks. The read/write FIFO sides are attached to *user design logic* (i.e., logic derived from the original System Generator model) that attached to the From FIFO and To FIFO blocks, respectively (Figure 4-4). Because both FIFO sides attach to user logic in hardware, the PC does not share control of the FIFO with the design. Instead, the FIFO behavior is similar to a System Generator design that includes a traditional FIFO block.

Single shared FIFO blocks are treated differently than shared FIFO pairs. A single To FIFO or From FIFO block is replaced by an asynchronous FIFO core when it is compiled for hardware co-simulation. One side of the FIFO (i.e., the unused shared FIFO half in System Generator) is connected to PC interface logic. The other side is connected to user design logic that attached to the original To or From FIFO block. In this manner, control over the FIFO is distributed between the PC and FPGA design.

When a To FIFO block is compiled for hardware co-simulation, the write side of the FIFO is connected to the same logic that attached to To FIFO block in user design. The read side of the FIFO is connected to memory map interface logic that allows the PC to read data from the FIFO during simulation. The opposite wiring approach is used when a From FIFO block is compiled for hardware co-simulation. In this case, the write side of the FIFO is connected to PC interface logic, while the read side is connected to the user design logic. The host PC writes data into the FIFO and the design logic can read data from the FIFO.

Shared FIFO pairs are typically distributed between software and FPGA hardware. In other words, one half of the pair is implemented in the FPGA while the other half is simulated in software using a To or From FIFO block. Together, the software and hardware portions form a fully functional asynchronous FIFO. When a software/hardware shared FIFO pair is co-simulated, System Generator transparently manages the necessary transactions between the PC and FPGA hardware.

When data is written to a software To FIFO block during simulation, the same data is written to the FIFO in hardware. The design in hardware can then retrieve this data by reading from the FIFO. Similarly, when data is written into the hardware FIFO by design logic, the data can be read by the From FIFO software block. Note that the empty, full, read and write count ports on the shared FIFO blocks pessimistically reflect the state of the hardware FIFO counterpart. A software shared FIFO can connect to a hardware shared FIFO simply by specifying the name of the shared FIFO as it was compiled for hardware co-simulation.

Pad

FPGA platforms often include a variety of on-board devices (e.g., external memory, analog to digital converters, etc.) that the FPGA can communicate with. For a variety of reasons, it may be useful to form connections to these components in your System Generator models and to use these components during hardware co-simulation. For example, if your board includes external memory, the user can define the control and interface logic to this memory in the user's System Generator design and use the physical memory during hardware co-simulation.

The user can interface to these types of components by including board-specific I/O pads in the user's System Generator models. A board-specific I/O pad is a special port that is wired to a specific FPGA pad when the model is compiled for hardware co-simulation. This type of port differs from standard co-simulation ports that are controlled by a corresponding port on a hardware co-simulation block. This means that a hardware co-simulation run-time block does not include board-specific ports on its external interface.

A board-specific I/O pad is implemented using special *non-memory mapped gateway blocks* that tell System Generator to wire the signals to the appropriate FPGA pins when the model is compiled into hardware. To connect a System Generator signal to a board-specific port, the user should connect the appropriate wire to the special gateway (in the same way as is done for a traditional gateway).

The user can define non-memory mapped ports for a design using the System Generator Board Descriptor Builder application. There are two ways to invoke this application:

1. By typing `xlSBDBuilder` at the MATLAB command window.

2. By selecting `Hardware Co-Simulation->New Compilation Target...` under the Compilation submenu on the System Generator parameters dialog box.

Shared Memory Read/Write Blocks

System Generator supports burst transfers to and from the FPGA during co-simulation using the Shared Memory Read and Write blocks (Figure 4-5). These blocks provide high-speed interfaces for lockable shared memories and shared FIFO blocks that have been compiled for co-simulation. These blocks rely on the burst-transfer support of the co-simulation platform to read or write the contents of an entire vector or matrix signal in a single-hardware transaction. Both blocks are described below.
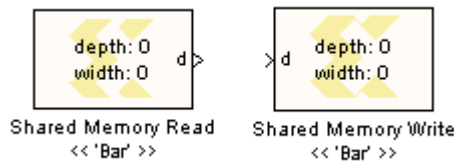
Figure 4-5:   **Shared Memory Read and Write Blocks**

The Shared Memory Read block provides a high-speed interface for reading data from a shared memory or FIFO in hardware. The requested data is read out of the shared memory and into a Simulink scalar, vector, or matrix signal which is written to the block's output port.

The Shared Memory Write block provides a high-speed interface for writing data into a shared memory or FIFO in hardware. The Shared Memory Write block input port should be driven by the Simulink scalar, vector, or matrix signal containing the data you would like written into the shared memory object. Note that the width of the input data must match the width of the shared memory, and the total number of elements in the input must not be bigger than the depth of the shared memory or FIFO.

The bracketed text beneath each block indicates the shared memory or FIFO with which the block interfaces. The depth and width displays on the block indicate the size of the shared memory. These values are updated at runtime when the block makes the connection to the shared memory object.

Co-Simulation Interfaces

System Generator provides multiple co-simulation interfaces that support a broad range of FPGA hardware platforms. There are certain platforms, such as the XtremeDSP Development Kit, that employ specialized co-simulation interfaces to take advantage of the high-speed connection between the FPGA platform and the host PC. For example, the XtremeDSP Development Kit provides a PCI interface that enables high-speed communication with a host PC. Ethernet-based co-simulation supports hardware co-simulation with FPGA platforms that provide network connections. System Generator also provides a general purpose JTAG hardware co-simulation interface that uses the JTAG port to communicate with the FPGA. These interfaces are discussed in detail below.

JTAG

The JTAG hardware co-simulation interface takes advantage of the ubiquity of JTAG to extend System Generator's hardware in the simulation loop capability to numerous other FPGA platforms. A platform can support the JTAG hardware co-simulation interface, provided it includes the following hardware components:

- A Xilinx FPGA part that is available in System Generator as a supported device (i.e., can be chosen in the *Part* field of the System Generator block dialog box)
- An on-board oscillator that supplies the FPGA with a free-running clock source
- A JTAG header that provides access to the FPGA.

If the user has a board that supports JTAG hardware co-simulation, there are additional hardware installation requirements:

- The power supply and all other required cables must be attached correctly to your development board.
- A Parallel Cable IV or Platform Cable USB programming must be connected to your machine's ECP printer port (LPT1 only) or USB port.
- The Parallel Cable IV or Platform Cable USB header must be connected to the JTAG port on your FPGA development board using either the flying leads or ribbon cable connectors.

PCI

Certain co-simulation platforms (e.g., the XtremeDSP Development Kit) provide a PCI interface that allows the FPGA hardware to communicate with the PC. These boards plug directly into an available PCI slot on the PC. For these platforms, System Generator supports PCI-based hardware co-simulation. Note that PCI is a specialized interface which includes hardware and software components that are tailored to the specific requirements of a particular platform.

Network-Based Ethernet Co-Simulation

The network-based Ethernet hardware co-simulation interface provides co-simulation access to an FPGA platform over an IPv4 network infrastructure. Because IPv4 networks are widespread, the interface provides a straightforward way to communicate with remote hardware connected to either a wired or wireless network. This interface is ideal in situations where the FPGA platform is remote (e.g., across the office or across the country) or when multiple designers must shared a single development board.

The network-based Ethernet interface supports operations in 10/100 Mb/s half/full duplex modes. For FPGA device configuration, the interface supports Ethernet-based configuration over the same network connection for co-simulation. This means that a separate programming cable (e.g., Parallel Cable IV) is *not* required.

Point to Point Ethernet Co-Simulation

Point-to-point Ethernet Hardware Co-Simulation provides a co-simulation interface using a raw Ethernet connection. The raw Ethernet connection refers to a Layer 2 (a.k.a. Data-Link Layer) Ethernet connection, between a supported FPGA development board and a host PC, with no routing network equipment along the path. By taking the advantage of the ubiquity and advancement of Ethernet technologies, the interface facilitates a convenient and high-bandwidth co-simulation to an external FPGA device.

The point-to-point Ethernet interface supports operations in 10/100/1000 Mb/s half/full duplex modes. Jumbo Frames are also supported on a Gigabit Ethernet connection, provided it is enabled by the underlying connection. For FPGA device configuration, the interface supports either JTAG-based configuration over a Parallel IV or a Platform USB cable, or Ethernet-based configuration over the same point-to-point Ethernet connection for co-simulation.

Third Party Co-Simulation

The System Generator software includes hardware co-simulation support for the XtremeDSP Development Kit, ML402 development board, and MicroBlaze Multimedia Demonstration board. The user can add support for new or third-party FPGA development boards by installing additional System Generator plugins. These plugins are distributed as ZIP files that configure System Generator to support the user's board.

System Generator provides an installer to install user development board plugins. A System Generator plugin can be installed by following the steps provided below:

1. Download the plugin to a temporary directory.

2. From the MATLAB command window, change directories to the temporary directory where the plugin is saved.

3. From the MATLAB command window, type

```
>> xlInstallPlugin('myplugin.zip')
```

Note: `myplugin.zip` is the name of the plugin file the user is installing.

This installs the plugin. A status bar is displayed in to show the progress of the installer.
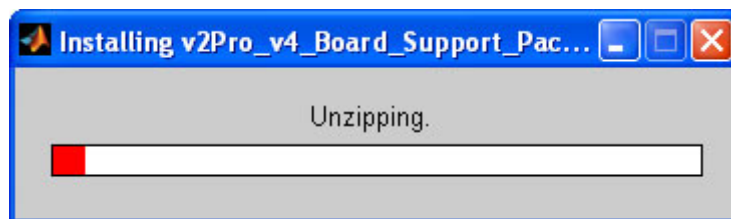
Figure 4-6:   **Status Bar**

4.   After the plugin is installed, the user can choose the development board as a compilation target in the System Generator block dialog box.

Building a Co-Sim Project

The starting point for hardware co-simulation is the System Generator model or subsystem the user would like to run in hardware. A model can be co-simulated, provided it meets the requirements of the underlying hardware platform. This model must include a System Generator block; this block defines how the model should be compiled into hardware. The first step, after the user has a model that is ready to run in hardware, is to open the System Generator block dialog box and select a compilation type under **Compilation**.

Choosing a Compilation Target

The user can choose the hardware co-simulation platform for System Generator to compile code by selecting an appropriate compilation type in the System Generator block dialog box. Hardware co-simulation targets are organized under the **Hardware Co-Simulation** submenu in the **Compilation** dialog box field (Figure 4-7). When the user installs System Generator, several hardware co-simulation compilation targets are automatically installed.
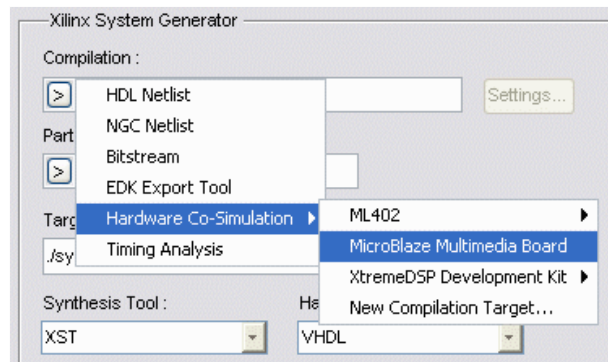
Figure 4-7:   **Hardware Co-Simulation Targets**

When a compilation target is selected, the fields on the System Generator block dialog box are automatically configured with settings appropriate for the selected compilation target. System Generator remembers the dialog box settings for each compilation target. These settings are saved when a new target is selected, and restored when the target is recalled.

Invoking the Code Generator

After the user has selected a compilation target, the user can invoke the System Generator code generator to compile the model for hardware co-simulation. The code generator is invoked by pressing the **Generate** button in the System Generator block dialog box (Figure 4-8).
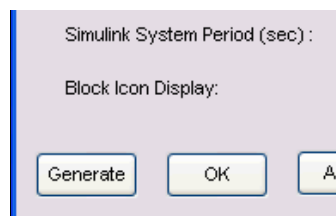
Figure 4-8:   **Code Generator Generate Button**

The code generator produces an FPGA configuration bitstream for the user's design that is suitable for hardware co-simulation. System Generator not only generates the HDL and netlist files for the user's model during the compilation process, but it also runs the downstream tools necessary to produce an FPGA configuration file.

Note: A status dialog box appears after the user presses the **Generate** button. During compilation, the status box (Figure 4-9) provides a **Cancel** and **Show Details** button. Pressing the **Cancel** button stops compilation. Pressing the **Show Details** button exposes details about each phase of compilation (or tool) as it is run. It is possible to hide the compilation details by pressing the **Hide Details** button on the status dialog box.
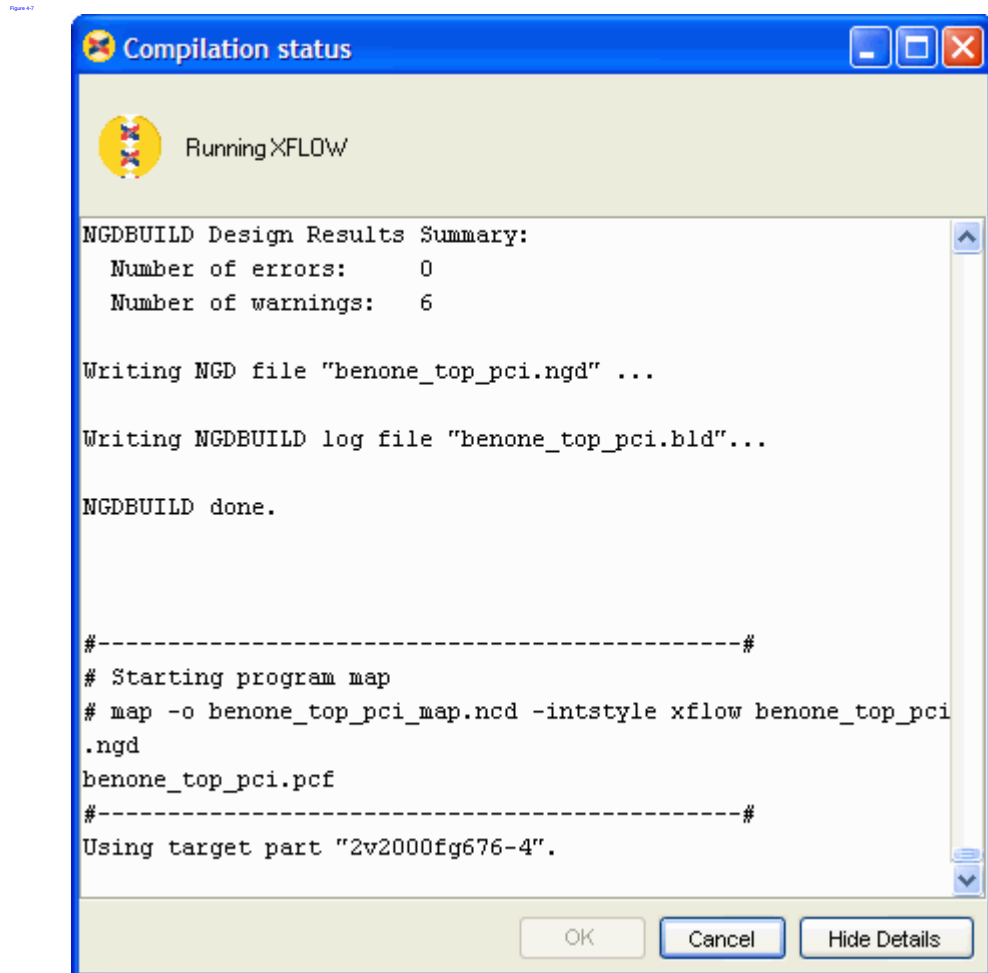
Figure 4-9: **Compilation Status**

The configuration bitstream contains the hardware associated with the user's model, and also contains additional interfacing logic that allows System Generator to communicate with the user's design via a physical interface between the platform and the PC. This logic includes a memory map interface over which System Generator can read and write values to the input and output ports on your design. It also includes any platform-specific circuitry (e.g., DCMs, external component wiring) that is required for the target FPGA platform to function correctly.

Hardware Co-Simulation Blocks

System Generator automatically creates a new hardware co-simulation block after it has finished compiling the design into an FPGA bitstream. A Simulink library is also created to store the hardware co-simulation block. At this point, the user can copy the block out of the library and use it in the System Generator design as the user would for other Simulink and System Generator blocks.

The hardware co-simulation block assumes the external interface of the model or subsystem from which it is derived. The port names on the hardware co-simulation block match the ports names on the original subsystem. The port types and rates also match the original design (Figure 4-10).
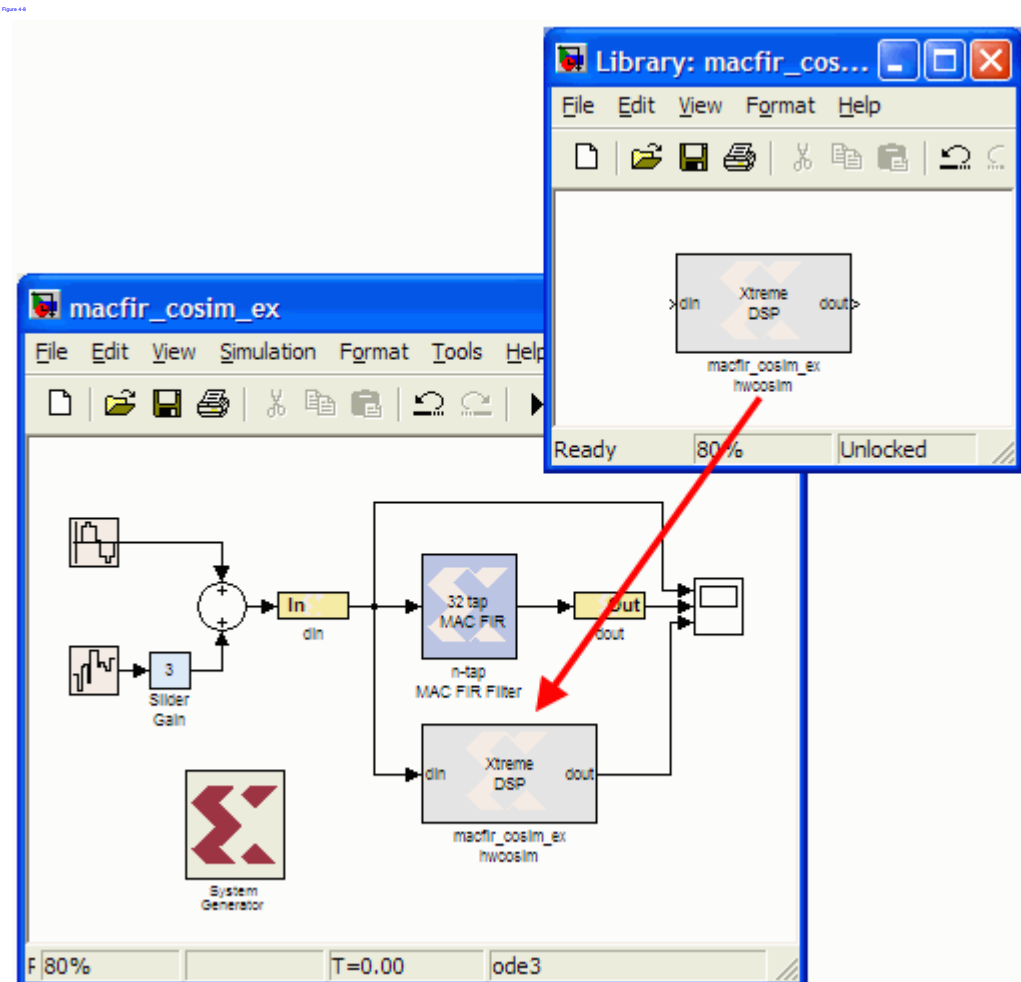
Figure 4-10: **Example of a Run-time Hardware Co-Simulation Block Inserted in the Original Model**

Hardware co-simulation blocks are used in a Simulink design the same way other blocks are used. During simulation, a hardware co-simulation block interacts with the underlying FPGA platform, automating tasks such as device configuration, data transfers, and clocking. A hardware co-simulation block consumes and produces the same types of signals that other System Generator blocks use. When a value is written to one of the block's input ports, the block sends the corresponding data to the appropriate location in

hardware. Similarly, the block retrieves data from hardware when there is an event on an output port.

Hardware co-simulation blocks can be driven by Xilinx fixed-point signal types, Simulink fixed-point signal types, or Simulink doubles. Output ports assume a signal type that is appropriate for the block they drive. If an output port connects to a System Generator block, the output port produces a Xilinx fixed-point signal. Alternatively, the port produces a Simulink data type when the port drives a Simulink block.
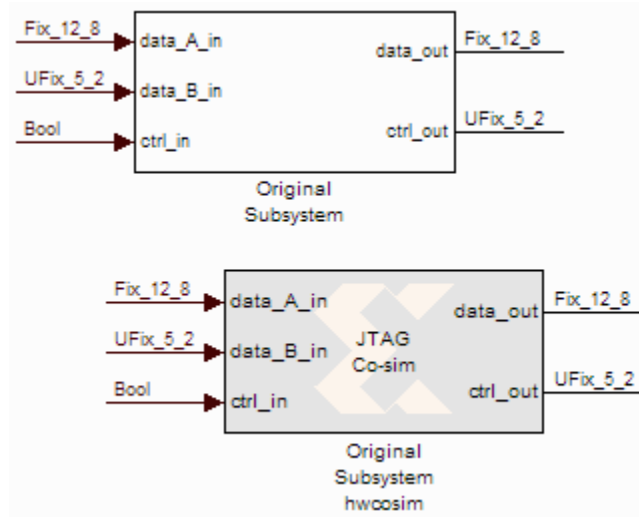
Figure 4-11: **Port Interface of a Run-time Co-Simulation Block Matches the Port Interface of the Original Design**

Note: When Simulink data types are used as the block signal type, quantization of the input data is handled by rounding, and overflow is handled by saturation.

Like other System Generator blocks, hardware co-simulation blocks provide parameter dialog boxes that allow them to be configured with different settings. The parameters that a hardware co-simulation block provides depend on the FPGA platform the block is implemented for (i.e., different FPGA platforms provide their own customized hardware co-simulation blocks).

Ethernet Co-Sim Setup

The procedure for setting up the Xilinx ML402 development board to support System ACE-based configuration with hardware co-simulation is described below. Two configuration modes are supported for point-to-point Ethernet co-simulation:

- System ACE configuration
- Standard JTAG configuration using a Parallel Cable IV or Platform Cable USB.

Note: If using JTAG configuration, the user can skip the section below as it deals with setup and configuration of the System ACE card.

A diagram of the ML402 and controls required for setup and validation is provided in Figure 4-12.
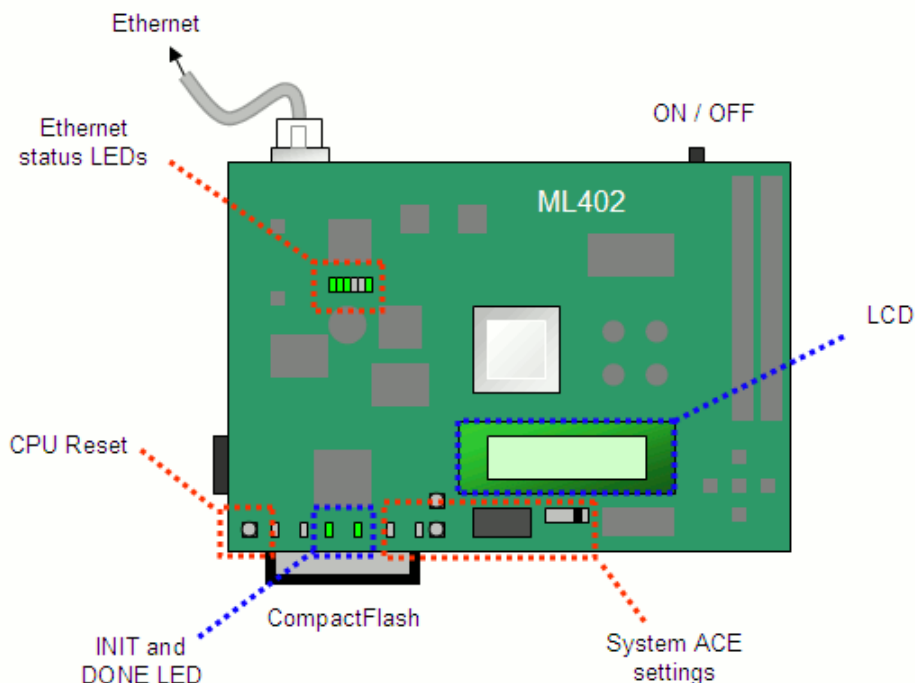
Figure 4-12: **ML402 Board Diagram**

System ACE Setup

System Generator utilizes the System ACE solution to support device configuration over an Ethernet-connection for both network and point-to-point Ethernet hardware co-simulation interfaces. Using this interface eliminates the need for a second programming cable (e.g., a Xilinx Parallel Cable IV) for device configuration.

For point-to-point Ethernet co-simulation, the System ACE configuration is available as the `Point-to-point Ethernet` option under the `Configuration` tab on the block parameters dialog box. For network-based Ethernet co-simulation, the System ACE configuration solution is the only configuration option.

Prepare the System ACE Compact Flash Card

1. Insert the Compact Flash (CF) card into a Compact Flash reader/writer.

2. Extract the default CF image in `$SYSGEN\ml402\sysace_cf.zip`, which is bundled with System Generator, onto the card.

   Note: *The card might need to be formatted to a FAT16 file system before the CF image can be correctly extracted onto card. Use the* ***mkdosfs*** *utility to format the card. The mkdosfs program can be obtained from* http://www.xilinx.com/products/boards/ml310/current/utilities/mkdosfs.zip.

The following example command formats the card as FAT16 file system:

```
mkdosfs -v -F 16 e:
```

where `e:` is the drive name associated with the Compact Flash reader.

Warning: **Ensure the drive name (e.g., 'e:') is specified correctly prior to running the program**.

Assign an Ethernet MAC Address and IPv4 Address

3. After writing the CF image to the card, the user will find two files, `mac.dat` and `ip.dat`, in the CF card's root directory. The `mac.dat` and `ip.dat` files specify the Ethernet MAC address and IPv4 address associated with the board, respectively. These addresses are used to uniquely identify a target board during Ethernet hardware co-simulation.

   Note: Steps 2 and 3 are optional and are necessary only when:

   ◆ The default MAC and IP addresses conflict with the user's default network settings.

   ◆ The user wishes to co-simulate two or more ML402 boards concurrently.

4. Open `mac.dat` in a text editor to change the Ethernet MAC address. The MAC address must be specified as a six pair of two-digit hexadecimal separated by colons (e.g. `00:0a:35:11:22:33`). All-zeros, broadcast, or multicast MAC addresses are not supported.

5. Open `ip.dat` in a text editor to change the IP address. The IP address must be specified in IPv4 dotted decimal notation (e.g. `192.168.8.1`). All-zeros, broadcast, multicast, or loop-back IP address are not supported.

Adjust On-Board Settings for System ACE

1. Turn off the ML402 board.

2. Install the Compact Flash card (as prepared in the previous steps) securely into the corresponding slot on the development board.

3. Adjust the on-board settings as follows:

   a. Configuration Address jumpers:

   ```
   [1: on, 2: off, 3: off, 4: on, 5: off, 6: on](Address 4 with JTAG mode)
   ```

   b. Configuration Source Selector Switch set to `SYS ACE` (System ACE method).
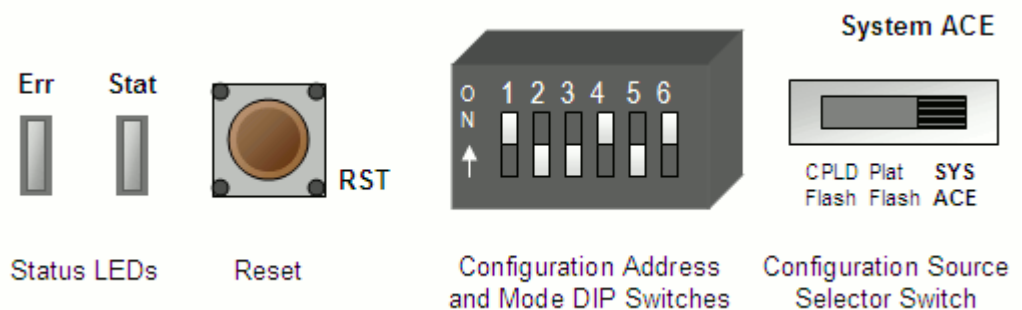
Figure 4-13: **On-Board Settings**

System ACE Troubleshooting

Verify System ACE Settings

1. Power on the ML402 board. Check the on-board status LEDs to ensure the FPGA is configured with the initial bootloader image from the Compact Flash card. If the configuration succeeded, the DONE LED should on and all error LEDs should be off. Otherwise, reexamine the steps in the System ACE setup procedure.

2. Check the information displayed on the 16-character x 2-line LCD screen of the board (Figure 4-14). If no error occurred, the Ethernet MAC address (without colons) should appear on the first line of the LCD and the IPv4 address should appear on the second line.
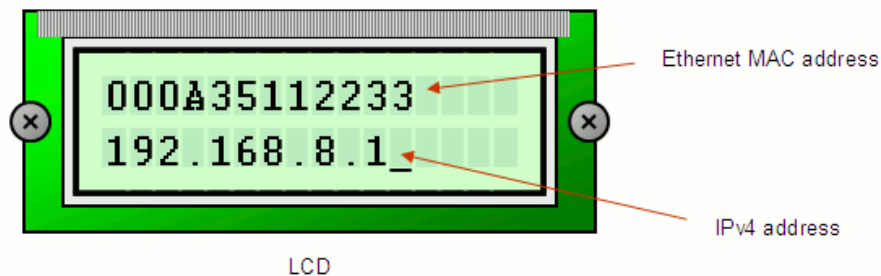
Figure 4-14: **Board LCD Screen**

3. If the LCD does not display the information correctly, press the System ACE Reset button to reset the System ACE controller and reconfigure the FPGA. Check the status LEDs again to ensure device configuration completed successfully.

Verify Ethernet Interface And Connection Status

1. Connect the Ethernet interface of the board to a network connection, or directly to a host.

2. Check the on-board Ethernet status LEDs (Figure 4-15) to make sure the Ethernet interface is attached to an active Ethernet segment. The LEDs should reflect the link speed and the duplex mode at which the interface is operating. If no LED is on, press the CPU Reset button to reset the FPGA, and also examine whether the Ethernet segment is active.
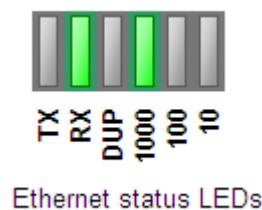
Figure 4-15: **Ethernet Status LEDs**

3. To ensure the board is reachable by the host, issue ICMP ping from the host to check the connectivity. For example, type `ping 192.168.8.1` on a console to test the connectivity to a board with IP address 192.168.8.1.

Ensuring a Correct Setup

After following the System ACE setup directions, the user can test to see that the hardware and software are installed and configured appropriately. This section provides a step-by-step hardware co-simulation walkthrough to verify a proper setup. In the process, the user can also learn how to configure the point-to-point Ethernet hardware co-simulation block dialog box with settings that are appropriate for your machine.

A design has already been prepared that contains a run-time block for point-to-point Ethernet hardware co-simulation. The example includes a reloadable 5x5 reloadable filter operator and demonstrates how hardware co-simulation can be used for high-throughput signal processing applications. In-depth information on this design can be found in the tutorial entitled *Real-time Signal Processing using Hardware Co-Simulation.*

1. From the MATLAB console, change directory to:
    `/Examples/SysgenTutorial/eth_cosim_validation_ex/`

2. Open the `conv5x5_video_testbench.mdl` model from the MATLAB console.

3. In the model, move into the `FPGA Processing` subsystem.
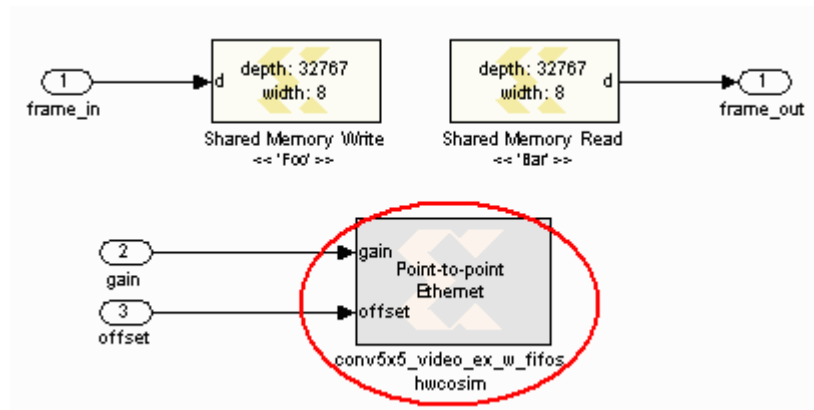
Figure 4-16: **FPGA Processing Subsystem**

This design includes a point-to-point Ethernet run-time co-simulation block that has already been compiled for the user. Before co-simulating the design, the user must first configure the block's parameters dialog box with settings appropriate for your PC.

4. Double-click on the `conv5x5_video_ex_w_fifos` block to open the point-to-point Ethernet co-simulation parameters dialog box.

    This design relies on a free-running clock source to process video frames in real-time.

5. Select `Free running` clock source mode under the `Basic` tab.
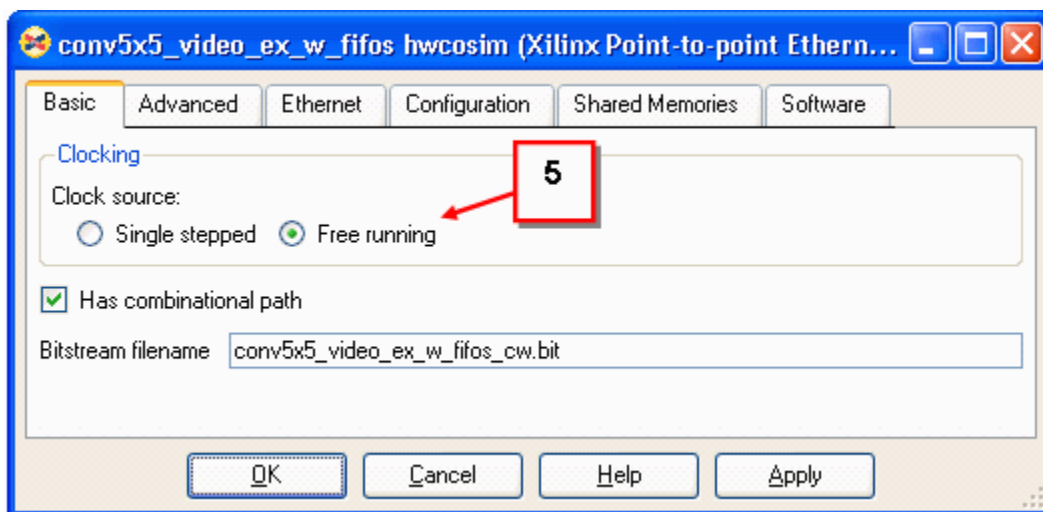
Figure 4-17: **Select Free Running Clock Source Mode**

6. If there is a Video I/O daughter card attached to the ML402 board, check the `Has Video I/O Daughter Card (VIODC) on the ML402 board` checkbox on the `Advanced` tab.
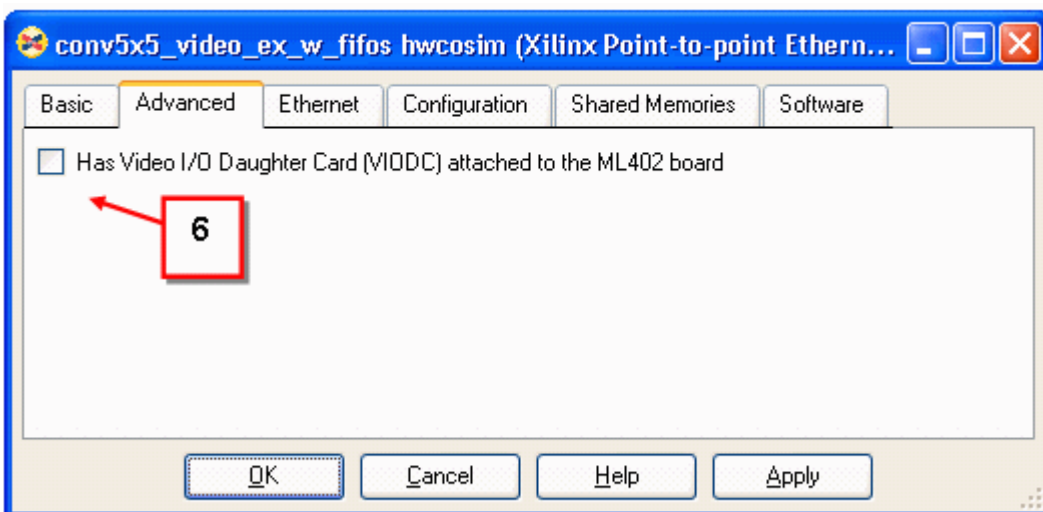
Figure 4-18: **Check the Has Video I/O Daughter Card (VIODC)**

Choose the Configuration Method

7. Select the `Configuration` tab.

8. Choose the `download cable` for configuring the target board.

9. For JTAG-based download cables (Parallel IV or Platform USB), change the `cable speed` if the default value is not suitable for the cable in use.

Note: Change the `configuration timeout` value only when necessary. The default value should suffice in most cases. A larger value is needed when it takes a considerable amount of time to re-establish a network connection with the FPGA platform after device configuration completes.
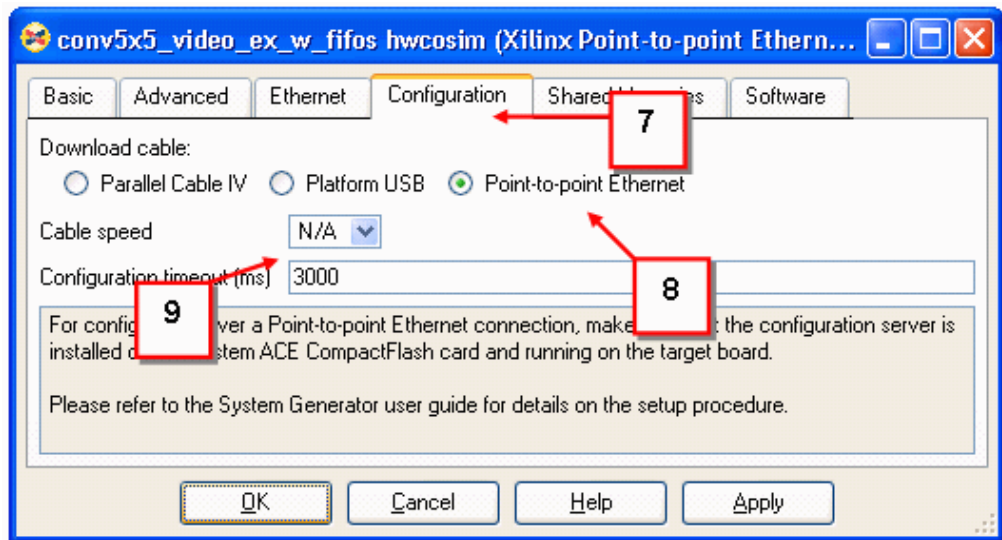
Figure 4-19: **Choose the Configuration Method**

Configure the Ethernet Interface Settings

10. Select the `Ethernet` tab.

From the `host interface` panel, navigate the pull down list and select the appropriate network interface card that you are using for hardware co-simulation.
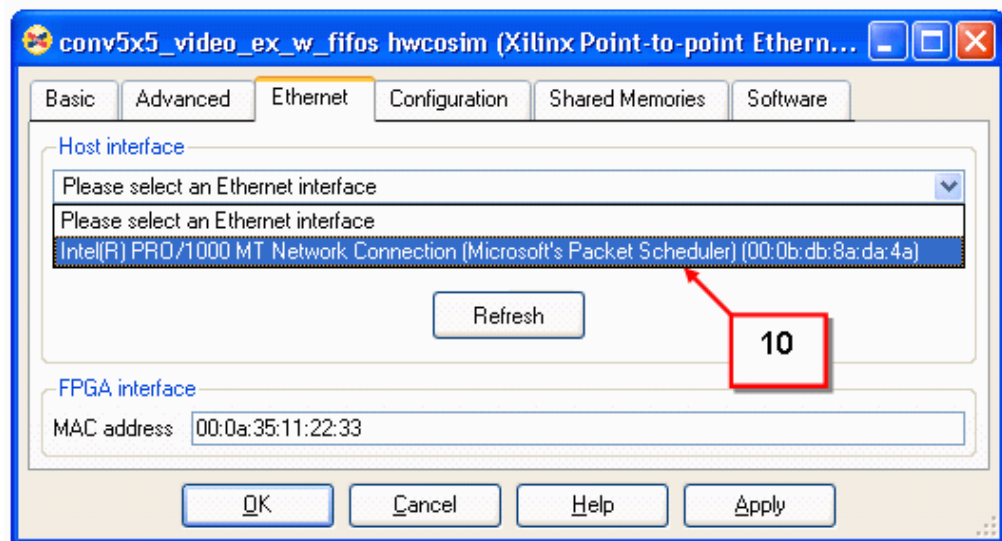
Figure 4-20: **Configure the Ethernet Interface Settings**

Note: *The pull down list only shows those Ethernet-compatible network interfaces installed on the host, which support 10/100/1000 Mb/s and are currently enabled and attached to an active Ethernet segment. If the target interface is not listed as expected, examine the connection and click the `Refresh` button to update the list.*

11. The information box beneath the pull down list provides the details about the selected interface. Examine the information to ensure the appropriate interface is chosen, and adjust the network settings in the operating system when necessary.
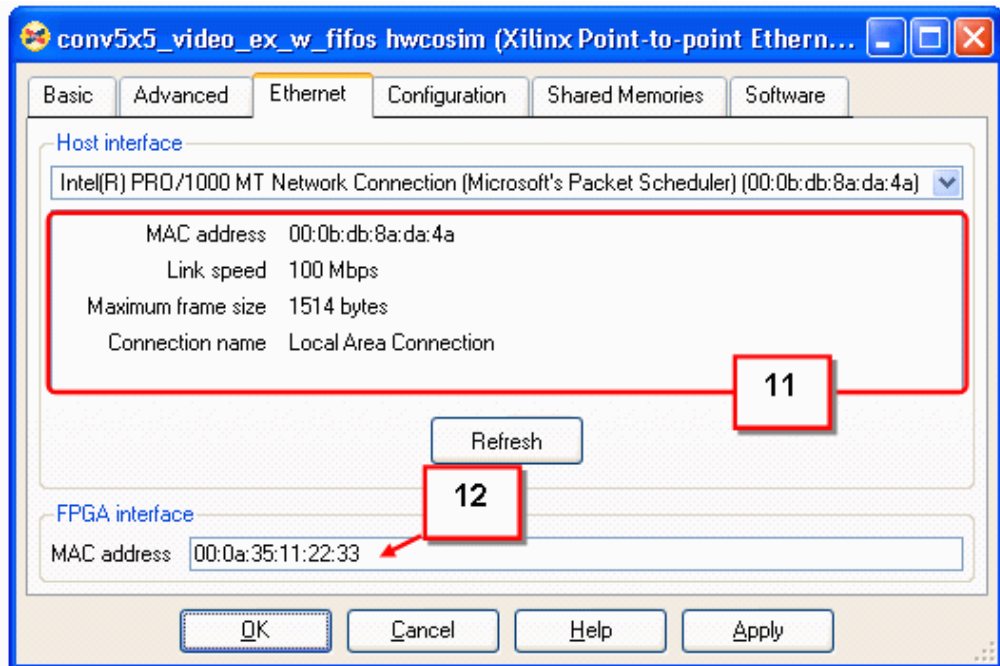
Figure 4-21: **Ensure the Appropriate Interface is Chosen**

12. Depending on which configuration method is chosen, the MAC address in the FPGA interface panel might need to be changed (Figure 4-22).

   a. *For JTAG-based configuration over a Parallel IV or a Platform USB cable:*

   The MAC address need not be changed unless default value conflicts with other network equipment on the same Ethernet segment, or when the co-simulation is running over multiple boards. In either case, an arbitrary but non-conflicting MAC address can be assigned to each point-to-point Ethernet co-simulation block.

   b. *For point-to-point Ethernet-based configuration:*

   Observe the MAC address displayed on the LCD screen of the target board when the configuration boot-loader is running. Change the FPGA MAC address in the co-simulation block if the default value does not match the target board. Refer to the System ACE Setup section for details about assigning the MAC address on a ML402 board.
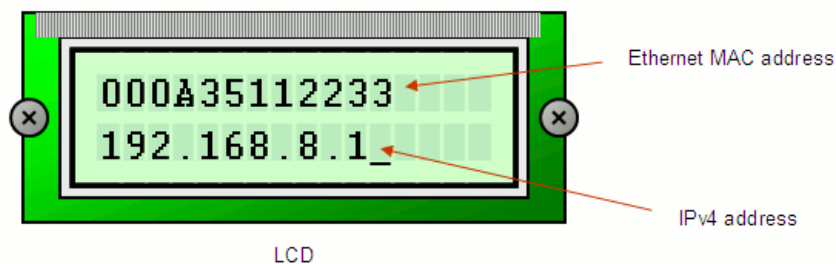
Figure 4-22: **Ethernet Parameters Displayed on ML402 LCD Display**

> *Note:* The MAC address must be specified using six pairs of two-digit hexadecimal number separated by colons (e.g., 00:0a:35:11:22:33).

13. Close the parameters dialog box.

Co-Simulating the Design

After setting the block parameters appropriately, the user can begin co-simulation, which is started by pressing the Simulink `Play` button. System Generator automates the device configuration process and transfers the design under test (DUT) into FPGA device for co-simulation. A dialog box is shown in to describe the status of the process (Figure 4-23).
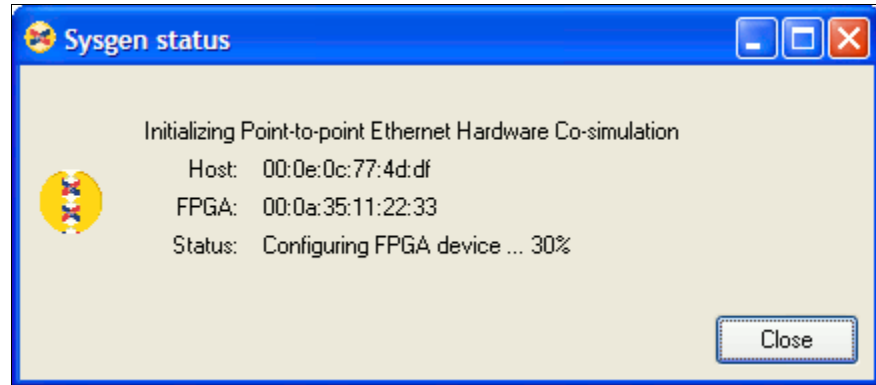
Figure 4-23: **Status Dialog Box**

1. The final configuration file is first generated based on the input bitstream specified in the block parameters.

2. The final configuration file is then transferred to the target board using the selected download cable, and finally used to configure the FPGA device. The progress of configuration is shown in the dialog box when the configuration is performed over a point-to-point Ethernet connection.

3. Upon the completion of device configuration, the co-simulation engine re-establishes the connection to the target board, and starts co-simulating the design.
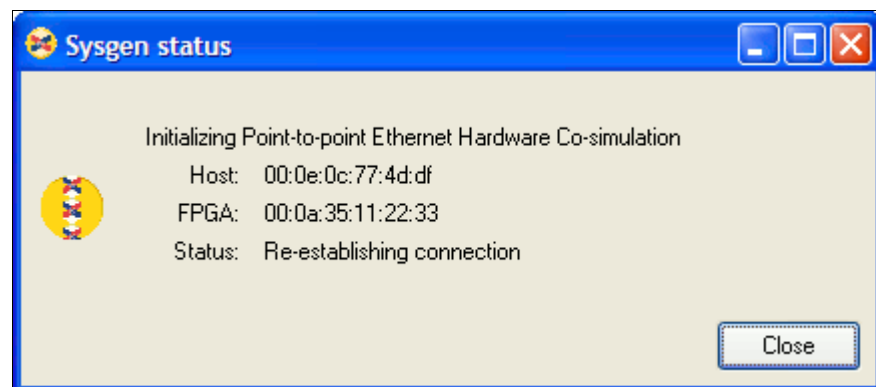
Figure 4-24: **Status Showing Reconnection**

Two windows appears after configuration completes, showing the original and filtered video streams. At this point, setup of your board is successful.
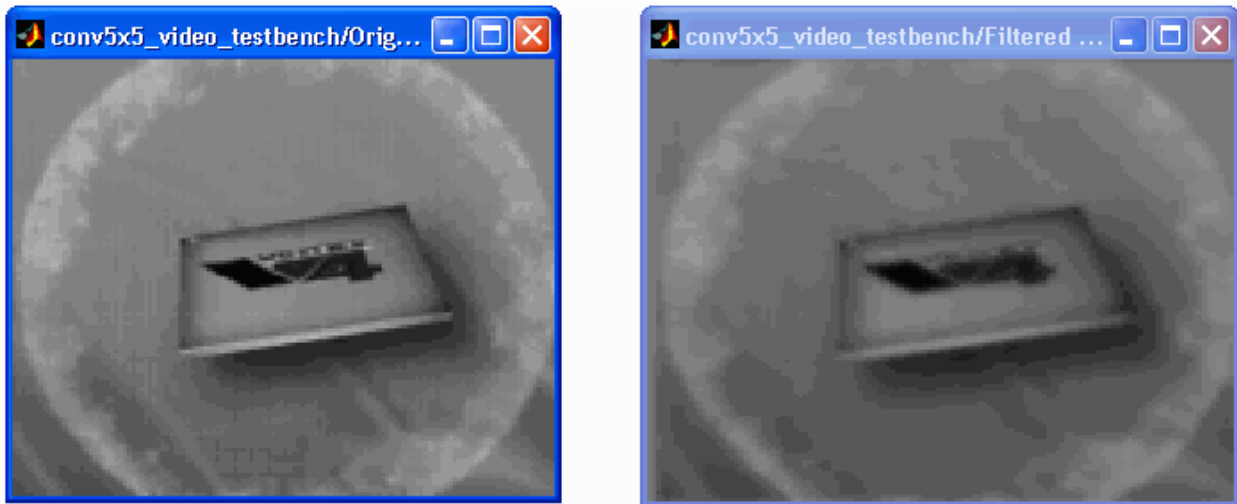
Figure 4-25: **Two Windows Shown after Configuration**

Frame Based Co-Simulation Tutorial

System Generator provides high-speed hardware co-simulation interfaces that allow the full contents of a Simulink vector or matrix signal to be read from or written to FPGA hardware in a single transaction. By using these interfaces, the user can significantly reduce the number of PC/hardware transactions during a simulation and further accelerate simulation speeds beyond what is traditionally possible with hardware co-simulation. A tutorial is provided that includes a step-by-step MAC FIR filter design to demonstrate how these interfaces can be used. For more information, refer to the *Frame-based Acceleration using Hardware Co-Simulation* tutorial under the Additional Topic and Tutorials section in the *System Generator User Guide*.

VSK Diagnostics and Support Tool Kit

Overview

The VSK diagnostics program serves to tie together the components of the VSK development toolkit (Figure 5-1) into a program to configure the ML402 and VIODC boards for video processing applications and to provide simple loopback and video processing functions. The VSK support toolkit consists of both hardware and software modules. The VSK support software can be used to manage all the overhead of interfacing to video streams on the VIODC and to provide simple video stream interface to and from a user design. Use of the VSK support software is optional. The software contains a set of basic features that can be used to develop video IP which is usable in custom environments. See Table 5-1.

ug217_ch6_01_120205
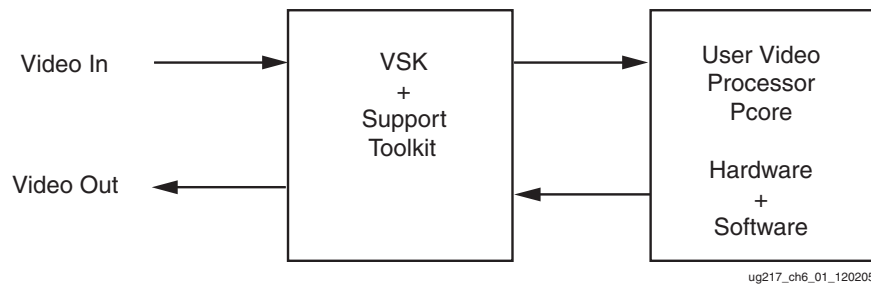
Figure 5-1: **VSK Support Toolkit to Develop a Video Processor Pcore**

Table 5-1: **VSK Support Toolkit Components**

Component Name	Current Filename	Type	Function
VIODC	vsk_viodc_xx.mdl	System Generator Multiple Subsystem	Configures the VIODC card and provides video streams to the ML402.
VIO_IF	vio_if.mdl	System Generator Pcore	Interfaces to VIODC and provides video streams to other pcores.
VIO driver	vsk_vio.c	C code	Configures VIODC and ML402 boards.
DDR_IF	vsk_ddr_xx.mdl	System Generator Pcore	Interfaces DDR memory to video streams and processor.
DDR driver	vsk_ddr.c	C code	Communicates with DDR and configures video into and out of memory.

Table 5-1: **VSK Support Toolkit Components** *(Continued)*

Component Name	Current Filename	Type	Function
VOP	vsk_vid_opx.mdl	System Generator Pcore	A simple video processor with color space, gamma correction, and a Bayer filter
VOP driver	vsk_vop.c	C code	Configures the video pipeline
VTOP	vsk_top.c	C code	Top-level diagnostics program

Notes:

1. Be sure to check the Xilinx VSK website for new updates to the VSK diagnostics.

VIODC Design

The VIODC design is constructed using the Xilinx System Generator. It contains interfaces to each of the various video interface ICs and an interface to the ML402 FPGA. The individual interfaces are simple designs with a control and status registers, pads, and a FIFO. See Figure 5-2.
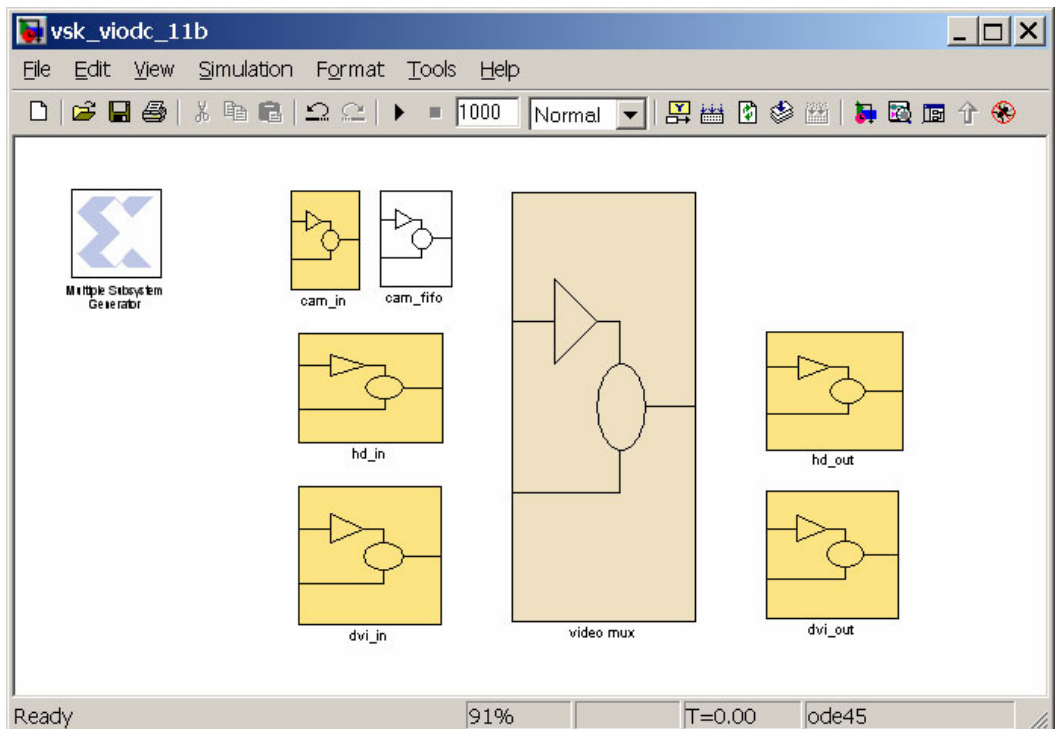
Figure 5-2: **VIODC – Top-Level Design with Seven Independent Clock Domains**

The video mux contains a large routing mux to select video from various inputs and route it to video outputs. It also contains a test pattern generator. In operation, the video mux runs at 100 MHz synchronous with the VIO interface back to the ML402 FPGA. This will be increased to 167 MHz in a future version of the VSK support package. Lower rate video is carried on the 100 MHz mux by using clock enables to identify the valid pixels. For instance, the VGA requires an approximate 25 MHz clock, and only one of four cycles is used to carry video. See Figure 5-3.
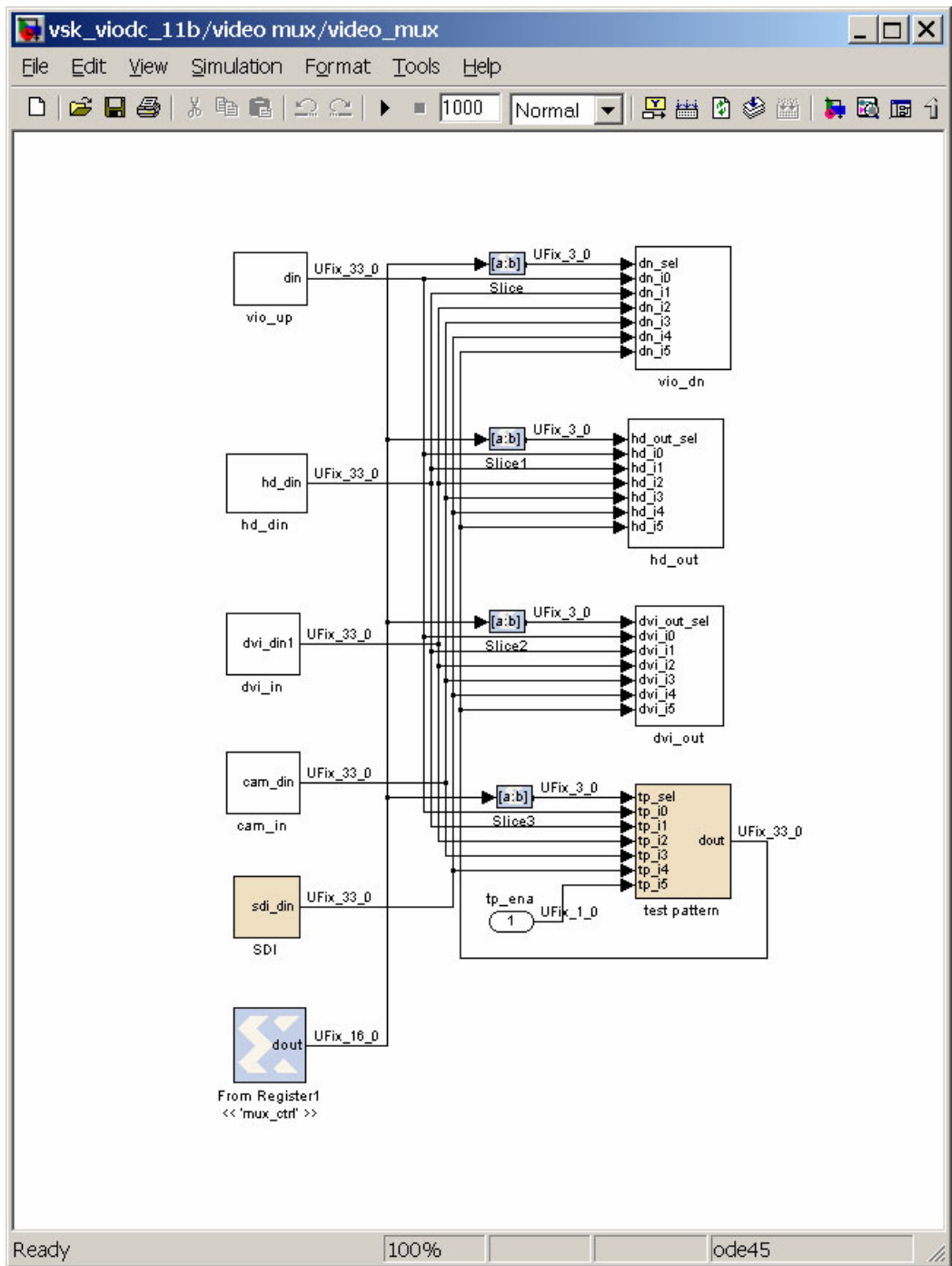
Figure 5-3: **VIODC Video Routing MUX**

The video is packed into a 33-bit word for transport inside the VIODC. It is arranged as shown in Table 5-2.

Table 5-2: **VIODC Video Format**

Bits	32	31	30	29:20	19:10	9:0
Function	pixel_valid	vsync_n	hsync_n	Red/Pr	Green/Y	Blue/Pb

IIC Interface

In addition to routing video data, the VIODC includes support for the IIC interface associated with each of the video interface ICs. While most of the interfaces are grouped into a common IIC bus called VID_SDA, VID_SCL, several other discrete IIC buses are used. Table 5-3 shows the IIC devices that are available.

Table 5-3: **Available IIC Devices**

Device Name	VIODC IC	IC Pin Name	IIC Bus Name	Device IIC Address
iic_clockgen	ICS1523	scl,sda	VID_SDA, VID_SCL	0x4c
iic_hd_out	ADV7321	scl,sda	VID_SDA, VID_SCL	0x54
iic_hd_in_ctl	ADV7403	scl,sda	VID_SDA, VID_SCL	0x40
iic_hd_in_vbi	ADV7403	scl2,sda2	ADV7403_SDA2	0x20
iic_dvi_out	TP410	dsel_sda, bsel_scl	VID_SDA, VID_SCL	0x70
iic_dvi_out_ddc	DVI OUT Connector	ddc_data, ddc_scl	DVI_OUT_DDC_SDA, DVI_OUT_DDC_SDA,	0xXX
iic_dvi_in_ddc	AD9887A	ddc_sda,scl	AD9887_SDA AD9887_SCL	0xA0
iic_dvi_in_vga	EDID PROM	sda,scl	VGA_IN_SDA, VGA_IN_SCL	0xXX
iic_dvi_in	AD9887A	sda.scl	VID_SDA, VID_SCL SCL	0x9A
iic_camera	MT9V022	camera_conn_sda, camera_conn_scl	CAMERA_SDA, CAMERA_SCL	0x98

VIODC-ML402 Serial Port

VIODC Serial Port Interface

The VIODC Serial Port (SPort) is used to write and read the registers of the VIODC FPGA design. Table 5-4, page 70 defines the 32 registers in the VIODC. The SPort consists of four signals: sport_clk, sport_sync, sport_up, and sport_dn. The ML40x outputs sport_clk, sport_sync and sport_up and inputs sport_dn.

The sport_clk is a clock that is provided by the ML40x. The clock is derived by dividing the ML40x system clock (~100 MHz) by 8 resulting in a sport_clk of ~12.5 MHz. The VIODC derives its system clock from the LVDS up_clk that should be driven by the ML40x FPGA. The up_clk should be driven by the ML40x system clock (~100 MHz).

The sport_sync denotes the beginning of a data write/read transfer. Sport_sync is High the first clock cycle of the transfer and then stays Low. A complete write cycle takes 32 clock cycles, 16 cycles of address and 16 cycles of data. The next 32 cycles are used to read the value of the same address that was used in the write cycle.

A new write/read cycle can be started every 32 cycles. For example:

Write cycle (sport_up): <00 xx> <01 xx> <02 xx> <03 xx> <04 xx> <05 xx> …

Read cycle (sport_dn): <xx xx> <00 xx> <01 xx> <02 xx> <03 xx> <04 xx> …

Sport_up is the serial write signal. The format for a write cycle is 16-bits of address followed by 16-bits of data.

Note: The address must be shifted by 1 bit to the left. The data is not shifted.

In Figure 5-4, *sport_up* shows a write cycle using address 0x5 and data 0xCCC5. Notice that the address is actually 0xA.
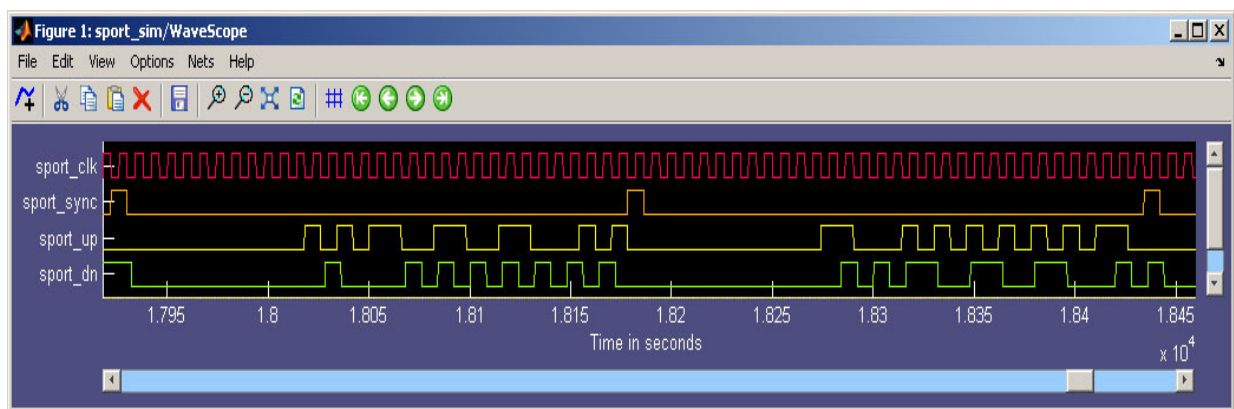
Figure 5-4: **SPort Waveform**

In Figure 5-4, waveform sport_up writes addr 0x0005, data 0xCCC5, addr 0x0006 data 5556 (actual: 0x000A,0xCCC5,0x000C,0x5556). Sport_dn reads addr 0x0004, data 0x5554, addr 0x0005, 0xCCC5 (actual 0x0008,0xAAA4,0x000A,0xCCC5) delay one extra clock cycle.

Sport_dn is the serial read signal. The format of the read cycles is 16-bits address followed by 16-bits data.

Note: The sport_dn read begins 33 clock cycles after the corresponding sport_sync signal. The address is shifted by one bit to the left. In Figure 5-4, sport_dn shows a read cycle.

Note: A problem with the read cycle causes it to return the previous value instead of the newly written value. To read the correct value, two reads are required. Since a read cycle only follows a write cycle, the value must be written into the register twice in order to successfully read the value back.

The SPort interface implemented for the VSK Diagnostics demonstration used two 16-bit dual-port memories with depths of 32 entries. One memory was used as a *write* memory and the second was used as a *read* memory. New register values were written into the address location of the *write* memory that corresponded to the address of the VIODC register to be written. A counter was used to continually cycle through the 32 entries so that the contents of the *write* memory was continually updated to the VIODC. The *read* memory was continually updated by reads from the VIODC registers.

VIODC Registers

Table 5-4: **VIODC Registers**

Register Name	Address	Bit Fields
VERSION_ID	0x0	
VIO_CTRL	0x1	[3]=cam_shift, [2:0]=vio_dn 0=up 1=hd_in 2=dvi_in 3=camera, 4=sdi 5=tp
CLOCK_CTRL	0x2	[15]=invcamclk [10:8]=dviout, [6:4]=hdout2, [2:0]=hdout1, 0=vio_up, 1=hd_in, 2=dvi_in 3=camera, 4=pll 5=clkdiv4
IIC_CTRL	0x3	201 = select camera, 100=dvi_in 80=vid 40=dvi_out_ddc 20=dvi_out 10=hd_in_dat 8= hd_in_ctl 4= hdout
PLL_CTRL	0x4	[15:12] = ics664_sel [11:8] = pll502_sel [7:5] = clk_mux_select 4 = gunlock_clk 3 = lvds_clkdiv 2 = lvds_clk 1 = sstl3_clkdiv 0 = sstl3_clk [4] = dac_cs_n [3] = dac_ldac_n [2] = dac_clr_n [1] = dac_sclk [0] = dac_din
VIO_UP_CTRL	0x5	[15:0]=vio_up_stat
VIO_DN_CTRL	0x6	[15:0]=vio_dn_stat

Table 5-4: **VIODC Registers** *(Continued)*

Register Name	Address	Bit Fields
HD_IN_CTRL	0x7	[1]=~fifo_rst, [0]=~reset
HD_OUT_CTRL	0x8	[7](1=invert H_sync, 0=default) [6](1=invert V_sync, 0=defalut) [5](0=HD mode, 1=SD mode) [4]=~fifo_rst, [3](0= sd hsync,vsync=1), [2](1=sd hsync,vsync 3-state), [1](0=dup green), [0]=reset(1=not reset)
DVI_IN_CTRL	0x9	[5]=~fifo_rst, [4]=coast [3]=clk_inv [2]=xclamp [1]=clk_ext, [0]=sel_b
DVI_OUT_CTRL	0xA	[3]=~fifo_rst, [2](1=vga_out) [1](1=vga_pd_n) [0](1=dvi_out_pd_n),
CAM_CTRL	0xB	[7]=hw/sw phase alignment (0=software, 1=hardware) (default = 0) [6]=~fifo_rst, [5:4]=sync inv, [3:0]=phase
MUX_CTRL	0xC	[14:12]=pat [10:8]=dvi [6:4]=hd [2:0]updn 0=updn, 1=hdin, 2=dviin, 3=cam, 4=sdi, 5=test;
PATTERN_CTRL	0xD	[1:0]=test_pattern
	0xE	Reserved
	0xF	Reserved
VIODC_VERSION	0x10	[15:0]= viodc fpga version
VIO_STAT	0x11	0xEBDB
CLOCK_STAT	0x12	[15:0]=clock_ctrl

Table 5-4: **VIODC Registers** *(Continued)*

Register Name	Address	Bit Fields
I2C_STAT	0x13	[3]=sda_dn, [2]=scl_dn, [1]=sda_up, [0]=scl_up
PLL_STAT	0x14	[1]=lock, [0]=func
VIO_UP_STAT	0x15	[15:0]=vio_up_ctrl
VIO_DN_STAT	0x16	[15:0]=vio_dn_ctrl
HD_IN_STAT	0x17	[1]=field, [0]=genlock
HD_OUT_STAT	0x18	[2]=s_vsync, [1]=s_hsync, [0]=s_blank
DVI_IN_STAT	0x19	[3]=in_de, [2]=ctl, [1]=scdt, [0]=sogout
DVI_OUT_STAT	0x1A	[0]=msen
CAM_STAT	0x1B	[11:0]=LVDS camera output
	0x1C	Reserved
RESET	0x1D	[0] = ML402 Reset
DIP_SWITCH	0x1E	[7:0]=dip_switch
BOARD_VERSION	0x1F	[3:0]=VIODC board version

Clock Routing

The VIODC contains a mux to select clock from video sources and route it to video outputs destinations.
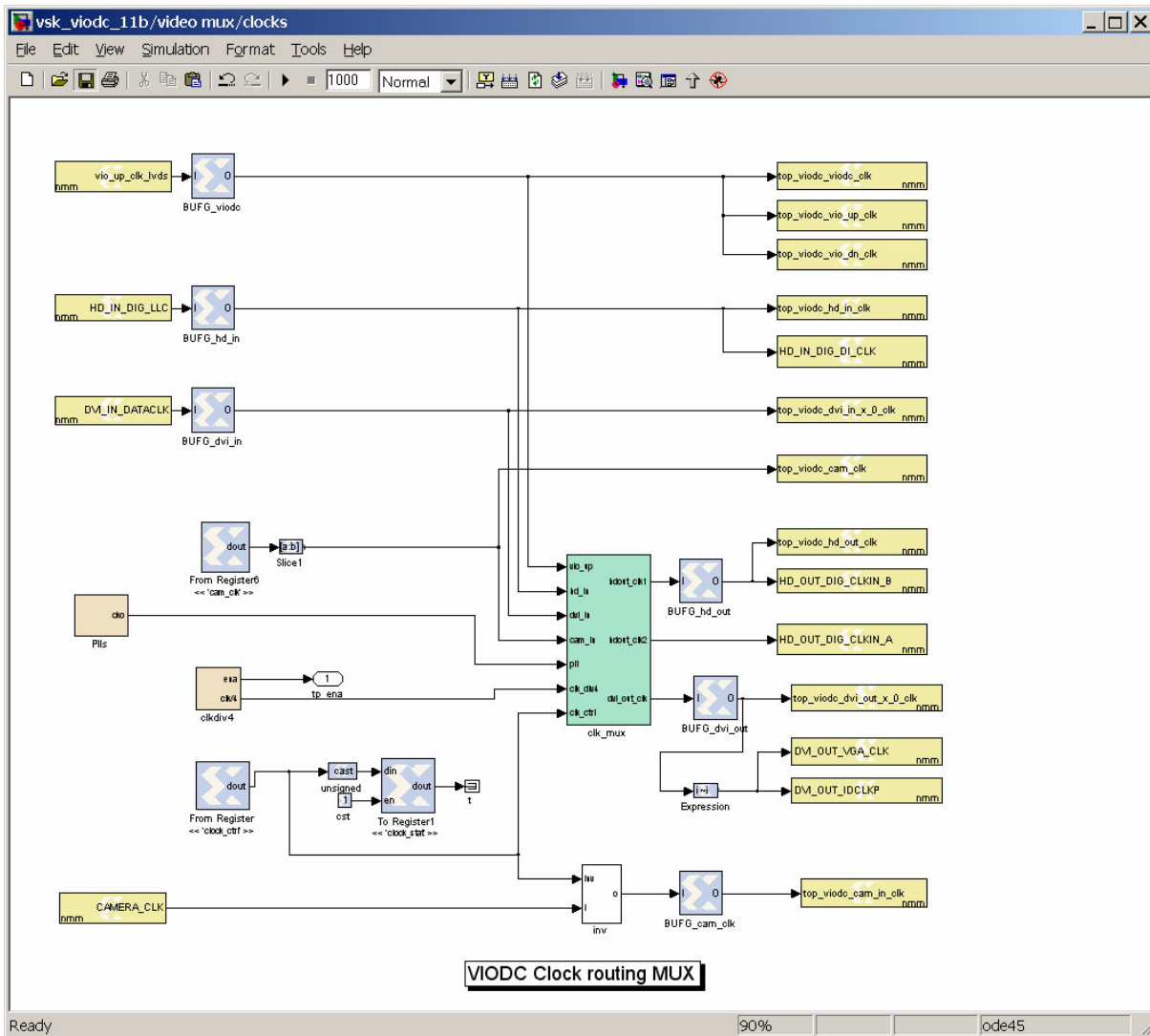
Figure 5-5:   **VIODC Clock Routing MUX**

VIO Design

The VIO design is the counter part of the VIODC design described in the previous section. The VIO communicates with the VIODC over the VIOBUS. It supports the SPort serial bus which writes/reads the VIODC registers, the IIC interface which programs the VIODC peripheral chips, and the upstream and downstream data transfers. It also contains a small test pattern generator and a video mux to route video between various destinations. The VIO can be exported as a Pcore which plugs into an EDK project, or it can be used as part of a larger System Generator design that is exported directly to a bitstream that can be used to program an FPGA.

Figure 5-7 shows the top-level view of the VIO Pcore. The *vio if* block has a mask associated with it that can be viewed in Figure 5-7. To access the mask, double click on the *vio if* block. The user can access the vio design under the *vio if* block by right-clicking on the block and selecting *Look Under Mask*. Figure 5-8 shows the *Look Under Mask* view of the *vio if* block.
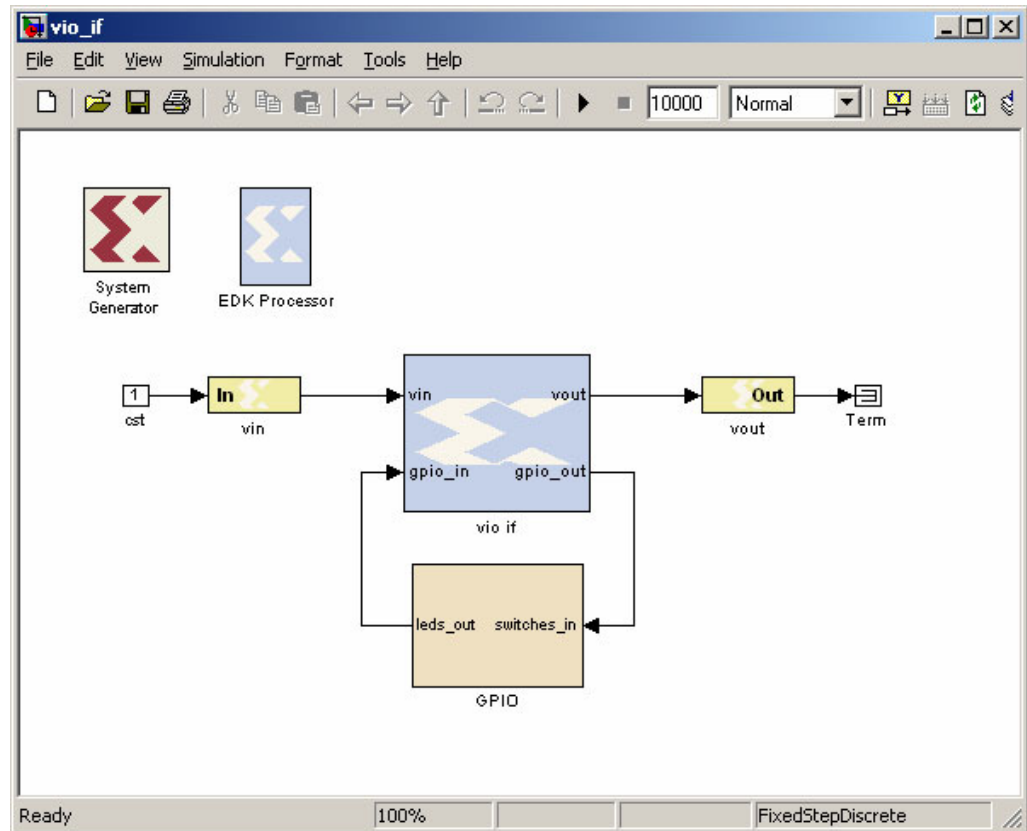
Figure 5-6: **VIO Pcore Top-Level Diagram**
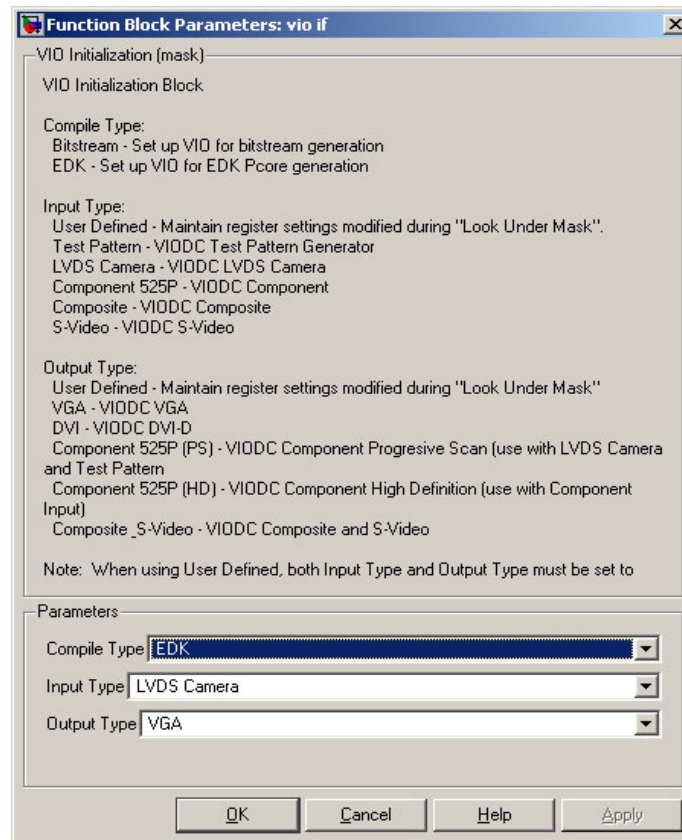
Figure 5-7:   **VIO Parameter Mask**
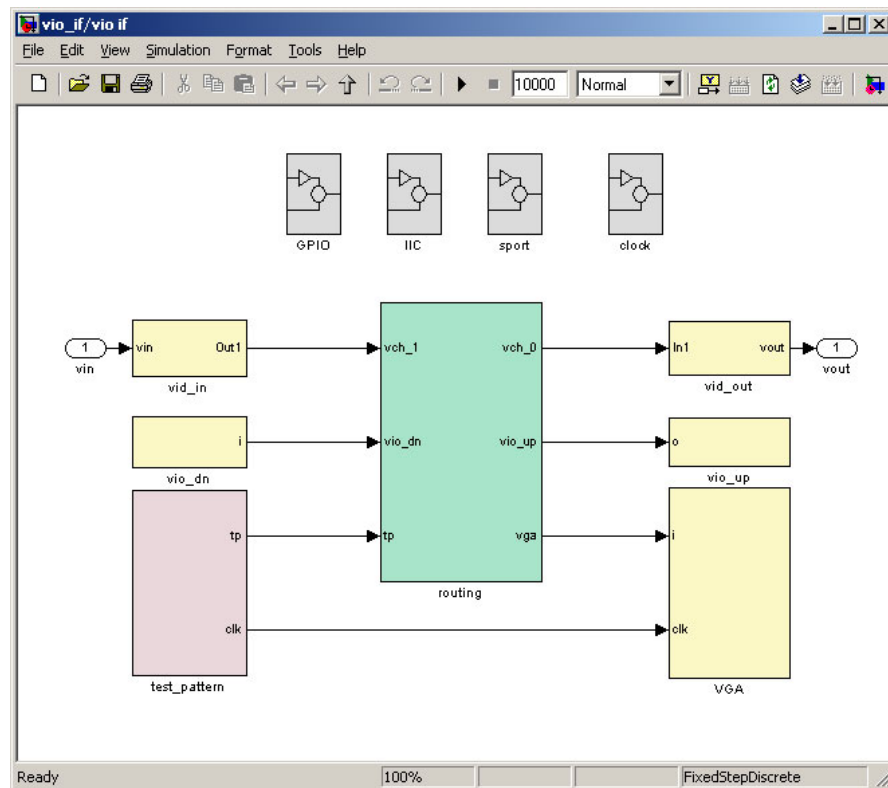
Figure 5-8: ***Look Under Mask* View of the *vio if* Block**

VIO Mask

The mask for the *vio if* block is used to select the compilation mode of the block and to select the input and output initializations. The mask can be accessed by double clicking on the *vio if* block.

Compile Type

The Compile Type field of the VIO mask is used to specify whether the block is being targeted towards a Pcore design that will be exported to EDK or towards a bitstream design that will be generated strictly from within System Generator. It is the user's responsibility to verify that the correct mode is selected before running "generate" from the System Generator block in their design. Errors can result if the incorrect compile mode is selected. When using the VIO block in a Pcore design, the user needs to regenerate the memory map of the EDK Processor block if the compile mode is ever modified.

When using the VIO block in EDK mode, the design is modified to use *shared memories* for the registers and memories in the design. This allows the creation of a memory map that a processor can use to access the various registers and memories.

When the VIO is compiled as an EDK Pcore, it can then be imported into an EDK project. From there, the registers and memories in the VIO can be accessed by the MicroBlaze processor in the EDK project. The accesses are made by using function calls that are auto-generated during the Pcore generation in System Generator.

Bitstream mode modifies the design to use *local memories*. Local memories are used because a processor is not used in this type of design. As a result, the memory map interface is not needed.

Input Type

The Input Type field of the VIO mask is used to specify the input that will be initialized at start up. The user can select from a Test Pattern (4:4:4 RGB mode), the LVDS Camera (4:4:4 Bayer format), the Component input set to 525P format (4:4:4 RGB), the Composite input (4:4:4 YCrCb), and the S-Video input (4:4:4 YCrCb).

A User Defined option allows the user to select if they plan to make changes directly to the initialization fields of the various registers and memories. The User Defined option keeps the mask from overwriting the edits the user has made.

Note: When using the User Defined option, both the Input Type and the Output Type must be set to User Defined.

Output Type

The Output Type field of the VIO mask is used to specify the output that will be initialized at start up. The user can select from VGA output (4:4:4 RGB mode), DVI output (4:4:4 RGB), Component output set to 525P (Progressive Scan or High Definitions formats) (4:4:4 YCrCb), Composite output (4:4:4 YCrCb), and the S-Video input (4:4:4 YCrCb).

A User Defined option allows the user to select if they plan to make changes directly to the initialization fields of the various registers and memories. The User Defined option keeps the mask from overwriting the edits the user has made.

Note: When using the User Defined option, both the Input Type and the Output Type must be set to User Defined.

Mask Modifications

The script for the VIO mask is unlocked so that the user has the option to modify the input/output settings or add new input/output options to meet their needs. It is highly recommended that the user only modify the variable initializations at the top of the script. The script can by found by right clicking on the *vio if* block and selecting Edit Mask. In the Mask editor window, select the Initialization tab. The Initialization commands window contains the script for the mask.

EDK Pcore

The VIO block can be used in a design that is targeted at being exported as an EDK pcore. Figure 5-6 is an example of such a design. This is the recommended way to use the VIO with EDK.

It is also possible to copy the VIO block in to a larger design that is to be exported as a pcore. The VIO contains input and output gateways that are expected to be connected to external pins in the EDK project. The user is responsible for resolving all of those issues. The VSK Diagnostics EDK design can be use as a reference.

The user is responsible for making sure that the Compile Type of the VIO mask is set to EDK before the pcore is generated. The user is also responsible for making sure that the memory map was properly generated in the EDK Processor block.

Bitstream

The VIO block can be used in a design that is targeted for being generated as a bitstream directly from System Generator. Figure 5-9 is an example of a standalone bitstream design. In this design, the VIO receives video from the VIODC and provides that data on its vout port to the VOP block for processing. The output of the VOP block is routed around to the vin input of the VIO block. The VIO then routes that data up to the VIODC for display.
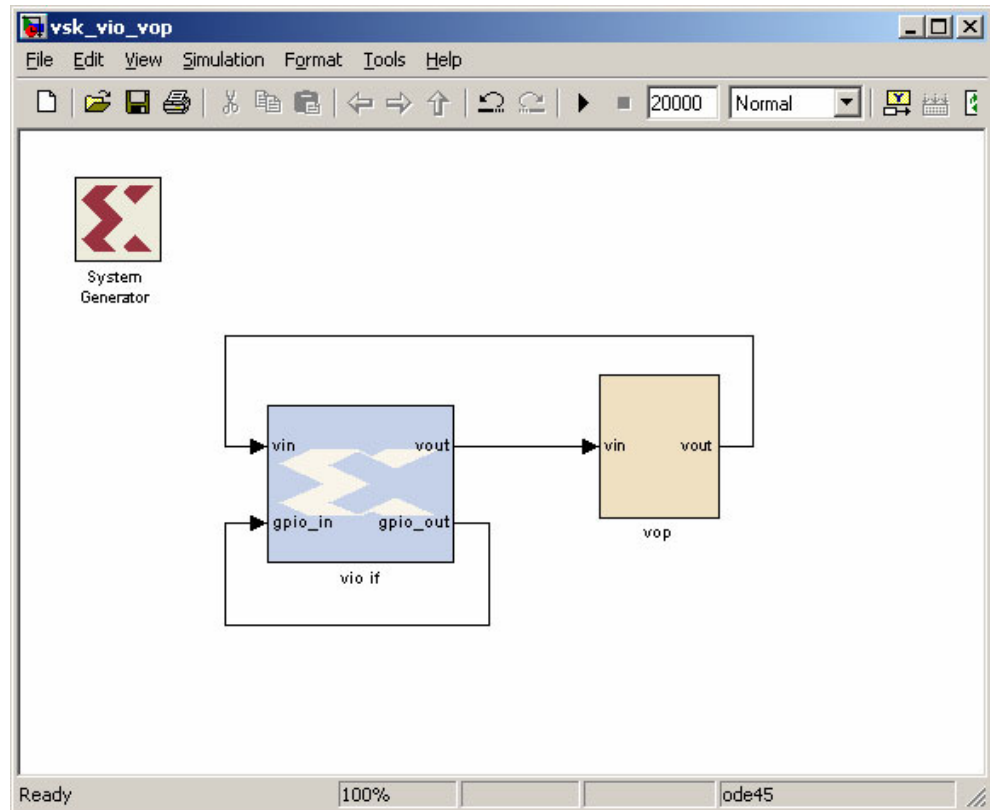
Figure 5-9: **VIO Bitstream Design**

The user can use the VIO mask to select the input and output types that the design will initialize at start-up. The user is responsible for making sure that the Compile Type is set to Bitstream before the design bitstream is generated.

There are input and output gateways in the VIO that must be connected to the proper FPGA pins. To designate those pin locations, there is a .ucf file that must accompany a bitstream design that uses the VIO block. An example of this .ucf file can be found on the VSK CDROM at `Examples\VSK_StandAlone`. The file is called `DesignName_cw.ucf`. This file must be copied into the target directory that the user specifies for building the bitstream. The file should be renamed to `<design_name>_cw.ucf`. (e.g., `vsk_vio_vio_cw.ucf`)

There are two ways to accomplish this:

1. Create the directory structure and copy the .ucf file into the target directory before generating the bitstream.

2. Generate the bitstream, copy the .ucf file into the created target directory, and regenerate the bitstream. If the .ucf file is not present in the target directory when the bitstream is created, the bitstream is generated, but the FPGA pins are selected

randomly by the software, instead of being locked to the correct locations as specified by the .ucf file.
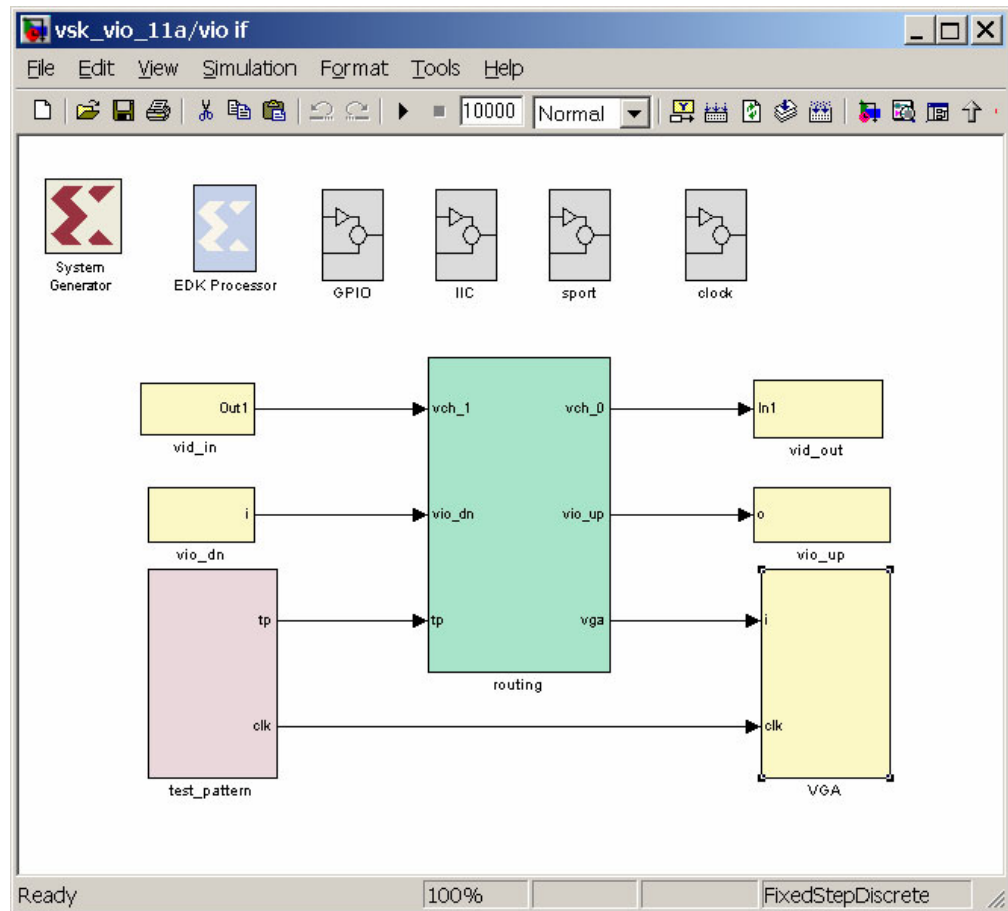
Figure 5-10: **VIO Pcore Top-Level Diagram**

VIO I/O Buses

The vin and vout buses of the vio_if block are defined in Table 5-2. The gpio_in and gpio_out buses are defined in Table 5-5.

Table 5-5: **VIO_IF GPIO Format**

Bits	12:9	8:5	4	3	2	1	0
gpio_out	dip_sw (8:5)	dip_sw (4:1)	sw_N	sw_E	sw_w	sw_S	sw_C
gpio_in	-	led(3:0)	led_N	led_E	led_W	led_S	led_C

VIO Registers

Table 5-6: **VIO Registers**

Register Name	Bit Fields
VIO_IF_IIC_CTRL	[3]=ena_iic_ram, [2]=sel_iic_ram, [1]=iic_scl, [0]=iic_sda
VIO_IF_IIC_STAT	[0]=sdai
VIO_IF_LED_CTRL	[4:0]=newsc_leds
VIO_IF_PATTERN_CTR	[2:0] VGA pattern control
VIO_IF_SPORT_CTRL	[4:0]= sport input delay compensation
VIO_IF_SPORT_STAT	[4:0]=current word counter
VIO_IF_SWITCH_STAT	[15:8]= dip_sw, [4:0]=newsc_button
VIO_IF_VGA_CTRL	[2]fifo reset [1:0]=vga_sel(0=test_pattern, 1=vid_out, 2=vid_sync) copied to vga stat
VIO_IF_VGA_STAT	[2:0] from to vga ctrl
VIO_IF_VIO_CTR	[4:3]vch_src [2:1]=vga-up_src(0=vio_dn, 1=vch_up, 2=tp [0]=vio_reset
VIO_IF_VIO_STAT	[9:0]=vio_dn[9:0]

DDR Design

The DDR pcore is used to buffer video streams in memory. In addition, it provides access to the DRR memory to the MicroBlaze processor. To do this, it uses the DDR controller included in the VSK examples. Figure 5-11 shows how the DDR read and write ports are attached to Shared FIFOs which are accessible by the processor.
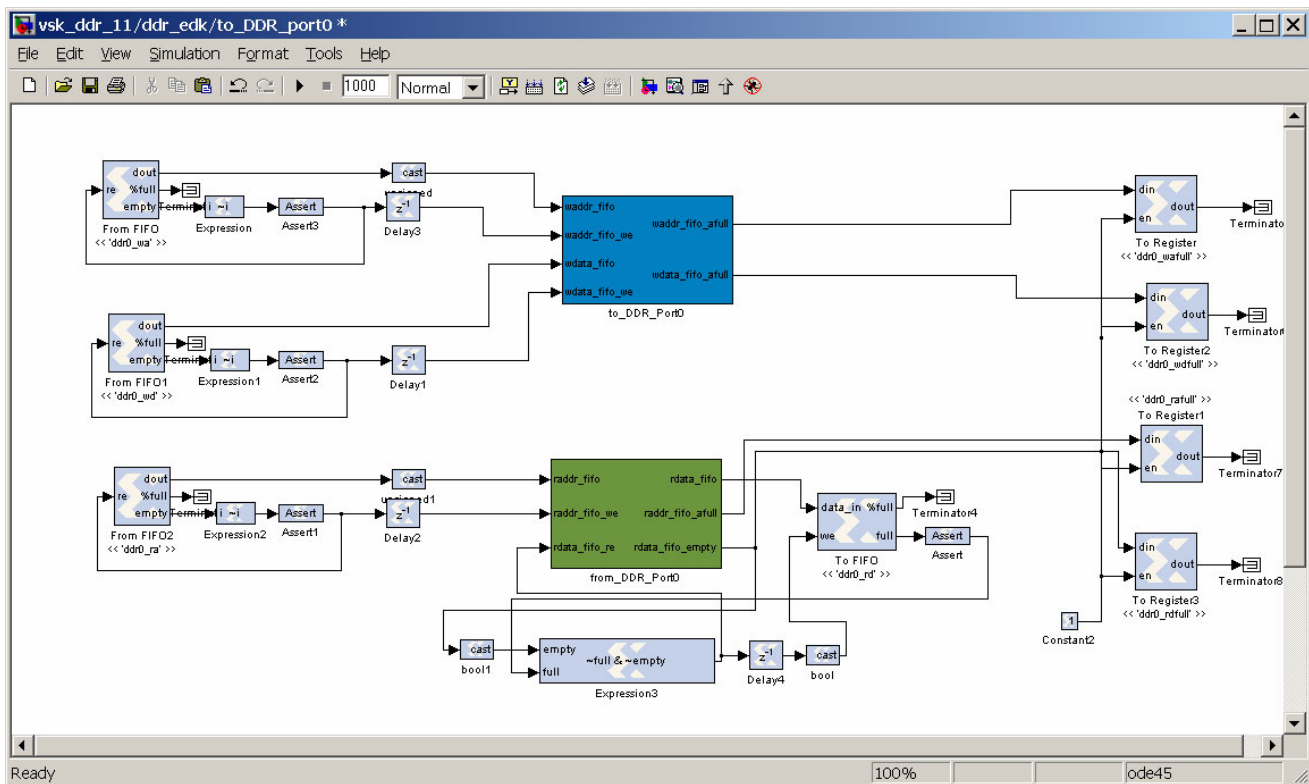
Figure 5-11:   **DDR Design**

VOP Design

The VOP design is a simple video processing pipeline. It is available to be modified or replaced by other video processing system. The vid_in and vid_out streams connect to the DRR memory, and VIO Pcores. See Figure 5-12.
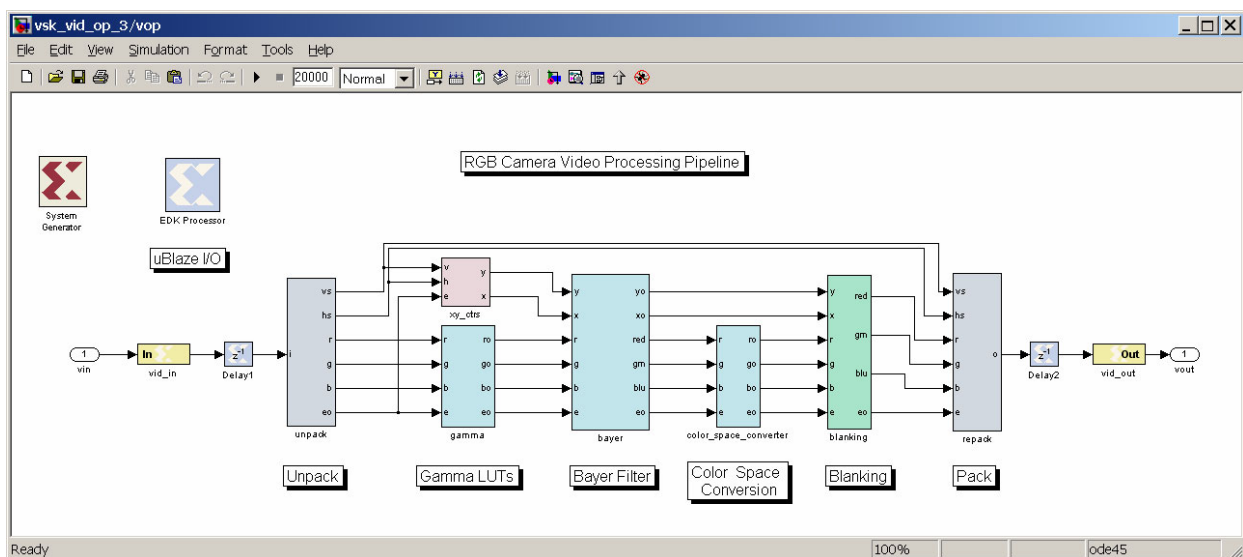
Figure 5-12:   **RGB Camera Video Processing Pipeline**

Running the Diagnostics

The Video Starter Kit can be configured to run a demo and diagnostics program. This program can be used to verify the operation of the VSK. This program can be used to enable the VIODC video interfaces in a passthrough mode, and demonstrate FPGA video processing on live video streams.

To run the demo, the hardware needs to be setup as shown in Figure 5-13 to talk to a PC over an RS-232 serial port.
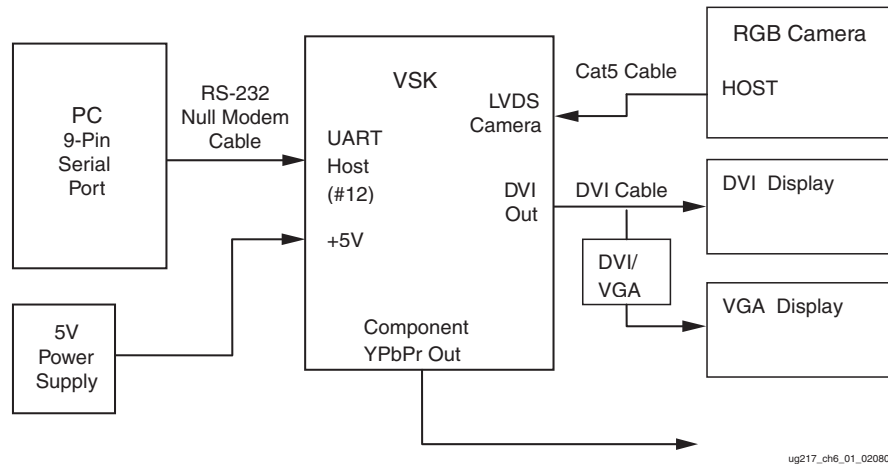
Figure 5-13: **VSK Demo Setup**

The VSK diagnostics also require LVDS camera to be connected to the VSK. Note that the VSK consists of a VIODC plugin card attached to a ML402 main board. The RS-232 and power connect to the ML402 main board and the RGB LVDS camera and DVI/VGA output connect to the VIODC plugin card.

Hardware Setup

Refer to Figure 5-14 and Figure 5-15 for the locations of each interface port.
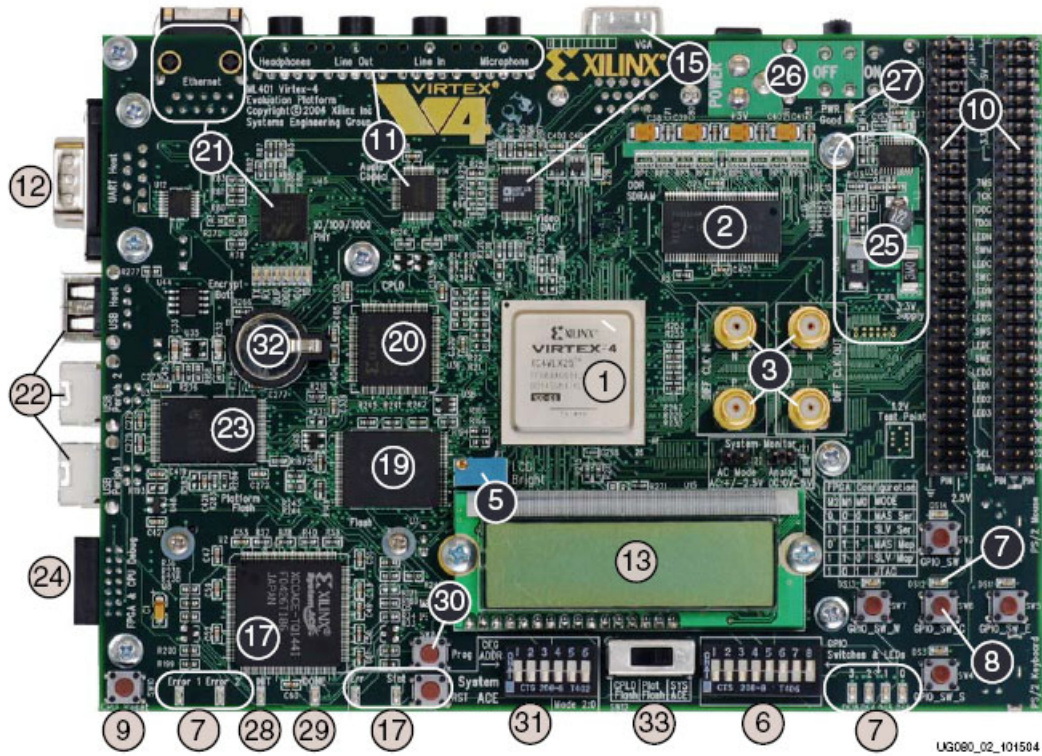
Figure 5-14: **ML402 Board - Top View**
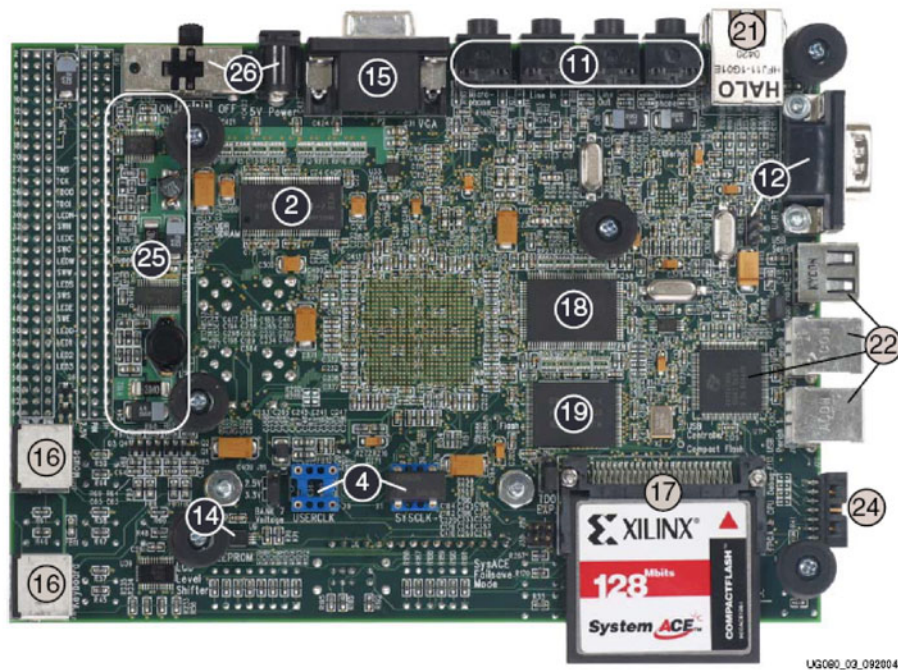
Figure 5-15: **Virtex-4 ML40x Evaluation Platform Components (Back)**

1. Insert the System Ace Flash Memory Card into the System ACE connector.

2. Connect the ML402 UART Host port (#12) to a PC with a 9-pin RS-232 serial port cable.

3. Connect the VIODC LVDS Camera port to the Irvine Sensors RGB Camera's HOST port using the CAT6 network cable.

4. Connect the VIODC DVI Out port to a DVI, VGA or component (YpbPr) video display. If connecting to a VGA device, use the DVI to VGA adapter included in the VSK.

5. With the power switch set to OFF, connect the ML402 5V power supply input (#26) to the output of the 5V power brick.

Software Setup

1. Open a HyperTerminal window in
   `Windows/Accessories/Communications/HyperTerminal`,
   (or use any other terminal program such as [Tera Term Pro](#)).

2. Configure the HyperTerminal for 115,200 bits per second, 8-bit, no parity, no flow control.

Configure the ML402 Board to Run the Diagnostics

1. Set the DIP switch CFG ADDR[2:0] to 0 and Mode[2:0] to 0 to as shown in Figure 5-16.

2. Switch the ML402 power switch to ON. The Power Good LED should light on the VIODC.

3. After power up, or pressing the RST button, select the VSK diagnostics program by pressing the center button. There are five push switches arranged labeled GPIO_SW_N, GPIO_SW_S GPIO_SW_E, GPIO_SW_W GPIO_SW_C. You can use the north-south pair to select a demo program. Alternately, a demo menu will appear on the ML402 board's VGA output. A PC with an RS-232 serial port can be used to select the demo.
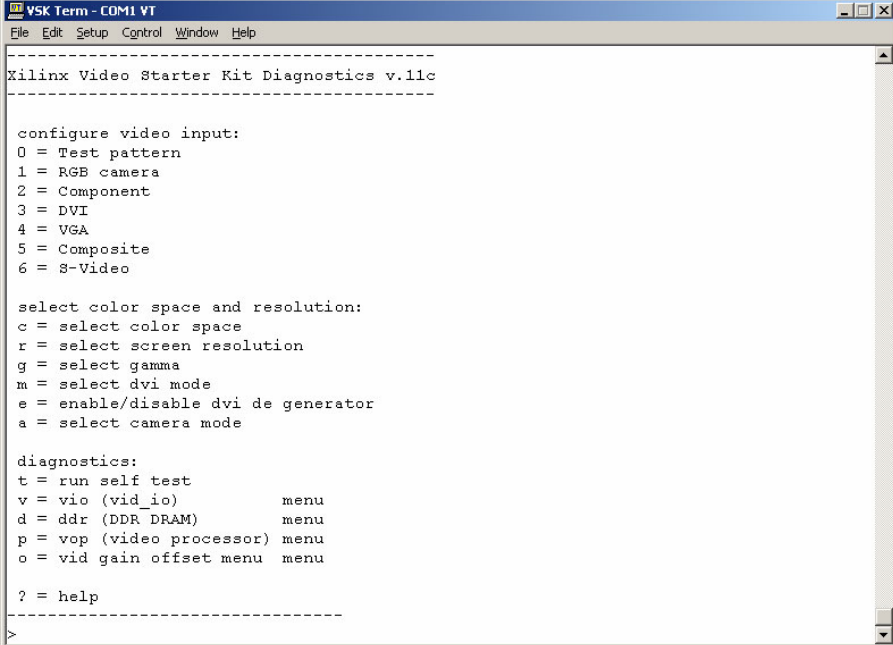
Figure 5-16: **Configure the ML402 Board**

4. After the ML402 and VIODC FPGAs are configured with the VSK diagnostics program, a menu should appear on the HyperTerminal screen.

Running the VSK Diagnostics

After power up, the HyperTerm RS-232 terminal window appears as in Figure 5-17. This indicates that the ML402 board has booted correctly and is able to communicate with the PC.


```
VSK Term - COM1 VT
File  Edit  Setup  Control  Window  Help
----------------------------------------
Xilinx Video Starter Kit Diagnostics v.11c
----------------------------------------

configure video input:
0 = Test pattern
1 = RGB camera
2 = Component
3 = DVI
4 = VGA
5 = Composite
6 = S-Video

select color space and resolution:
c = select color space
r = select screen resolution
g = select gamma
m = select dvi mode
e = enable/disable dvi de generator
a = select camera mode

diagnostics:
t = run self test
v = vio (vid_io)         menu
d = ddr (DDR DRAM)       menu
p = vop (video processor) menu
o = vid gain offset menu  menu

? = help
------------------------------
>
```
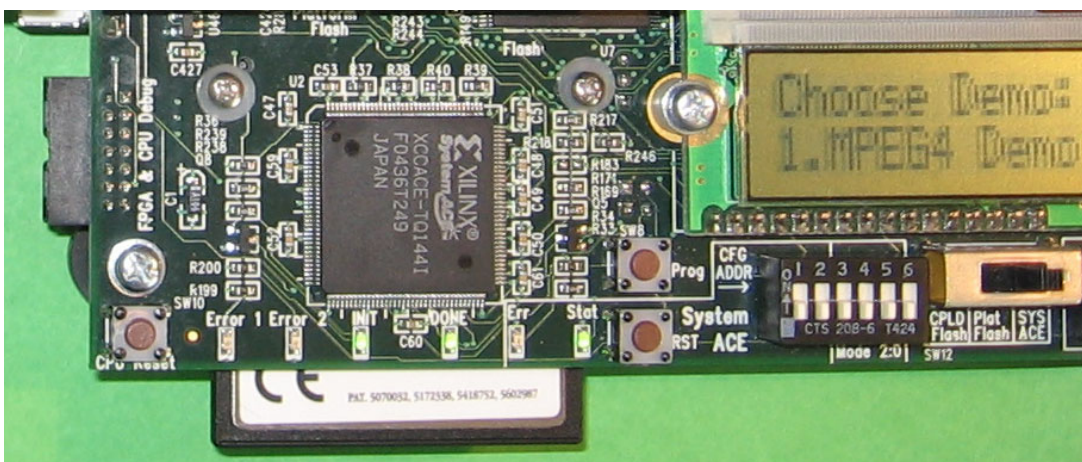
Figure 5-17:   **HyperTerm RS-232 Terminal Window**

1. Press "?" anytime to display the menu.

2. Then run the VSK diagnostics Self-Test program by pressing t. (No return is necessary). The self-test program reports any errors. If errors are encountered, check the cables and retry. If the self-test still reports an error, contact Xilinx Support.

RGB Camera Test

1. To configure the RGB camera to display on the video outputs, press 1.

2. After configuring the RGB camera to display, a color image from the camera should be visible on the output video screen. If the display is connected to the component video, the colorspace is incorrect since the camera is RGB, and the component video is YPrPb. To change colorspace, press the c key, to cycle between several color space options until the color is corrected.

The VSK also supports VGA, DVI, and component video inputs and outputs. These video interfaces can be tested as described next. Note that not all resolutions are supported.

Component Video Input Test

To test the component video input:

1. Connect any component video source, such as a component capable DVD player.

2. Select option 2 in the diagnostics menu.

3. Select the appropriate resolution using the r key.

4. The VSK supports 525P, 720P, and 1080I component video.

Depending on the output monitor, the user should see a component image on the VGA and DVI outputs. If using component output, select the output resolution and color space using the r key.

DVI Input Test

To test the DVI input:

1. Connect a DVI video source, such as a PC or a DVI capable DVD player to the VSK DVI input connector.

2. Select option 3 in the diagnostics menu.

3. Select the appropriate resolution using the r key.

4. The VSK supports 525P, 720P, and 1080I DVI video. Other standards will be supported in the future.

Depending on the output monitor, the user should see the DVI image on the VGA and DVI outputs. If using component output, select the output resolution using the r key.

VGA Input Test

To test the VGA input:

1. Connect a PC output to the VGA input on the VSK VIODC board using a VGA cable.

2. Set the PC screen resolution to 800 x 600.

3. Configure the VSK input to VGA.

4. Select option 4 in the diagnostics menu.

5. Other VGA screen resolutions are not currently supported. Other standards will be supported in the future.

The user should see the PC screen on the VSK VGA or DVI output. Component video devices typically will not sync to the VGA signal.

Composite Input Test

To test the Composite input:

1. Connect the Composite input port on the VSK VIODC board to a Composite video source, such as a DVD player using a Composite cable (RCA cable).

2. Connect the Composite output port of the VSK VIODC board to a Composite display, such as a TV using a Composite cable (RCA cable).

    Note: The S-Video output port of the VSK VIODC can also be used.

3. Select option 5 in the diagnostics menu.

The user should see the video on the Video display. Only the Composite video and S-Video outputs are presently supported for this option.

S-Video Input Test

To test the S-Video input:

1. Connect the S-Video input port on the VSK VIODC board to an S-Video source, such as a DVD player using an S-Video cable.

2. Connect the S-Video output port of the VSK VIODC board to an S-Video sink, such as a TV using an S-Video cable.

Note: The Composite video output port of the VSK VIODC can also be used.

3. Select option 6 in the diagnostics menu.

The user should see the video on the Video display. Only the Composite video and S-Video outputs are presently supported for this option.

Additional Diagnostics and Controls

The VSK diagnostics package contains additional functionality which can be accessed by selecting the VIO, DDR, and Video Processor menus. To exit these submenus, press the escape key.

VIO Diagnostics Peek and Poke Facility

The VIO diagnostics contains a simple keystroke menu which provides access to all the registers in the ML402 FPGA, the VIODC FPGA, and the ICS Interface on the VIODC board. The menu uses key pairs to select a device and register address. The selected register can be read or written with a data value. The data value can be incremented or decremented to desired value. Although simple, this facility helps to setup device registers in an interactive mode.

Table 5-7: **Keystroke Menu**

Key	Parameter	Action	Function
-	device	device--	Decrements the device id
=	device	device++	Increments the device id
[	addr	addr--	Decrements the device register address
]	addr	addr++	Increments the device register address
;	data	data--	Decrements the data parameter
'	data	data++	Increments the data parameter
,		read	Data=register[device][addr]
.		write	Register[device][addr]=data

Table 5-11 shows the devices that are available.

Table 5-8: **Available Devices**

Device	Device ID
ML402	0
VIODC	1
iic_loopback	2
iic_clockgen	3
iic_hd_out	4
iic_hd_in_ctl	5
iic_hd_in_vbi	6

Table 5-8:   **Available Devices**

Device	Device ID
iic_dvi_out	7
iic_dvi_out_ddc	8
iic_dvi_in_vga	9
iic_dvi_in	10
iic_camera	11

VIO Diagnostics - Device Configure Facility

The peek and poke menu makes it simple to read and write registers, but setting groups of registers can become tedious. The VIO diagnostics contain a feature to configure groups of registers into specific modes to support applications level such as configuring the VIODC to route the LVDS Camera to the VGA output. Use the <q> and <w> key to select an application mode and <a> to configure the VIODC. The VSK diagnostics are written to allow these same to be called from other software programs to configure the VIODC and ML402 into various operational modes.

Troubleshooting

1. Flat panel displays cannot display all video standards, or video which is out of spec.

2. VGA and flat panel displays require RGB data, component video, and S-video require YCC or YPbPr encoded video. If the incorrect video format is selected, the video will appear on the display, but with incorrect colors.

3. VGA and flat panel displays require RGB data and component video, and S-video require YCC or YPbPr encoded video. If the incorrect video format is displayed, the video appears with incorrect colors.

4. VGA and component require gamma corrected video, where flat panels usually apply a gamma correction. If the display has too much contrast or is washed out, it is likely that the gamma correction is incorrect.   In addition, the image is shifted with regions of black.

VSK Tutorial

Overview

Video processing systems often contain a mix of both hardware and software components. For instance, a video processing block may contain registers which need to be initialized by an embedded processor. This tutorial illustrates the process of creating a video *processing core* or pcore which is compatible with systems constructed with the Xilinx Embedded Development Kit (EDK). EDK pcores are reusable peripherals which can be imported into any EDK project. The Video Starter Kit (VSK) can be used with System Generator to develop EDK pcores that process live video streams.

Creating a Video Gain and Offset Peripheral

The tutorial creates a simple peripheral to apply basic gain and offset controls to a video stream. It covers the following:

- Gain and offset theory
- System architecture overview
- Design entry
- Testing the video function in System Generator
- Generating the pcore
- Importing the pcore into an EDK project
- Importing the pcore software drivers
- Controlling the pcore from a demo menu
- Running the tutorial with live video on the VSK

Gain and Offset Theory

Figure 6-1 illustrates the video gain and offset processing system that is applied to a video stream.
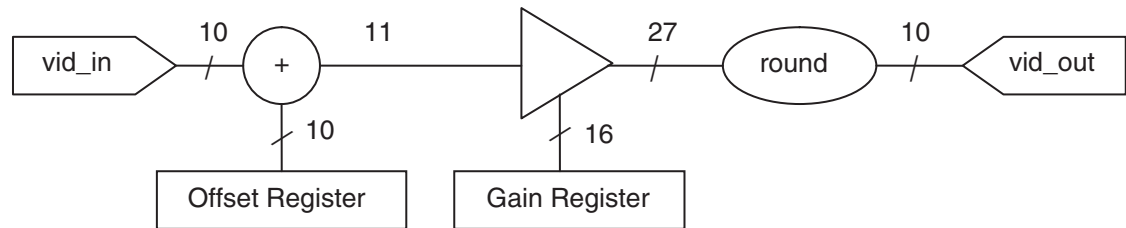
ug217_ch7_01_121205

Figure 6-1: **Gain and Offset**

The function implements the following equation for three video channels:

$$Vid\_out = (vid\_in(channel) + offset(channel)) * gain(channel)$$

System Architecture

In the tutorial system, the gain and offset is implemented as a new pcore and inserted into the VSK diagnostic/demo system. See Figure 6-2. Gain and offset for each channel are controlled by registers that are writeable from a MicroBlaze processor. The video input stream and output stream are supplied by the vio_if pcore which controls the VIODC board. The VIODC pixel clock rates are determined by the video source pixel clock, while the MicroBlaze and video processing pcores run at 100 MHz. The video sources are buffered in FIFOs and converted to 100 MHz streams with an accompanying pixel enable signal.
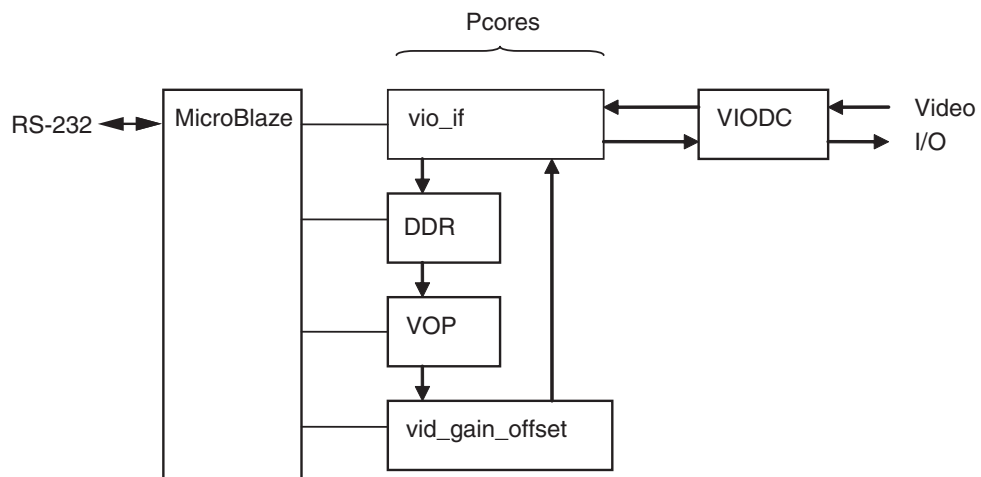
ug217_ch7_02_121205

Figure 6-2: **Gain and Offset System Architecture**

Video Stream Format

The VSK demo system assumes that video streams are packed into 33-bit fields. Packing the various video signals into a single field simplifies routing the video channel. The model

includes a test pattern generator that generates a packed video stream and pack and unpack functions. The packed format for VSK video streams is:

```
vid[32:0]=
{
 pix_enable,
 vsync_n,
 hsync_n,
 red[9:0],
 green[9:0],
 blue[9:0]
};
```

Pixel Enable

The video stream format includes a pix_enable signal that indicates the presence of an active video pixel. The use of the pix_enable signal allows video streams with arbitrary pixel clocks rates to be processed on a system with a greater but fixed frequency. For instance, the RGB camera uses a 26.6 MHz pixel clock, but the processing clock is 100 MHz. This means that roughly one in four clocks carry active pixels. The processing blocks can use the pix_enable as a clock enable or take advantage of the higher processing rate by using multiple clocks to process each pixel.

Tutorial Files

The `vsk_vid_pcore_tutorial` files can be found in the `%VSK\Examples\SysgenTutorial\VSK_pcore_tutorial` directory (%VSK refers to the directory where the VSK CD is installed).

The following files and directories are used in the tutorial:

- `vid_go_start.mdl`   // a starting model for the tutorial
- `vid_go_solution.mdl` // the finished model
- `edk_vid_go_start` // the starting EDK project directory
- `edk_vid_go_solution` // the completed EDK project directory
- `viodc_wrapper_v11b.bit` // the VIODC bit file
- `vsk_vid_gain_offset.c` // the C driver software for gain offset
- `vsk_tutorial_top.c`   // the top-level c-file for the tutorial

Building the Gain Offset Pcore in System Generator

The gain and offset peripheral is created using Xilinx System Generator toolkit and MATLAB Simulink. After testing in System Generator, the design is exported as a pcore and integrated into an existing EDK project. To create the gain and offset system, follow these steps.

1. Open MATLAB.
2. Open the model `vsk_vid_pcore_tutorial/vid_go_start.mdl.` Save the design as `vid_go.mdl.`

   The model includes a test pattern generator which generates a packed video stream with pack and unpack functions. The test pattern simulates a 100 pixel x 10 line video frame, with R,G,B ramps in the horizontal direction.

3. Run the model by clicking on the start simulation button. The user should see the test pattern displayed on the scope as shown in Figure 6-3.
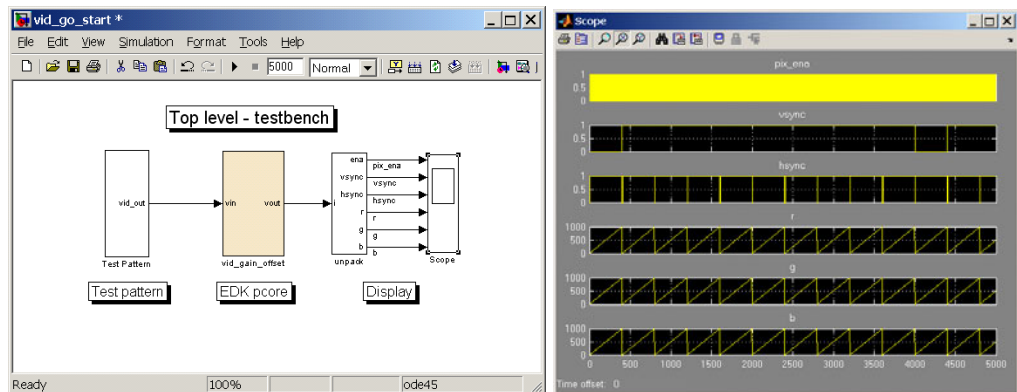

Figure 6-3: `vid_go_start` **Simulation Results**

4. Click on the block labeled `vid_gain_offset` to open the block to be generated as a pcore. The name of this block will be the name of the generated pcore.
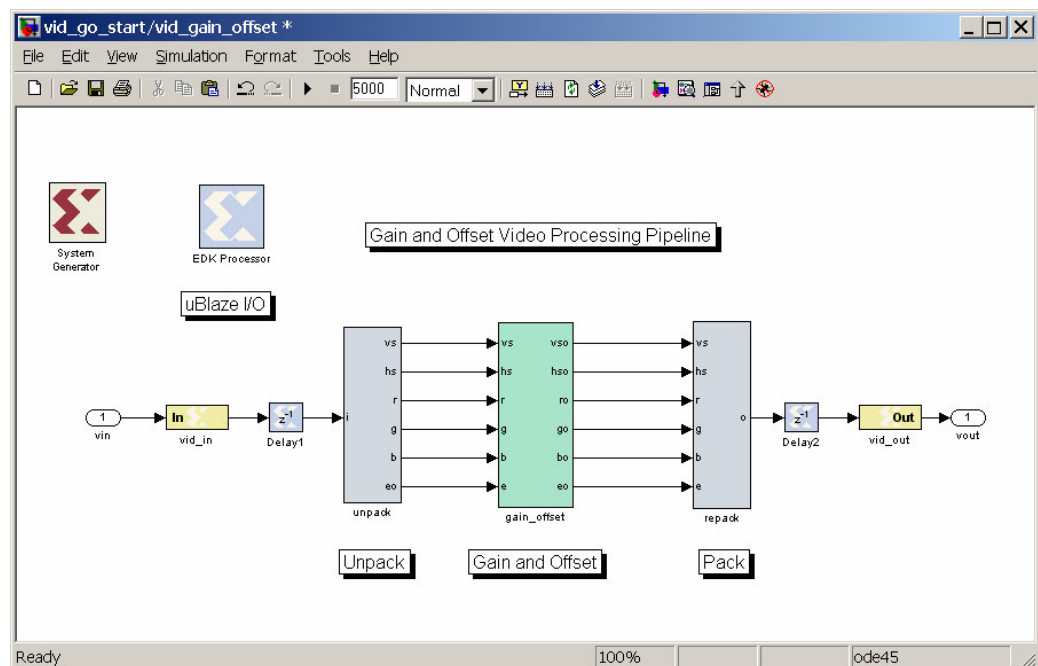
Figure 6-4: **Gain and Offset Processing Pcore**

The model contains a number of blocks and subsystems that are used to generate and define the pcore.

◆ System Generator block – used to generate the pcore.

◆ EDK Processor block – generates the interface to the MicroBlaze using FSL interface. For more details on the EDK flow, refer to Chapter 3, "EDK Integration."

◆ Input and output gateways – supply simulation vectors and are connected to video streams

♦ Pack and unpack subsystem – convert the VSK video stream format to RGB plus hsync, vsync and pix_enable.

♦ Delay block – serves to pipeline the video stream to improve FPGA timing.

5. Open the subsystem labeled `gain_offset` and then open the subsystem labeled `go_red`. The user will implement a gain and offset function in this subsystem and copy it to the green and blue channel.

6. Open the Library Browser (**View->Library Browser**) and select the Xilinx blockset. Find a multiplier and adder, and convert blocks and drag them into the `go_red` subsystem. Now add a couple of `from registers` to the diagram. These allow the MicroBlaze to set the offset and gain values by writing to these registers.

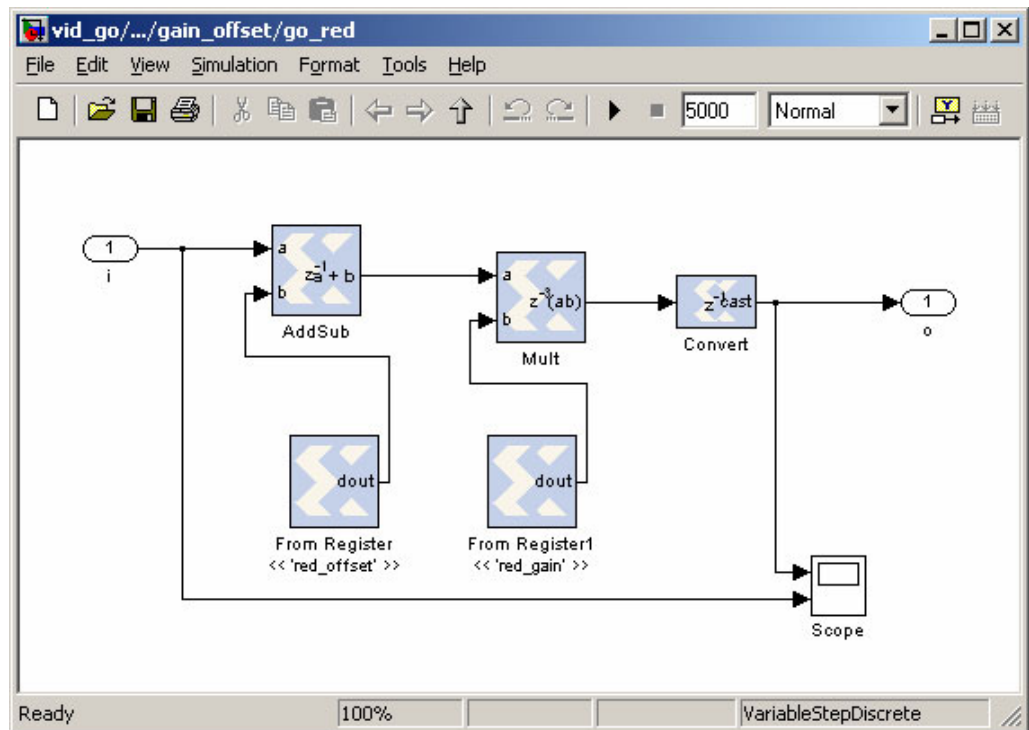7. Add a scope and connect the blocks as shown in Figure 6-5.

Figure 6-5: **Connecting the Blocks**

8. Configure the blocks as follows:

♦ Set the output type of the convert block to 10-bits unsigned, set round to round-even, and set overflow to saturate. Set the latency to 1 to add a pipeline register. Set binary point to 0.

♦ Set the name of the offset register to `red_offset`, and the output type to 16-bit signed, with a binary point at bit 0. Set the initial value to 0.

♦ Set the name of the gain register to `red_gain`, and the output type to 16-bit signed, with a binary point at bit 12. Set the initial value to 1.

♦ Set the adder latency to 1 to add a pipeline register.

9. Open the EDK processor block in the next to the top-level subsystem. Configure the EDK interface by clicking on the add button followed by **Apply** to generate the processor interface to the `from registers`.
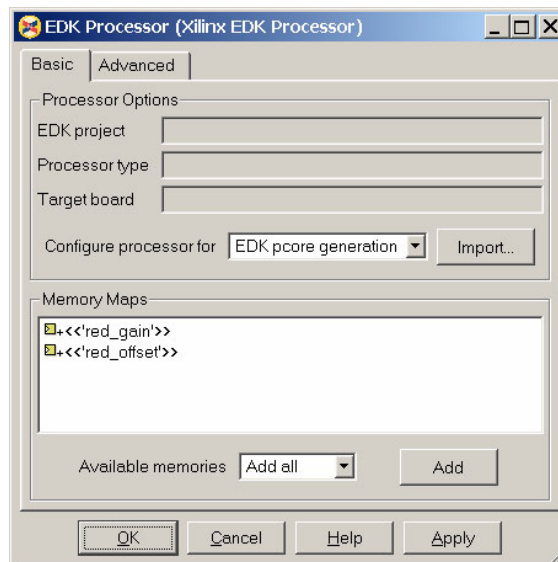
Figure 6-6: **EDK Processor Configuration**

Double click on the red gain in the EDK processor menu to set the initial gain to 1. Press **Play** in the Simulink model and open the output scope. and observe the input and output signals on the scope. If the design is correct, the output is seen as a delayed version of the input. To see the effect of the gain and offset circuit, the user can alter the initial values of the gain and offset register.

Note: The user must leave the EDK processor menu open for new values to take effect. This is because a new memory map is created when the menu is closed which resets the gain to the initial value.

10. Now create the green and blue channel by copying the go_red subsystem. Change the names in the from registers to green_gain, green_offset, blue_gain, and blue_offset. Click each gain or offset register to set the initial values to 1 and 0. Add the new registers to the processor interface as follows:

   a. Close the EDK processor block and reopen it.

   b. Right click in the memory map pane and select **Delete_All** followed by **Apply**.

   c. Select **Add All** followed by Add and Apply. A pop-up generating the memory map should now be seen.

   d. Close the EDK processor menu and right click on the EDK processor block and select **Look Under Mask** to see the generated memory map.

   e. Run the design and check the output for correctness. All three outputs at the top level scope can be seen.

11. Set the delays for hsync, vsync, and pixel_enable in the gain_offset subsystem to match the latency of the gain_offset blocks. See Figure 6-7.
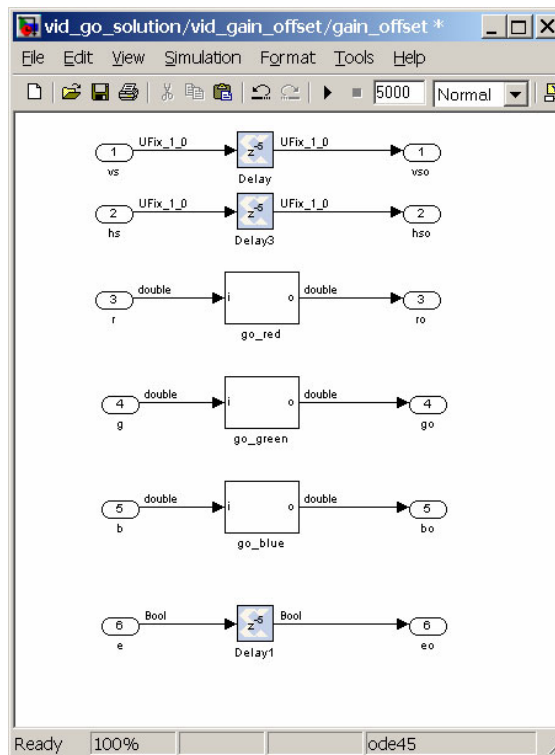
12. Save the design as vid_go.mdl.

Figure 6-7: **Design Saved as `vid_go.mdl`**

Testing the Video Function in System Generator

Run the `vid_go.mdl` design at the top level to verify the expected results. If a gold model is available, it can be compared to the System Generator results at the top level.

Generating the Pcore

To generate the pcore, generate an HDL netlist for the pcore using the pcore-export output flow.

1. Before generating the pcore, create a new EDK project named `edk_vid_go` by copying the VSK diagnostics EDK project named `edk_vid_go_start` to a new directory.

    Copy the `%VSK\Examples\SysgenTutorial\VSK_pcore_tutorial\` `edk_vid_go_start` directory to a new directory. Rename the directory to `edk_vid_go`. An EDK project containing the finished tutorial is also available as `edk_vid_go_solution`.

2. Open the completed model `vid_go.mdl` and open the `vid_gain_offset` subsystem.

3. Open the EDK processor block and make sure **EDK Pcore Generation** is selected. Close the EDK Processor block.

4. Open the System Generator block. Select **Export as a pcore to EDK** as a compilation target. Select **Virtex-4 V4SX35-10ff668** as a Part and select a **10** ns clock period.

5. Under `Compilation Settings`, select the pcore version and select the EDK project directory that contains the `edk_vid_go` EDK project. The EDK pcore will be created and moved to the pcore directory in the EDK project.
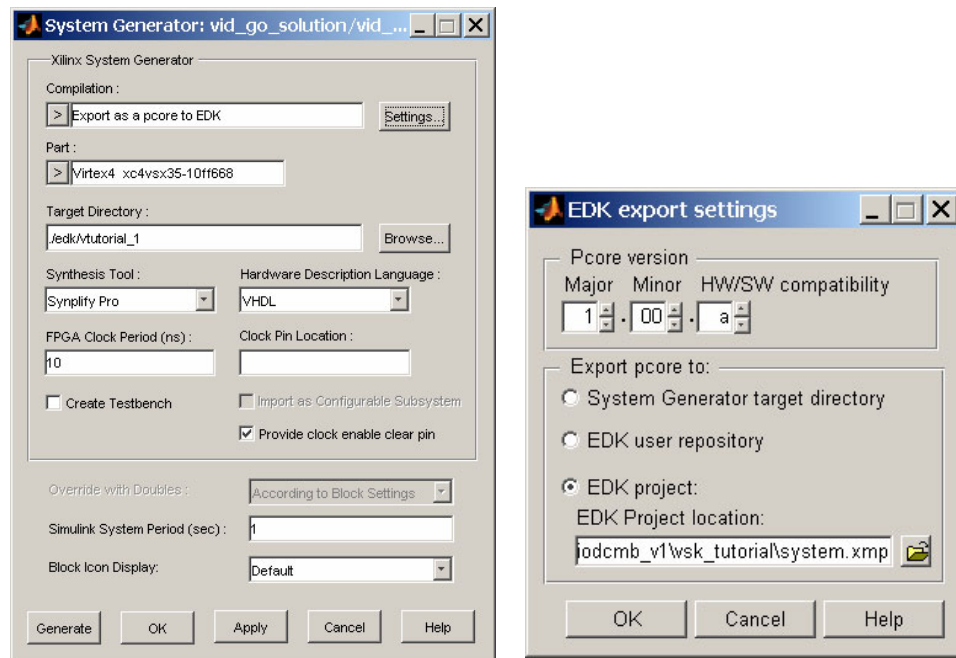

Figure 6-8: **Generating the Pcore**

6. To generate a pcore, a target directory needs to be specified. This is the directory the pcore will be created in. After generation is complete, the pcore will be copied to the EDK project. Select a target directory. Click on **Generate** to generate the pcore.

7. Save and close the model after generation completes.

Importing the Pcore into an EDK Project

The pcore is now imported into the EDK project using the **Configure Coprocessor** option. This option connects the pcore to the MicroBlaze processor using FSL ports. The input and output video streams need to be connected manually using the Add/Edit port map GUI.

1. Open Xilinx Platform Studio.

2. Open the `%VSK\Examples\SysgenTutorial\VSK_pcore_tutorial\ edk_vid_go\system.xmp` project.

3. Make sure the pcore is recognized by running **Project->Rescan User Repositories**.

4. Open **Hardware->Configure Coprocessor**.

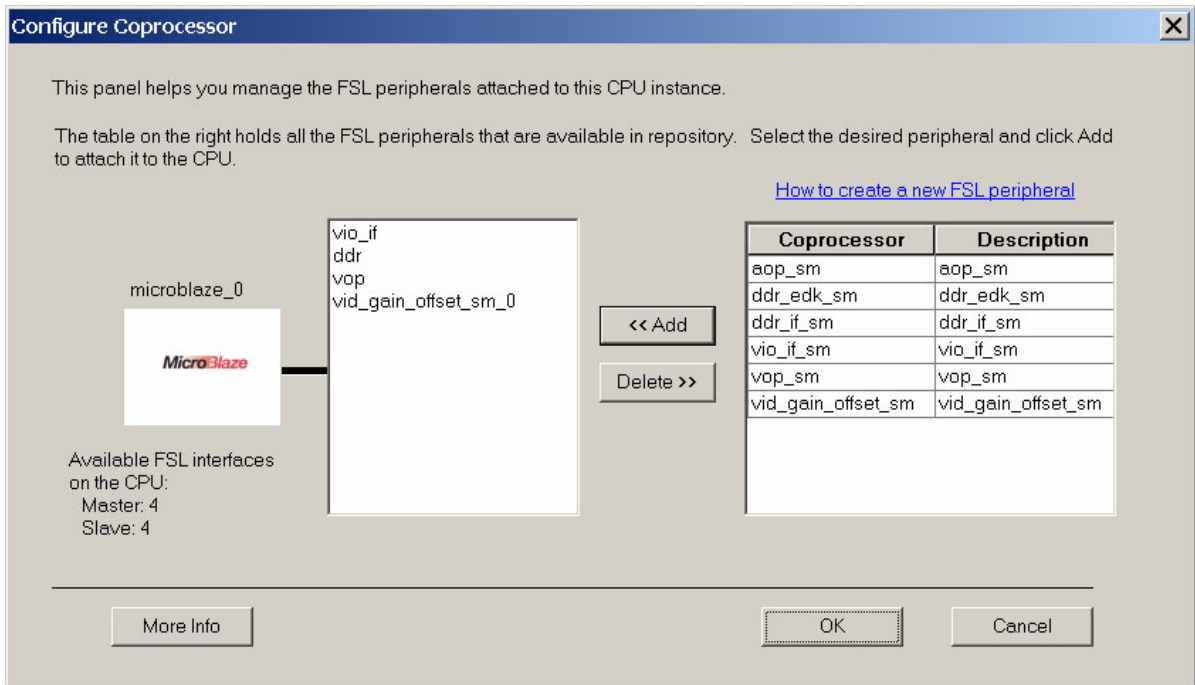5. Select `vid_gain_offset_sm` and add the pcore to the project as shown in Figure 6-9.

Figure 6-9: **Configure Coprocessor Panel**

After selecting OK, the user is directed to the add/edit cores menu to wire up the unconnected ports.

6. In the top-level System Assembly pane, select the **Ports** filter and menu; change the instance name of the imported pcore by clicking on the pcore name and renaming the block to `vid_gain_offset`. The software libraries will be generated with this name.

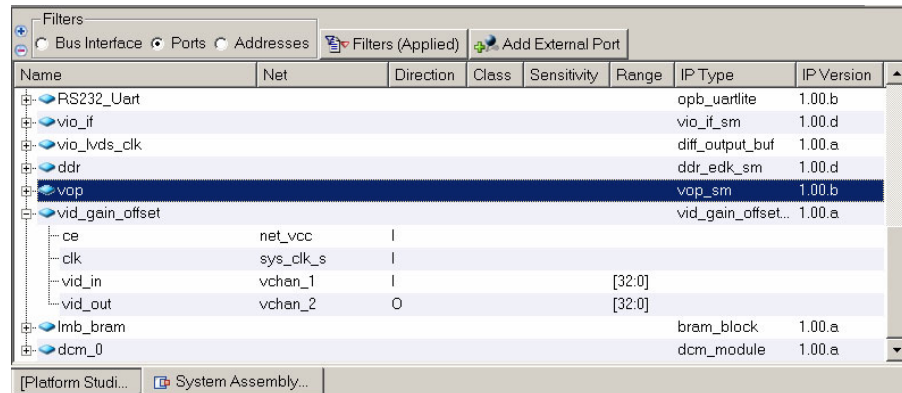*Figure 6-10:* **System Menu Showing New Imported Pcore**

7. Expand the `vid_gain_offset` pcore to show the port signal names by clicking on the [+] icon.

8. Select the port name. Select the Net column of the ce port and modify the net name to `net_vcc`.

9. Now wire the 33-bit video data buses between the pcores. The `vid_gain_offset` pcore will be inserted in the video pipeline after the `vop` pcore.

a. Select the `vid_in` port and modify the net connection to `vchan_1`.

b. Select the `vid_out` port and modify the net connection to `vchan_2`

c. Open the `vio_if` pcore port list and modify the `vin` net connection to `vchan_2`.

d. Open the `ddr` pcore port list and modify the `vchan_1` net connection to `vchan_3`.

   Note: The `ddr` pcore is not actually used in the data path of this tutorial. It is included so that this design can be used as a starting point for more complex designs that might require the `ddr` pcore.

10. After configuring the pcore port names, the video buses between the pcores should be wired as shown in Figure 6-11.
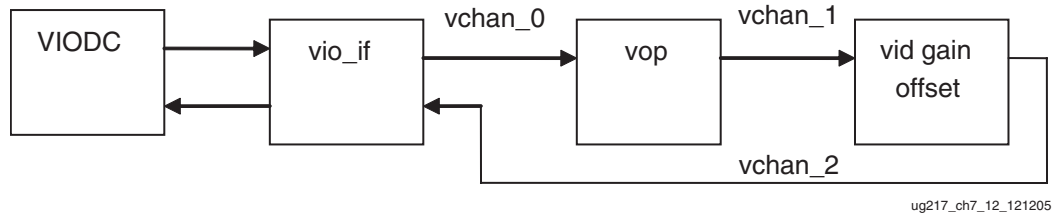

ug217_ch7_12_121205

Figure 6-11:   **Pcore Wiring with vid_gain_offset Pcore Inserted into Video Pipeline**

11. Select **Hardware->Generate Netlist** to generate a new netlist and check that the pcore import is error free.

12. Select **Hardware->Generate Bitstream** to generate the bitstream

These last steps take 10-20 minutes to the new hardware implementation.

Importing the Pcore Software Drivers

Next, the user needs to create the software drivers to load the gain and offset registers. Base I/O functions are already created when the pcore was created and can be found in the file `<vid_gain_offset.h>`, but the base IO functions will be wrapped with another C-function to load the offset and gain.

The completed C functions are found in the file named `vsk_vid_gain_offset.c`. This file includes the function below which sets the video gain according to a channel number:

```
void set_vid_gain(int chan, int gain)
{
 if(chan==0)
 {
 vid_gain_offset_Write(VID_GAIN_OFFSET_RED_GAIN,
  VID_GAIN_OFFSET_RED_GAIN_DIN, gain);
 }
 if(chan==1)
 {
 vid_gain_offset_Write(VID_GAIN_OFFSET_GREEN_GAIN,
  VID_GAIN_OFFSET_GREEN_GAIN_DIN, gain);
 }
 if(chan==2)
 {
```

```
vid_gain_offset_Write(VID_GAIN_OFFSET_BLUE_GAIN,VID_GAIN_OFFSET
_BLUE_GAIN_DIN, gain);
 }
}
```

1. Copy the files `vsk_vid_gain_offset.c` and `vsk_turorial_top.c` into the `%VSK\Examples\SysgenTutorial\VSK_pcore_tutorial\edk_vid_go\VSK_diagnostics\src` directory.

2. Under the application pane, select **VSK_Diagnostics/Sources** and right click. Select **Add Existing Files**.

3. Add the selected file, `vsk_vid_gain_offset.c`.

4. Add `vsk_tutorial_top.c` to `VSK_Diagnostics/Sources`.

5. Right click on the file `vsk_top.c`. Select **Remove**.

6. Select **Device Configuration->Update Bitstream** to compile the software.

Controlling the Pcore from a Demo Menu

To demonstrate the vid gain and offset function, the user can construct a simple means of controlling the gain and offset from a *keystroke* menu over an RS-232 port. The imported `.c` files contain a menu function named `main_vid_offset_gain` for the purpose of controlling the gain and offset.

The vsk_vid_offset_gain menu can be called from a top-level menu in the `vsk_tutorial_top.c` file. From the top-level menu, call the gain offset menu using the 'o' key.

Running the Tutorial with Live Video

To run the tutorial with live video, the user also needs to load the VIODC bit file. To do so:

1. Open a command window in Windows XP and type `impact`. This brings up the iMPACT GUI (Figure 6-12).

2. Click the Boundary Scan mode in the iMPACT modes pane and the select **File->initialize Chain** to scan the JTAG chain, and select the XCV2P7 device. Select **Cancel** when asked to select a configuration file.

3. Right click on the XCV2P7 icon, and select **Assign a New Configuration File**. Select the file `viodc_wrapper_11b.bit` in the tutorial directory and load it to the FPGA using `program` using **Operations -> Program**.
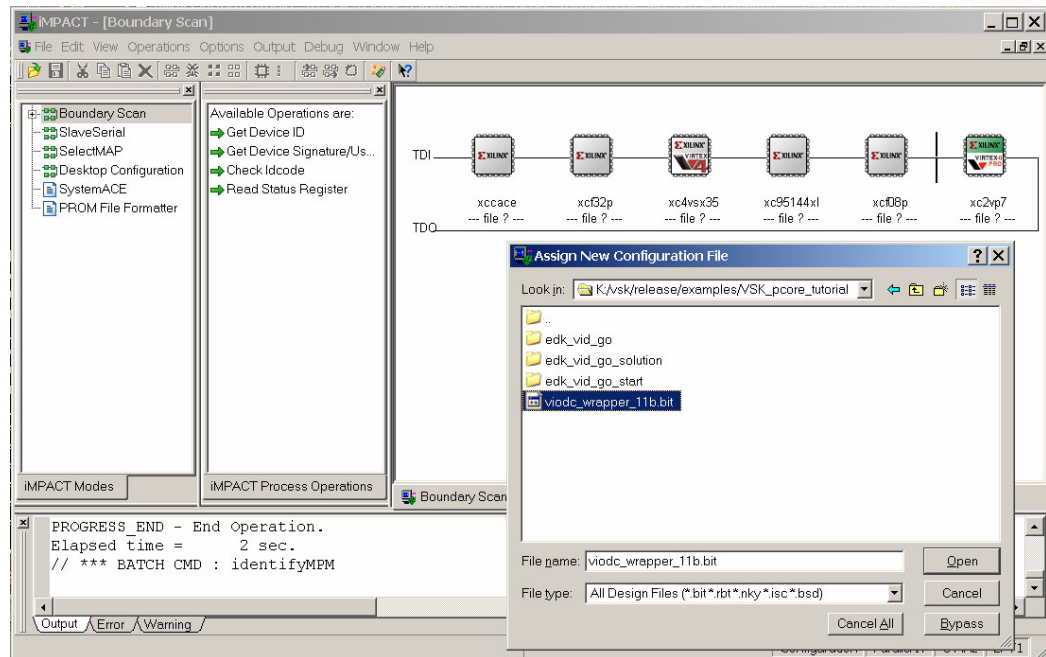
Figure 6-12: **iMPACT Window**

To run the tutorial:

4. Open the project in Xilinx Platform Studio. To load the project in Xilinx Platform Studio, use **File -> Open** and select .../`vsk_tutorials/edk_vid_go/system.xmp`.

5. Connect the Parallel JTAG cable to the JTAG port (FPGA & CPU Debug) on the bottom board of the VSK and the PC.

6. Using EDK (**Tools -> Download**), download the EDK project to the XC4VSX35 device. Alternately, load the `download.bit` file from the EDK project to the XC4VSX35 using iMPACT.

7. Connect an RS-232 terminal to the VSK serial port and configure the terminal program for 115,200 Baud, 8-N-1, no flow control.

8. If it has compiled and loaded correctly, it should print the top-level menu on screen.

9. Select **1** to configure the VSK to use the RGB camera input.

10. Select **O** to enter the `vid_gain_offset` menu.

11. Control the gain and offset using the '[ ]' and '{}' keys.

12. Select a color channel and mode using the 'r', 'g', 'b' keys.

13. **ESC** to quit.

Compiling the VIODC FPGA Design

This chapter describes how to compile the System Generator `vsk_viodc_xxx.mdl` design to a bitstream (*xxx* is the version number). The chapter covers the following:

- Tutorial overview
- Overview of VIODC design compilation process
- Incrementing the VIODC version ID
- Generating the design using the multiple subsystem generator
- Using ISE Project Navigator to add a VHDL wrapper
- Loading the VIODC design to the XCV2P7 FPGA on the VIODC board
- Verifying the operation of the VIODC

Tutorial Overview

This tutorial is intended to illustrate the process of compiling the VIODC FPGA design using System Generator and Xilinx ISE. Source files for this tutorial are available on the CDROM under the Examples directory:

`Examples/vsk_diagnostics/viodc`

Overview of VIODC Design Compilation Process

The VIODC board includes a Xilinx XCV2P7 FPGA to interface to the various video interfaces. The VIODC FPGA design uses seven independent clock domains to interface to the various video interface devices. Sysgen designs using multiple clocks require the use of the Multiple Subsystem Generator (MSG) block to generate an HDL design. The HDL is then wrapped with a top-level VHDL design and associated with a UCF user constraint file. The wrapper is then compiled using ISE Project Navigator. After a bitfile is obtained, it is loaded to the board using iMPACT software. Optionally, it can be compiled into a System Ace Flash image and loaded automatically.

VIODC Design Components

- `vsk_viodc_xxx.mdl` – The System Generator VIODC design source file. This file also requires two additional files `xl_bufg.vhd` and `xl_bufg_config.m` which are required to support the BUFG UNISIM primitive.
- `viodc_sgl_xxx.vhd` – The VHDL wrapper design. This design wraps the System Generator design. It is required to add various LVDS and 3-state buffers to the design. The '`_sgl_`' qualifier denotes that this is a design for the VIODC which uses the single-ended version of the VIOBUS to communicate with the ML402 FPGA platform.

- `viodc_sgl_11c.ucf` – The .ucf constraint file associated with the top-level FPGA design

Incrementing the VIODC Version ID

The VIODC design includes a register which is used as a version ID. In this design, the version is simply incremented from 0x11c to 0x11d.

1. Open the `vsk_viodc_xxx.mdl` design and save a new copy and increment the version in the name. Example: copy `vsk_viodc_11c.mdl` and save as `vsk_viodc_11d.mdl`.

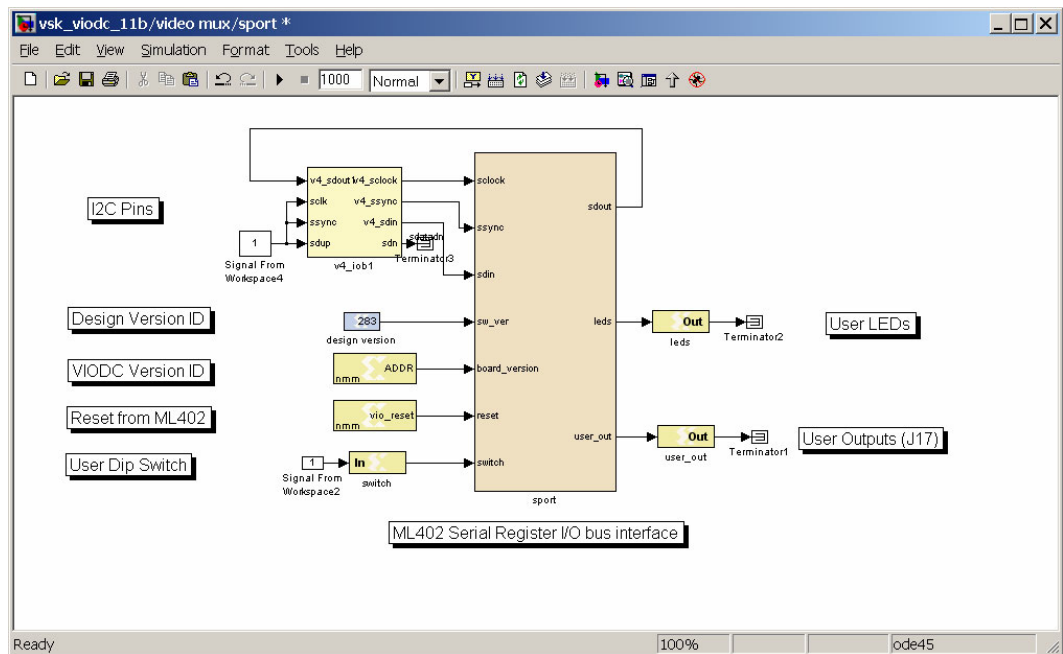2. From the top level, open the `video_mux/sport` (Figure 7-1).

Figure 7-1: **VIODC Serial Register I/O Block**

3. Open the constant block `design_version`.

4. Increment the version ID. For example, change hex **011c** to **011d.**

5. Save the file.

Generating the Design Using the Multiple Subsystem Generator

To generate the design, use the Multiple Subsystem Generator (MSG) block. When the design is generated, SysGen will generate each subsystem individually, followed by generating a top-level wrapper for the entire SysGen design.

1. Open the top-level design `vsk_viodc_xxx.mdl` (Figure 7-2) and open the Multiple Subsystem Generator block (Figure 7-3).
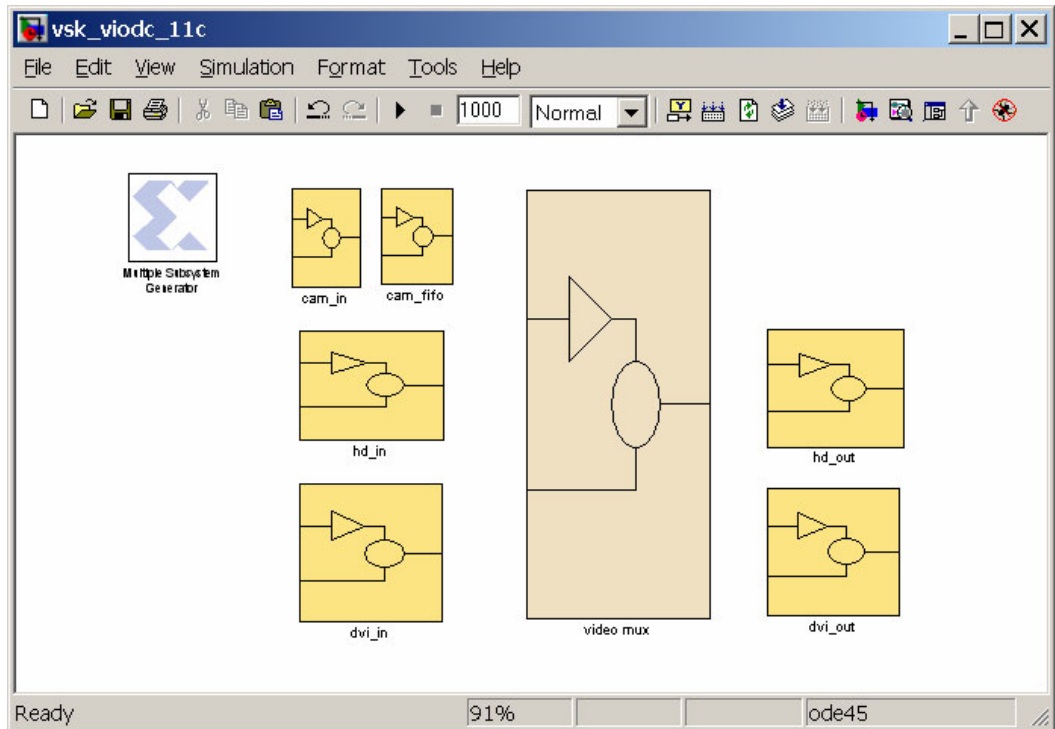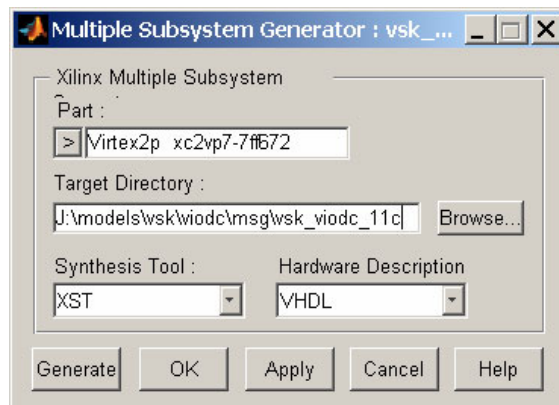
Figure 7-2: `vsk_viodc_xxx.mdl` **Top Level**

Figure 7-3: **MSG Generate Block**

2. Change the Target Directory to `.\msg\vsk_viodc_xxx`. For example, `.\msg\vsk_viodc_11d`

3. Select the **xcv2p7-7ff672** as a part.

4. Click **Generate**. The design will generate to the specified directory. This process will take a few minutes.

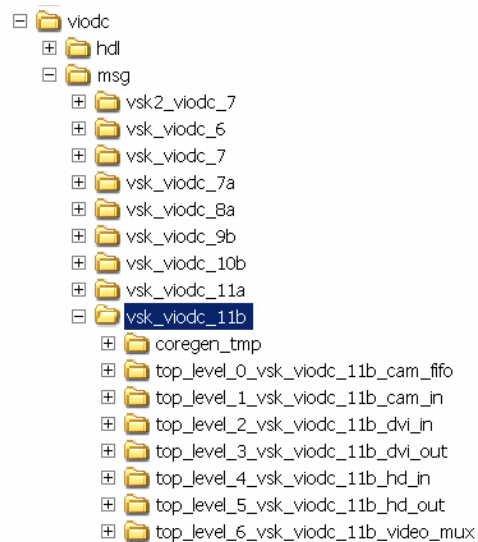5. The resulting directory will look similar to the following directory structure: (Figure 7-4).

Figure 7-4: **Directory Structure Generated by Multiple Subsystem Generator**

Using ISE Project Navigator to Add a VHDL Wrapper

Xilinx Project Navigator is used to create the final `viodc_design`.

1. Copy the included wrapper file `viodc_sgl_xxx.vhd` into the `./msg/vsk_viodc_xxx` directory and increment the `.vhd` name to match the new revision number. For example, copy the included wrapper file `viodc_sgl_11c.vhd` to `msg/vsk_viodc_11d/viodc_sgl_11d.vhd`.

2. Copy the included UCF file `viodc_sgl_xxx.ucf` into the `./msg/vsk_viodc_xxx` directory and increment the `.ucf` name to match the new revision number. For example, copy the included constraints file `viodc_sgl_11c.ucf` to `msg/vsk_viodc_11d/viodc_sgl_11d.ucf`.

3. Edit `viodc_sgl_xxx.vhd` wrapper to increment the entity and architecture names of the design (e.g., from `11c` to `11d`), and also increment the component declaration and instance names (`vsk_viodc_11c` to `11d`). Basically, do a find/replace (replace `11c` with `11d`).

4. Open the project (for ISE 8.2, this would be the `vsk_viodc_xxx.ise` file) **Project Add Source** `vsk_viodc_xxx.vhd` and `vsk_viodc_xxx.ucf`. The top level should now show `viodc_wrapper_xxx` as the top entity name.

5. Select that entity (already done in ISE 8.2) and double-click on **Generate Programming File**. This will take a few minutes. The project tree will look similar to Figure 7-5.

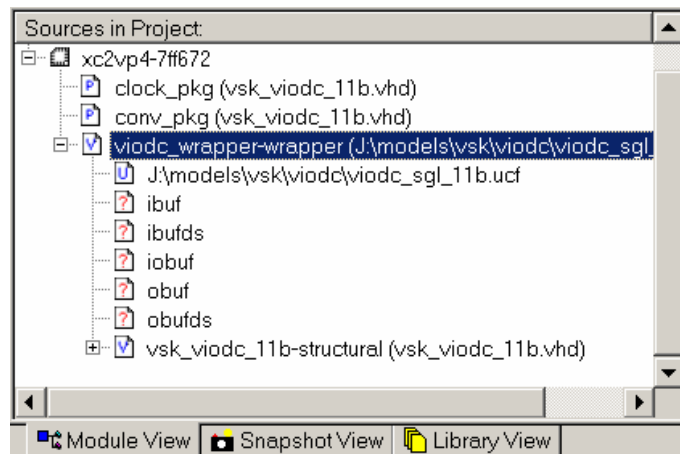6. **Generate** the bitfile from the ISE Tools menu.

Figure 7-5: **Project Navigator Source File View**

Loading the VIODC Design to the XCV2P7 FPGA on the VIODC Board

The modified VIODC design can now be loaded to the VSK.

1. Load the VSK diagnostics from the Compact Flash (or the EDK project). This will load the XC4VSX35 device on the ML402 board with the diagnostics design (`download.bit`), which is capable of reading the VIODC's ID register. The XC4VSX35 device needs to be programmed to communicate to the registers in the XC2VP7 device with the serial link.

2. Open the Xilinx iMPACT program either in Project Navigator or from a command line.

3. Connect a JTAG parallel cable IV to the VSK ML402 JTAG port (labeled *FPGA & CPU Debug port*). Power up the VSK.

4. Scan the JTAG chain and select the XCV2P7 device.

5. Assign the bitfile named `vsk_viodc_xxx.bit` to the XCV2P7 device.

6. Load the `VSK_diagnostics` program.

7. Load the bitfile to the XCV2P7 device.

Verifying the VIODC Operation

First connect a terminal for the UART. The terminal will communicate with the UART connected to the MicroBlaze running the VSK diagnostic program in the XC4VSX35 FPGA.

1. 1Connect a PC to the ML402 RS-232 serial port using a null-modem cable.

2. Set up a terminal program to for 115,200 baud, 8-bit, no parity, and no flow control.

Now see if the new version number can be read by the VIODC. The terminal program should already be running. Press '1' to initialize the camera. Pressing '?' will being up a help menu.

3. In the top-level menu, type 'v' to enter the VIO diagnostics menu.

4. Using the VIO diagnostics menu, read the version register from the VIODC board.

   a. Use the '-', '=' key pair to select the VIODC registers.

b. Use the '[',']' key pair to select address `0x10`, which points to the VIODC version register.

c. Read the VIODC version register using the ',' key. It should read `0x11d`.

d. Alternately, use the 'd' key to display all the register values in the VIODC.

Modifying the VSK Diagnostic Software EDK Project

The `vio.c` also needs to be modified because the VSK diagnostic software checks for the expected version of the VIODC in the `vio_if_init()` routine. If it detects an incorrect value, the following message appears when booting:

```
---------------------------------------
-- Video Starter Kit - press ? for help
---------------------------------------
Error Incorrect VIODC version found
Expected =0x11C, found 0x11D.
```

To fix this problem, change the `#define VIODC_VERSION` in the `vio.c` file to the expected value. When the EDK program is recompiled and run, the expected version will now match the value read from the VIODC.
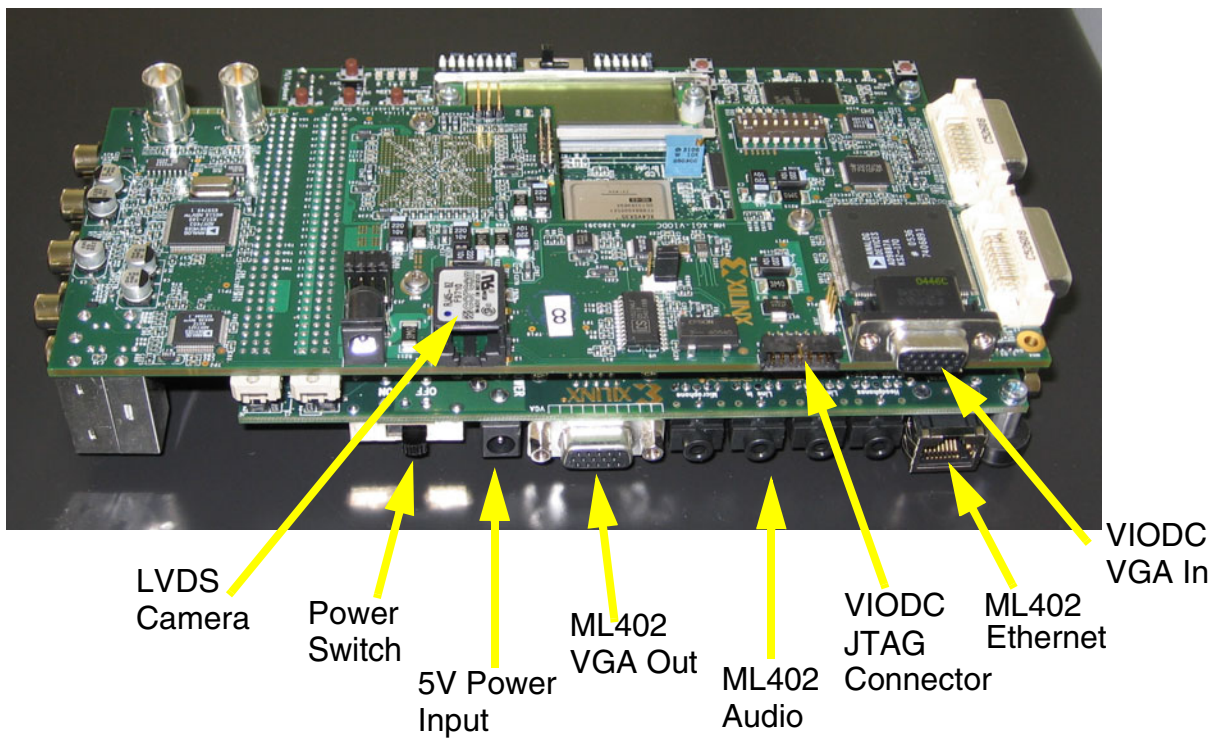
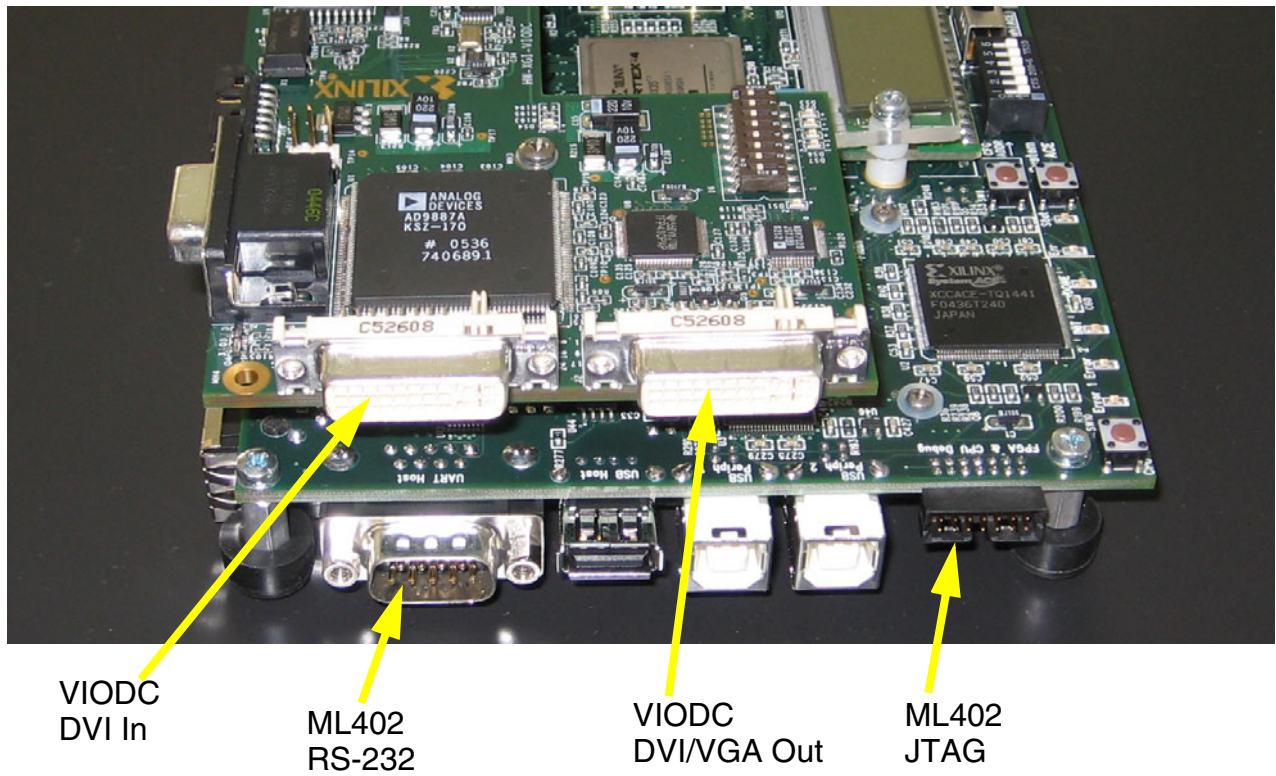# VSK I/O Connector Location Pictures

VIODC Connectors

LVDS
Camera

Power
Switch

5V Power
Input

ML402
VGA Out

ML402
Audio

VIODC
JTAG
Connector

ML402
Ethernet

VIODC
VGA In

Figure A-1:   **VIODC Rear View**

VIODC
DVI In

ML402
RS-232

VIODC
DVI/VGA Out

ML402
JTAG

Figure A-2: **VIODC Left Side View**
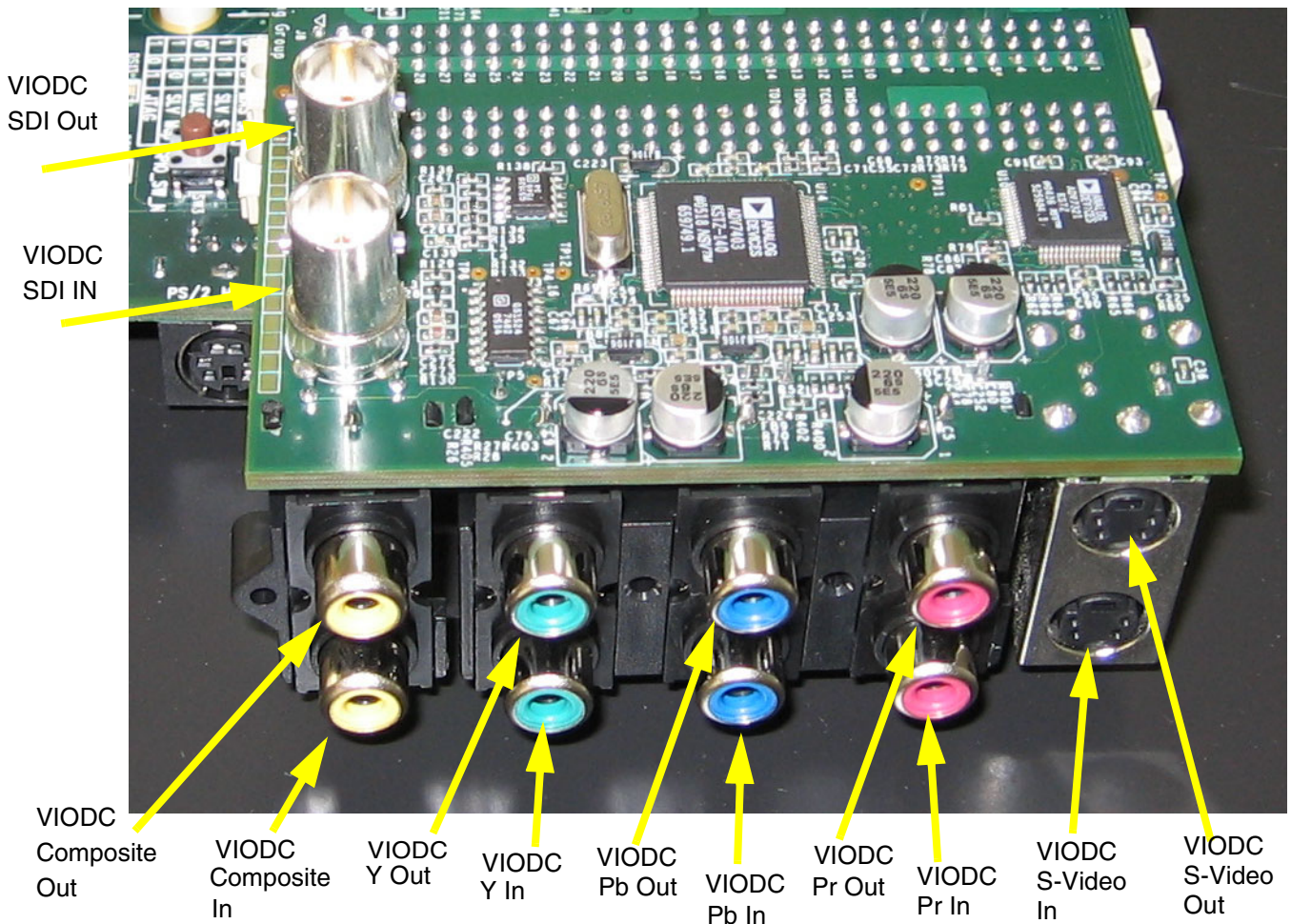
Figure A-3: **VIODC Right Side View**

LVDS Camera

Figure A-4: **LVDS Camera**

ML402 Board
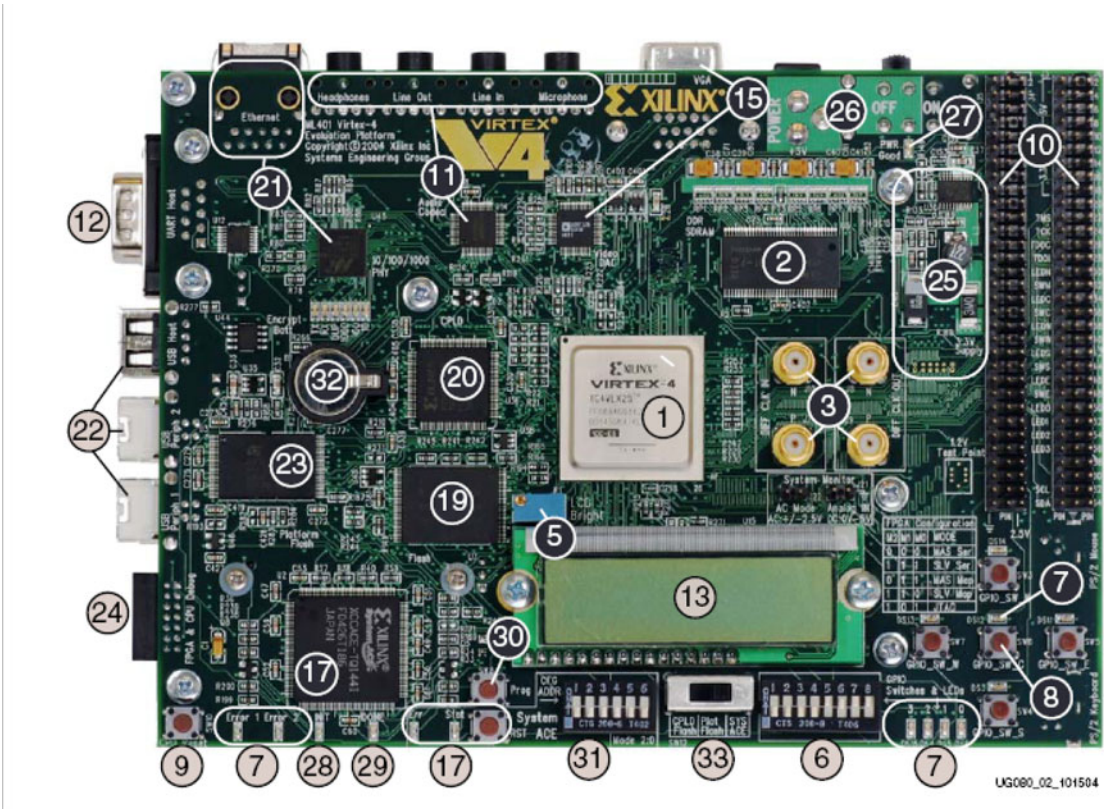
Figure A-5:   **ML402 Board**
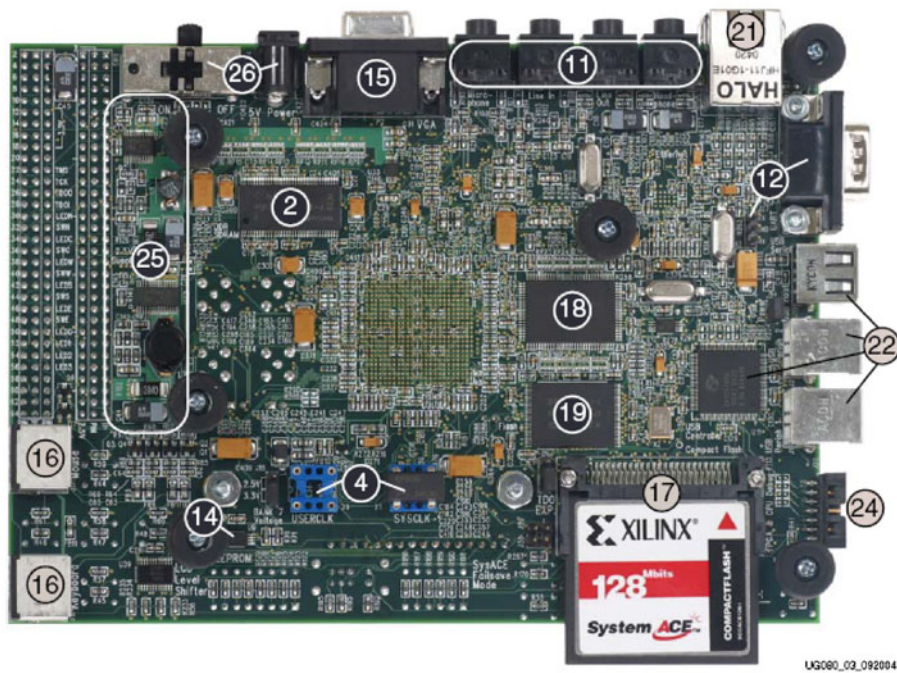
Figure A-6: **ML402 Evaluation Platform**