Vivado Design Suite User Guide

Embedded Processor Hardware Design

UG898 (v2015.3) September 30, 2015

XILINX
ALL PROGRAMMABLE™

Revision History

The following table shows the revision history for this document.

Date	Version	Revision
09/30/2015	2015.3	Added information related to different MEM file formats supported by Vivado synthesis and UpdateMEM in Memory (MEM) Files in Chapter 6.
04/01/2015	2015.1	Updated Chapter 4, Designing with the MIG Core to include UltraScale™ architecture. Update Figure 2-4, page 10 to show the new IP Catalog Details window. Added information about the Platform Board Flow in IP Integrator in Chapter 2. Added details related to Debug Options under MicroBlaze Configuration Wizard Debug Page in Chapter 3.

Table of Contents

Introduction

Overview

This chapter provides an introduction to using the Xilinx® Vivado® Design Suite flow for programming an embedded design using the Zynq®-7000 All Programmable (AP) SoC device or the MicroBlaze™ processor.

Embedded systems are complex. Hardware and software portions of an embedded design are projects in themselves. Merging the two design components so that they function as one system creates additional challenges. Add an FPGA design project, and the situation can become very complicated.

To simplify the design process, Xilinx offers several sets of tools. It is a good idea to know the basic tool names, project file names, and acronyms for these tools.

The Vivado Integrated Design Environment (IDE) includes the IP integrator tool, which you can use to *stitch* together a processor-based design. This tool, combined with the Xilinx Software Development Kit (SDK), provides an integrated environment to design and debug microprocessor-based systems and embedded software applications.

Hardware and Software Tool Flow Overview

The Vivado tools provide specific flows for programming, based on the processor. The Vivado IDE uses the IP integrator with graphic connectivity screens to specify the device, select peripherals, and configure hardware settings.

The Zynq-7000 AP SoC uses the Vivado IP integrator to capture hardware platform information in XML format applications, along with other data files. These are used in software design tools to create and configure Board Support Package (BSP) libraries, infer compiler options, program the PL, define JTAG settings, and automate other operations that require information about the hardware. The Zynq-7000 SoC solution reduces the complexity of an embedded design by offering an ARM Cortex A9 dual core as an embedded block, and programmable logic along with it, on a single SoC.

Xilinx provides the following design tools for developing and debugging software applications for Zynq-7000 AP SoC and MicroBlaze processor devices:

- Software IDE
- GNU-based compiler tool-chain
- JTAG debugger

These tools let you develop both bare-metal applications that do not require an operating system, and applications for the open-source Linux operating system. The Vivado IP integrator captures information about the Processing System (PS) and peripherals, including configuration settings, register memory map, and associated logic in the Processing Logic (PL) fabric. Bitstream can then be generated for PL initialization.

Software solutions are also available from third-party sources that support Cortex-A9 processors, including but not limited to:

- Software IDEs
- Compiler tool-chains
- Debug and trace tools
- Embedded OS and software libraries
- Simulators
- Models and virtual prototyping tools

Third-party tool solutions vary in the level of integration and direct support for Zynq-7000 devices.

See the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 1] for more information about the SDK and programming for Zynq devices. The SDK is a standalone product, and is available for download from www.xilinx.com.

Figure 1-1 illustrates the tools flow for embedded hardware.

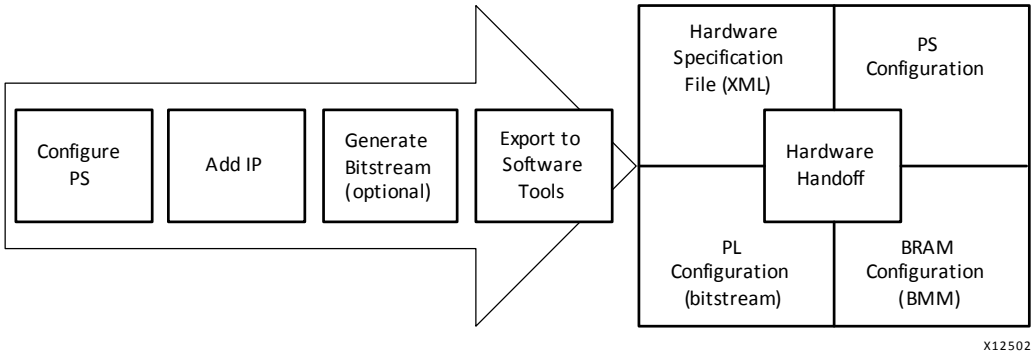


Figure 1-1: **Hardware Design Tool Handoff to Software Tools**

Completing an Embedded Processor Design

To complete an embedded processor design, do the following steps:

1. Create a new Vivado Design Suite project.
2. Create a block design in the IP Integrator tool and instantiate the Zynq Processing System 7 IP core, or a MicroBlaze processor, along with any other Xilinx IP or your custom IP.
3. Create a top-level wrapper and instantiate the block design into a top-level RTL design.
4. Run the top-level design through synthesis and implementation and export the hardware to SDK.
5. Create your software application, in SDK, associate the Executable Linkable File (ELF) file with the hardware design.
6. Use UpdateMem to merge the ELF and Memory Map Information (MMI) for the Block Rams with the hardware device bitstream.
7. Program into the target board.

Chapter 2

Using a Zynq-7000 Processor in an Embedded Design

Introduction

This chapter describes how to use the Xilinx® Vivado® Design Suite flow for using the Zynq®-7000 All Programmable (AP) SoC device.

The examples target the Xilinx ZC702 Rev 1.0 evaluation board and the tool versions in the 2013.2 Vivado Design Suite release.

IMPORTANT: *The Vivado IP integrator is the replacement for Xilinx Platform Studio (XPS) for embedded processor designs, including designs targeting Zynq devices and MicroBlaze™ processors. XPS only supports designs targeting MicroBlaze processors. Both IP integrator and XPS are available from the Vivado integrated design environment (IDE).*

Designing for Zynq-7000 Devices in the Vivado IDE

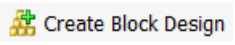
Designing for Zynq-7000 AP SoC devices is different using the Vivado IDE than in the ISE® Design Suite and Embedded Development Kit (EDK).

The Vivado IDE uses the IP integrator tool for embedded development. The IP integrator is a GUI-based interface that lets you stitch together complex IP subsystems.

A variety of IP are available in the Vivado IDE IP Catalog to accommodate complex designs.

You can also add custom IP to the IP Catalog. See the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 3] for more information.

Creating an IP Integrator Design with the Zynq-7000 Processor

Click the IP integrator **Create Block Design** button 

to open the Create Block Design dialog box, where you can enter the **Design Name** as shown in the figure below.

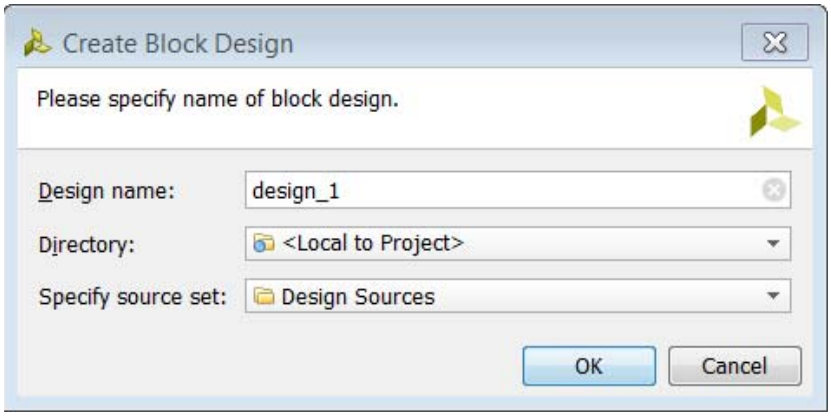


Figure 2-1: **Design Name Dialog Box**

The Block Design can be created as a part of a project, or it can be created in a different location that you can specify in the **Directory** field.

You can also Specify the source type by setting the field **Specify source set** from the pull-down menu. The Block Design window opens, as shown in Figure 2-2.

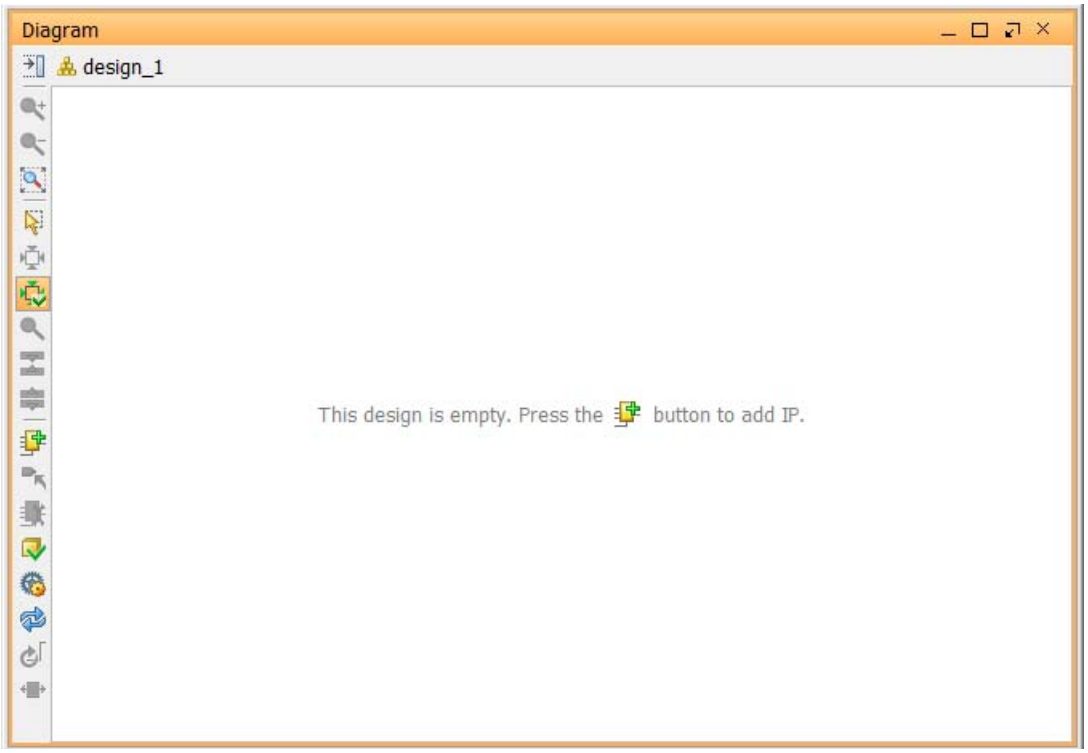


Figure 2-2: **Block Design Window**

1. In the empty block design canvas, you are prompted to **Add IP** from the IP Catalog (Figure 2-3). You can also right-click in the canvas and select **Add IP**.

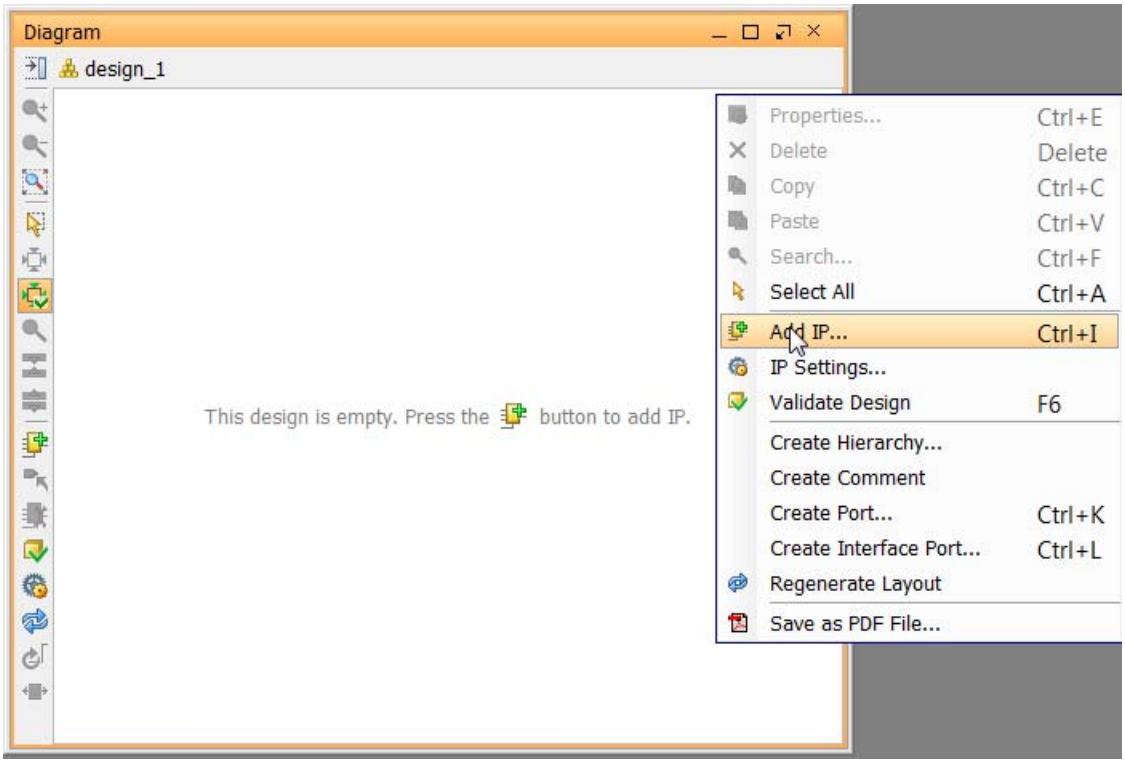


Figure 2-3: **Adding IP in the Block Design Canvas**

2. Select the **Add IP** option, and a Search box opens where you can search for, and select the **ZYNQ7 Processing System**, shown in Figure 2-4.

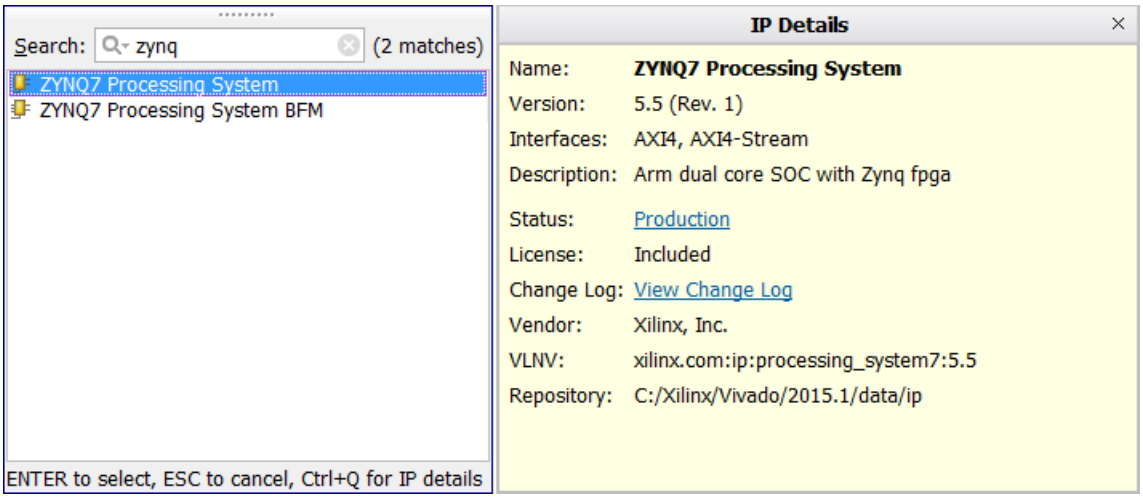


Figure 2-4: **Search for Zynq in the IP Catalog**

When you select the Zynq IP, the Vivado IP integrator adds the IP to the design, and a graphical representation of the processing system is displayed, as shown in Figure 2-5.

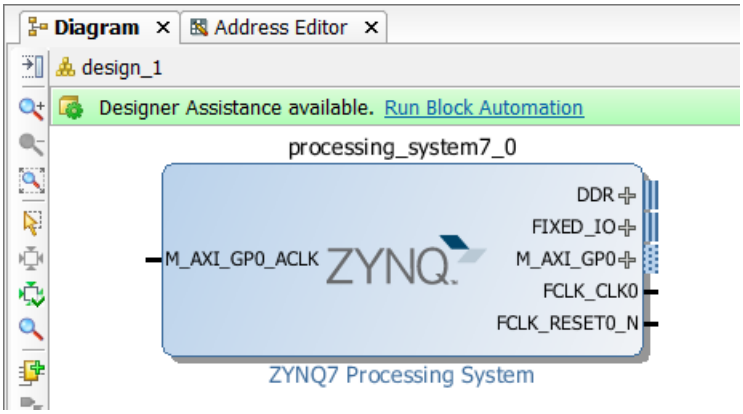


Figure 2-5: **Graphical Display of Default ZYNQ7 Processing System**

Tcl Command:

```
create_bd_cell -type ip -vlnv xilinx.com:ip:processing_system7:5.5
processing_system7_0
```

3. Double-click the processing system graphic to invoke the **Re-customize IP** process, which displays the Re-customize IP for the ZYNQ7 Processing System dialog box as shown in Figure 2-6.

4. Review the contents of the block design. The green colored blocks in the ZYNQ7 Processing System are configurable items. You can click a green block to open the coordinating configuration options.

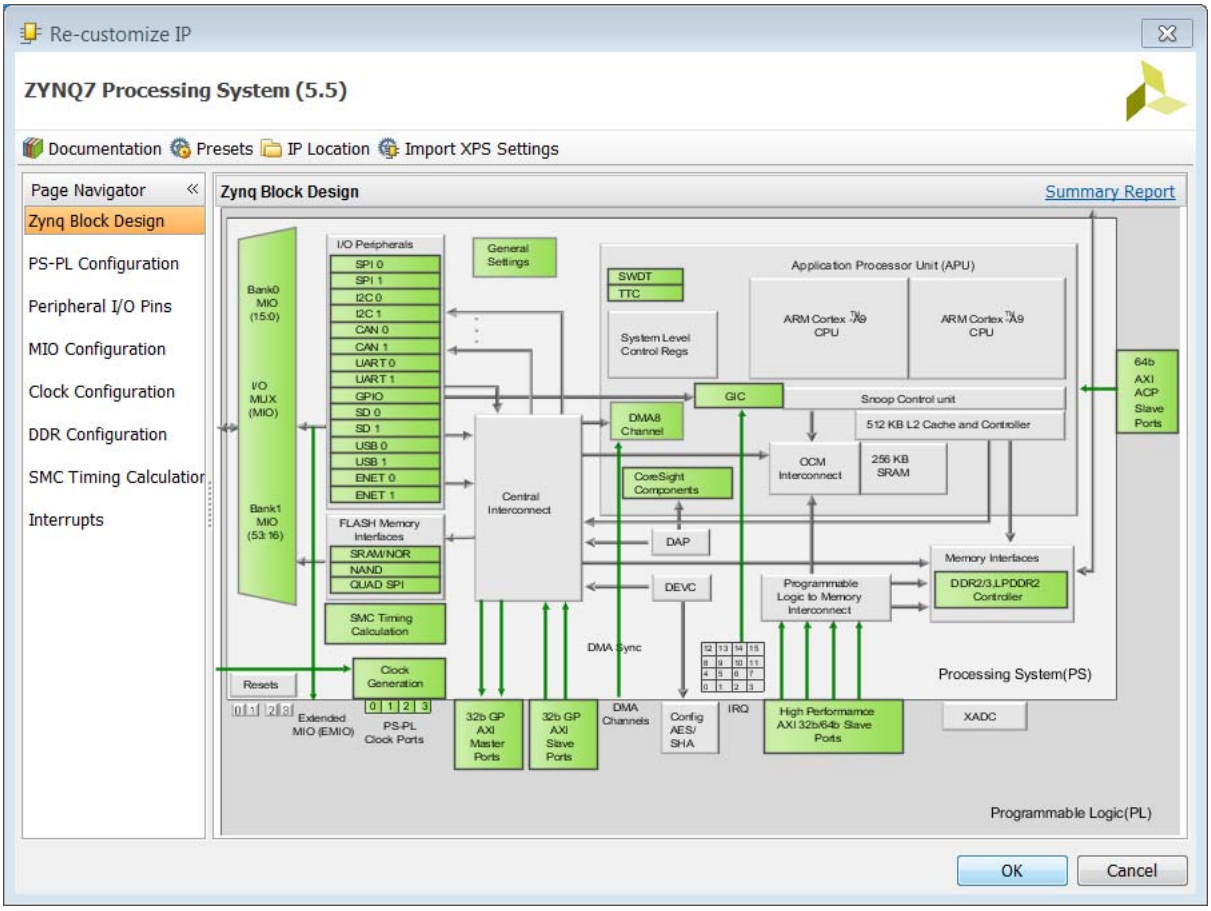


Figure 2-6: **ZYNQ7 Processing System Configuration Dialog Box**

Alternatively, you can select the options from the Page Navigator on the left, as shown in Figure 2-6.

Overview of the Zynq Block Design and Configuration Window

The *Zynq-7000 AP SoC Technical Reference Manual* (UG585) [Ref 4] provides details on the default options available in the Page Navigator. The following subsections describe in brief the Page Navigator selection options.

Processing System (PS)-Processing Logic (PL) Configuration Options

The PS-PL Configuration option tree displayed with the collapsed options as shown here.

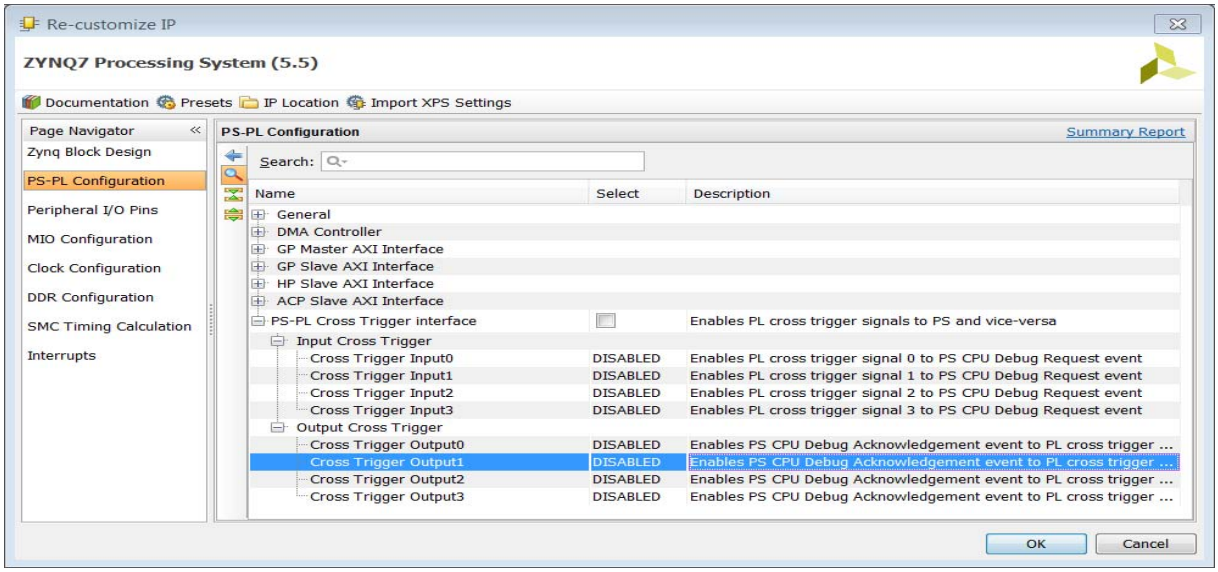


Figure 2-7: **PL-PS Configuration Pane**

Note the four buttons at the top of the window:

- **Documentation**: Click this button to open the documentation page of the Xilinx website, where you can find documentation pertaining to Zynq.
- **Presets**: Click this button to view information about the available preset options. User can save the current configuration of PS7 to a file or apply a pre-existing configuration to configure the current instance of the processors. Presets can also be applied to a target board. The available options are Default, ZC702, ZC706, and Zedboard as seen in Figure 2-8, page 14.

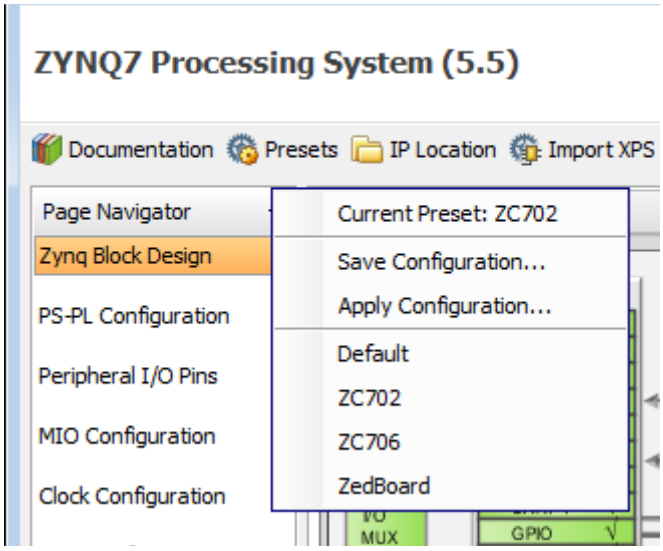


Figure 2-8: **Preset Options**

- **IP Location**: The IP can be created local to the project. However, you can also create IP at a remote location.

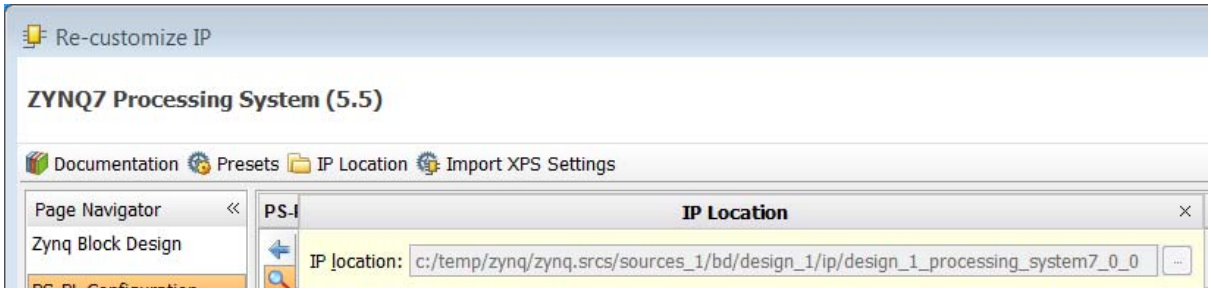


Figure 2-9: **Specify IP Location**

- **Import XPS Settings**: If you have an XML file describing the configuration of a Zynq processor from a XPS-based project, you can use this button to import that settings file to quickly configure the Zynq processor.

General Options

When you expand **General Options**, the following selections are available.

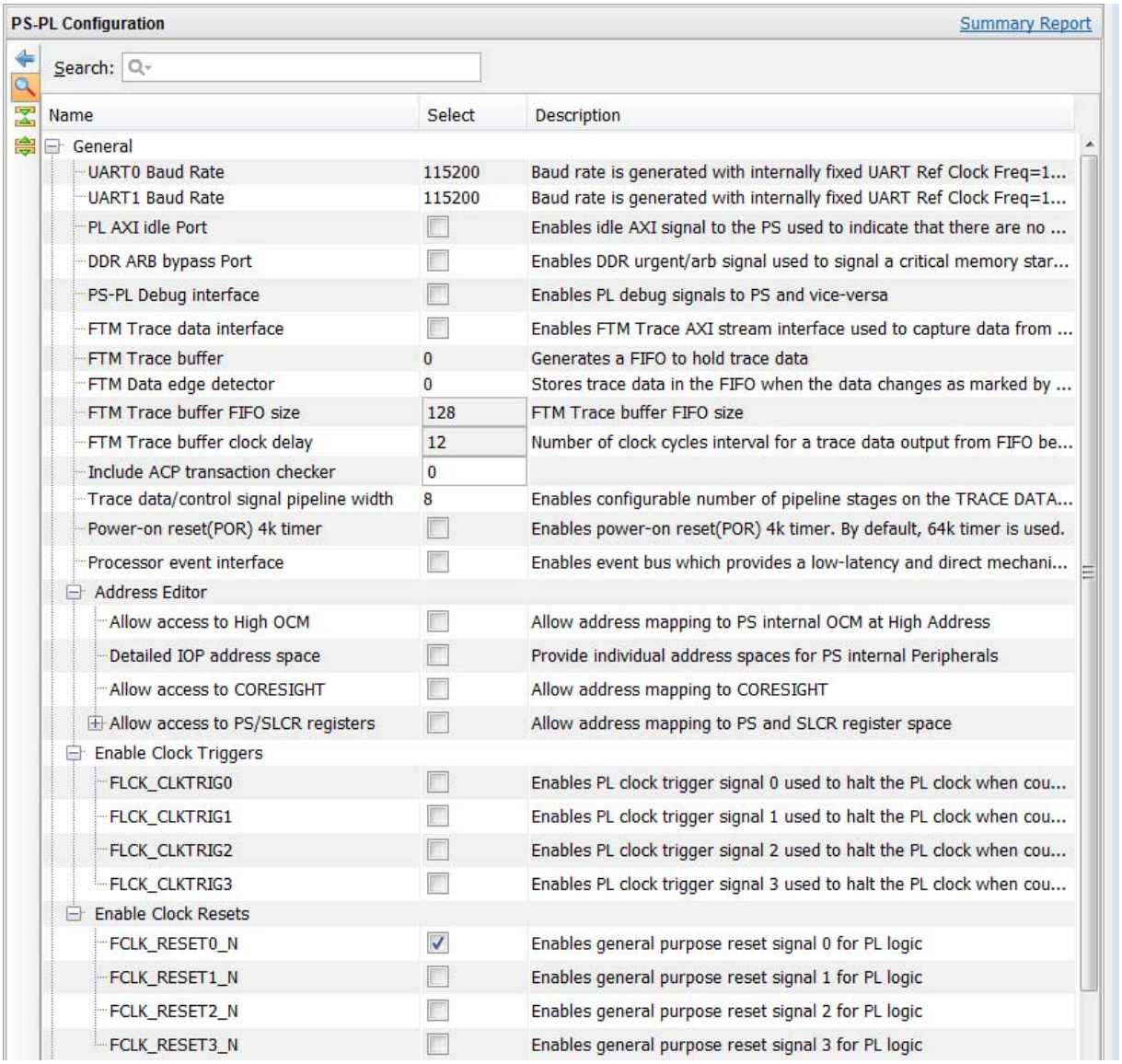


Figure 2-10: **General Options (First Tier)**

MIO and EMIO Configuration

From the Page Navigator, you can view and configure I/O pins by either clicking on the Peripheral I/O Pins option or MIO Configuration option.

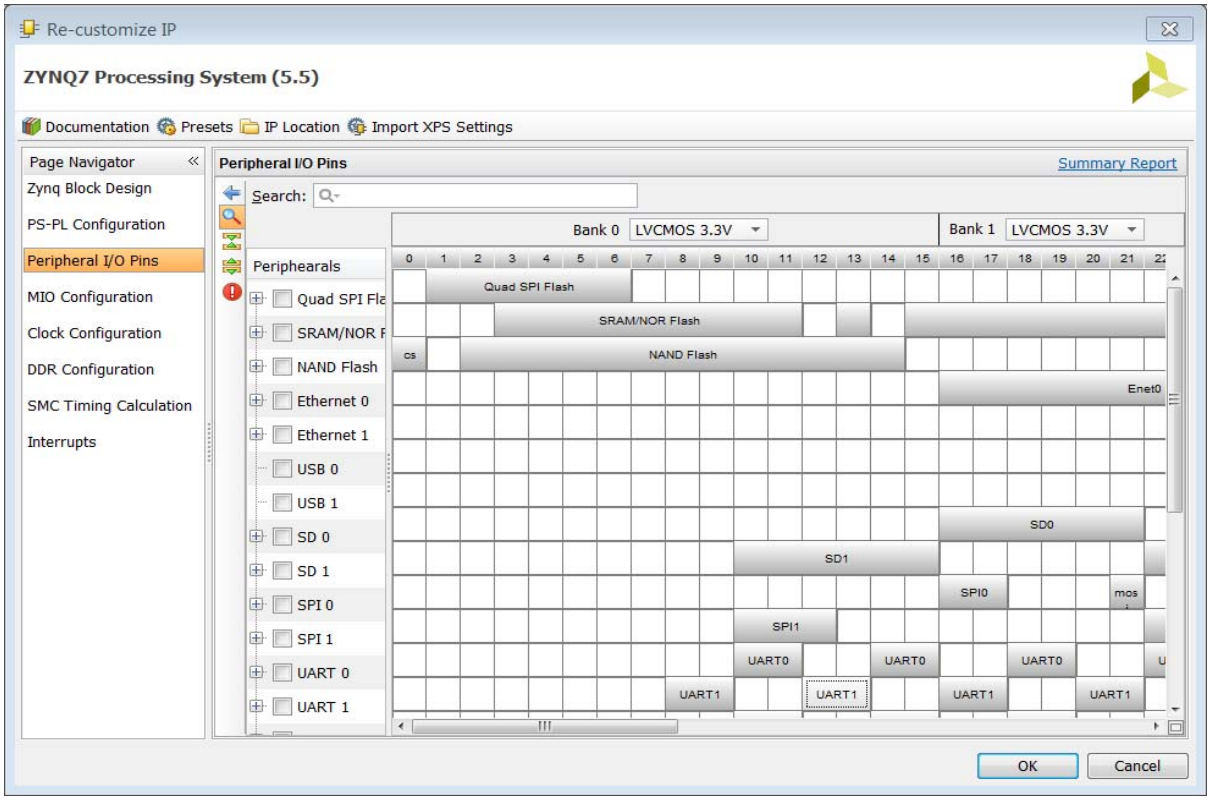


Figure 2-11: **Configuring Peripheral I/O Pins Using the Peripheral I/O Pins Menu**

The Zynq-7000 PS has over 20 peripherals available. You can route these peripherals directly to the dedicated Multiplexed I/Os (MIO) on the device, or through the Extended Multiplexed I/Os (EMIOs) routing to the fabric.

The configuration interface also lets you select I/O standards and slew settings for the MIO. The I/O peripheral block appears with a checkmark when you enable a peripheral. The block design depicts the status of enabled and disabled peripherals.

From the MIO Configuration option, you can do the same as shown in Figure 2-12.

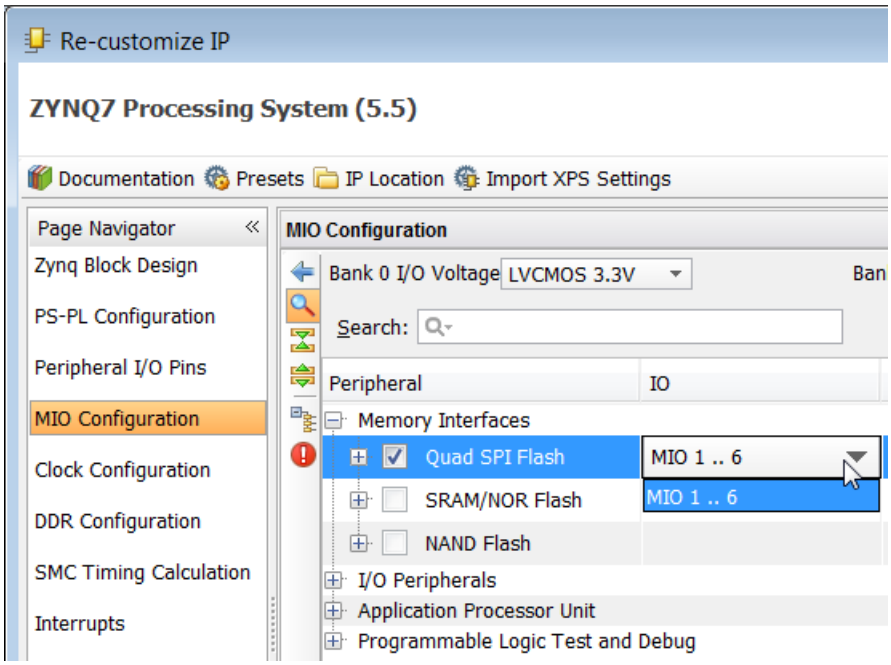


Figure 2-12: **Configuring Peripheral I/O Pins Using the MIO Configuration Menu**

Chapter 2, "Signals, Interfaces, and Pins" of the *Zynq-7000 AP SoC Technical Reference Manual* (UG585) [Ref 4] describes the MIOs and EMIOs for the 7z010 CLG225 device.

Pin Limitations

The 32 MIO pins available in the 7z010 CLG225 device restrict the functionality of the PS as follows:

- Either one USB or one Ethernet controller is available using MIO.
- Cannot boot from SDIO.
- No NOR/SRAM interfacing.
- The width of NAND Flash is limited to 8 bits.

Bank Settings

After you select peripherals, the individual I/O signals for the peripheral appear in the respective MIO locations. Use this section primarily for selecting I/O standards for the various peripherals. The PS MIO I/O buffers split into two voltage domains. Within each domain, each MIO is independently programmable.

There are two I/O voltage banks:

- Bank 0 consists of pins 0:15.
- Bank 1 consists of pins 16:53.

Each MIO pin is individually programmed for voltage signaling:

- 1.8 and 2.5/3.3 volts
- CMOS single-ended or HSTL differential receiver mode

IMPORTANT: *The entire bank must have the same voltage, but the pins can have different I/O standards.*

When you configure MIOs in the MIO Configuration dialog box on the Zynq tab, you can view a read-only image of the peripheral and respective MIO selections. The left side of the window lists the available peripherals. A checkmark on the peripheral indicates that a peripheral is selected.

Flash Memory Interfaces

Select one of the following in the configuration wizard:

- Quad-SPI Flash
- SRAM/NOR Flash
- NAND Flash

Quad-SPI Flash

Available options for Quad SPI Flash are shown in Figure 2-13.

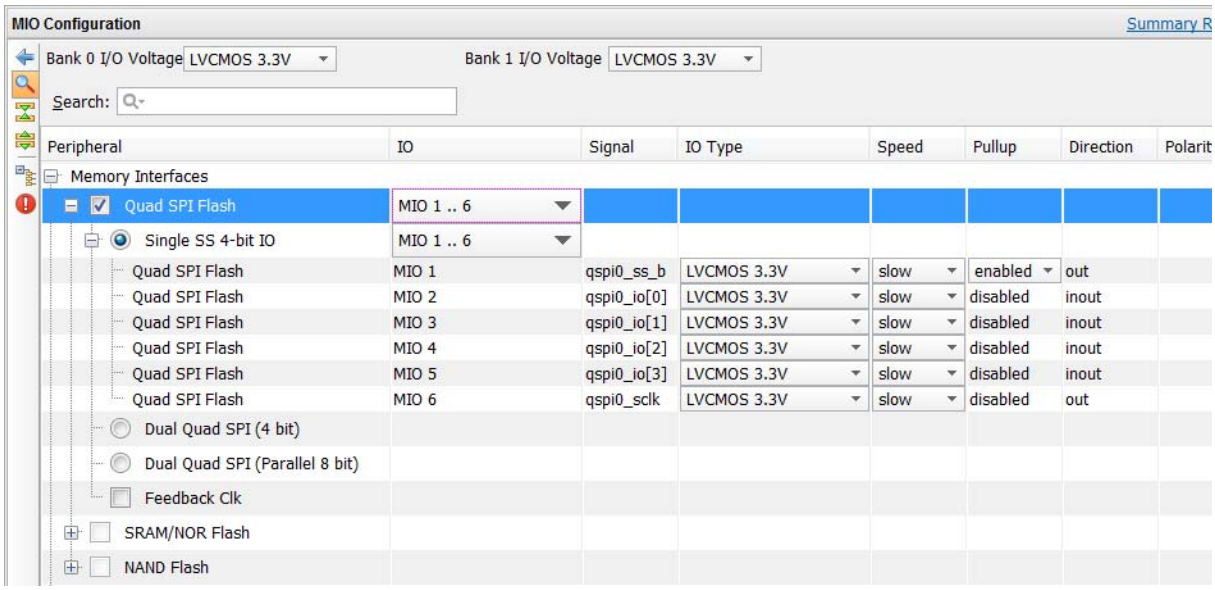


Figure 2-13: **Quad SPI Flash Options**

Key features of the linear Quad-SPI Flash controller are:

- Single or dual 1x and 2x read support
- 32-bit APB 3.0 interface for I/O mode that allows full device operations including program, read, and configuration
- 32-bit AXI linear address mapping interface for read operations
- Single chip select line support
- Write protection signal support
- Four-bit bidirectional I/O signals
- Read speeds of x1, x2, and x4
- Write speeds of x1 and x4
- 100 MHz maximum Quad-SPI clock at master mode
- 252-byte entry FIFO depth to improve Quad-SPI read efficiency
- Support for Quad-SPI device up to 128 Mb density
- Support for dual Quad-SPI with two Quad-SPI devices in parallel

Additionally, the linear address mapping mode features include:

- Regular read-only memory access through the AXI interface
- Up to two SPI flash memories
- Up to 16 MB addressing space for one memory and 32 MB for two memories
- AXI read acceptance capability of four
- Both AXI incrementing and wrapping-address burst read
- Automatically converts normal memory read operation to SPI protocol, and vice versa
- Serial, Dual, and Quad-SPI modes

SRAM/NOR Flash

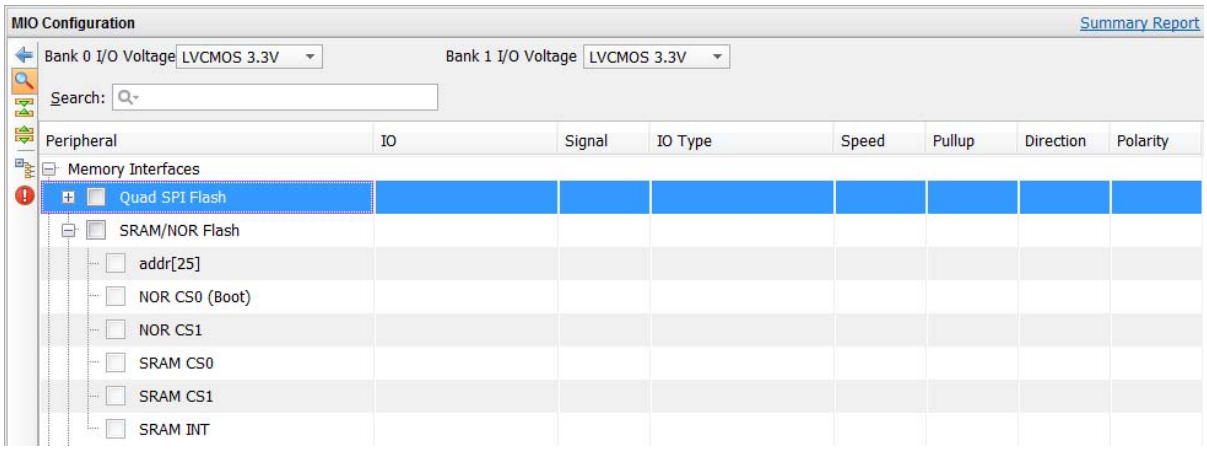


Figure 2-14: **SRAM/NOR Flash Configuration Options**

The SRAM/NOR controller has the following features:

- 8-bit data bus width
- One chip select with up to 26 address signals (64 MB)
- Two chip selects with up to 25 address signals (32 MB + 32 MB)
- 16-word read and 16-word write data FIFOs
- 8-word command FIFO
- Programmable I/O cycle timing on a per-chip select basis
- Asynchronous memory operating mode

NAND Flash

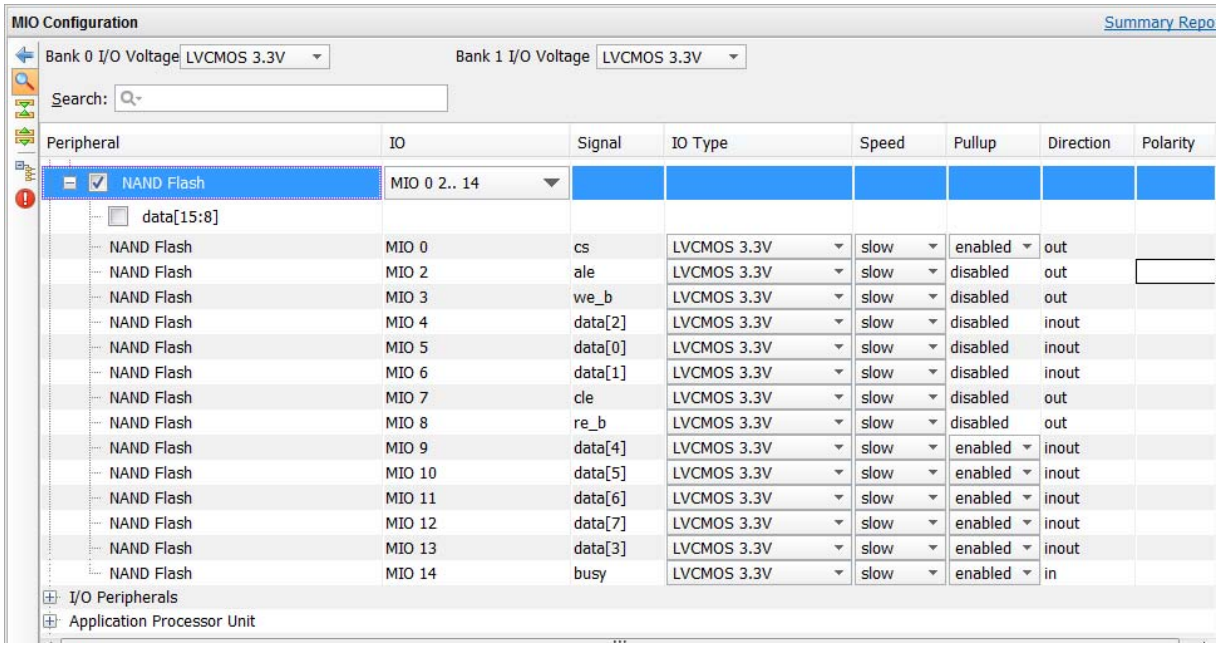


Figure 2-15: **NAND Controller Options**

The NAND controller has the following features:

- 8/16-bit I/O width with one chip select signal
- ONFI specification 1.0
- 16-word read and 16-word write data FIFOs
- 8-word command FIFO
- Programmable I/O cycle timing
- ECC assist
- Asynchronous memory operating mode

Clock Configuration

Figure 2-16: **Clock Configuration**

You can configure clocks in the Zynq-7000 device using one of the following methods:

- From the Page Navigator, click **Clock Configuration**.
- In the Zynq block design, click the **Clock Configuration** block.

Figure 2-17 shows the Clock Configuration page.

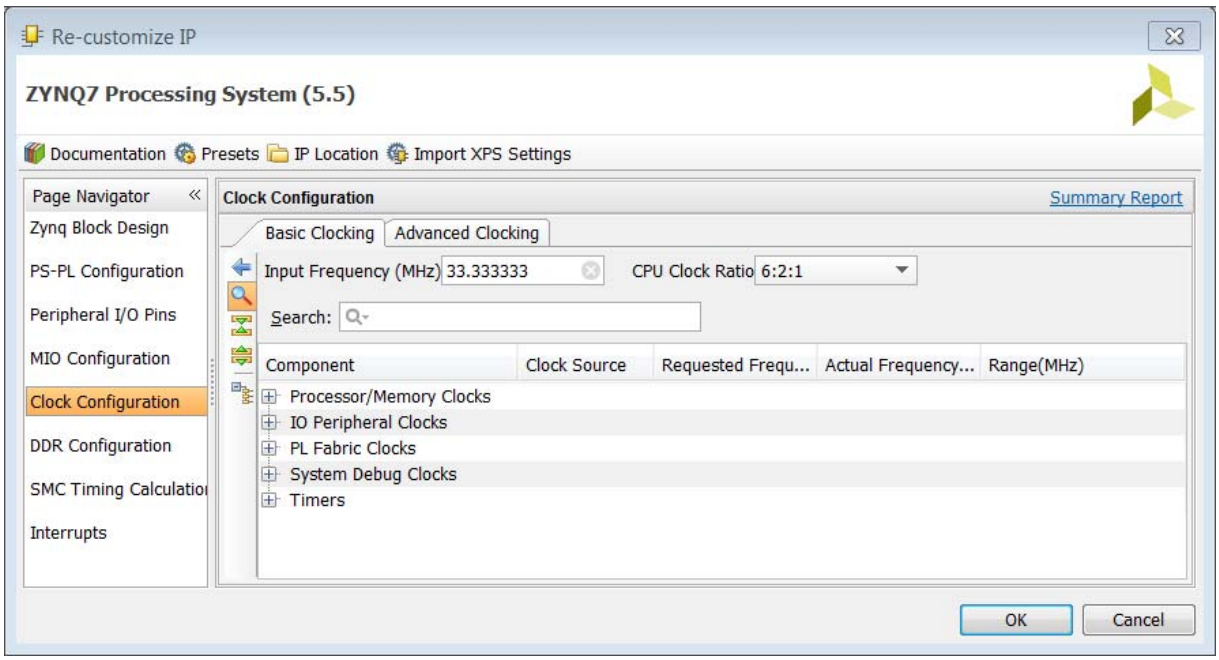


Figure 2-17: **Clock Configuration Page**

Figure 2-18 shows the Clock configuration page.

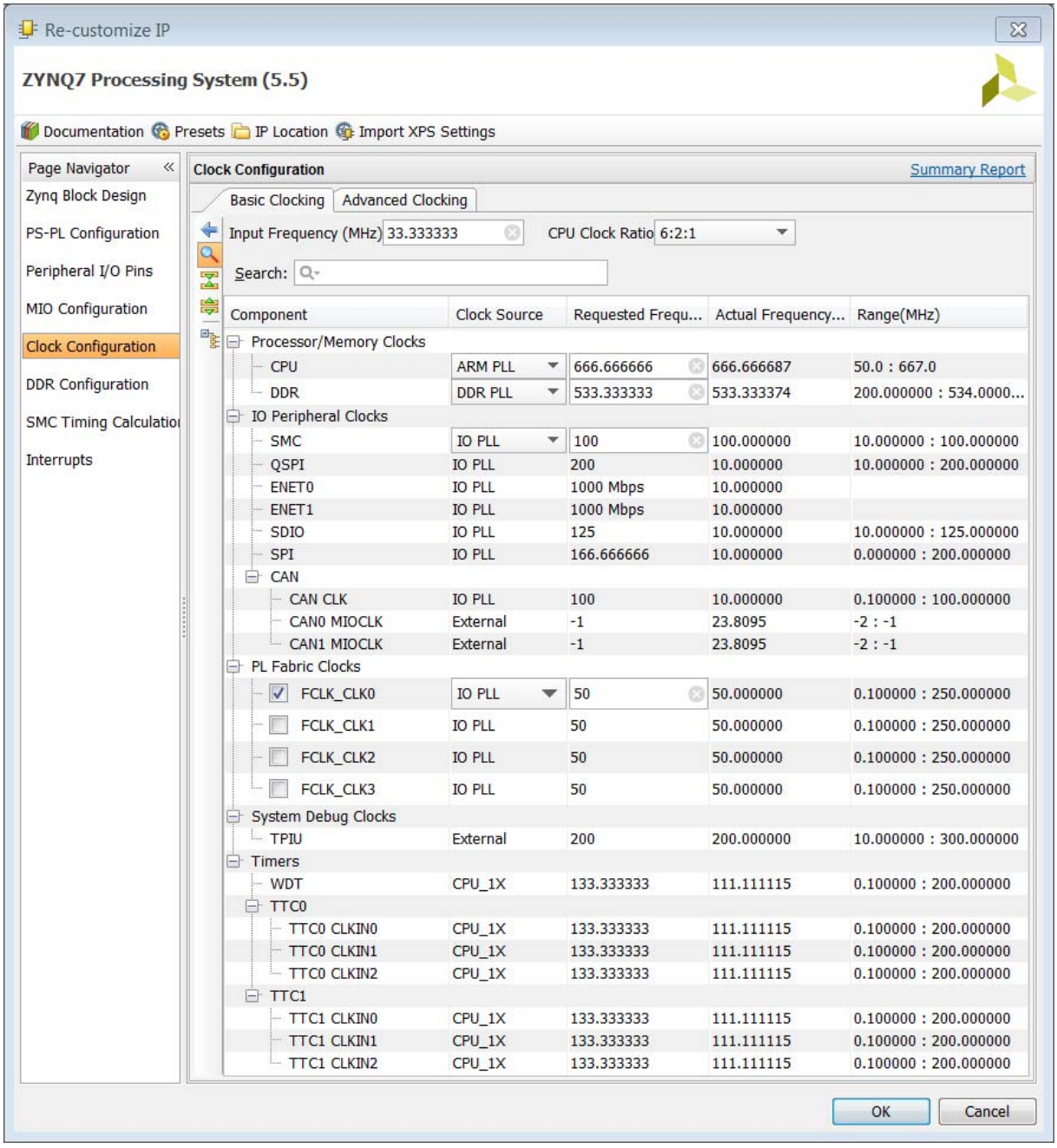


Figure 2-18: **Processor and Memory Clock Configurations Page**

The *Zynq-7000 AP SoC Technical Reference Manual* (UG585) [Ref 4] describes the clocking of the PS in detail. The Zynq clocking dialog box lets you set the peripheral clocks. The peripherals on the PS typically allow clock source selection from internal PLLs or from an external clock source. Most of the clocks can select the PLL to generate the clock.

Because the same PLL generates multiple frequencies, it might not be possible to get the exact frequency entered in the Requested Frequency (MHz) column. The achievable frequency is in the Actual Frequency (MHz) column.

Note: The frequency for a specific peripheral depends on many factors, such as input frequency, frequency for other peripherals driven from the same PLL, and restrictions from the architecture. Details of the M & D values chosen by the tool are available in the log file.

DDR Configuration

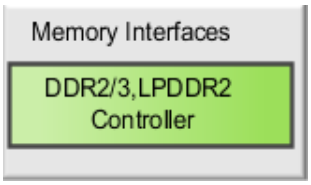


Figure 2-19: **DDR Controller**

You can configure DDR using one of two methods:

- From the Page Navigator, select the **DDR Configuration**.
- In the Zynq block design, click the **DDR2/3, LPDDR2** Controller block.

The DDR memory controller supports DDR2, DDR3, DDR3L, and LPDDR2 devices and consists of three major blocks: an AXI memory port interface - DDR interface (DDRI), a core controller with transaction scheduler (DDRC), and a controller with digital PHY (DDRP).

The DDRI block interfaces with four 64-bit synchronous AXI interfaces to serve multiple AXI masters simultaneously. Each AXI interface has a dedicated transaction FIFO. The DDRC contains two 32-entry content addressable memories (CAMs) to perform DDR data service scheduling to maximize DDR memory efficiency. It also contains a "fly-by" channel for a low-latency channel to allow access to DDR memory without going through the CAM.

The PHY processes read and write requests from the controller and translates them into specific signals within the timing constraints of the target DDR memory. The PHY uses signals from the controller to produce internal signals that connect to the pins using the digital PHYs. The DDR pins connect directly to the DDR device(s) using the PCB signal traces.

The system accesses the DDR using the DDRI through its four 64-bit AXI memory ports:

- One AXI port is dedicated to the L2-cache for the CPUs and ACP
- Two ports are dedicated to the AXI_HP interfaces
- The other masters on the AXI interconnect share the fourth port

The DDRI arbitrates the requests from the eight ports (four reads and four writes). The arbiter selects a request and passes it to the DDR controller and transaction scheduler (DDRC).

The arbitration is based on a combination of how long the request has been waiting, the urgency of the request, and if the request is within the same page as the previous request.

The DDRC receives requests from the DDRI through a single interface for both read and write flows. Read requests include a tag field that the DDR returns with the data. The DDR controller PHY (DDRP) drives the DDR transactions.

Figure 2-20 shows the DDR Controller configuration page.

Note: 8-bit interfaces are not supported; however, 8-bit parts can be used to create 16/32-bit interfaces.

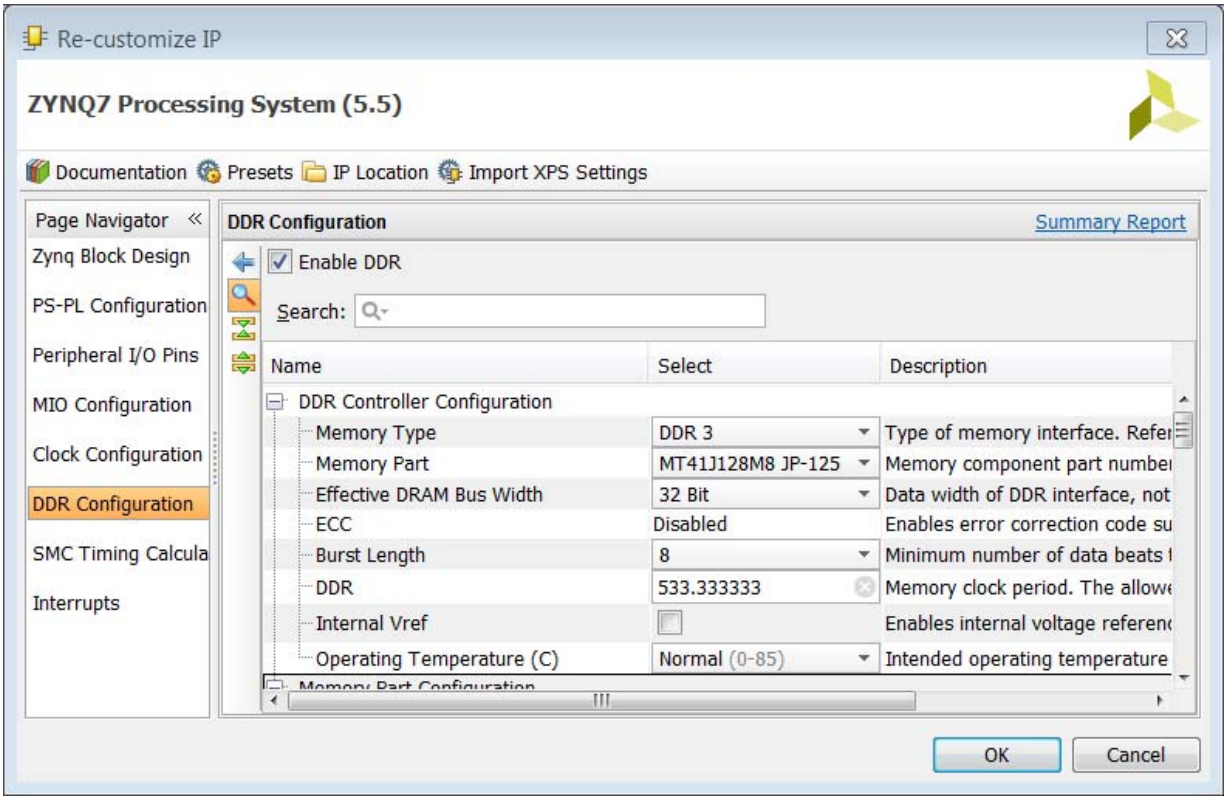


Figure 2-20: **DDR Controller Configurations Page**

GIC - Interrupt Controller

You can configure the Generic Interrupt Controller (GIC) in one of two methods:

- In the Page Navigator, click **Interrupts**.
- In the Zynq block design, click the GIC block.

Figure 2-21: **Generic Interrupt Controller**

Figure 2-22 shows the Interrupt Port Configuration page.

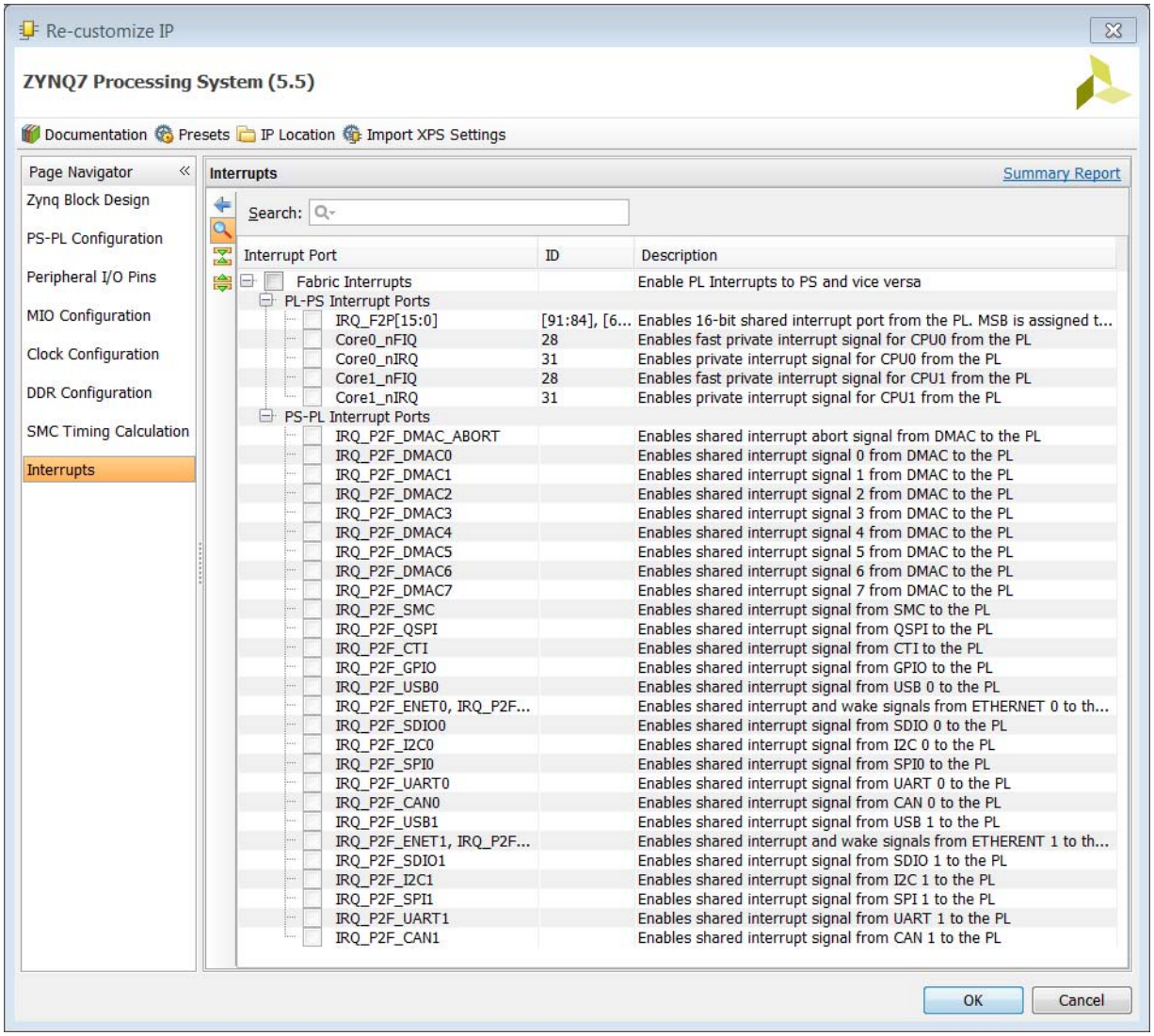


Figure 2-22: **GIC Interrupts**

GIC is a centralized resource for managing interrupts sent to the CPUs from the PS and PL. The controller enables, disables, masks, and prioritizes the interrupt sources and sends them to the selected CPU (or CPUs) in a programmed manner as the CPU interface accepts the next interrupt. In addition, the controller supports security extension for implementing a security-aware system.

The controller is based on the ARM Generic Interrupt Controller Architecture version 1.0 (GIC v1), non-vectored.

The private bus on the CPU accesses the registers for fast read/write response by avoiding temporary blockage or other bottlenecks in the Interconnect.

The interrupt distributor centralizes all interrupt sources before dispatching the one with the highest priority to the individual CPUs.

The GIC ensures that, when you target an interrupt to several CPUs, only one CPU takes the interrupt at a time. All interrupt sources contain a unique interrupt ID number. All interrupt sources have their own configurable priority and list of targeted CPUs.

Both the *Zynq-7000 AP SoC Technical Reference Manual* (UG585) [Ref 4] and the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 1] contain information regarding the logic blocks in the Zynq-7000 device.

Interconnect between PS and PL

AXI_HP Interfaces

Figure 2-23: **AXI_HP Interfaces**

The four AXI_HP interfaces provide PL bus masters with high-bandwidth data paths to the DDR and OCM memories. Each interface includes two FIFO buffers for read and write traffic. The PL to the memory Interconnect routes the high-speed AXI_HP ports either to two DDR memory ports or to the OCM. The AXI_HP interfaces are also referenced as AXI FIFO interfaces (AFI), to emphasize their buffering capabilities.

IMPORTANT: *You must enable the PL level shifters using LVL_SHFTR_EN before PL logic communication can occur.*

Enable these interfaces by selecting **PS-PL Configuration** from the Page Navigator and expanding the **HP Slave AXI Interface** option as shown in Figure 2-24.

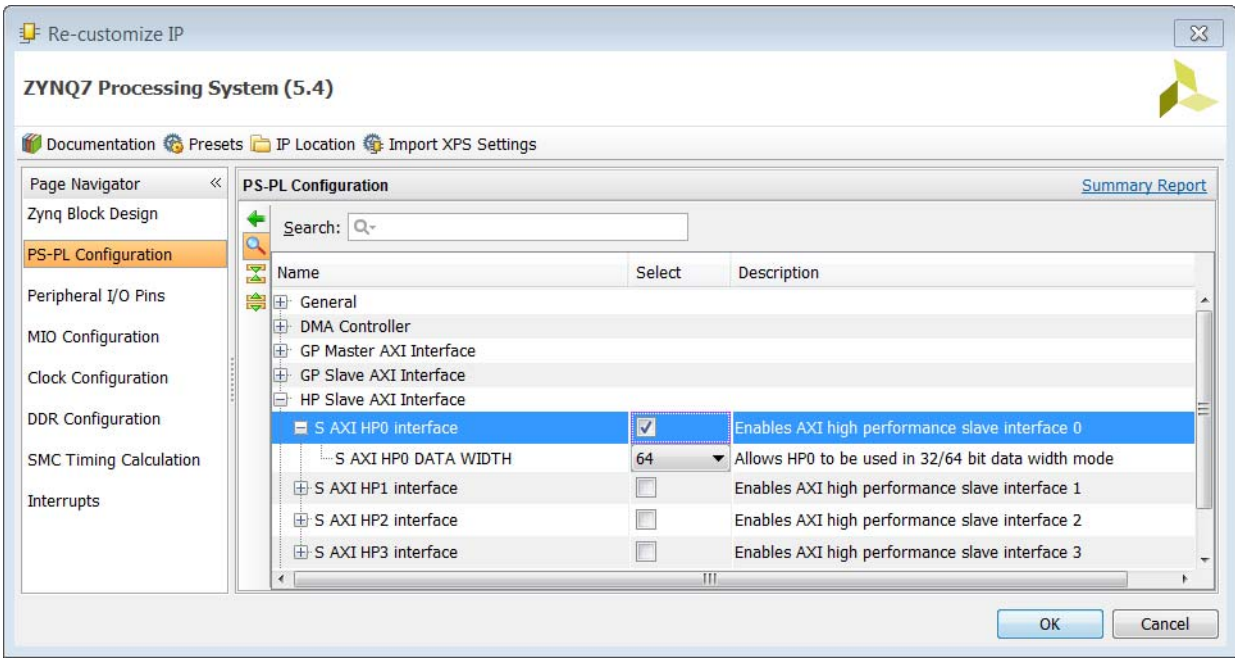


Figure 2-24: **Enabling AXI HP Interfaces**

The interfaces provide a high-throughput data path between PL masters and PS memories including the DDR and on-chip RAM. The main features include:

32- or 64-bit data wide master interfaces (independently programmed per port)

- Efficient dynamic upsizing to 64 bits for aligned transfers in 32-bit interface mode, controllable using `AxCACHE`
- Automatic expansion to 64-bits for unaligned 32-bit transfers in 32-bit interface mode
- Programmable release threshold of write commands
- Asynchronous clock frequency domain crossing for all AXI interfaces between the PL and PS
- Smoothing out of "long-latency" transfers using 1 KB (128 by 64 bit) data FIFOs for both reads and writes
- QoS signaling available from PL ports
- Command and Data FIFO fill-level counts available to the PL
- Standard AXI 3.0 interfaces support
- Programmable command issuance to the interconnect, separately for read and write commands
- Large slave interface read acceptance capability in the range of 14 to 70 commands (burst length dependent)
- Large slave interface write acceptance capability in the range of 8 to 32 commands (burst length dependent)

AXI_ACP Interface

The Accelerator Coherency Port (ACP) provides low-latency access to programmable logic masters, with optional coherency and L1 and L2 cache.

From a system perspective, the ACP interface has similar connectivity as the APU CPUs. Due to this close connectivity, the ACP directly competes for resource access outside of the APU block.

IMPORTANT: *You must enable the PL level shifters using LVL_SHFTR_EN before PL logic communication can occur.*

In the ZYNQ7 block design, click the **64b AXI ACP Slave Ports** block to configure the AXI_ACP.

Figure 2-25: **AXI_ACP Configuration**

Alternatively, select the **PS-PL Configuration** and expand **ACP Slave AXI Interface**.

Figure 2-26 shows the ACP AXI Slave Configuration page.

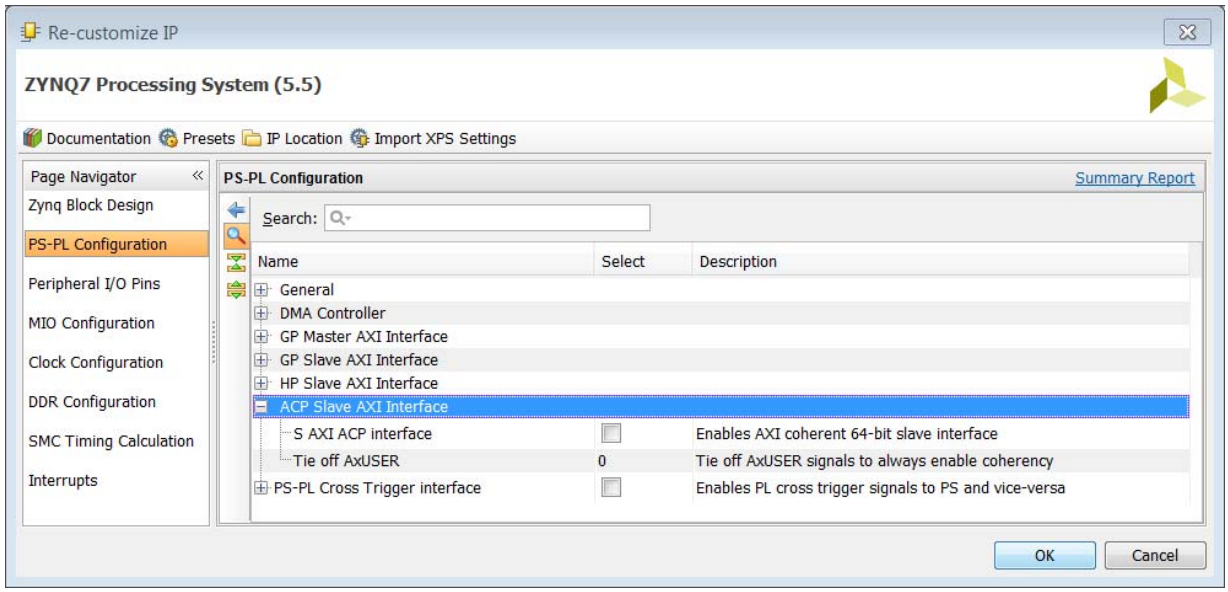


Figure 2-26: **ACP Slave AXI Interface Page**

AXI_GP Interfaces

AXI_GP features include:

- Standard AXI protocol
- Data bus width: 32
- Master port ID width: 12
- Master port issuing capability: 8 reads, 8 writes
- Slave port ID width: 6
- Slave port acceptance capability: 8 reads, 8 writes

These interfaces are connected directly to the ports of the master interconnect and the slave interconnect without additional FIFO buffering, unlike the AXI_HP interfaces, which have elaborate FIFO buffering to increase performance and throughput. Therefore, the performance is constrained by the ports of the master interconnect and the slave interconnect. These interfaces are for general-purpose use only; they are not intended to achieve high performance.

IMPORTANT: *You must enable the PL level shifters using LVL_SHFTR_EN before PL logic communication can occur.*

In the ZYNQ7 block design, click the following block to configure the AXI_GP interface.

Figure 2-27: **AXI_GP Configuration**

Alternatively, in the Page Navigator, select the **PS-PL Configuration** and expand the **GP Master AXI Interface** and **GP Slave AXI Interface** options.

Figure 2-28 shows the GP AXI Master and Slave Configuration page.

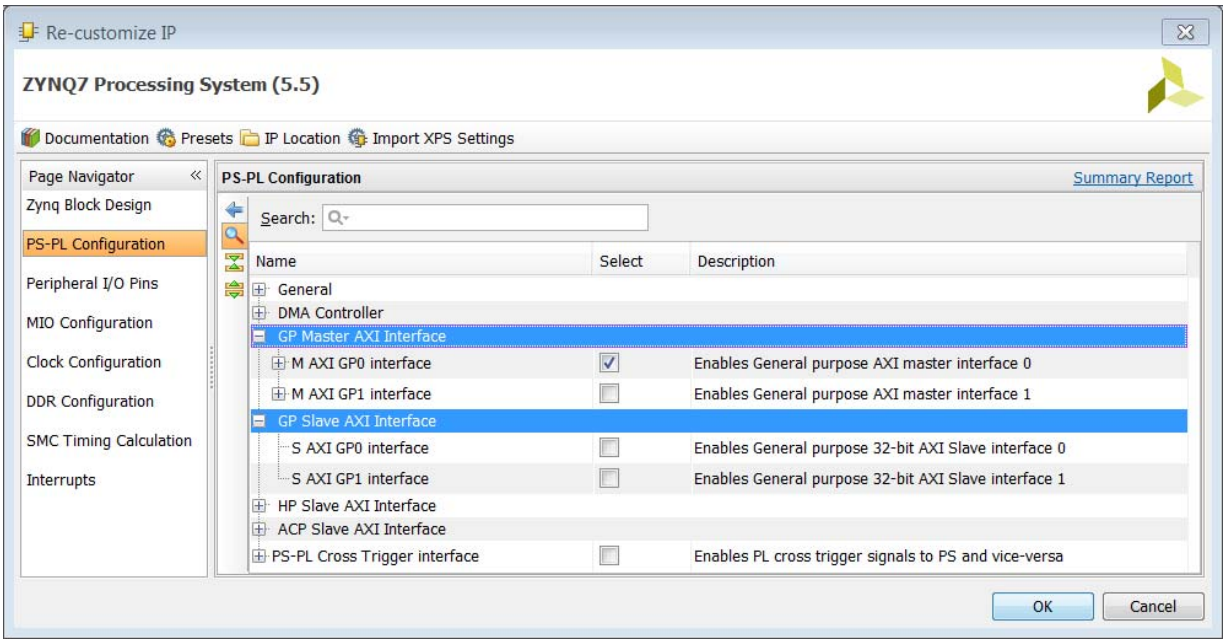


Figure 2-28: **GP Master and Slave AXI Interfaces**

PS-PL Cross Trigger Interface

Embedded Cross Trigger (ECT) is the cross-triggering mechanism. Through ECT, a CoreSight component can interact with other components by sending and receiving triggers. ECT is implemented with two components:

- Cross Trigger Matrix (CTM)
- Cross Trigger Interface (CTI)

One or more CTMs form an event broadcasting network with multiple channels. A CTI listens to one or more channels for an event, maps a received event into a trigger, and sends the trigger to one or more CoreSight components connected to the CTI. A CTI also combines and maps the triggers from the connected CoreSight components and broadcasts them as events on one or more channels. Both CTM and CTI are CoreSight components of the control and access class.

ECT is configured with:

- Four broadcast channels
- Four CTIs

Note: Power-down is not supported.

You can enable cross trigger by selecting the PS-PL Cross Trigger Interface in the ZYNQ7 Processing System configuration dialog box.

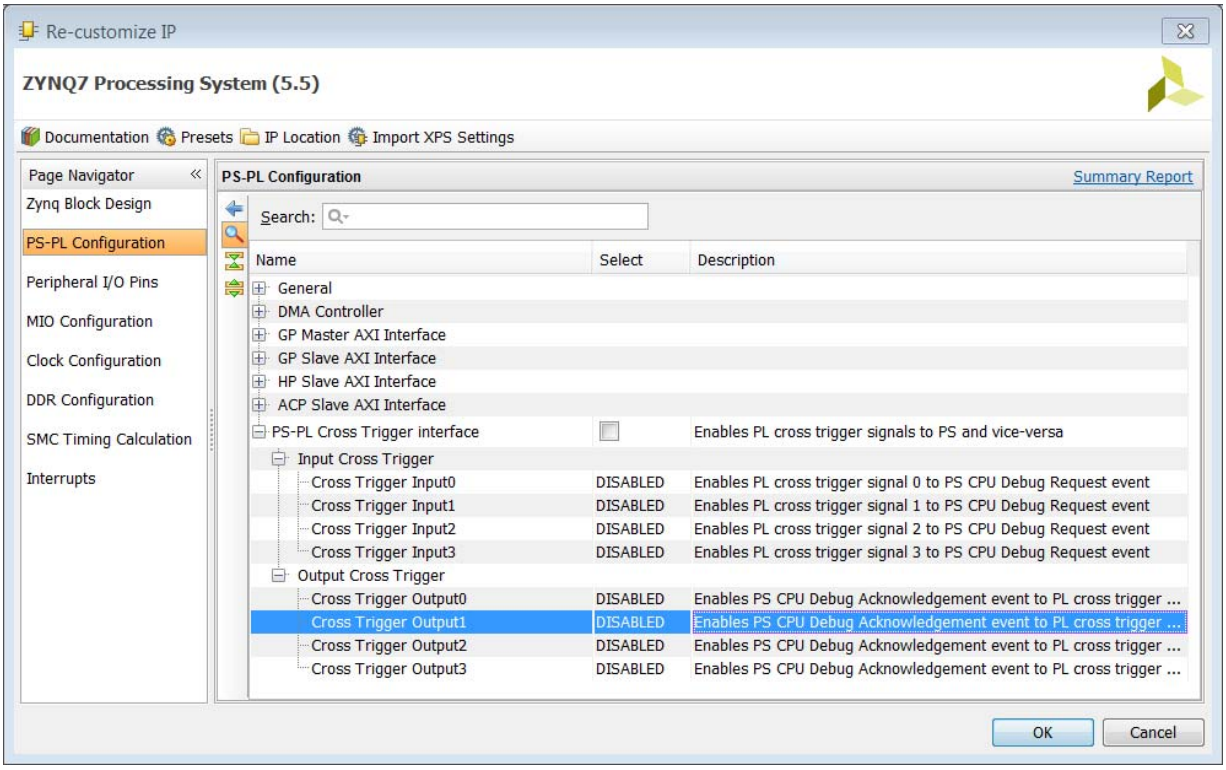


Figure 2-29: **PS-PL Cross Trigger Interface**

Using the Programmable Logic (PL)

The PL provides a rich architecture of user-configurable capabilities.

Configurable logic blocks (CLB)

- 6-input look-up tables (LUTs) with memory capability within the LUT
- Register and shift register functionality
- Adders that can be cascaded

36 Kb block RAM

- Dual ports, up to 72 bits wide
- Configurable as dual 18 Kb
- Programmable FIFO logic
- Built-in error correction circuitry

Digital signal processing - DSP48E1 Slice

- 25 × 18 two's complement multiplier/accumulator high-resolution (48 bit) signal processor
- Power-saving 25-bit pre-adder to optimize symmetrical filter applications
- Advanced features: optional pipelining, optional ALU, and dedicated buses for cascading

Clock management

- UHigh-speed buffers and routing for low-skew clock distribution
- Frequency synthesis and phase shifting
- Low-jitter clock generation and jitter filtering

Configurable I/Os

- High-performance SelectIO™ technology
- High-frequency decoupling capacitors within the package for enhanced signal integrity
- Digitally controlled impedance that can be tri-state for lowest power, high-speed I/O operation
- High range (HR) I/Os support 1.2 V to 3.3 V
- High performance (HP) I/Os support 1.2 V to 1.8 V (7z030, 7z045, and 7z100 devices)

Low-power gigabit transceivers

- (7z030, 7z045, and 7z100 devices)
- High-performance transceivers capable of up to 12.5 Gb/s (GTX)
- Low-power mode optimized for chip-to-chip interfaces
- Advanced transmit pre- and post-emphasis, and receiver linear (CTLE) and decision feedback equalization (DFE), including adaptive equalization for additional margin

Analog-to-digital converter (XADC)

- Dual 12-bit 1 MSPS analog-to-digital converters (ADCs)
- Up to 17 flexible and user-configurable analog inputs
- On-chip or external reference option
- On-chip temperature (±4°C max error) and power supply (±1% max error) sensors
- Continuous JTAG access to ADC measurements

Integrated interface blocks for PCI Express designs (7z030, 7z045, and 7z100 devices)

- Compatible to the PCI Express base specification 2.1 with Endpoint and Root Port capability
- Supports Gen1 (2.5 Gb/s) and Gen2 (5.0 Gb/s) speeds

Advanced configuration options, advanced error reporting (AER), end-to-end CRC (ECRC)

Custom Logic

The Vivado® IP packager lets you and third-party IP developers use the Vivado IDE to easily prepare an Intellectual Property (IP) design for use in the Vivado IP catalog. The IP user can then instantiate this third party IP into a design in the Vivado Design Suite.

When IP developers use the Vivado Design Suite IP packaging flow, the IP user has a consistent experience whether using Xilinx IP, third-party IP, or customer-developed IP within the Vivado Design Suite.

IP developers can use the IP packager feature to package IP files and associated data into a ZIP file. The IP user receives this generated ZIP file and installs the IP into the Vivado Design Suite IP Catalog. The IP user then customizes the IP through parameter selections and generates an instance of the IP. See the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 3] and *Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator* (UG995) [Ref 5] for more information.

RECOMMENDED: *To verify the proper packaging of the IP before handing it off to the IP user, Xilinx® recommends that the IP developer run each IP module completely through the IP user flow to validate that the IP is ready for use.*

Zynq-7000 Processing System Simulation

The Zynq®-7000 Bus Functional Model (BFM) supports the functional simulation of Zynq-7000-based applications. It enables the functional verification of Programmable Logic (PL) by mimicking the PS-PL interfaces and OCM/DDR memories of Processor System (PS) logic. This BFM is delivered as a package of encrypted Verilog modules. A sequence of Verilog tasks in a Verilog syntax file controls the BFM operation.

Features

- Pin compatible and Verilog-based simulation model
- Supports all AXI interfaces
  - AXI 3.0 compliant
- Sparse memory model (for DDR) and a RAM model (for OCM)
- Verilog task-based API
- Delivered in Vivado Design Suite
- Blocking and non-blocking interrupt support
- Requires license to AXI BFM models

Applications

The Zynq-7000 BFM provides a simulation environment for the Zynq-7000 PS logic, typically replacing the processing_system7 block in a design. The Zynq-7000 BFM models the following:

- Transactions originating from PS masters through the AXI BFM master API calls
- Transactions terminating through the PS slaves to models of the OCM and DDR memories through interconnect models
- FCLK reset and clocking support
- Input interrupts to the PS from PL
- PS register map

For more information on the Zynq BFM, see *Zynq-7000 Bus Functional Model* (DS897).

Embedded IP Catalog

The Vivado Design Suite IP Catalog is a unified repository that lets you search, review detailed information, and view associated documentation for the IP. After you add the third-party or customer IP to the Vivado Design Suite IP catalog, you can access the IP through the Vivado Design Suite flows.

Figure 2-30 shows a portion of the Vivado IDE IP integrator IP Catalog.

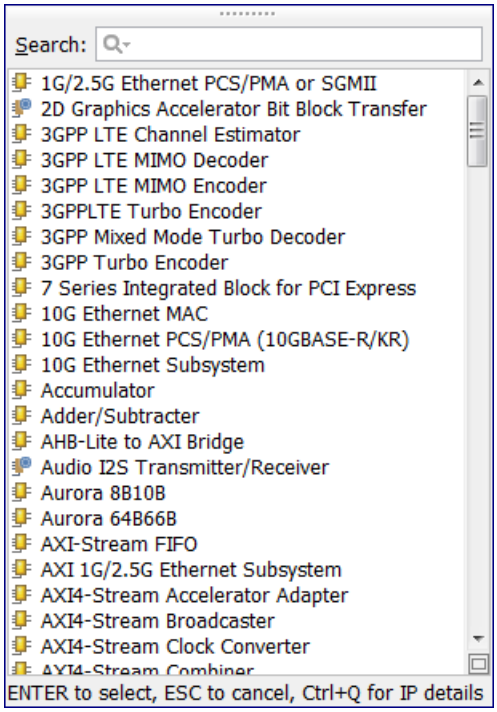


Figure 2-30: **IP Integrator IP Catalog**

Completing Connections Using Designer Assistance

After you have configured the ZYNQ-7000 PS, you can instantiate other IP that go in the programmable logic portion of the device.

In the IP integrator diagram area, right-click and select **Add IP**.

You can use two built-in features of the IP integrator to complete the rest of the IP subsystem design: Block Automation and Connection Automation. These features help you put together a basic microprocessor system in the IP integrator tool and connect ports to external I/O ports.

Block Automation

Block Automation is available when a microprocessor such as the Zynq-7000 PS or MicroBlaze™ processor is instantiated in the block design of the IP integrator tool.

Click **Run Block Automation** to get assistance with putting together a simple **ZYNQ Processing System**, as shown in Figure 2-31.

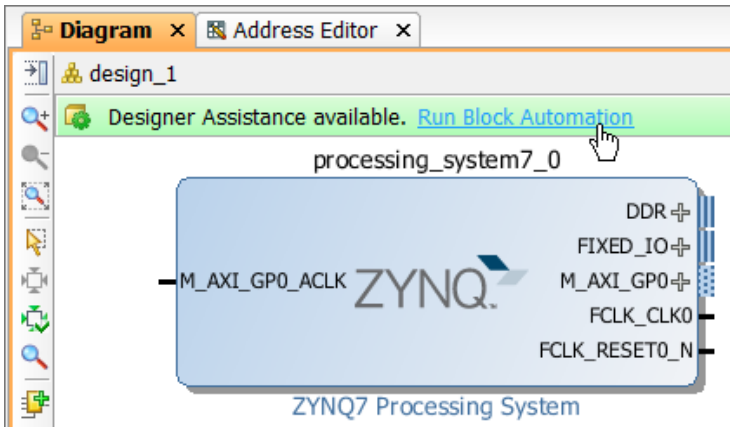


Figure 2-31: **Run Block Automation Feature**

The Run Block Automation dialog box shows the options available for automation, as shown in Figure 2-32.

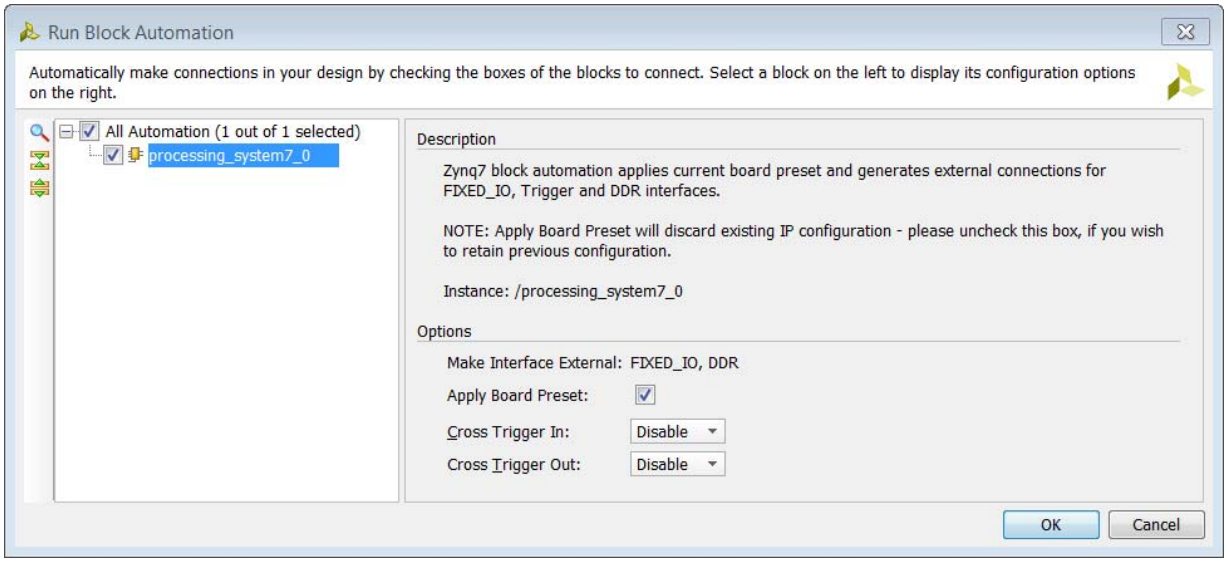


Figure 2-32: **Run Block Automation for ZYNQ7 Dialog Box**

After you click **OK**, the Block Automation feature creates the basic system, as shown in Figure 2-33.

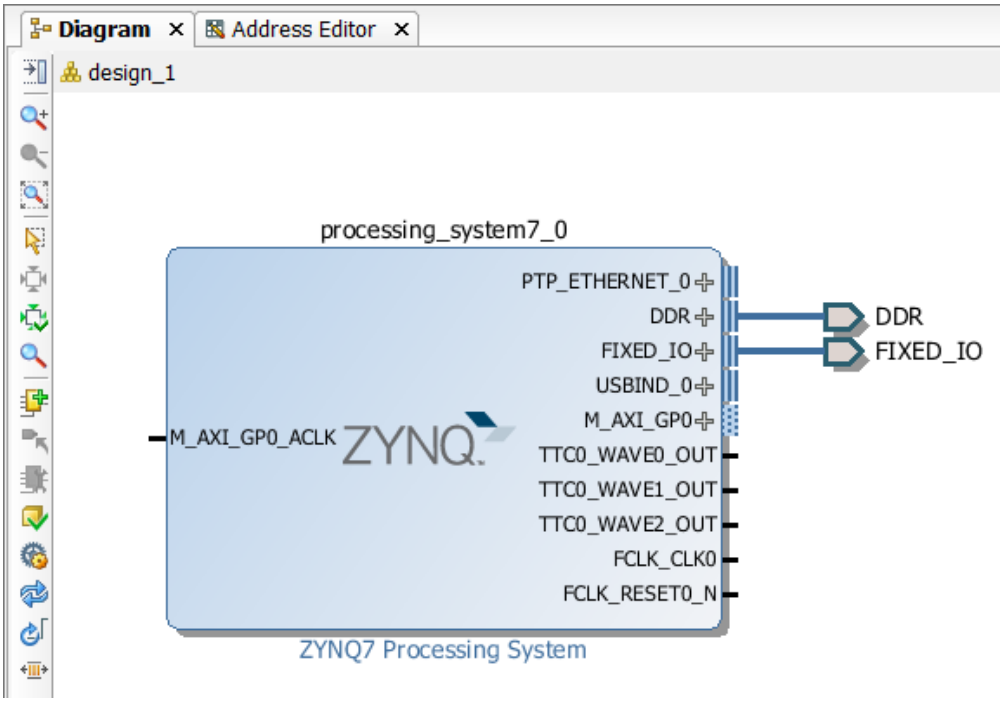


Figure 2-33: **IP Integrator Canvas after Running Block Automation**

You can also enable the cross-trigger feature by selecting the appropriate function using the **Cross Trigger In** and **Cross Trigger Out** fields of the Block Automation dialog box.

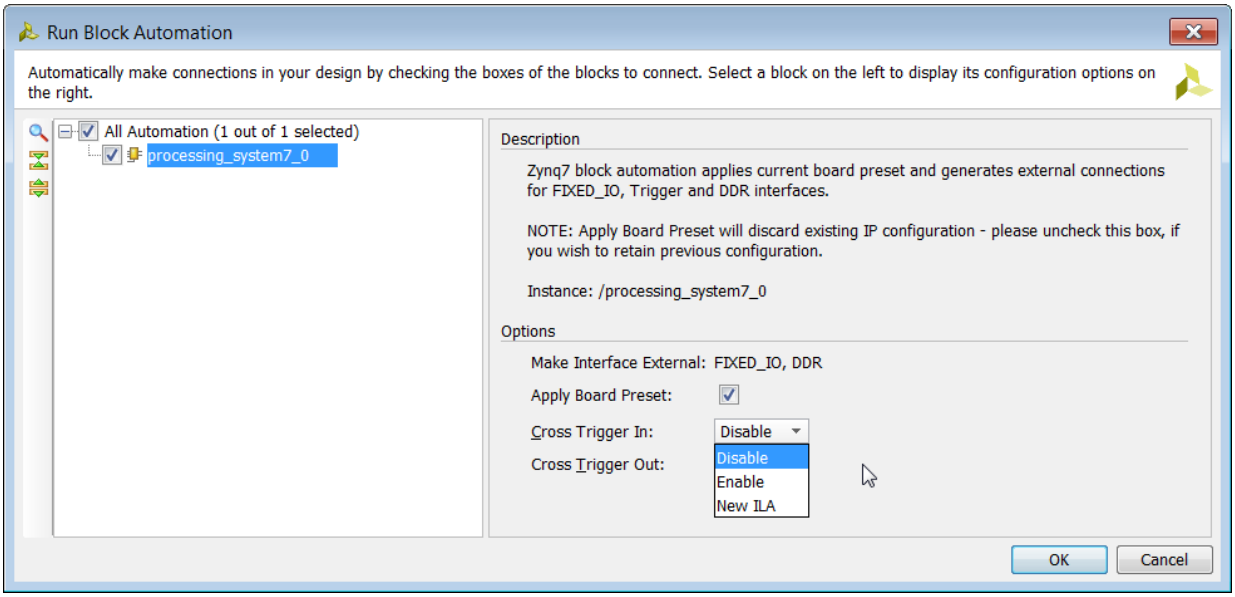


Figure 2-34: **Using Run Block Automation Dialog Box to Enable Cross Trigger Feature**

The default value for the **Cross Trigger In** and **Cross Trigger Out** fields is **Disable**. However, you can use the cross trigger by selecting the **Enable** and **New ILA** options.

Selecting **Enable** for **Cross Trigger In** and **Cross Trigger Out** exposes only one of the available cross trigger pins in ZYNQ7. The connectivity to these pins is left for you to complete.

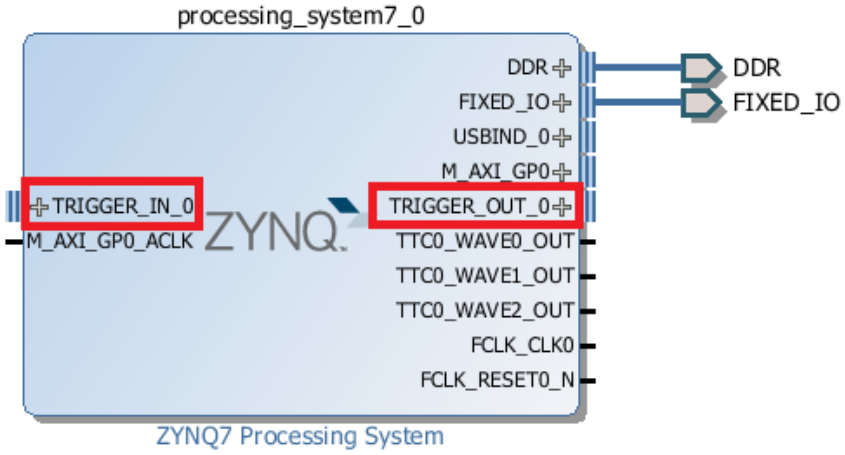


Figure 2-35: **Cross Trigger Pins in ZYNQ7**

When you select the New ILA option, it not only enables the cross-trigger pins; it also connects them to an Integrated Logic Analyzer (ILA) core.

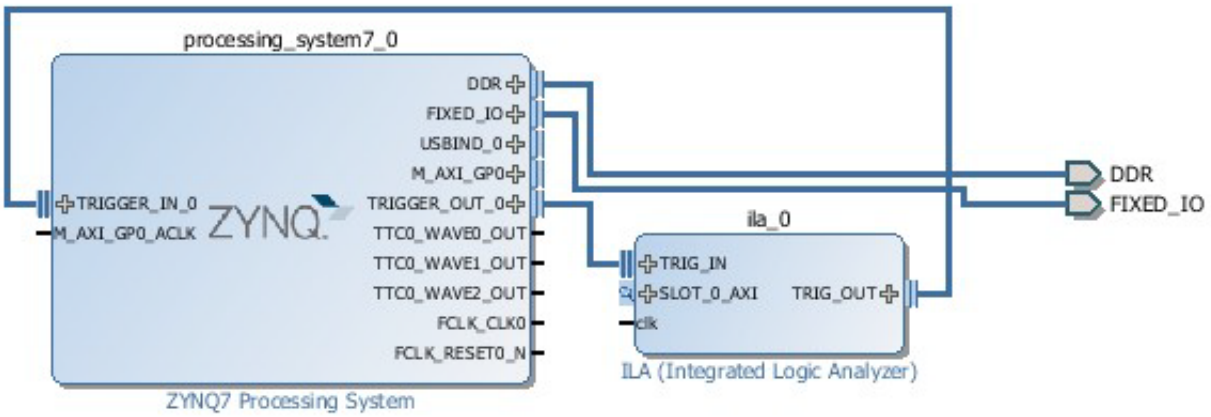


Figure 2-36: **Cross Trigger Pins Connected to an ILA Using Block Automation**

The Vivado IP integrator tool also provides a Board Automation feature when using a Xilinx Target Reference Platform, such as the ZC702. See Platform Board Flow in IP Integrator, page 44 for more information.

This feature provides connectivity of the ports of an IP to the FPGA pins on the target board. The IP configures accordingly, and based on your selections, connects the I/O ports. Board Automation automatically generates the physical constraints for those IP that require physical constraints.

In Figure 2-33, observe that the external `DDR` and `FIXED_IO` interfaces connect to external ports.

Using Connection Automation

If the IP integrator tool determines that a potential connection exists among the instantiated IP in the canvas, it opens the Connection Automation feature.

In Figure 2-37, the AXI BRAM Controller and the Block Memory Generator IP are instantiated along with the ZYNQ7 Processing System IP.

The IP integrator tool determines that a potential connection exists between the AXI BRAM Controller and the ZYNQ7 IP; consequently, Connection Automation is available.

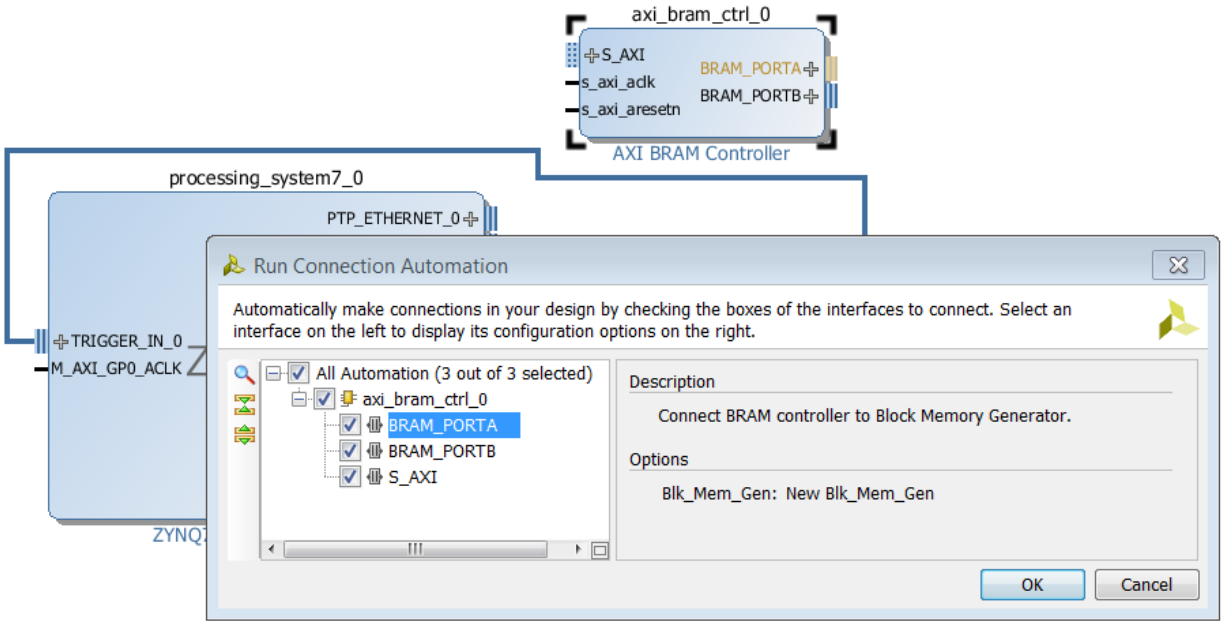


Figure 2-37: **Using Run Connection Automation Feature to Complete Connectivity**

Clicking **Run Connection Automation** does the following:

- Instantiates an AXI Interconnect, a Block Memory Generator, and a Proc Sys Reset IP
- Connects the AXI BRAM Controller to the ZYNQ7 PS IP using the AXI Interconnect
- Appropriately connects the Proc Sys Reset IP as shown in Figure 2-38

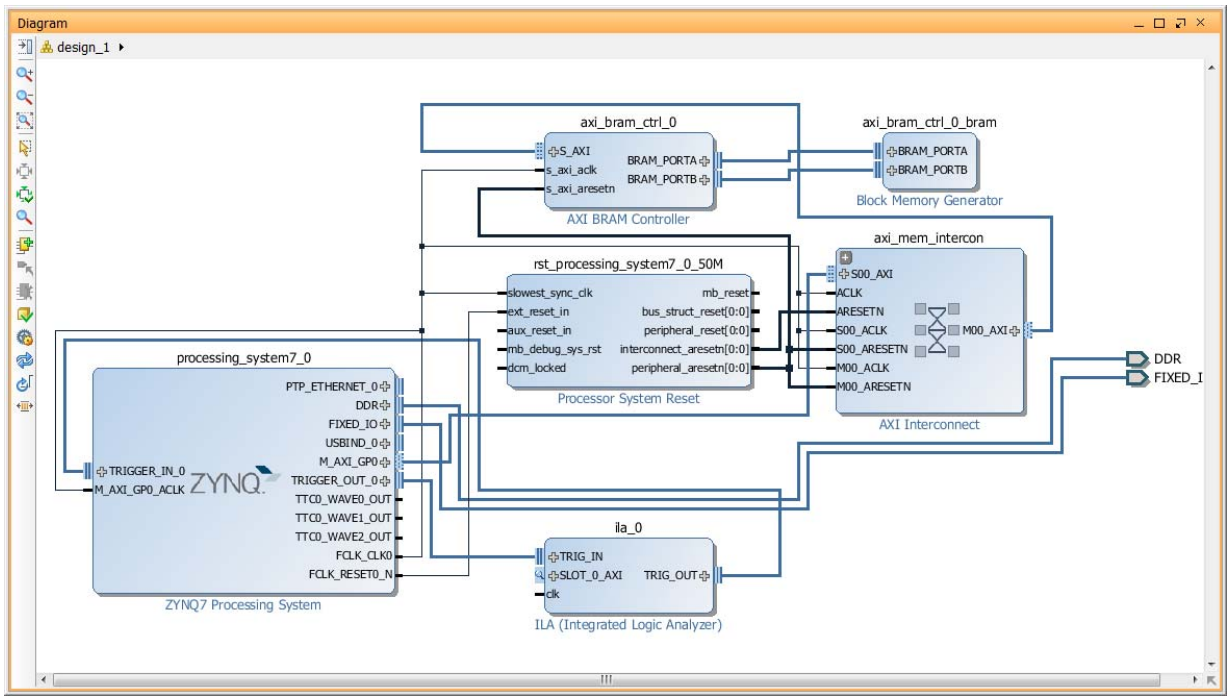


Figure 2-38: **Block Design After Using Connection Automation**

Manual Connections in a Design

Figure 2-39 shows how you can connect the ILA SLOT_0_AXI or the clk pin to the clock and the AXI interface that needs to be monitored in the design. You can do this manually.

As you move the cursor near an interface or pin connector on an IP block, the cursor changes to a pencil. Click an interface or pin connector on an IP block, and drag the connection to the destination block.

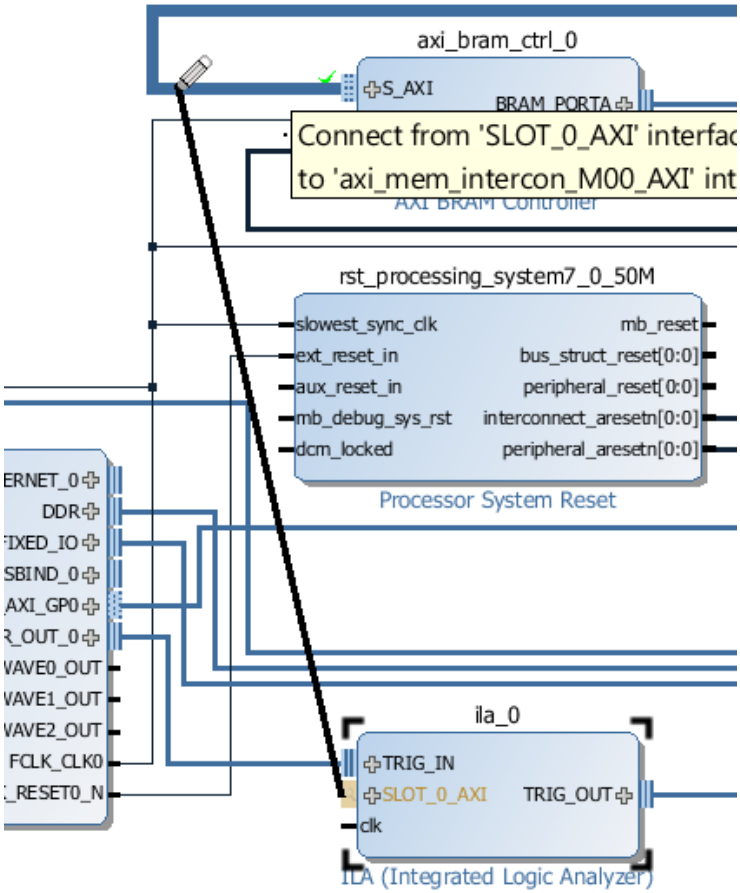


Figure 2-39: **Manually Connecting Ports**

Manually Creating and Connecting to I/O Ports

You can manually create external I/O ports in the Vivado IP integrator. You can connect signals or interfaces to external I/O ports by selecting a pin, a bus, or an interface connection.

To manually create/connect to an I/O port, right-click the port in the block diagram, and then select one of the following from the right-click menu:

- **Make External**. You can use the **Ctrl+Click** keyboard combination to select multiple pins and invoke the **Make External** connection. This command ties a pin on an IP to an I/O port on the block design.
- **Create Port**. Use this command for non-interface signals, such as a `clock`, `reset`, or `uart_txd`.

  The Create Port option gives more control in terms of specifying the input/output, the bit-width and the type (`clk`, `reset`, or `data`). In case of a clock, you can even specify the input frequency.

- **Create Interface Port**. This command creates ports on the interface for groupings of signals that share a common function.

  For example, the S_AXI is an interface port on several Xilinx IP. The command gives more control in terms of specifying the interface type and the mode (master or slave).

Enhanced Designer Assistance

The IP integrator tool offers enhanced designer assistance when an AXI Streaming Interface is to be connected to an AXI memory mapped interface. As an example in Figure 2-40, a FIR Compiler IP with a streaming interface is to be connected to the slave ACP port of the PS7.

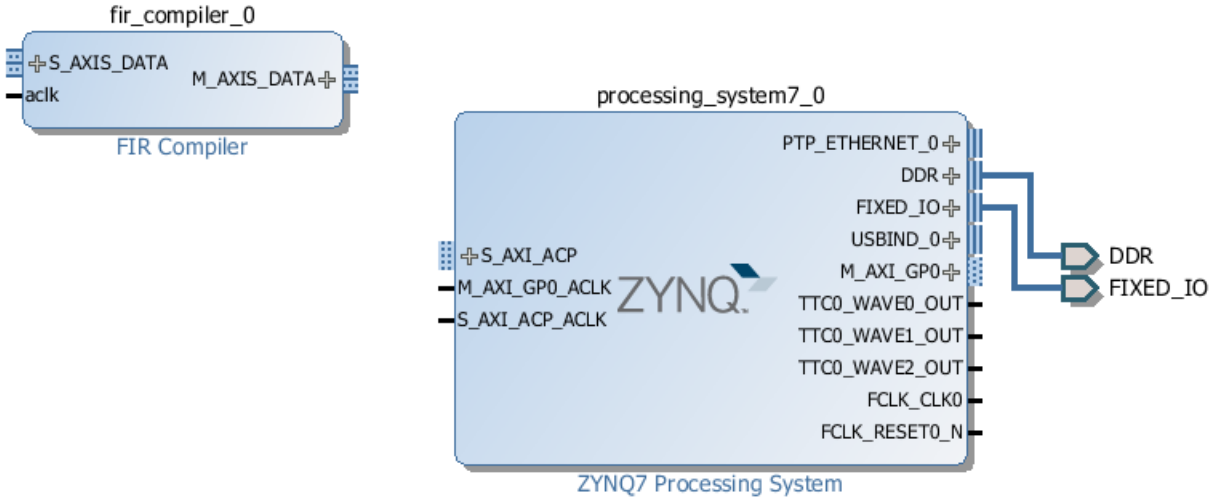


Figure 2-40: **Connecting Streaming Interface to a Memory Mapped Interface**

To use the enhanced designer assistance the user needs to make a direct connection between the M_AXIS_DATA interface pin of the FIR Compiler and the S_AXI_ACP Port of the

ZYNQ7 Processing System as shown in Figure 2-41.

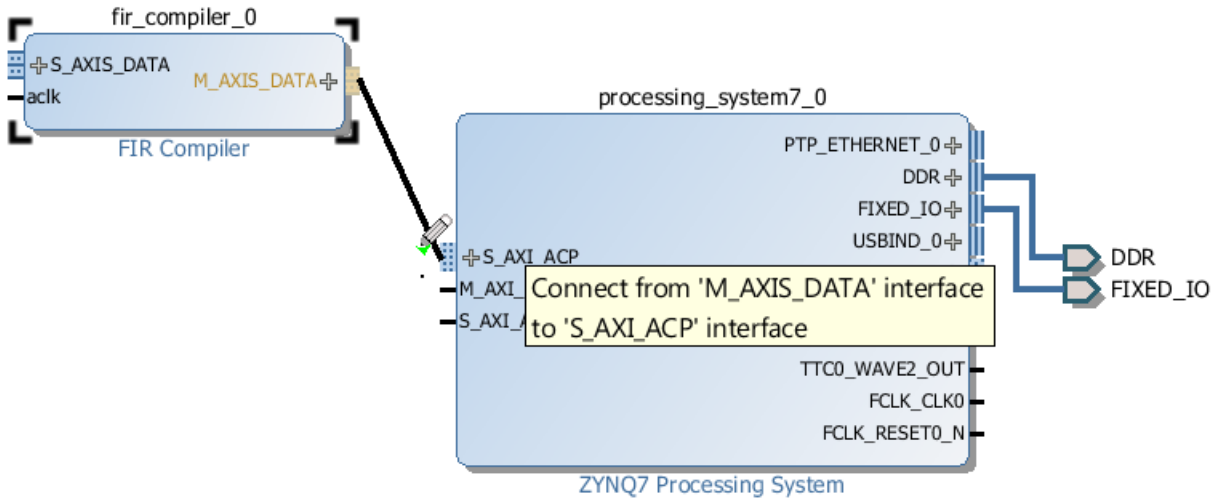


Figure 2-41: **Invoking enhanced designer assistance**

The Make Connection dialog box pops up informing the user that the Stream Bus Interface `/fir_compiler_0/M_AXIS_DATA` will be connected to the Memory Mapped bus-interface `/processing_system7_0/S_AXI_ACP`. It also offers the user difference options for clocking on the Streaming Interface and Memory Mapped Interface. The default is **Auto**.

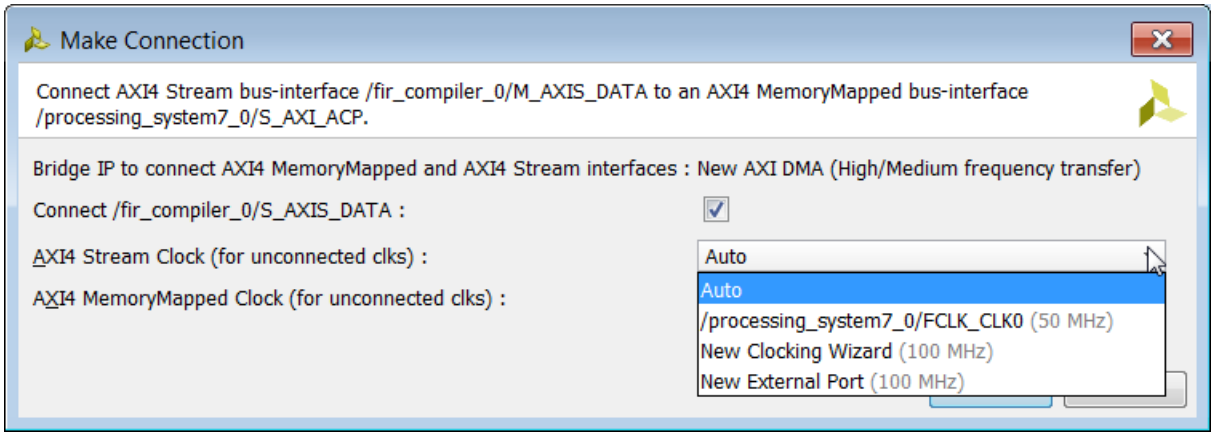


Figure 2-42: **Make Connection dialog for enhanced designer assistance**

The enhanced designer assistance instantiates a DMA core configured to do High/Medium frequency transfers and makes the appropriate connection when the user chooses to click on OK after selecting the proper settings.

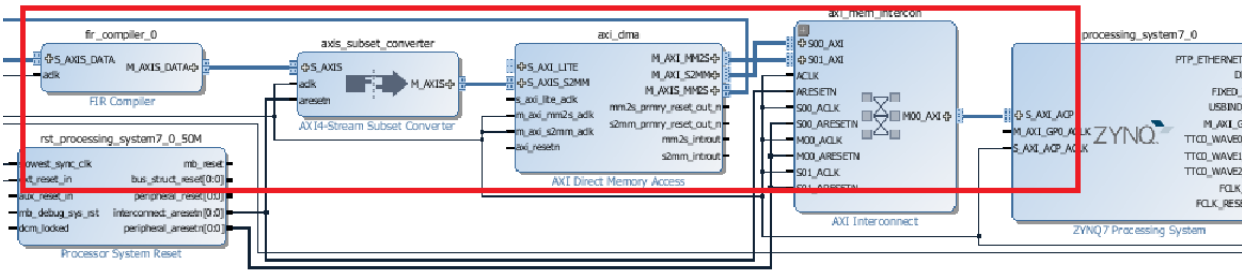


Figure 2-43: **Connections made after using enhanced designer assistance**

The enhanced designer assistance instantiates an AXI Subset Converter, an AXI Direct Memory Access and an AXI Interconnect to make the connection between the streaming interface of the FIR Compiler and the ACP port of PS7. The AXI4-Stream Subset Converter provides a solution for connecting slightly incompatible AXI4-Stream signal sets together. The IP has configurable AXI4-Stream signals for each interface that allows one to convert one signal set to another in a consistent manner.

Platform Board Flow in IP Integrator

The Vivado® Design Suite is board aware. The tools know the various components present on the target board and can customize an IP to be instantiated and configured to connect to the components of a particular board. Several 7 series boards and a Kintex UltraScale board are currently supported.

The IP integrator shows all the components present on the board in a separate tab called the Board tab. When you use this tab to select the desired components and the Designer Assistance offered by IP integrator, you can easily connect your design to the components of your choice. All the I/O constraints are automatically generated as a part of using this flow. Refer to this link in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (*UG994*)* [Ref 3] for more information.

Memory Mapping in Address Editor

While memory mapping of the peripherals (slaves) instantiated in the block design are automatically assigned, users can also manually assign the addresses. To generate the address map for this design:

1. Click the Address Editor tab above the diagram.
2. Click the **Auto Assign Address** button (bottom on the left side).

You can manually set addresses by entering values in **Offset Address** and **Range** columns. Refer to this link in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP*

Integrator (UG994) [Ref 3] for more information.

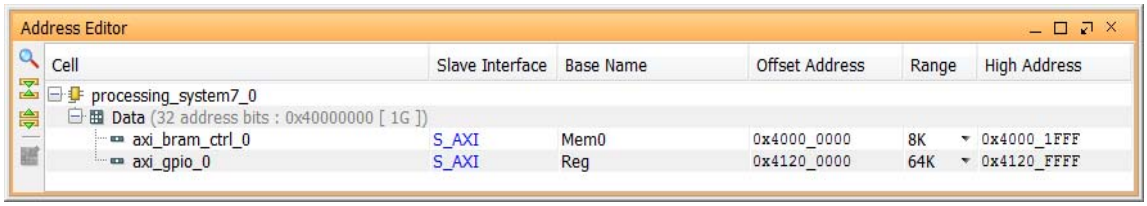


Figure 2-44: **Memory Mapping Peripherals**

TIP: *The **Address Editor** tab only opens if the diagram contains an IP block that functions as a bus master (such as the ZYNQ7 processor) in the design.*

Running Design Rule Checks

The Vivado IP integrator runs basic DRCs in real time as you put the design together. However, errors can occur during design creation. For example, the frequency on a clock pin might not be set correctly.

To run a comprehensive DRC, click the **Validate Design** toolbar button .

If no warnings or errors occur in the design, a validation dialog box displays to confirm that there are no errors or critical warnings in your design,

Integrating a Block Design in the Top-Level Design

After you complete the block design and validate the design, there are two more steps required to complete the design:

- Generate the output products
- Create a HDL wrapper

Generating output products makes the source files and the appropriate constraints for the IP available in the Vivado IDE Sources window.

Depending upon what you selected as the target language during project creation, the IP integrator tool generates the appropriate files. If the Vivado IDE cannot generate the source files for a particular IP in the specified target language, a message displays in the console.

Generating Output Products

To generate output products, do one of the following:

- In the Block Design panel, expand the Design Sources hierarchy and select **Generate Output Products**.
- In the Flow Navigator panel, under IP Integrator, click **Generate Block Design**.

The Vivado Design Suite generates the HDL source files and the appropriate constraints for all the IP used in the block design. The source files are generated based upon the Target Language that you selected during project creation, or in the Project Settings dialog box. Refer to the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 3], for more information on generating output products.

Creating an HDL Wrapper

You can integrate an IP integrator block design into a higher-level design. To do so, instantiate the design in a higher-level HDL file.

To instantiate at a higher level, in the Design Sources hierarchy of the Block Design panel, right-click the design and select **Create HDL Wrapper**.

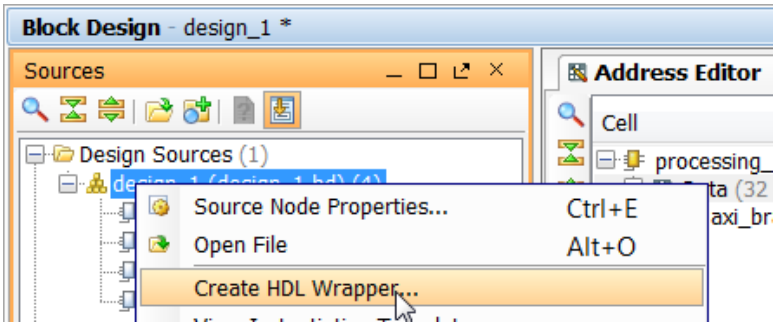


Figure 2-45: **Creating an HDL Wrapper**

Vivado offers two choices for creating an HDL wrapper:

- Let Vivado create and automatically update the wrapper, which is the default option
- Create a user-modifiable script, which you can edit and maintain. Choosing this option requires that you update the wrapper every time you make port-level changes in the block design.

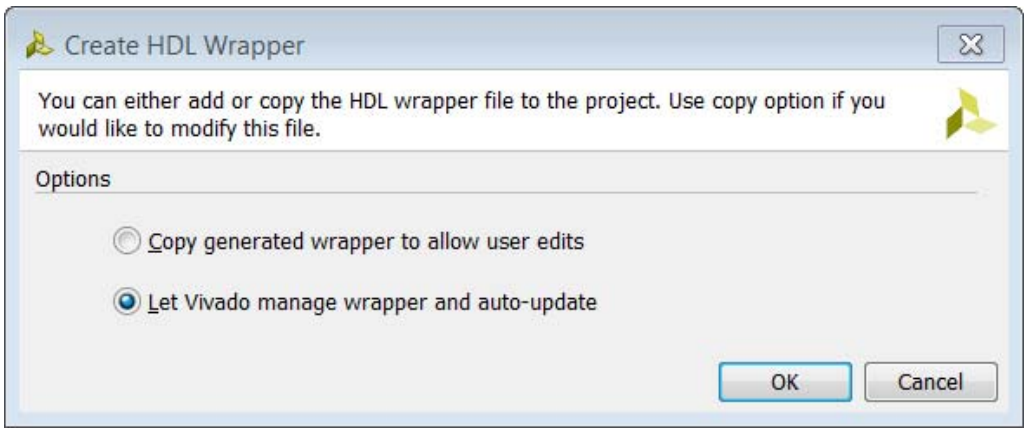


Figure 2-46: **Create HDL Wrapper Dialog Box**

This generates a top-level HDL file for the IP integrator subsystem. You can now take your design through the other design flows: elaboration, synthesis, and implementation.

Vivado Pin Planner View of PS I/O

The *Zynq-7000 All Programmable SoC PCB Design and Pin Planning Guide* (UG933) [Ref 6] provides a detailed description of guidelines for PCB Design and Pin Planning for Zynq-7000 devices.

Vivado IDE Generated Embedded Files

When you export a Zynq-7000 processor hardware design from the IP integrator tool to SDK, the IP integrator generates the following files:

Table 2-1: **Files Generated by IP Integrator**

File	Description
`system.xml`	This file opens by default when you launch SDK and displays the address map of your system.
`ps7_init.c` `ps7_init.h`	The ps7_init.c and ps7_init.h files contain the initialization code for the Zynq Processing System and initialization settings for DDR, clocks, PLLs, and MIOs. SDK uses these settings when initializing the processing system so applications can run on top of the processing system. Some settings in the processing system are in a fixed state for the ZC702 evaluation board.
`ps7_init.tcl`	This is the Tcl version of the INIT file.
`ps7_init.html`	The INIT file describes the initialization data.

See the *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821) [Ref 1] for more information about generated files.

Using the Software Development Kit (SDK)

The Xilinx Software Development Kit (SDK) provides a complete environment for creating software applications targeted for Xilinx embedded processors. It includes a GNU-based compiler toolchain (GCC compiler, GDB debugger, utilities, and libraries), JTAG debugger, flash programmer, drivers for Xilinx IP and bare-metal board support packages, middleware libraries for application-specific functions, and an IDE for C/C++ bare-metal and Linux application development and debugging. Based upon the open source Eclipse platform, SDK incorporates the C/C++ Development Toolkit (CDT).

Features of SDK include:

- C/C++ code editor and compilation environment
- Project management
- Application build configuration and automatic make file generation
- Error navigation
- Integrated environment for debugging and profiling embedded targets
- Additional functionality available using third-party plug-ins, including source code version control

SDK Availability

SDK is available from the Xilinx Vivado Design Suite installation package or as a standalone installation. SDK also includes an application template for creating a First Stage Bootloader (FSBL), as well as a graphical interface for building a boot image. SDK contains a help system that describes concepts, tasks, and reference information.

Exporting a Hardware Description

Once a design has been implemented and the bitstream generated, you can export the design to SDK for software application development. In rare cases where the Processing Logic does not contain any logic at all, you can also export the design without implementing or generating the bitstream.

To export your design to SDK, do the following:

1. In the main Vivado IDE, select **File > Export > Export Hardware**.

   The Export Hardware for SDK dialog box opens (Figure 2-47).

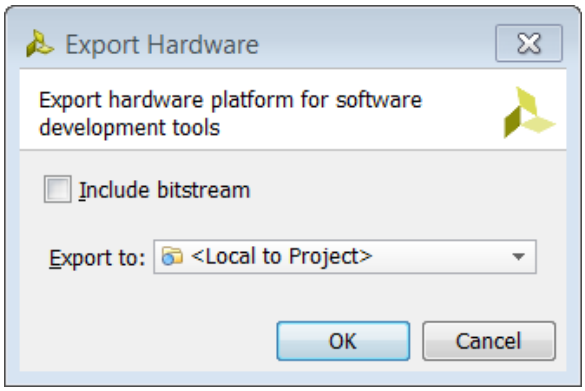


Figure 2-47: **Export Hardware for SDK**

2. In the Export Hardware for SDK dialog box, check the **Include bitstream** check box.

Note: In a project-based flow, typically the **Export to** field is set to **<Local to Project>**, but it can be changed as deemed appropriate.

3. After the hardware definition has been exported, select **File > Launch SDK** to launch SDK from Vivado

   The Launch SDK dialog box opens.

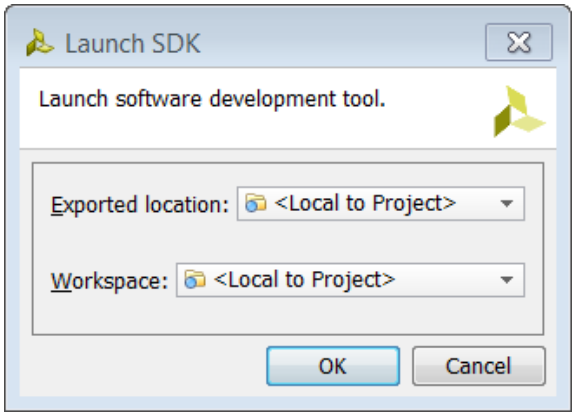


Figure 2-48: **Launch SDK Dialog Box**

   The **Exported location** and **Workspace** fields are typically set to **<Local to Project>** in a project based flow. However, if the user specified a different location for exporting the hardware definition, set the **Exported location** field to that particular location. Likewise, the Workspace location can be set to a the appropriate directory location if desired.

After you export the hardware definition to SDK, and launch SDK, you can start writing your software application in SDK.

You can do further debug and downloading of the software from SDK.

Alternatively, you can import the ELF file for the software back into the Vivado tools, and integrate it with the FPGA bitstream for further download and testing.

Using a MicroBlaze Processor in an Embedded Design

Introduction to MicroBlaze Processor Design

The Vivado IDE IP integrator is a powerful tool that lets you stitch together a processor-based system.

The MicroBlaze™ embedded processor is a Reduced Instruction Set Computer (RISC) core, optimized for implementation in Xilinx® Field Programmable Gate Arrays (FPGAs).

Figure 3-1 shows a functional block design of the MicroBlaze core.

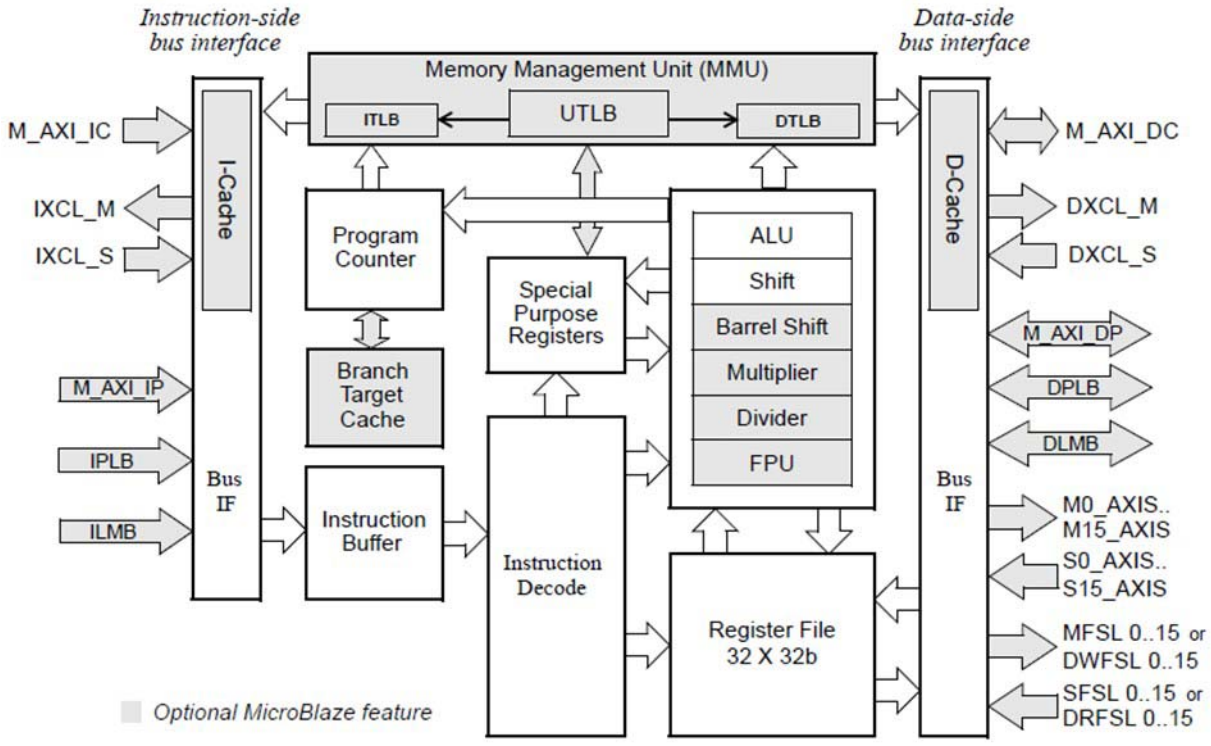


Figure 3-1: **Block Design of MicroBlaze Core**

The MicroBlaze processor is highly configurable: you can select a specific set of features required by your design.

The fixed feature set of the processor includes:

- Thirty-two 32-bit general purpose registers
- 32-bit instruction word with three operands and two addressing modes
- 32-bit address bus
- Single issue pipeline

In addition to these fixed features, the MicroBlaze processor has parameterized values that allow selective enabling of additional functionality.

RECOMMENDED: *Older (deprecated) versions of MicroBlaze support a subset of the optional features described in this manual. Only the latest (preferred) version of MicroBlaze (v9.0) supports all options. Xilinx recommends that new designs use the latest preferred version of the MicroBlaze processor.*

Refer to the *MicroBlaze Processor Reference Guide* (UG984) [Ref 7] for more information.

Creating an IP Integrator Design with the MicroBlaze Processor

Designing with a MicroBlaze processor is different using the Vivado IDE than it was using the ISE® Design Suite and Embedded Development Kit (EDK).

The Vivado IDE uses the IP integrator tool for embedded development. The IP integrator is a GUI-based interface that lets you stitch together complex IP subsystems.

A variety of IP are available in the Vivado IDE IP Catalog to meet the needs of complex designs. You can also add custom IP to the IP Catalog.

Designing with the MicroBlaze Processor

1. In the Flow navigator panel, under IP Integrator, click the **Create Block Design** button to open the Create Block Design dialog box.
2. Type the Design Name, as shown in Figure 3-2.

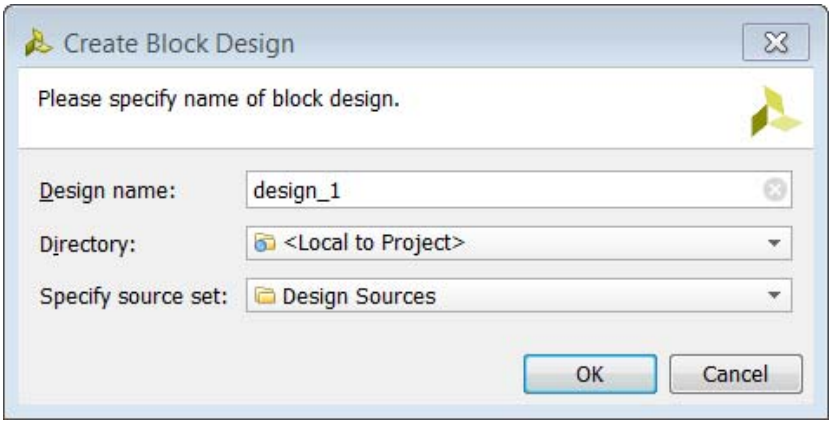


Figure 3-2: **Create Block Design Dialog Box**

The Block Design window opens (Figure 3-3).

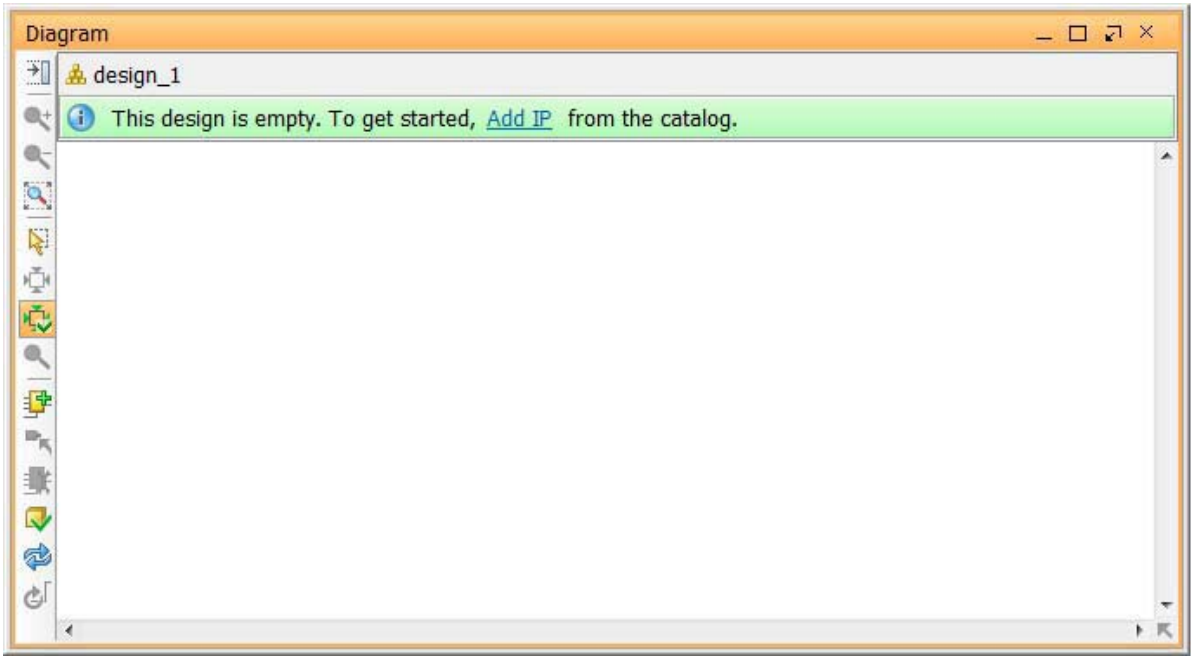
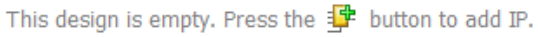


Figure 3-3: **The Block Design Canvas**

Within the empty design, there is a link to Add IP from the IP Catalog. You can also right-click in the canvas to open an option to add IP.

3. Within the empty design, there is an Add IP icon to add IP from the IP Catalog. You can also right-click in the canvas and then select Add IP from the context menu to add IP.



A Search box opens to let you search for and select the MicroBlaze processor, as shown in Figure 3-4.

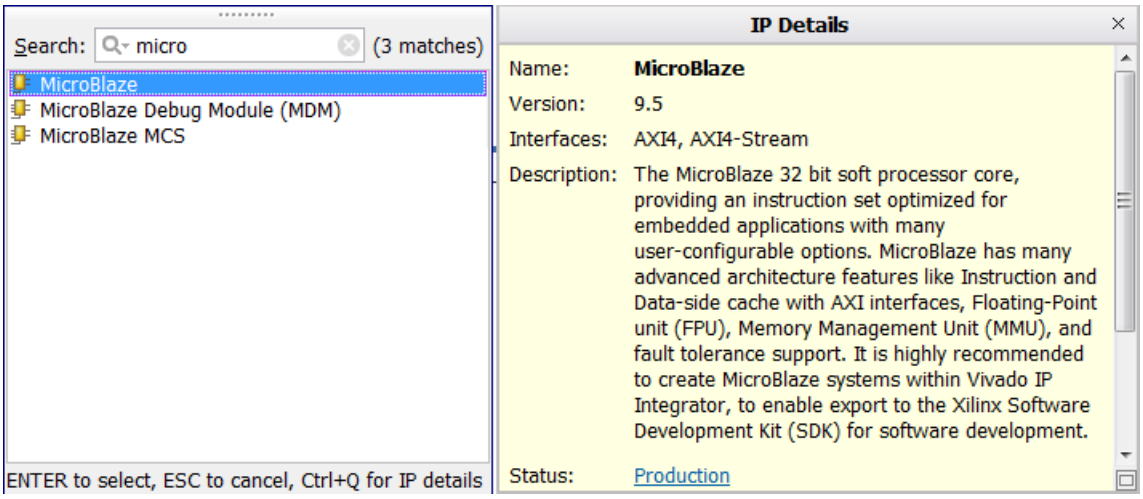


Figure 3-4: **Search the IP Catalog for MicroBlaze**

When you select the MicroBlaze IP, the Vivado IP integrator adds the IP to the design, and a graphical representation of the processing system displays, as shown in Figure 3-5.

Figure 3-5: **MicroBlaze Processor in Block Design Canvas**

Tcl Command:

```
create_bd_cell -type ip -vlnv xilinx.com:ip:microblaze:9.5 microblaze_0
```

4. Double-click the MicroBlaze IP in the canvas to invoke the Re-customize IP process, which displays the Re-customize IP for the MicroBlaze processor, dialog box.

MicroBlaze Configuration Window

The MicroBlaze Configuration wizard provides:

- A template-based configuration dialog box for one-click configuration
- Estimates of MicroBlaze relative area, frequency, and performance, based on options set in the dialog boxes, giving immediate feedback
- Guidance through the configuration process
- Tool tips for all configuration options to understand the effect of each option
- Direct access to all options in the tabbed interface using the **Advanced** button

The MicroBlaze Configuration wizard has the following wizard pages, which enable based on the selected General Settings options:

- **Configuration Wizard**: First page that showing template selection and general settings.
- **General**: Selection of execution units, optimization that is always shown
- **Exceptions**: Exceptions to enable, which is shown if exceptions are selected on the first page
- **Debug**: Number of breakpoints and watchpoints, which is shown if debug is enabled
- **Cache**: Cache settings, which is shown if caches are selected
- **MMU**: MMU settings, which is shown if memory management is selected
- **Buses**: Bus settings. Last page, always shown

Figure 3-6 shows the Welcome page of the MicroBlaze Configuration wizard.

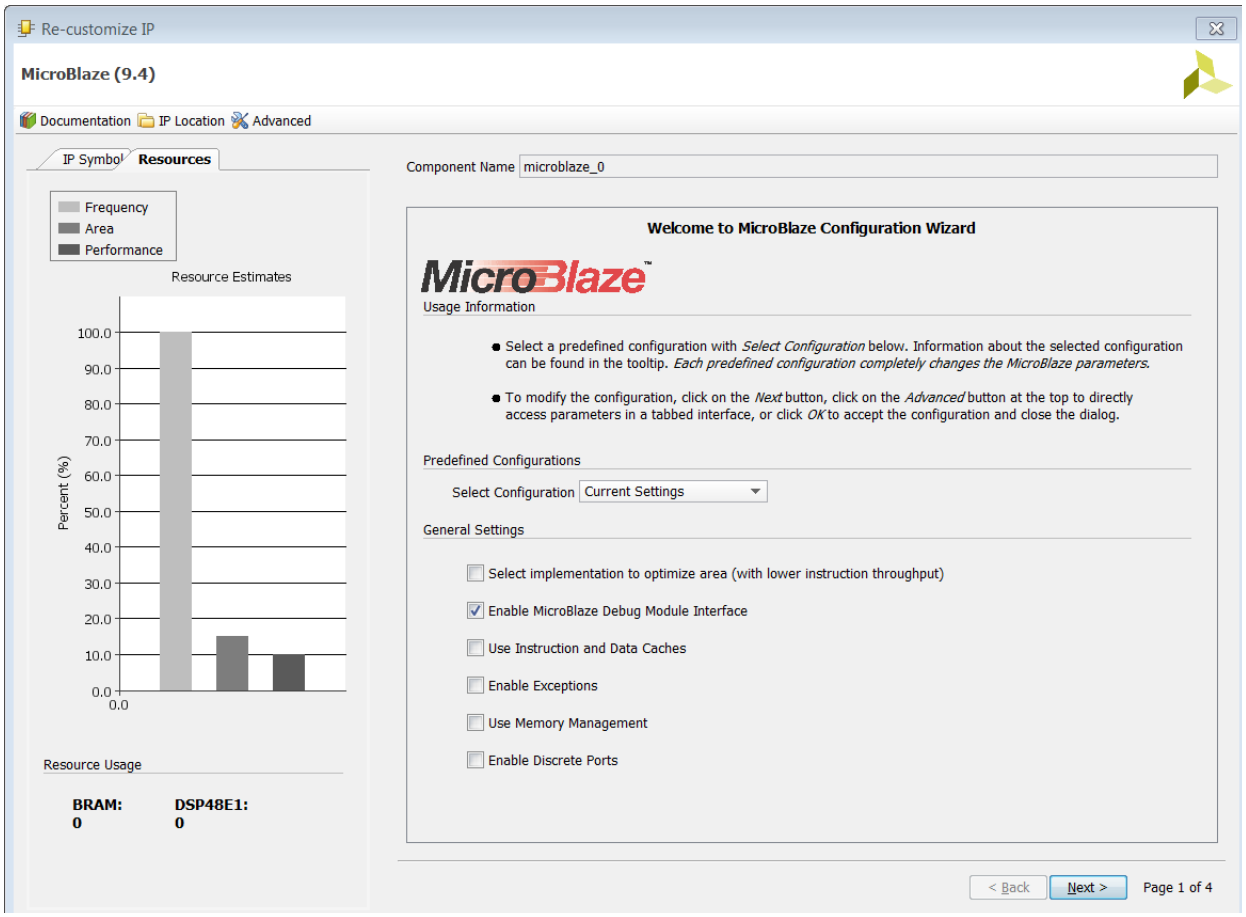


Figure 3-6: **MicroBlaze Configuration Wizard**

The left of the dialog box shows the relative values of the frequency, area and performance for the current settings.

- **Frequency:** This value is the estimated frequency percentage relative to the maximum achievable frequency with this architecture and speed grade, which gives an indication of the relative frequency that can be achieved with the current settings.

  Note: This is an estimate based on a set of predefined benchmark systems, which can deviate up to 30% from the actual value. Do not take this estimation as a guarantee that the system can reach a corresponding frequency.

- **Area:** This value is the estimated area percentage in LUTs relative to the maximum area using this architecture, which gives an indication of the relative MicroBlaze area achievable with the current settings.

  Note: This is an estimate, which can deviate up to 5% from the actual value. Do not take this estimation as a guarantee that the implemented area matches this value.

- **Performance:** This value indicates the relative MicroBlaze performance achievable with the current settings, relative to the maximum possible performance.

Note: This is an estimate based on a set of benchmarks, and actual performance can vary significantly depending on the user application.

- **BRAMs:** This value is the total number of block RAMs used by MicroBlaze. The instruction and data caches, and the branch target cache use block RAMS, and well as the Memory Management Unit (MMU), which uses one block RAM in virtual or protected mode.
- **DSP48** or **MULT18:** This value is the total number of DSP48 or MULT18 used by MicroBlaze. The integer multiplier, and the Floating Point Unit (FPU) use this total value to implement float multiplication.

MicroBlaze Configuration Wizard Welcome Page

The simplest way to use the MicroBlaze™ Configuration wizard is to select one of the six predefined templates, each defining a complete MicroBlaze configuration. You can use a predefined template as a starting point for a specific application, using the wizard to refine the configuration, by adapting performance, frequency, or area.

When you modify an option, you received direct feedback that shows the estimated relative change in performance, frequency, and area in the information display. The options are:

- **Minimum Area:** The smallest possible MicroBlaze core. No caches or debug.
- **Maximum Performance:** Maximum possible performance. Large caches and debug, as well as all execution units.
- **Maximum Frequency:** Maximum achievable frequency. Small caches and no debug, with few execution units.
- **Linux with MMU:** Settings suitable to get high performance when running Linux with MMU. Memory Management enabled, large caches and debug, and all execution units.
- **Low-end Linux with MMU:** Settings corresponding to the MicroBlaze Embedded Reference System. Provides suitable settings for Linux development on low-end systems. Memory Management enabled, small caches and debug.
- **Typical:** Settings giving a reasonable compromise between performance, area, and frequency. Suitable for standalone programs, and low-overhead kernels. Caches and debug enabled.

Figure 3-7 shows the Predefined Configurations in the Configuration wizard.

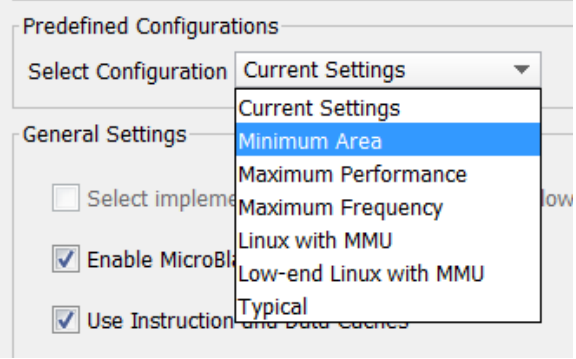


Figure 3-7: **MicroBlaze Predefined Configuration Settings**

General Settings

If a pre-defined template is not used, you can select the options from the pages, which are available for fine-tuning the MicroBlaze processor, based on your design needs. As you position the mouse over these different options, a tooltip informs you what the particular option means. The following bullets detail these options.

- **Select implementation to optimize area** (with lower instruction throughput): Enables area optimized MicroBlaze. When this parameter is set, the implementation optimizes area, particularly by reducing the pipeline from five stages to three.

RECOMMENDED: *It is recommended to enable optimization on architectures with limited resources such as Artix®-7 devices. However, if performance is critical, this parameter should not be set, because some instructions require additional clock cycles to execute.*

Note: You cannot use the Memory Management Unit (MMU), Branch Target Cache, Instruction Cache Streams, Instruction Cache Victims, Data Cache Victims, and AXI Coherency Extension (ACE) with area optimization.

- **Enable MicroBlaze Debug Module Interface:** Enable debug to be able to download and debug programs using Xilinx Microprocessor Debugger.

RECOMMENDED: *Unless area resources are very critical, it is recommended that debugging be always enabled.*

- **Use Instruction and Data Caches:** You can use MicroBlaze with an optional instruction cache for improved performance when executing code that resides outside the LMB address range. The instruction cache has the following features:
  - Direct mapped (1-way associative)
  - User selectable cacheable memory address range
  - Configurable cache and tag size

- Caching over AXI4 interface (M_AXI_IC) or CacheLink (XCL) interface
- Option to use 4 or 8 word cacheline
- Cache on and off controlled using a bit in the MSR
- Optional WIC instruction to invalidate instruction cache lines
- Optional stream buffers to improve performance by speculatively prefetching instructions
- Optional victim cache to improve performance by saving evicted cache lines
- Optional parity protection; invalidates cache lines if Block RAM bit error is detected
- Optional data width selection to either use 32 bits, an entire cache line, or 512 bits

Activating caches significantly improves performance when using external memory, even if you must select small cache sizes to reduce resource usage.

- **Enable Exceptions:** Enables exceptions when using an operating system with exception support, or when explicitly adding exception handlers in a standalone program.
- **Use Memory Management:** Enables Memory Management if planning to use an operating system - such as Linux -with support for virtual memory of memory protection.

  Note: When you enable area optimized MicroBlaze or stack protection, the Memory Management Unit is not available.
- **Enable Discrete Ports:** Enables discrete ports on the MicroBlaze instance, which is useful for:
  - Generating software breaks (Ext_BRK, Ext_NM_BRK)
  - Managing processor sleep and wakeup (Sleep, Wakeup, Dbg_Wakeup)
  - Handling debug events (Debug_Stop, MB_Halted)
  - Signaling error when using fault tolerance (MB_Error)

MicroBlaze Configuration Wizard General Page

Figure 3-8 shows the General Page of the MicroBlaze Configuration wizard.

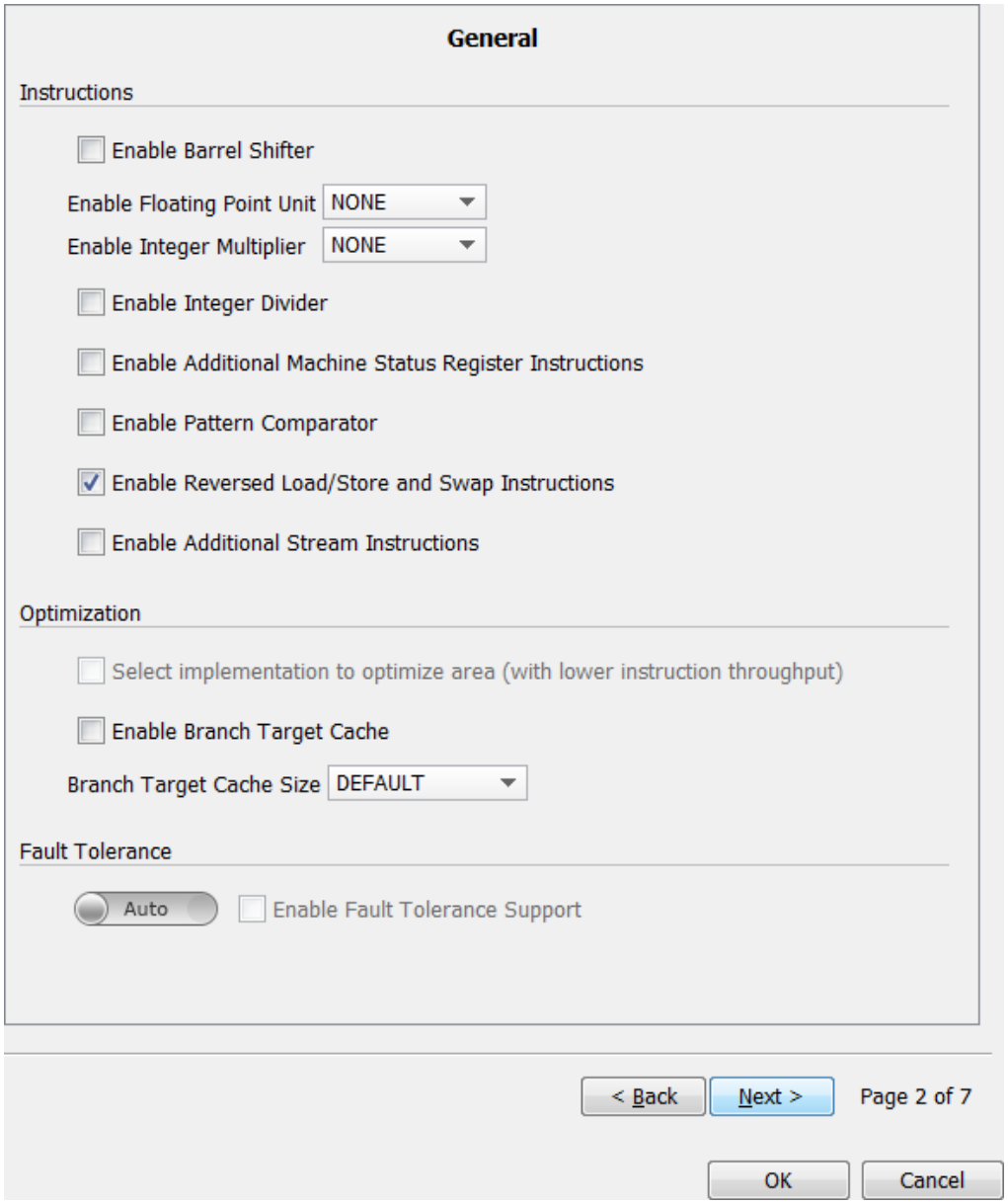


Figure 3-8: **General Page of the MicroBlaze Configuration Wizard**

Instructions

- **Enable Barrel Shifter:** Enables a hardware barrel shifter in MicroBlaze. This parameter enables the instructions `bsrl`, `bsra`, `bsll`, `bsrli`, `bsrai`, and `bslli`. Enabling the barrel shifter can dramatically improve the performance of an application, but increases the size of the processor. The compiler uses the barrel shift instructions automatically if this parameter is enabled.
- **Enable Floating Point Unit:** Enables a single-precision Floating Point Unit (FPU) based on the IEEE-754 standard. Using the FPU significantly improves the single-precision,

floating point performance of the application and significantly increases the size of MicroBlaze.

Setting this parameter to BASIC enables the instructions `fadd`, `frsub`, `fmul`, `fdiv`, and `fcmp`. Setting it to EXTENDED also enables the instruction `flt`, `finit`, and `fsqrt`. The compiler automatically uses the FPU instructions corresponding to setting of this parameter.

- **Enable Integer Multiplier:** Enables a hardware integer multiplier in MicroBlaze. This parameter enables the instructions `mul` and `muli` when set to MUL32.

  When set to `MUL64`, this enables the additional instructions `mulh`, `mulhu`, and `mulhsu` for 64-bit multiplication. This parameter can be set to NONE to free up MUL or DSP48 primitives in the device for other uses. Setting this parameter to NONE has a minor effect on the area of the MicroBlaze processor. When this parameter is enabled, the compiler uses the `mul` instructions automatically.
- **Enable Integer Divider:** Enables a hardware integer divider in MicroBlaze. This parameter enables the instructions, `idiv` and `idivu`. Enabling this parameter can improve the performance of an application that performs integer division, but increases the size of the processor. When this parameter is enabled, the compiler uses the `idiv` instructions automatically.
- **Enable Additional Machine Status Register Instructions:** Enables additional machine status register (MSR) instructions for setting and clearing bits in the MSR. This parameter enables the instructions `msrset` and `msrclr`. Enabling this parameter improves the performance of changing bits in the MSR.
- **Enable Pattern Comparator:** Enables pattern compare instructions `pcmpbf`, `pcmpeq`, and `pcmpne`. The pattern compare bytes find (`pcmpbf`) instructions return the position of the first byte that matches between two words and improves the performance of string and pattern matching operations. The SDK libraries use the `pcmpbf` instructions automatically when this parameter is enabled.
  - The `pcmpeq` and `pcmpne` instructions return 1 or 0 based on the equality of the two words. These instructions improve the performance of setting flags and the compiler uses them automatically.
  - Selecting this option also enables count leading zeroes instruction, `clz`. The `clz` instruction can improve performance of priority decoding, and normalization.
- **Enable Reversed Load/Store and Swap Instructions:** Enables reversed load/store and swap instructions `lbur`, `lhur`, `lwr`, `sbr`, `shr`, `swr`, `swapb`, and `swaph`. The reversed load/store instructions read or write data with opposite endianness, and the swap instructions allow swapping bytes or half-words in registers. These instructions are mainly useful to improve performance when dealing with big-endian network access with a little-endian MicroBlaze.
- **Enable Additional Stream Instructions:** Provides additional functionality when using AXI4-Stream links, including dynamic access instruction `GETD` and `PUTD` that use

registers to select the interface. The instructions are also extended with variants that provide:

- Atomic `GET`, `GETD`, `PUT`, and `PUTD` instructions
- Test-only `GET` and `GETD` instructions
- `GET` and `GETD` instructions that generate a stream exception if the control bit is not set

IMPORTANT: *The stream exception must be enabled to use these instructions, and at least one stream link must be selected.*

Optimization

Select implementation to optimize area (with lower instruction throughput):This option is the same as in the General Settings options. Enable Branch Target Cache: When set, implements the branch target, which improves branch performance by predicting conditional branches and caching branch targets.

Note: To be able to use the Branch Target Cache, do not enable area optimization.

Fault Tolerance

- **Enable Fault Tolerance Support:** When enabled, MicroBlaze protects internal Block RAM with parity, and supports Error Correcting Codes (ECC) in LMB block RAM, including exception handling of ECC errors. This prevents a bit flip in block RAM from affecting the processor function.
  - If this value is auto-computed (by not overriding it), fault tolerance is automatically enabled in MicroBlaze when ECC is enabled in connected LMB BRAM controllers.
  - If fault tolerance is explicitly disabled, the IP integrator tool enables ECC automatically in connected LMB BRAM Controllers.
  - If fault tolerance is explicitly disabled, ECC in connected LMB BRAM controllers is not affected.

MicroBlaze Configuration Wizard Exception Page

Figure 3-9 shows the MicroBlaze exception options page.

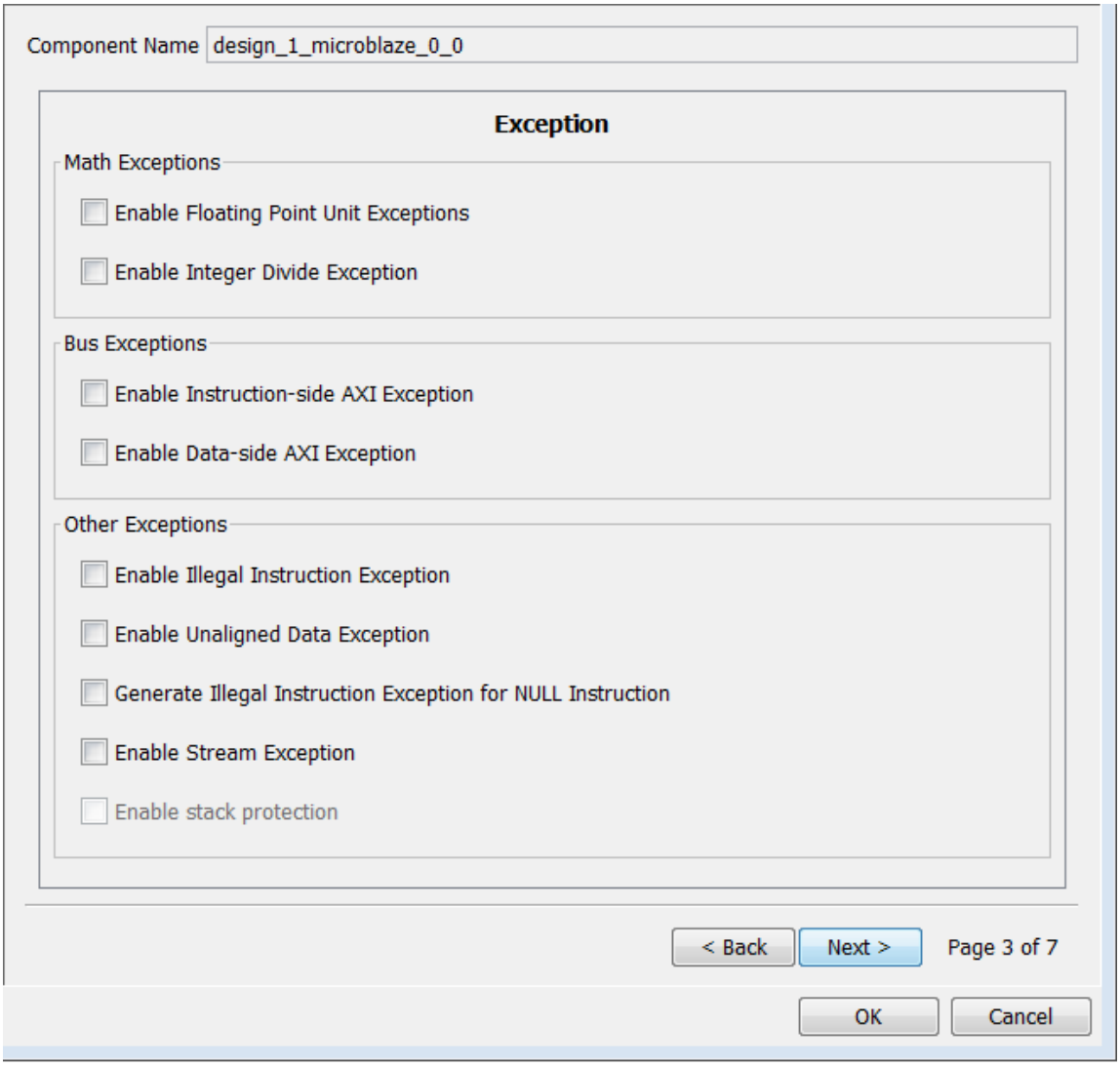


Figure 3-9: **Exception Options in the MicroBlaze Configuration Wizard**

IMPORTANT: *You must provide your own exception handler.*

Math Exceptions

- **Enable Floating Point Unit Exceptions:** Enables exceptions generated by the Floating Point Unit (FPU). The FPU throws exceptions for all of the IEEE standard conditions: underflow, overflow, divide-by-zero, and illegal operations. In addition, the MicroBlaze FPU throws a de-normalized operand exception.

- **Enable Integer Divide Exception:** Causes an exception if the divisor (rA) provided to the idiv or idivu instruction is zero, or if an overflow occurs for idiv.

Bus Exceptions

- **Enable Instruction-side AXI Exception:** Causes an exception if there is an error on the instruction-side AXI bus.
- **Enable Data-side AXI Exception:** Causes an exception if there is an error on the data-side AXI bus.

Other Exceptions

- **Enable Illegal Instruction Exception:** Causes an exception if the major opcode is invalid.
- **Enable Unaligned Data Exception:** When enabled, the tools automatically insert software to handle unaligned accesses.
- **Generated Illegal Instruction Exception for NULL Instructions:** MicroBlaze compiler does not generate, nor do SDK libraries use the NULL instruction code (0x00000000). This code can only exist legally if it is hand-assembled. Executing a NULL instruction normally means that the processor has jumped outside the initialized instruction memory.

  If C_OPCODE_0x_ILLEGAL is set, MicroBlaze traps this condition; otherwise, it treats the command as a NOP. This setting is only available if you have enabled Illegal Instruction Exception.

- **Enable Stream Exception:** Enables stream exception handling for Advanced eXtensible Interface (AXI) read accesses.

IMPORTANT: *You must enable additional stream instructions to use stream exception handling.*

- **Enable Stack Protection:** Ensures that memory accesses using the stack pointer (R1) to ensure they are within the limits set by the Stack Low Register (SLR) and Stack High Register (SHR). If the check fails with exceptions enabled, a Stack Protection Violation exception occurs. The Xilinx Microprocessor Debugger (XMD) also reports if the check fails.

MicroBlaze Configuration Wizard Cache Page

Figure 3-10 shows the Cache options page for the MicroBlaze Configuration.

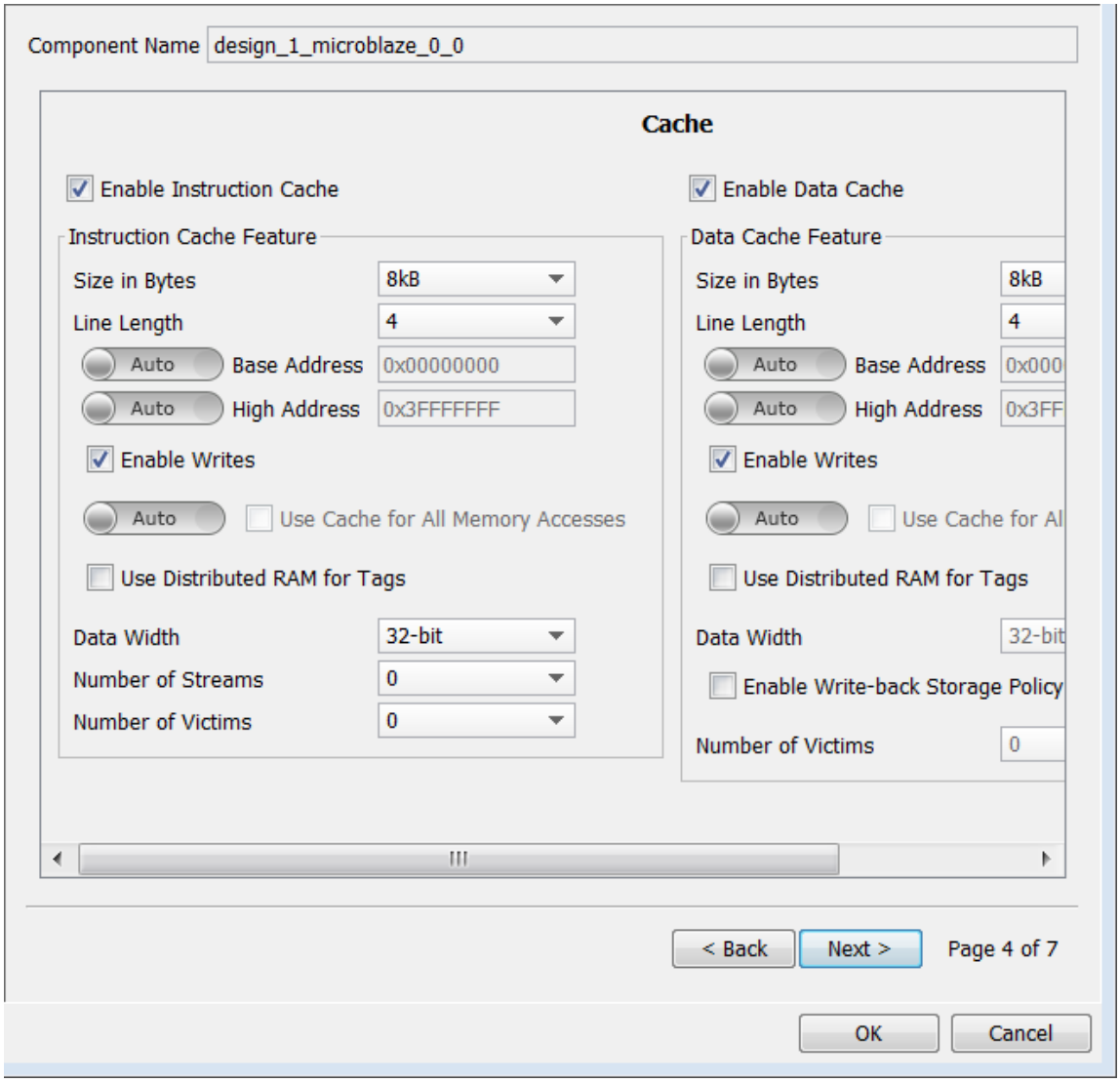


Figure 3-10: **Cache Options Page of the MicroBlaze Configuration Wizard**

- **Enable Instruction Cache:** Uses this cache only when it is also enabled in software by setting the instruction cache enable (ICE) bit in the machine status register (MSR).
- Instruction Cache Features:
  - **Size in Bytes:** Specifies the size of the instruction cache if C_USE_ICACHE is enabled. Not all architectures permit all sizes.
  - **Line Length:** Select between 4 or 8 word cache line length for cache miss-transfers from external instruction memory.

- **Base Address:** Specifies the base address of the instruction cache. This parameter is used only if C_USE_ICACHE is enabled.
- **High Address:** Specifies the high address of the instruction cache. This parameter is used only if C_USE_ICACHE is enabled.
- **Enable Writes:** When enabled, one can invalidate instruction cache lines with the wic instruction. This parameter is used only if C_USE_ICACHE is enabled.
- **Use Cache for All Memory Accesses:** When enabled, uses the dedicated cache interface on MicroBlaze is for all accesses within the cacheable range to external instruction memory, even when the instruction cache is disabled.

  Otherwise, the instruction cache uses the peripheral AXI for these accesses when the instruction cache is disabled. When enabled, an external memory controller must provide only a cache interface MicroBlaze instruction memory. Enable this parameter when using AXI Coherency Extension (ACE).
- **Use Distributed RAM for Tags:** Uses the instruction cache tags to hold the address and a valid bit for each cacheline. When enabled, the instruction cache tags are stored in Distributed RAM instead of Block RAM. This saves Block RAM, and can increase the maximum frequency.
- **Data Width:** Specifies the instruction cache bus width when using AXI Interconnect. The width can be set to:
- **32-bit:** Bursts are used to transfer cache lines for 32-bit words depending on the cache line length,
- **Full Cacheline:** A single transfer is performed for each cache line, with data width 128 or 256 bits depending on cache line length
- **512-bit:** Performs a single transfer, but uses only 128 or 256 bits, depending on cacheline length.

  The two wide settings require that the cache size is at least 8 KB or 16KB depending upon cache line length. To reduce the AXI interconnect size, this setting must match the interconnect data width. In most cases, you can obtain the best performance with the wide settings.

  Note: This setting is not available with area optimization, AXI Coherency Extension (ACE), or when you enable fault tolerance.
- **Number of Streams:** Specifies the number of stream buffers used by the instruction cache. A stream buffer is used to speculatively pre-fetch instructions, before the processor requests them. This often improves performance, because the processor spends less time waiting for instruction to be fetched from memory.

  Note: To be able to use instruction cache streams, do not enable area optimization or AXI Coherency Extension (ACE).
- **Number of Victims:** Specifies the number of instruction cache victims to save. A victim is a cacheline that is evicted from the cache. If no victims are saved, all evicted lines must be read from memory again, when they are needed. By saving

the most recent lines, they can be fetched much faster, thus improving performance.

RECOMMENDED: *It is possible to save 2, 4, or 8 cachelines. The more cachelines that are saved, the better performance becomes. The recommended value is 8 lines.*

Note: To be able to use instruction cache victims, do not enable area optimization or AXI Coherency Extension (ACE).

MicroBlaze Configuration Wizard MMU Page

Figure 3-11 shows the MMU page of the MicroBlaze Configuration.

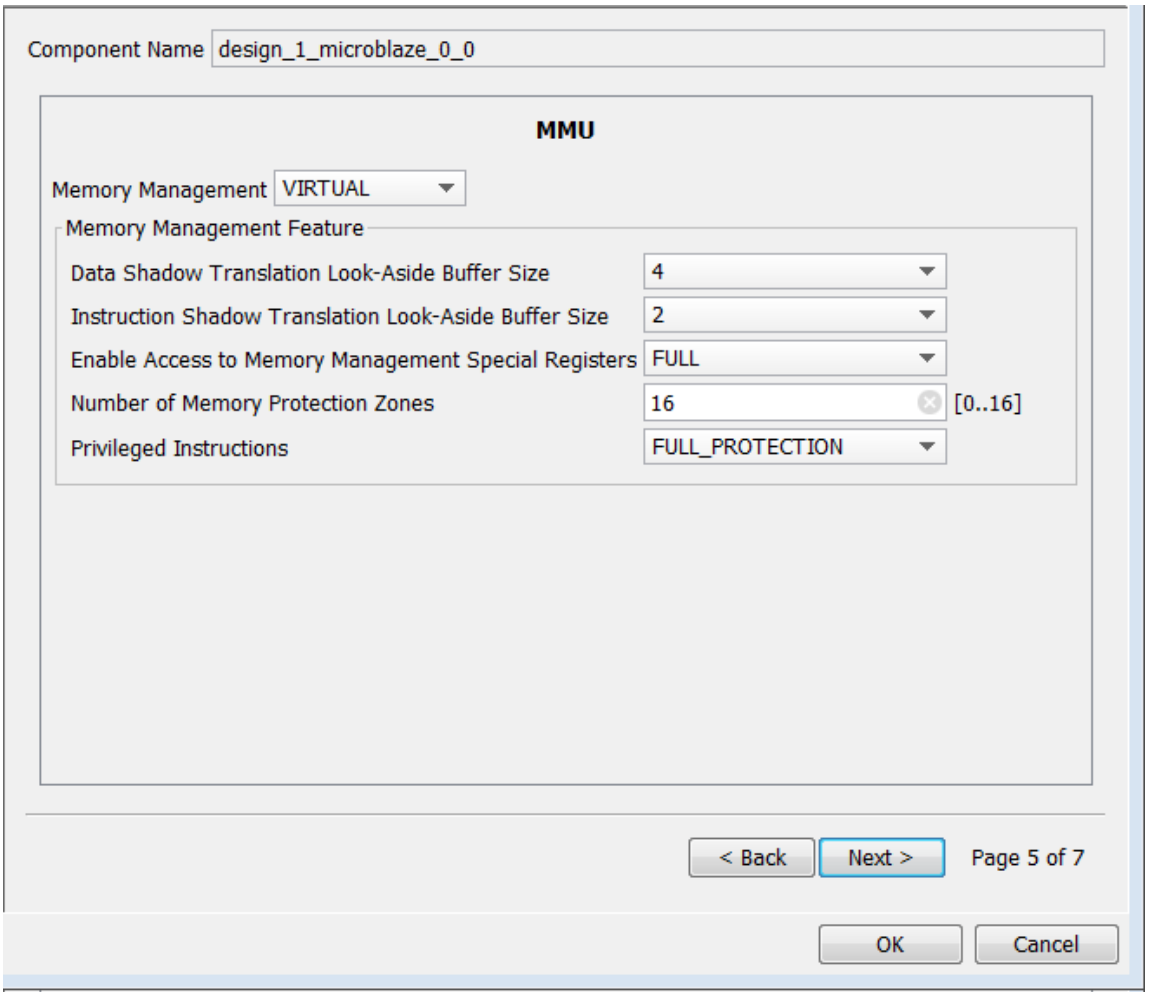


Figure 3-11: **MicroBlaze Configuration Wizard MMU Page**

Memory Management

The **Memory Management** field specifies the implementation of the Memory Management Unit (MMU).

- To disable the MMU, set this parameter to None (0), which is the default.
- To enable only the User Mode and Privileged Mode instructions, set this parameter to Usermode (1). To enable Memory Protection, set the parameter to Protection (2).
- To enable full MMU functionality, including virtual memory address translation, set this parameter to Virtual (3).

When Usermode is set, it enables the Privileged Instruction exception. When Protection or Virtual is set, it enables the Privileged Instruction exception and the four MMU exceptions (Data Storage, Instruction Storage, Data TLB Miss, and Instruction TLB Miss).

Memory Management Features:

- **Data Shadow Translation Look-Aside Buffer Size:** Defines the size of the instruction shadow Translation Look-Aside Buffer (TLB). This TLB caches data address translation information, to improve performance of the translation. The selection is a trade-off between smaller size and better performance: the default value is 4.
- **Instruction Shadow Translation Look-Aside Buffer Size:** Defines the size of the instruction shadow Translation Look-Aside Buffer (TLB). This TLB caches instruction address translation information to improve performance of the translation. The selection is a trade-off between smaller size and better performance: the default value is 2.
- **Enable Access to Memory Management Special Registers:** Enables access to the Memory Management Special Register using the MFS and MTS instructions:
  - Minimal (0) only allows writing TLBLO, TLBHI, and TLBX.
  - Read (1) adds reading to TLBLO, TLBHI, TLBX, PID, and ZPR.
  - Write (2) allows writing all registers, and reading TLBX.
  - Full (3) adds reading of TLBLO, TLBHI, TLBX, PID, and ZPR.

  In many cases, it is not necessary for the software to have full read access. For example, this is the case for Linux Memory Management code. It is then safe to set access to Write, to save area. When using static memory protection, access can be set to Minimal, because the software then has no need to use TLBSX, PID, and ZPR.
- **Number of Memory Protection Zones:** Specifies the number of memory protection zones to implement. In many cases memory management software does not use all available zones. For example, the Linux Memory Management code only uses two zones. In this case, it is safe to reduce the number of implemented zones, to save area.
- **Privileged Instructions:** Specifies which instructions to allow in User Mode.

- The Full Protection (0) setting ensures full protection between processes.
- The Allow Stream Instructions (1) setting makes it possible to use AXI4-Stream instructions in User Mode.

CAUTION! *It is strongly discouraged to change this setting from Full Protection, unless it is necessary for performance reasons.*

MicroBlaze Configuration Wizard Debug Page

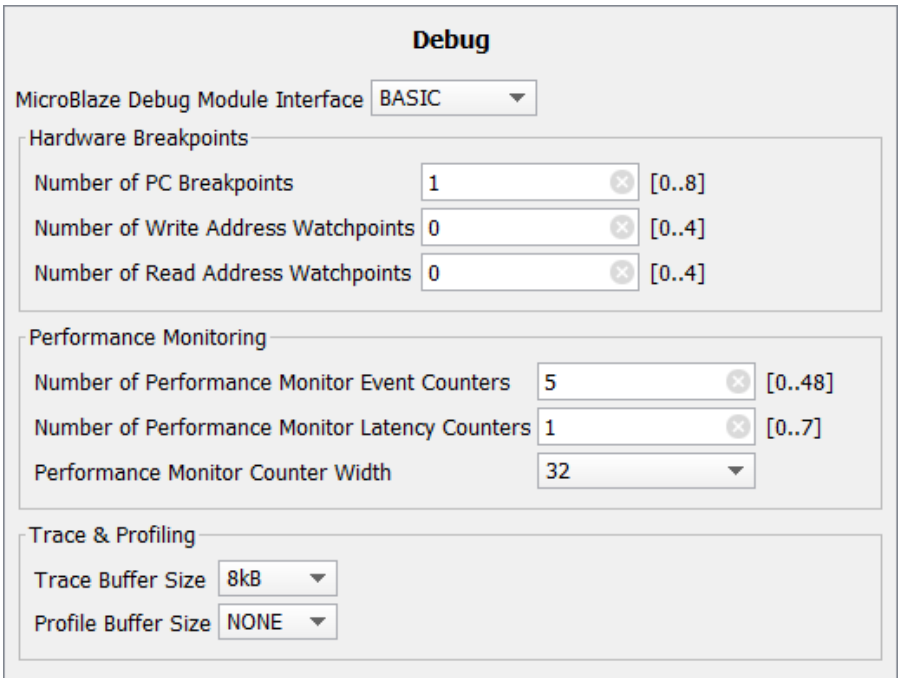


Figure 3-12: **MicroBlaze Configuration Wizard Debug Page**

Debug Options

MicroBlaze Debug Module Interface: **BASIC** mode enables the MicroBlaze Debug Module (MDM) interface to MicroBlaze for debugging. With this option, you can use Xilinx Microprocessor Debugger (XMD) to debug the processor over the Joint Test Action Group (JTAG) boundary-scan interface. The **EXTENDED** mode enables enhanced debug features of MicroBlaze such as Cross-Trigger, Trace and Profiling. You can disable this option after you finish debugging to reduce the size of MicroBlaze by setting this option to **NONE**.

Hardware Breakpoints:

- **Number of PC Breakpoints:** Specifies the number of program counter (PC) hardware breakpoints for debugging. This parameter controls the number of hardware breakpoints Xilinx Microprocessor Debugger (XMD) can set. This option only has meaning if C_DEBUG_ENABLED is on. The MicroBlaze processor takes a noticeable frequency hit the larger this parameter is set.
- **Number of Write Address Watchpoints:** Specifies the number of write address breakpoints for debugging. This parameter controls the number of write watchpoints

Xilinx Microprocessor Debugger (XMD) can set. This option only has meaning if C_DEBUG_ENABLED is on. MicroBlaze take a noticeable frequency hit, the larger this parameter is set.

- **Number of Read Address Watchpoints:** Specifies the number of read address breakpoints for debugging. This parameter controls the number of read watch points Xilinx Microprocessor Debugger (XMD) can set. This option only has meaning if C_DEBUG_ENABLED is on. The MicroBlaze processor takes a noticeable frequency hit the larger this parameter is set.

RECOMMENDED: *It is recommended that these two options be set to 0 if you are not using watch points for debugging.*

Performance Monitoring

With extended debugging, MicroBlaze provides the following performance monitoring counters to count various events and to measure latency during program execution:

- C_DEBUG_EVENT_COUNTERS - Use this to configure the event counters.
- C_DEBUG_LATENCY_COUNTERS - Use this to configure the latency counters.
- C_DEBUG_COUNTER_WIDTH - Use this to set the counter width to 32, 48 or 64 bits.

With the default configuration, the counter width is set to 32 bits and there are five event counters and one latency counter.

Trace and Profiling

With extended debugging, MicroBlaze provides program trace, storing information in the Embedded Trace Buffer to enable program execution tracing.

Use the parameter C_DEBUG_TRACE_SIZE to configure the size of the trace buffer from 8KB to 128KB.

By setting C_DEBUG_TRACE_SIZE to `0` (None), program trace is disabled.

MicroBlaze Configuration Wizard Buses Page

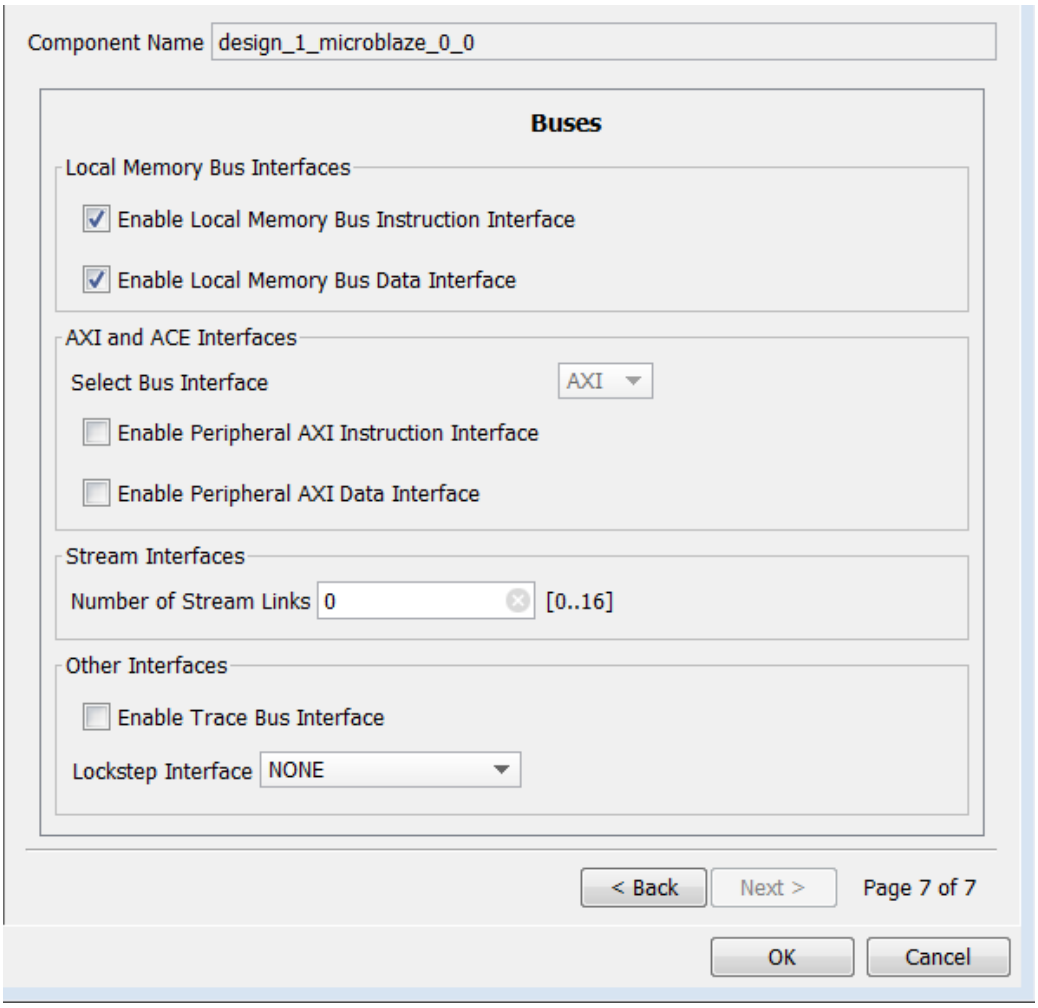


Figure 3-13: **MicroBlaze Configuration Wizard Buses Page**

Local Memory Bus Interfaces:

- **Enable Local Memory Bus Instruction Interface:** Enables LMB instruction interface. When this instruction is set, the Local Memory Bus (LMB) instruction interface is available. A typical MicroBlaze system uses this interface to provide fast local memory for instructions. Normally, it connects to an LMB bus using an LMB Bus Interface Controller to access a common Block RAM.
- **Enable Local Memory Bus Data Interface:** Enables LMB data interface. When this parameter is set, the Local Memory Bus (LMB) data interface is available. A typical MicroBlaze system uses this interface to provide fast local memory for data and vectors. Normally, it connects to an LMB bus using an LMB Bus Interface Controller to access a common Block RAM.

AXI and ACE Interfaces:

- **Select Bus Interface:** When this parameter is set to AXI, then AXI is selected for both peripheral and cache access. When this parameter is set to ACE, then AXI is selected for peripheral access and AXI Coherency Extension (ACE) is selected for cache access, providing cache coherency support.

  Note: To be able to use ACE, area optimization, write-back data cache, instruction cache streams or victims, and cache data widths other than 32-bit must not be set. You must set Use Cache for All Memory Accesses for both caches.

- **Enable Peripheral AXI Interface Instruction Interface:** When this parameter is set, the peripheral AXI4-Lite instruction interface is available. In many cases, this interface is not needed, in particular if the Instruction Cache is enabled and C_ICACHE_ALWAYS_USED is set.
- **Enable Peripheral AXI Data Interface:** When this parameter is set, the peripheral AXI data interface is available. This interface usually connects to peripheral I/O using AXI4-Lite, but it can be connected to memory also. If you enable exclusive access, the AXI4 protocol is used.

Stream Interfaces:

- **Number of Stream Links:** Specifies the number of pairs of AXI4-Stream link interfaces. Each pair contains a master and a slave interface. The interface provides a unidirectional, point-to-point communication channel between MicroBlaze and a hardware accelerator or coprocessor. This is a low-latency interface, which provides access between the MicroBlaze register file and the FPGA fabric.

Other Interfaces:

- **Enable Trace Bus Interface:** When this parameter is set, the Trace bus interface is available. This interface is useful for debugging, execution statistics and performance analysis. In particular, connecting interface to a ChipScope™ Logic Analyzer (ILA) allows tracing program execution with clock cycle accuracy.
- **Lockstep Interface:** When you enable lockstep support, two MicroBlaze cores run the same program in lockstep, and you can compare their outputs to detect errors.
  - When set to NONE, no lockstep interfaces are enabled.
  - When set to LOCKSTEP_MASTER, it enables the Lockstep_Master_Out and Lockstep_Out output ports.
  - When set to LOCKSTEP_SLAVE, it enables the Lockstep_Slave_in input port and Lockstep_Out output ports, and the C_LOCSTEP_SLAVE parameter is set to 1.

Cross-Trigger Feature of MicroBlaze Processors

With basic debugging, cross trigger support is provided by two signals: DBG_STOP and MB_Halted.

- When the DBG_STOP input is set to 1, MicroBlaze halts after a few instructions. XMD detects that MicroBlaze has halted, and indicates where the halt occurred. The signal can be used to halt MicroBlaze processors at any external event, such as when an Integrated Logic Analyzer (ILA) is triggered.
- The MB_Halted output signal is set to 1 whenever the MicroBlaze processor is halted, such as after a breakpoint or watchpoint is hit, after a stop XMD command, or when the DBG_STOP input is set. The output is cleared when MicroBlaze execution is resumed by an XMD command.

  The DBG_STOP and MB_Halted pins are hidden. To see those pins, you must enable the **Enable Discrete Ports** option in the MicroBlaze Configuration dialog box, as shown in Figure 3-14.

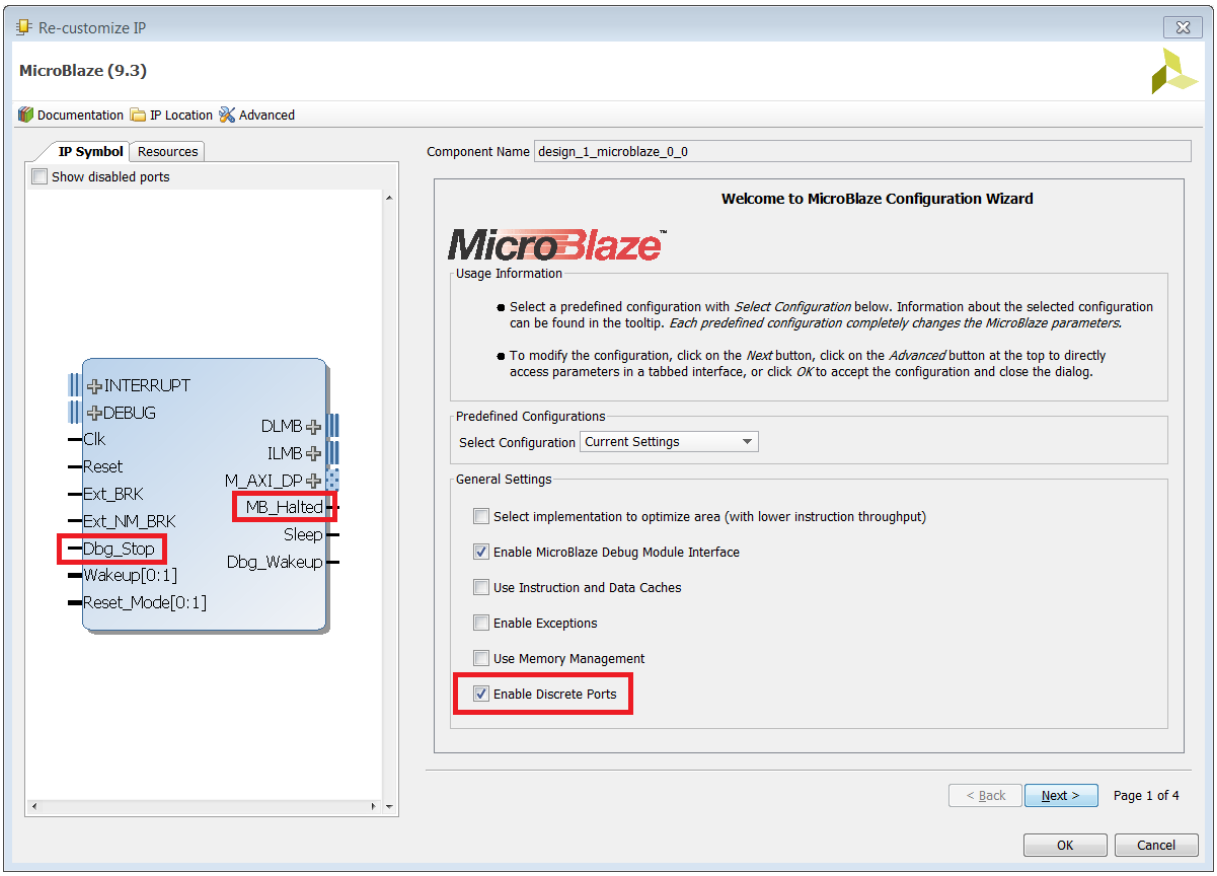


Figure 3-14: **Enable Discrete Ports Option**

You can use the MB_Halted signal to trigger an Integrated logic analyzer, or halt other MicroBlaze cores in a multiprocessor system by connecting the signal to their DBG_STOP inputs.

With extended debugging, cross trigger support is available in conjunction with the MDM. The MDM provides programmable cross triggering between all connected processors, as well as external trigger inputs and outputs. For details, see the *MicroBlaze Debug Module (MDM) v3.1 Product Guide (PG115)* [Ref 8].

MicroBlaze can handle up to eight cross trigger actions. Cross trigger actions are generated by the corresponding MDM cross trigger outputs, connected via the Debug bus.

To enable extended debug, set the MicroBlaze Debug Module Interface to **EXTENDED** in the Debug Page of the MicroBlaze Configuration Wizard.

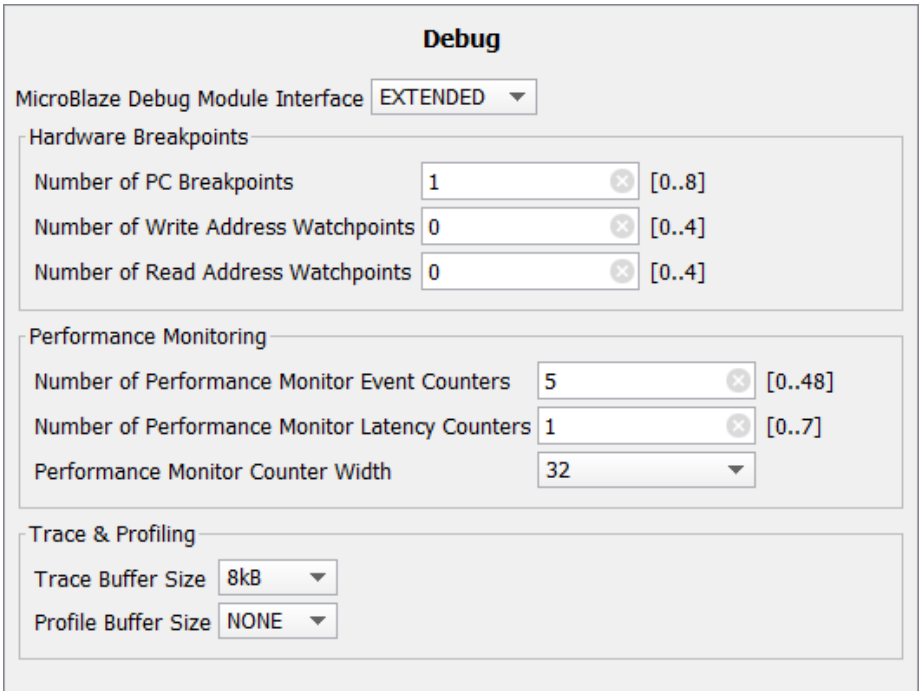


Figure 3-15: **Enable EXTENDED Debug for MicroBlaze**

You can also set the extended debug option when running Block Automation for the MicroBlaze processor.

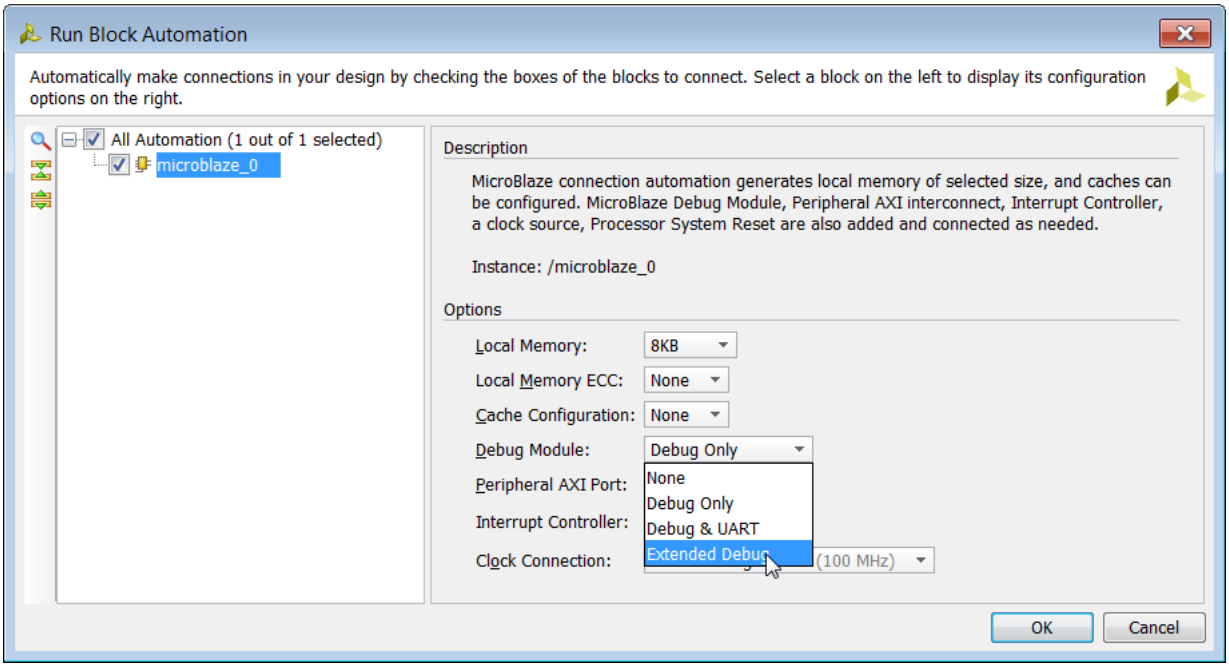


Figure 3-16: **Extended Debug Option**

Next, in the MicroBlaze Debug Module (MDM), the **Enable Cross Trigger** check box is enabled.

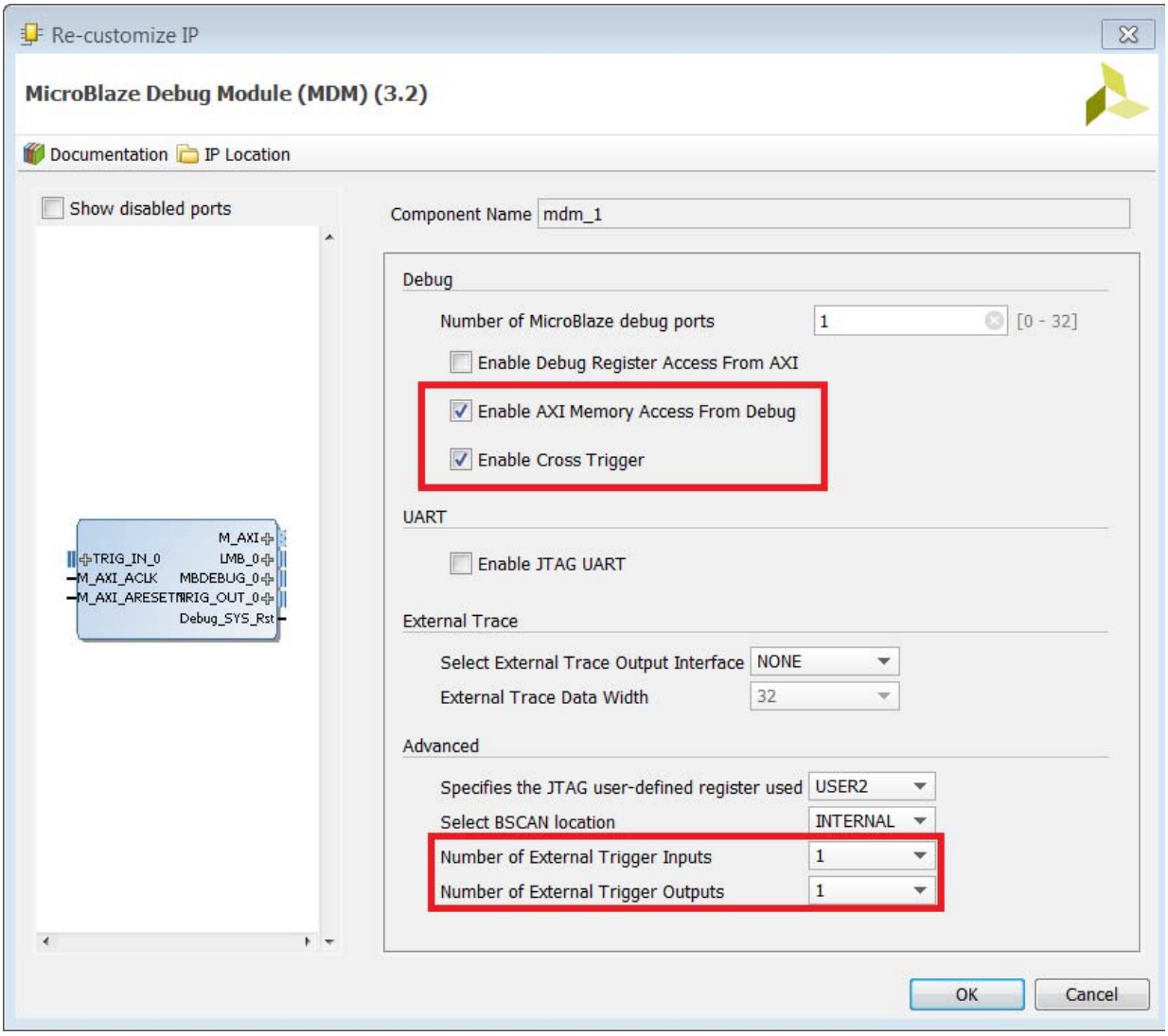


Figure 3-17: **Enable Cross Trigger Check Box in MDM**

You can also select up to four external trigger inputs and external trigger outputs. When enabled, the block design updates to show the MDM details.

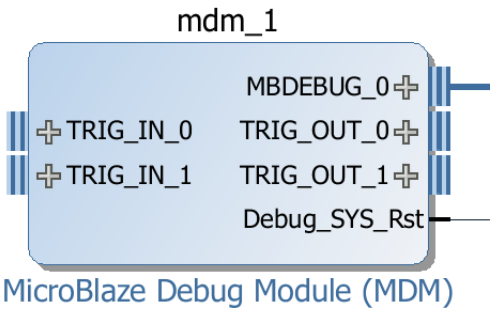


Figure 3-18: **MDM in Block Design After Enabling Cross Trigger**

Next, run Connection Automation to connect the cross trigger signals to an ILA.

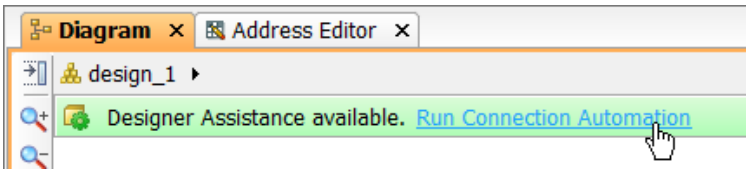


Figure 3-19: **Connecting the TRIG_IN_0 Interface Pin to an ILA**

The Run Connection Automation dialog box informs you that it will instantiate a new ILA and connect the `TRIG_IN_0` signal of the MDM to the corresponding pin of the ILA.

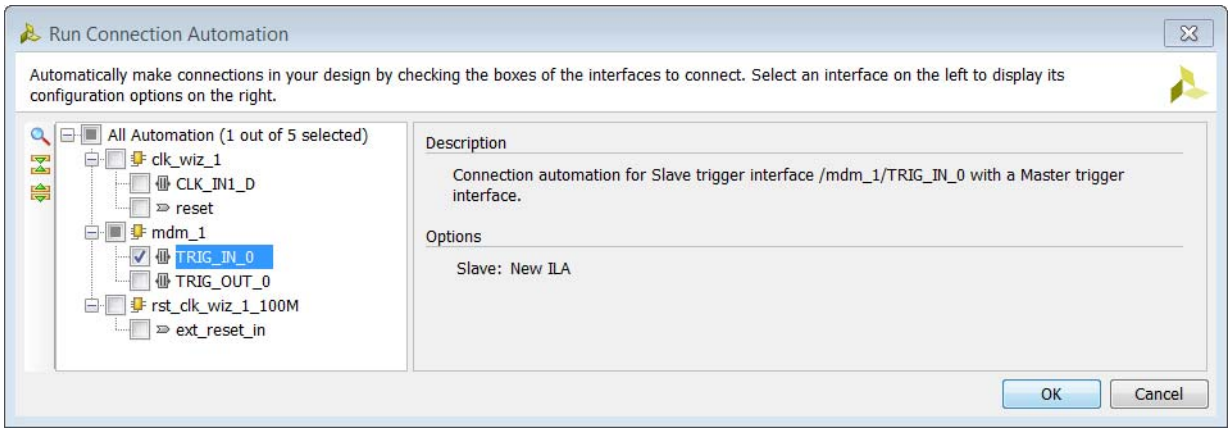


Figure 3-20: **Run Connection Automation Confirmation Dialog Box**

Run Connection Automation again and select the TRIG_OUT_0 interface. At this time, you have two options to choose from in the **Slave** field. You can either use the already instantiated ILA or you can instantiate a new ILA. Depending on your selection, either a new

ILA will be instantiated or the already instantiated ILA will be re-used to connect the cross-trigger pins of the MDM.

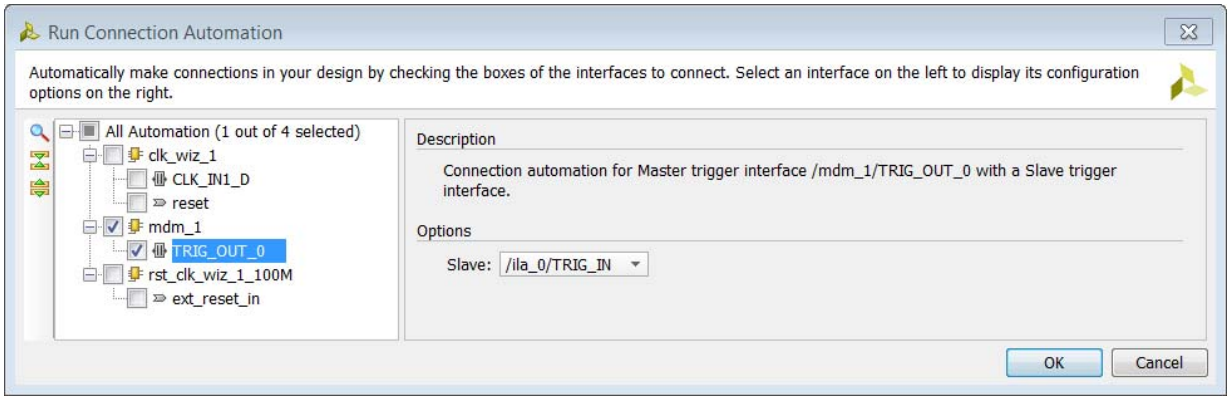


Figure 3-21: **Connecting the TRIG_OUT_0 Interface Pin to an ILA**

.

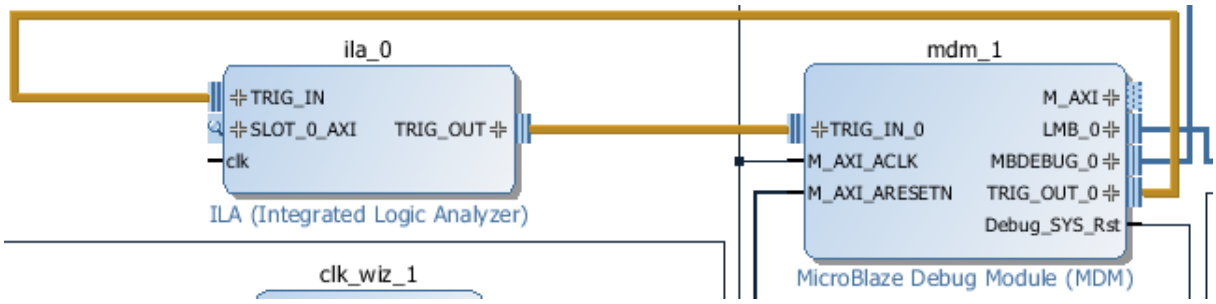


Figure 3-22: **Block Design After Connecting Cross Trigger Pins to the ILA**

Custom Logic

The Vivado IP packager lets you and third party IP developers use the Vivado IDE to prepare an Intellectual Property (IP) design for use in the Vivado IP catalog. The IP user can then instantiate this third party IP into a design in the Vivado Design Suite.

When IP developers use the Vivado Design Suite IP packaging flow, the IP user has a consistent experience whether using Xilinx IP, third party IP, or customer-developed IP within the Vivado Design Suite.

IP developers can use the IP packager feature to package IP files and associated data into a ZIP file. The IP user receives this generated ZIP file, installs the IP into the Vivado Design Suite IP Catalog. The IP user then customizes the IP through parameter selections and generates an instance of the IP.

RECOMMENDED: *To verify the proper packaging of the IP before handing it off to the IP user, Xilinx recommends that the IP developer run each IP module completely through the IP user flow to verify that the IP is ready for use.*

Embedded IP Catalog

The Vivado IDE IP Catalog is a unified repository that lets you search, review detailed information, and view associated documentation for the IP. After you add the third party or customer IP to the Vivado Design Suite IP catalog, you can access the IP through the Vivado Design Suite flows. Figure 3-23 shows a portion of the Vivado IDE IP Catalog.

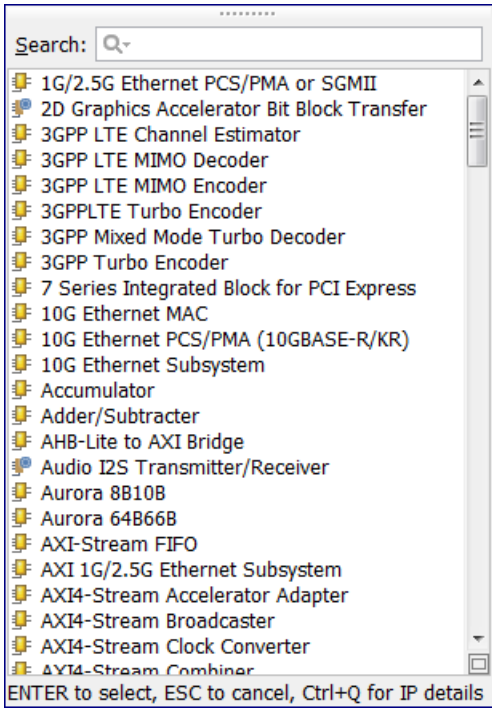


Figure 3-23: **IP Integrator IP Catalog**

Completing Connections

After you have configured the MicroBlaze processor, you can start to instantiate other IP that constitutes your design.

In the IP Integrator canvas, right-click and select **Add IP**.

You can use two built-in features of the IP integrator to complete the rest of the IP subsystem design: the Block Automation and Connection Automation features assist you with putting together a basic microprocessor system in the IP integrator tool and/or connecting ports to external I/O ports.

Block Automation

The Block Automation feature is available when a microprocessor such as the ZYNQ7 Processing System (PS) or the MicroBlaze Processor is instantiated in the block design of the IP integrator tool.

1. Click **Run Block Automation** to get assistance with putting together a simple MicroBlaze System.

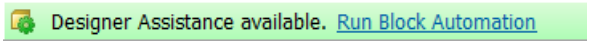


Figure 3-24: **Run Block Automation Using Designer Assistance**

The Run Block Automation dialog box lets you provide input about basic features that the microprocessor system requires.

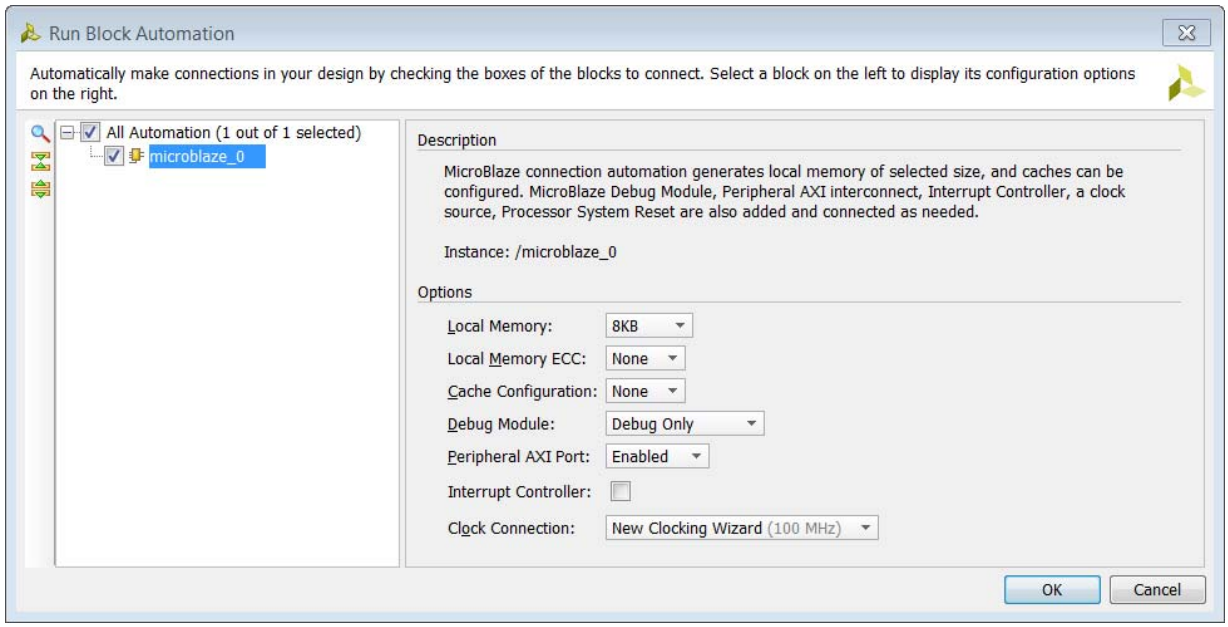


Figure 3-25: **Run Block Automation Dialog Box for MicroBlaze**

2. Select the required options and click **OK**.

Run Block Automation creates the following MicroBlaze system.

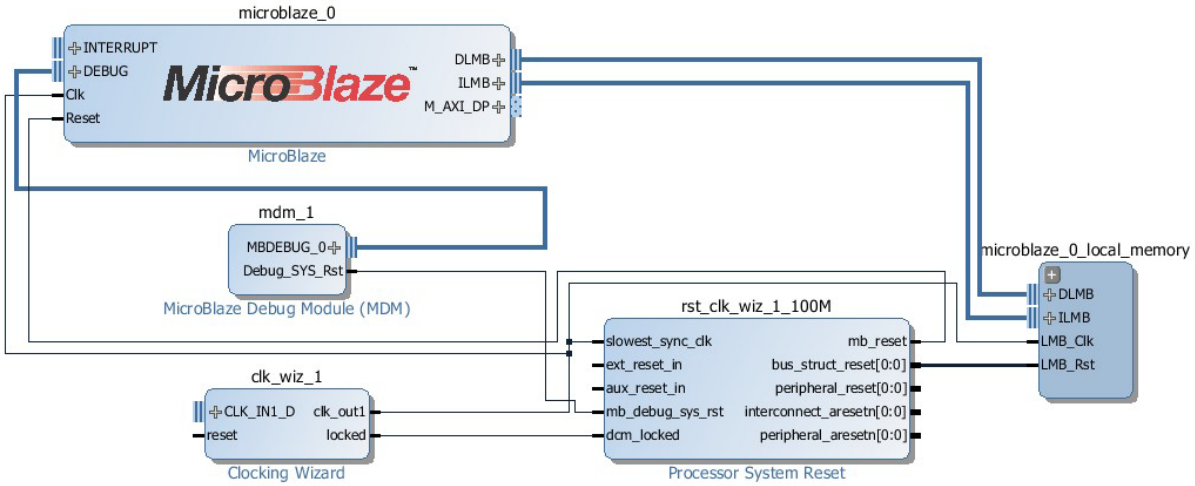


Figure 3-26: **MicroBlaze Design After Running Block Automation**

Using Connection Automation

When the IP integrator tool determines that a potential connection exists among the instantiated IP in the canvas, it opens the Connection Automation feature.

In Figure 3-27, two IP, the GPIO and the Uartlite, are instantiated along with the MicroBlaze subsystem.

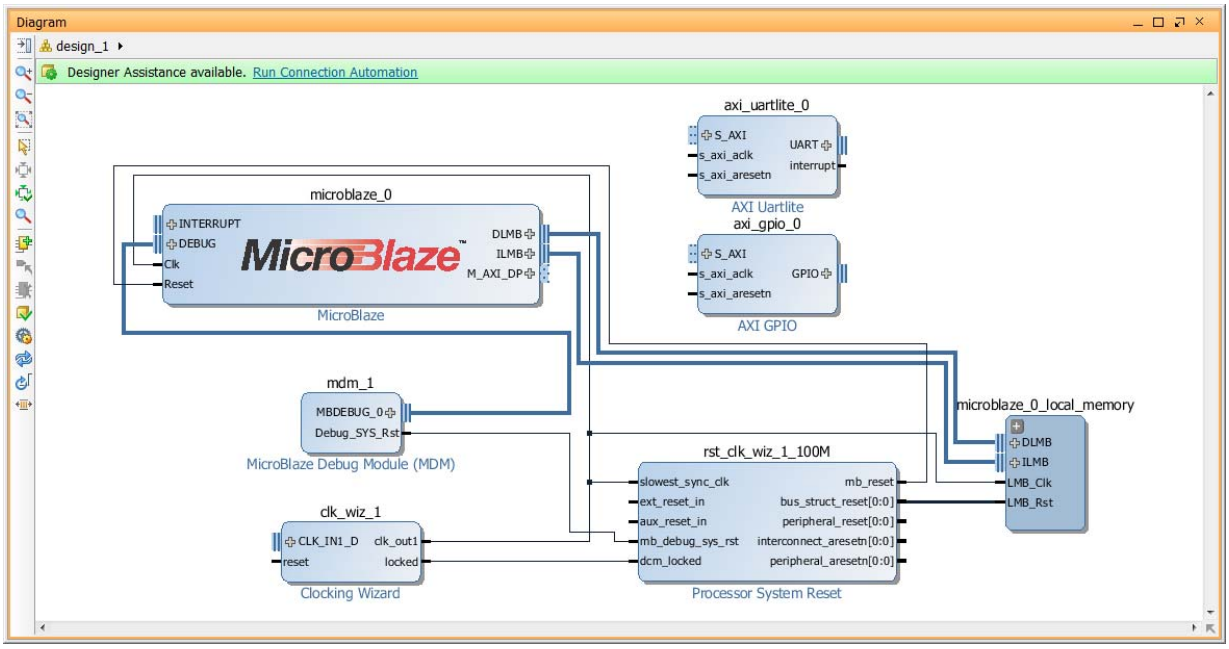


Figure 3-27: **Using Connection Automation Feature of IP Integrator**

When the Run Connection Automation link is clicked, the following dialog box opens.

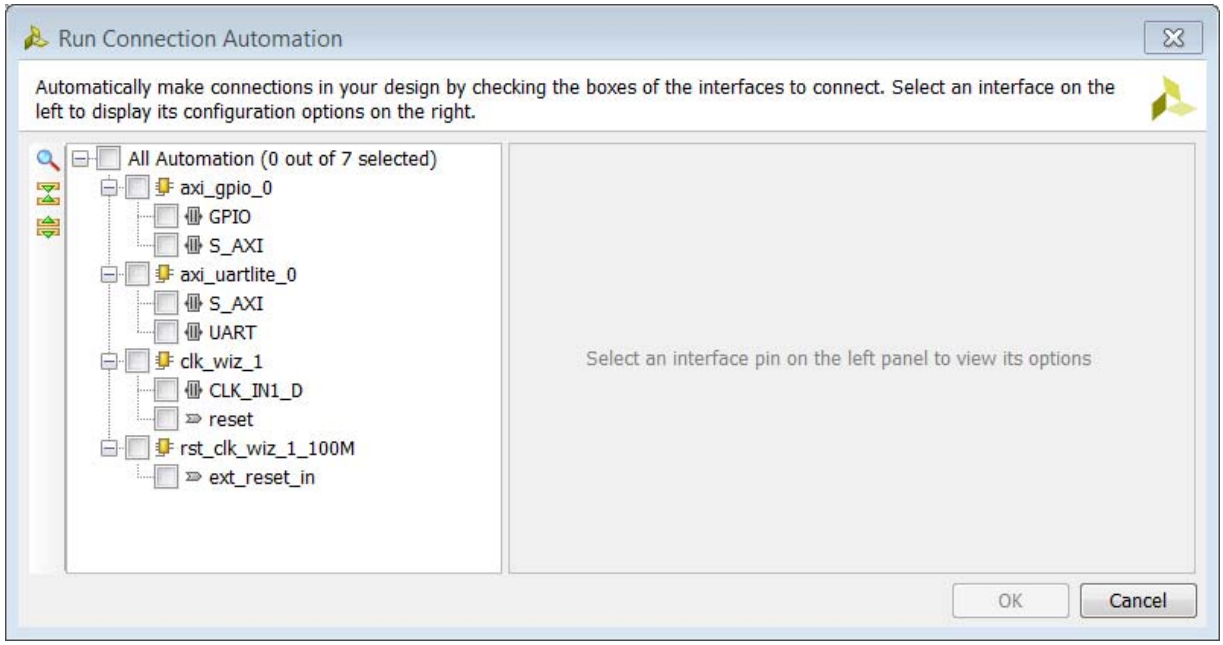


Figure 3-28: **The Run Connection Automation dialog box**

The IP integrator determines that there is a potential connection for the following objects:

- The Proc Sys Rst IP ext_reset_in pin must connect to a reset source, which can be either an internal reset source or an external input port.
- The Clocking Wizard CLK_IN_1_D pin must connect to either an internal clock source or an external input port.
- The AXI GPIO s_axi interface must connect to a master AXI interface.
- The AXI GPIO core gpio interface must connect to external I/Os.
- The Uartlite IP s_axi interface must connect to a master AXI interface.
- The Uartlite IP uart interface must connect to external I/Os.

When you run connection automation on each of those available options, the block design looks like Figure 3-29.

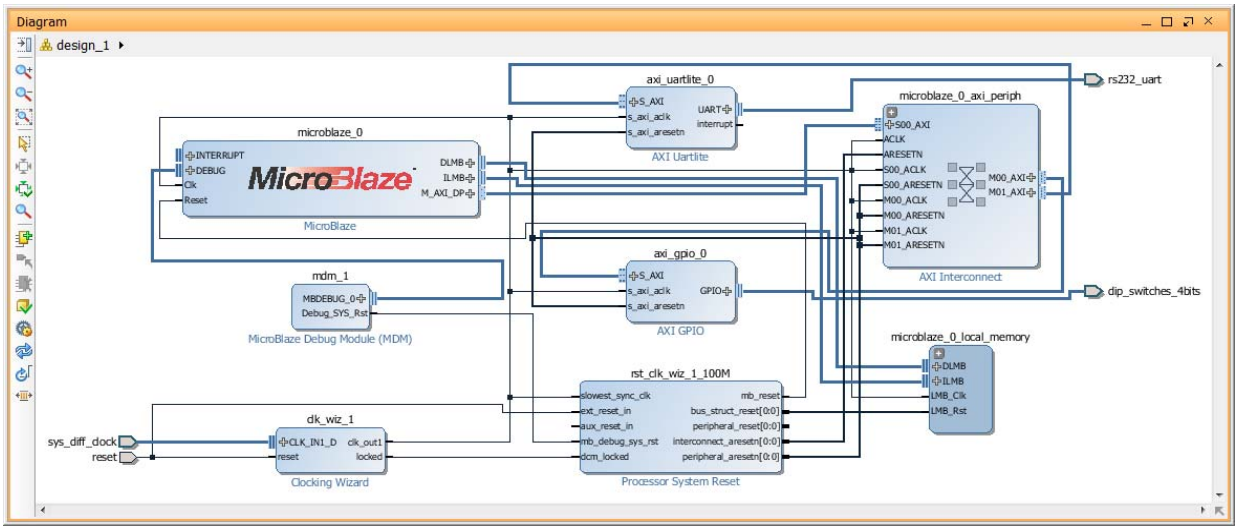


Figure 3-29: **Running Connection Automation for a Sample MicroBlaze Design**

Board Automation in IP Integrator

See Platform Board Flow in IP Integrator, page 44.

Manual Connections in an IP Integrator Design

See Manual Connections in a Design, page 41.

Manually Creating and Connecting to I/O Ports

See Manually Creating and Connecting to I/O Ports, page 41.

Memory Mapping in Address Editor

See Memory Mapping in Address Editor, page 44.

Running Design Rule Checks

See Running Design Rule Checks, page 45.

Integrating a Block Design in the Top-Level Design

See Integrating a Block Design in the Top-Level Design, page 45.

MicroBlaze Processor Constraints

The IP integrator generates constraints for IP generated within the tool during output products generation; however, you must generate constraints for any custom IP or higher-level code.

A constraint set is a set of XDC files that contain design constraints, which you can apply to your design. There are two types of design constraints:

- Physical constraints define pin placement, and absolute, or relative placement of cells such as: BRAMs, LUTs, Flip Flops, and device configuration settings.
- Timing constraints, written in industry standard SDC, define the frequency requirements for the design. Without timing constraints, the Vivado Design Suite optimizes the design solely for wire length and routing congestion.

  Note: Without timing constraints, Vivado implementation makes no effort to assess or improve the performance of the design.

IMPORTANT: *The Vivado Design Suite does not support UCF format. For information on migrating UCF constraints to XDC commands refer to the Vivado Design Suite User Guide [Ref 9] for more information.*

You have a number of options on how to use constraint sets. You can have:

- Multiple constraints files within a constraint set.
- Constraint sets with separate physical and timing constraint files.
- A master constraints file, and direct design changes to a new constraints file.
- Multiple constraint sets for a project, and make different constraint sets active for different implementation runs to test different approaches.
- Separate constraint sets for synthesis and for implementation.
- Different constraint files to apply during synthesis, simulation, and implementation to help meet your design objectives.

Separating constraints by function into different constraint files can make your overall constraint strategy more clear, and facilitate being able to target timing and implementation changes.

Organizing design constraints into multiple constraint sets can help you do the following:

- Target different Xilinx FPGAs for the same project. Different physical and timing constraints could be necessary for different target parts.
- Perform "what-if" design exploration. Using constraint sets to explore different scenarios for floorplanning and over-constraining the design.

- Manage constraint changes. Override master constraints with local changes in a separate constraint file.

TIP: *A good way to validate the timing constraints is to run the report_timing_summary command on the synthesized design. Problematic constraints must be addressed before implementation.*

For more information on defining and working with constraints that affect placement and routing, see the *Vivado Design Suite User Guide: Using Constraints* (UG903) [Ref 10].

Taking the Design through Synthesis, Implementation and Bitstream Generation

After you complete the design and constrain it appropriately, you can run synthesis and implementation, and then you can generate a bitstream.

Exporting Hardware to the Software Development Kit (SDK)

See Using the Software Development Kit (SDK), page 47 for more information.

In general, after you generate the bitstream for your design, you are ready to export your hardware definition to SDK.

This action exports the necessary XML files needed for SDK to understand the IP used in the design and also the memory mapping from the perspective of the processor. After a bitstream is generated and the design is exported, you can then download the device and run the software on the processor.

TIP: *If you want to start software development before a bitstream is created, you can export the hardware definition to SDK after generating the design.*

1. Select **File > Export > Export Hardware**.

   This launches the Export Hardware for SDK dialog box, where you can choose the available export options.

2. After the hardware is exported, select **File > Launch SDK** to launch SDK.

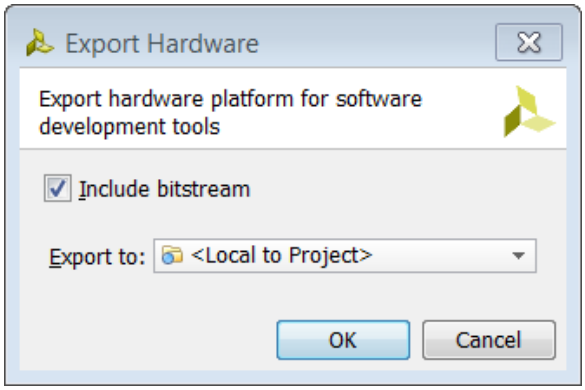


Figure 3-30: **Export Hardware for SDK Dialog Box**

After you export the hardware definition to SDK, and launch SDK, you can start writing your software application. Also, you can perform more debug and software from SDK.

Alternatively, you can import the software ELF file back into a Vivado IDE project, and integrate that file with an FPGA bitstream for further download and testing.

Designing with the MIG Core

Overview

The Xilinx® memory interface generator (MIG) core is a combined pre-engineered controller and physical layer (PHY) for interfacing UltraScale™ architecture and 7 series FPGA user designs with AMBA® advanced extensible interface (AXI4) slave interfaces to DDR SDRAM devices.

This chapter provides information about using, customizing, and simulating a LogiCORE™ IP DDR4, DDR3, or DDR2 SDRAM memory interface core in the Vivado IP integrator tool. This chapter describes the core architecture and provides details on customizing and interfacing to the core.

TIP: *Although the information in this chapter is tailored for the KC705, Kintex®-7 board, the differences for UltraScale devices, and the KCU105 board, are highlighted throughout this text. These guidelines can also be applied to Xilinx devices on custom boards.*

Project Creation

Although you can create entire designs using IP integrator, a typical design consists of HDL, IP, and IP integrator block designs. In the Vivado® tools, you can create a new project from the Quick Start page, as shown below.

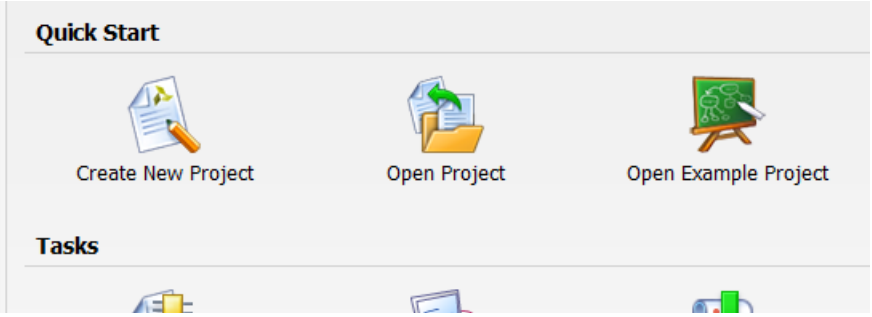


Figure 4-1: **Create New Project from the Quick Start Page**

1. Click **Create New Project** to open the New Project wizard.

2. Specify a name and location for the new project and click **Next**.
3. For Project Type of the New Project wizard, specify an RTL Project, and check "Do not specify sources at this time" as shown below.
4. Click **Next**.

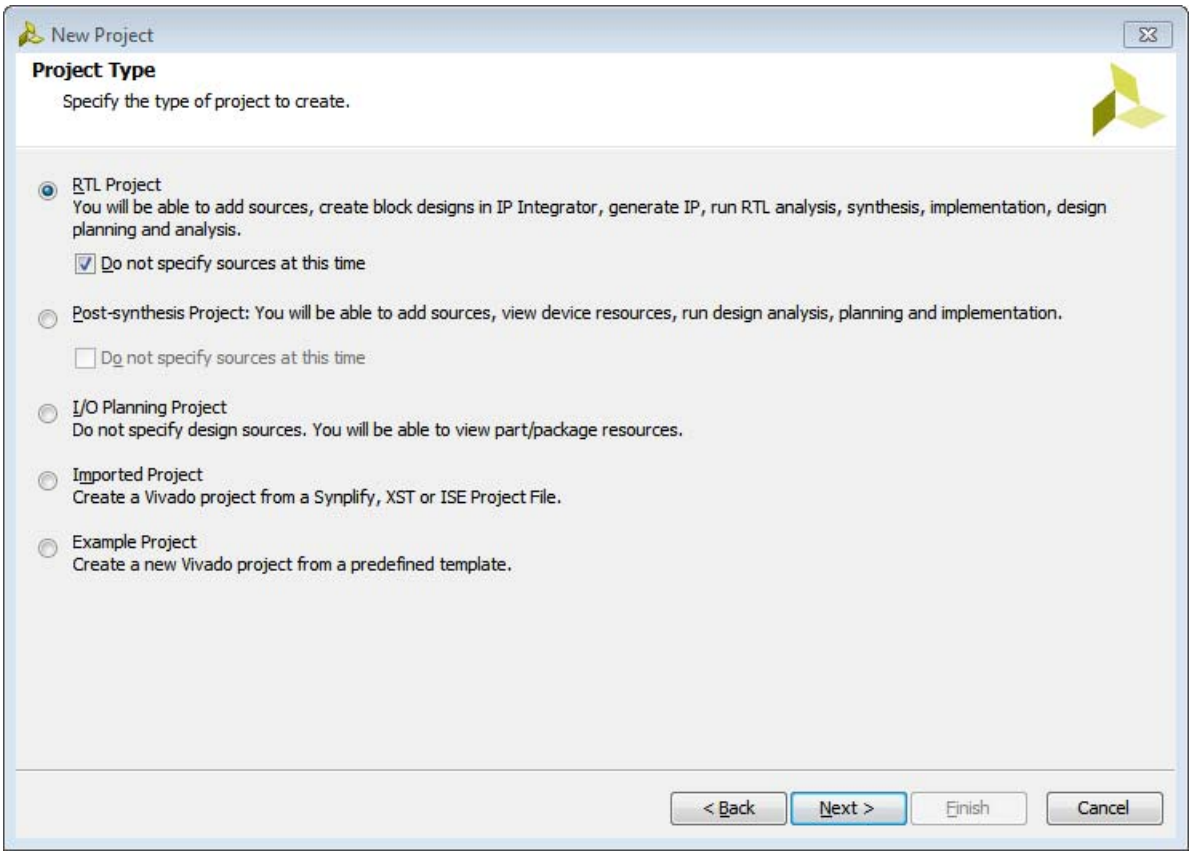


Figure 4-2: **Select the Target Board**

5. Select the KC705 board as the target board, and click **Next**.

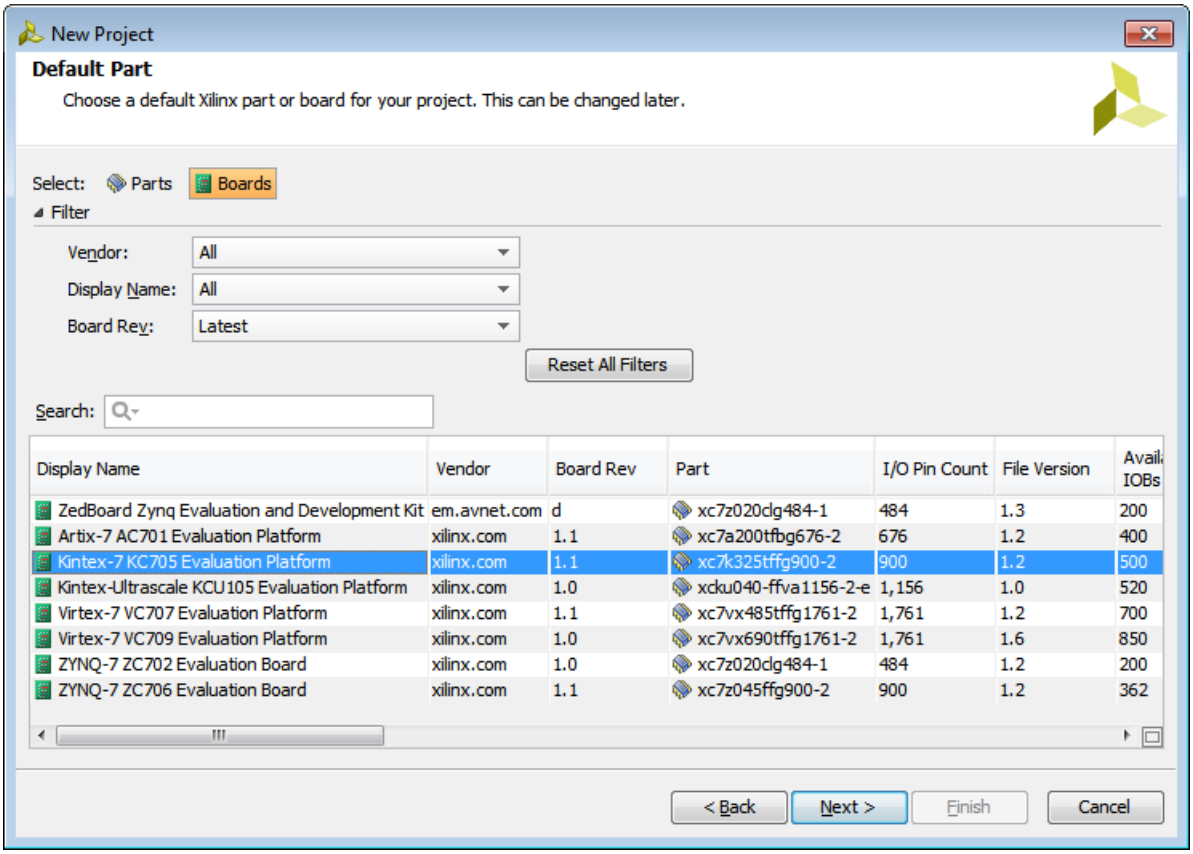


Figure 4-3: **Select the Target Board**

Selecting a board as the target for the design lets the Vivado Design Suite use the platform board flow. For more information on the platform board flow, see *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994) [Ref 3].

Note: You can also create the project using the following Tcl commands:

```
create_project <project_name> <path_to_project> -part xc7k325tffg900-2
set_property BOARD_PART xilinx.com:kc705:part0:1.2 [current_project]
set_property TARGET_LANGUAGE vhdl [current_project]
```

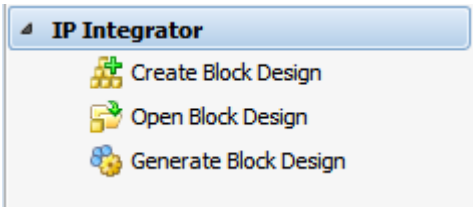
Where:

- *<project_name>* is the name of the project to create.
- *<path_to_project>* is the file path to write the project directory.

TIP: *In Tcl command examples, as shown above, angle brackets (< >) are used to indicate variables specific to your design. Provide a suitable value and do not include the angle brackets.*

Designing in IP Integrator

Create a new block design in the Flow Navigator by clicking **Create Block Design** under **IP Integrator**. .



You can also use the following Tcl Command:

```
create_bd_design <design_name>
```

Adding the MIG IP

To add the Memory Interface Generator IP, right-click in the IP integrator design canvas and select **Add IP**. A searchable IP Catalog opens. When you type the first few letters of an IP name, in this case MIG, only the IP cores matching the name are listed.

Alternatively, you can click the **Add IP** button on the left side of the canvas  .

Double-click to select the Memory Interface Generator IP and add it to your block design.

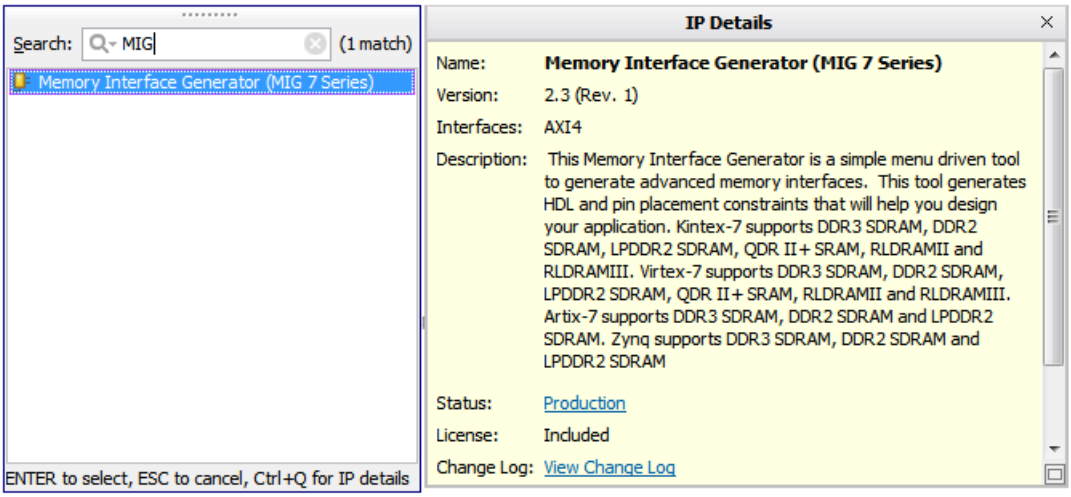


Figure 4-4: **Add the MIG IP by Searching in the IP Catalog**

This places the MIG IP core into the IP integrator block design. To make changes to the MIG configuration, right-click on the block to open the menu, and click **Customize Block**. You can also double-click the MIG IP block to open the Xilinx Memory Interface Generator dialog box. For more information on MIG configuration settings, refer to the *7 Series FPGAs Memory Interface Solutions User Guide* (UG586) [Ref 11], or to the *UltraScale Architecture-Based FPGAs Memory Interface Solutions* (PG150) [Ref 12].

Figure 4-5 shows both the MIG 7 series IP core in the upper-right, and the DDR4 MIG core for UltraScale devices in the lower-left. The MIG IP that is available in the IP Catalog

depends on the target part or platform board selected for your project. There are separate IP cores to support DDR3 and DDR4 memory controllers for UltraScale devices.

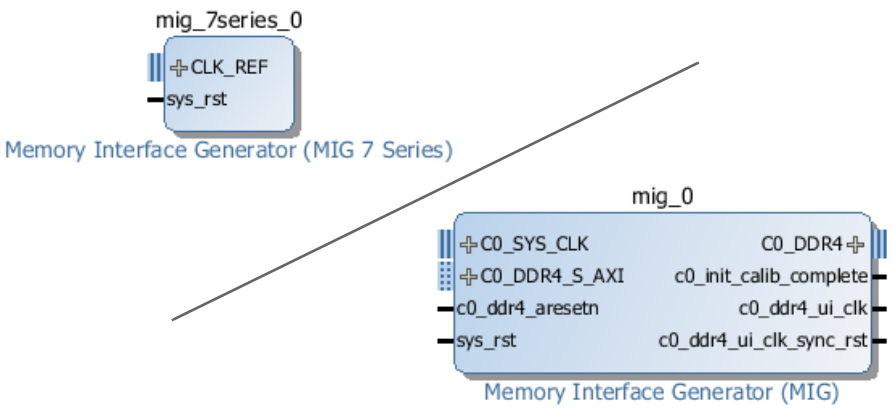


Figure 4-5: **Instantiate the MIG IP Core in the Block Design**

Because you have selected the KC705 board as the target for your project, as shown in Figure 4-3, page 88, the Board tab of the platform board flow is available to let you select components to interface to your design, as shown in Figure 4-6.

From the Board tab, drag and drop the DDR3 SDRAM component into the block design canvas.

Note: In the case of the UltraScale KCU105 board, the DDR4 SDRAM component may also used.

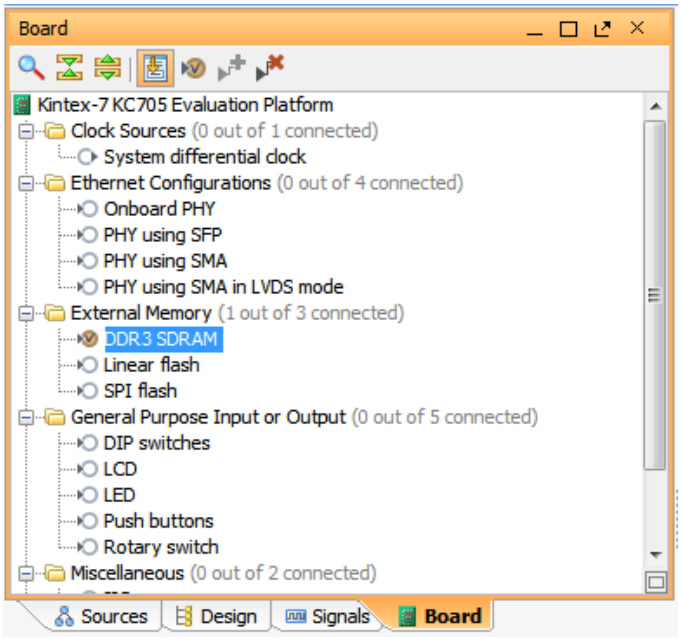


Figure 4-6: **Instantiating the MIG core using Platform Board Flow**

In order to connect the memory controller to the memory components on the target platform board, the Vivado IP integrator connects the SYS_CLK and DDR interfaces of the MIG IP to external interface ports, as seen below.

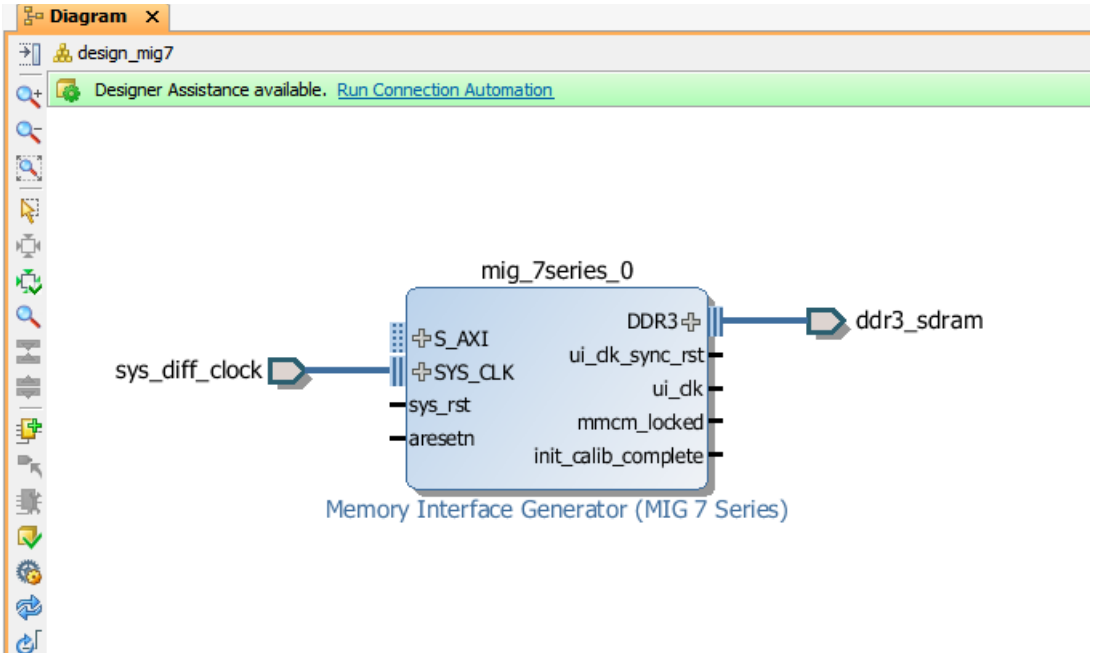


Figure 4-7: **Connecting SYS_CLK and DDR3 interfaces to the Board**

TIP: *You can also begin by simply dragging and dropping the DDR SDRAM component from the Board tab into an empty block design. In this case, the Vivado IP integrator instantiates the MIG IP onto the canvas and connects the SYS_CLK and DDR interfaces of the MIG to the components on the platform board.*

Select the **Run Connection Automation** link at the top of the design canvas, as seen in Figure 4-7. This connects the MIG IP to the system FPGA reset on the platform board.

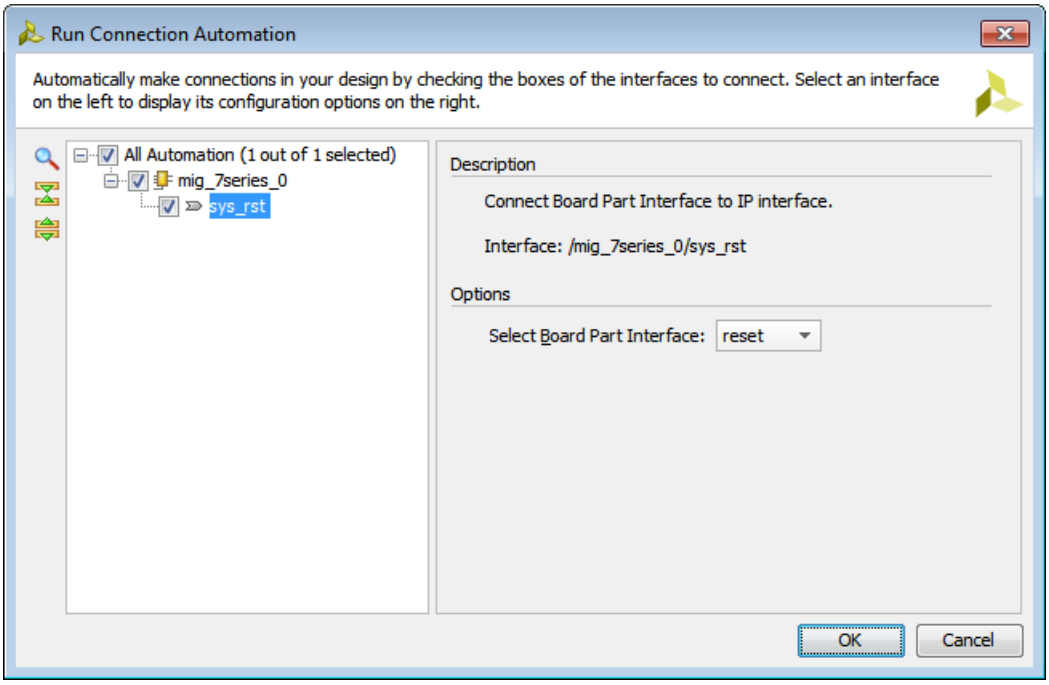


Figure 4-8: **Connecting SYS_CLK and DDR3 interfaces to the Board**

Note: For the KCU105 board, the Run Connection Automation dialog box includes both the `CO_SYS_CLK` and the `sys_rst` interfaces for the MIG IP.

Making Connections with Block Automation

As an alternative to dragging and dropping the DDR SDRAM component from the Board tab, you could use the Block Automation feature of IP integrator to configure the MIG and tie its SYS_CLK and DDR3 interfaces to the board interfaces.

Because the MIG core provides the clocking for the entire KC705 board, you should **Run Block Automation** for the MIG core prior to adding a clock controller. .

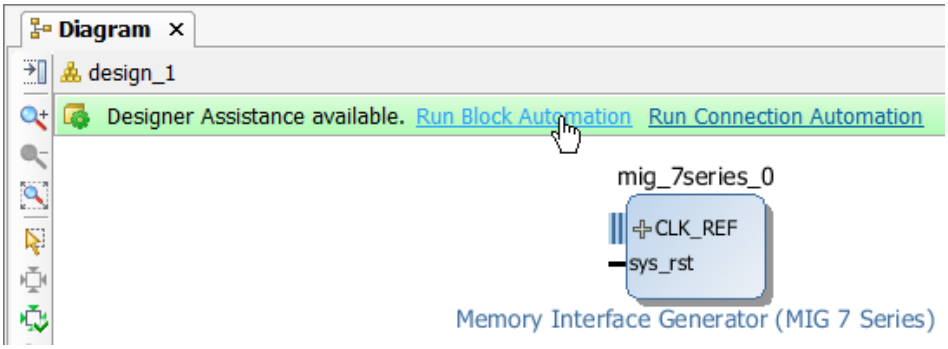


Figure 4-9: **Run Block Automation for the MIG Core**

This opens the Run Block Automation dialog box as shown below.

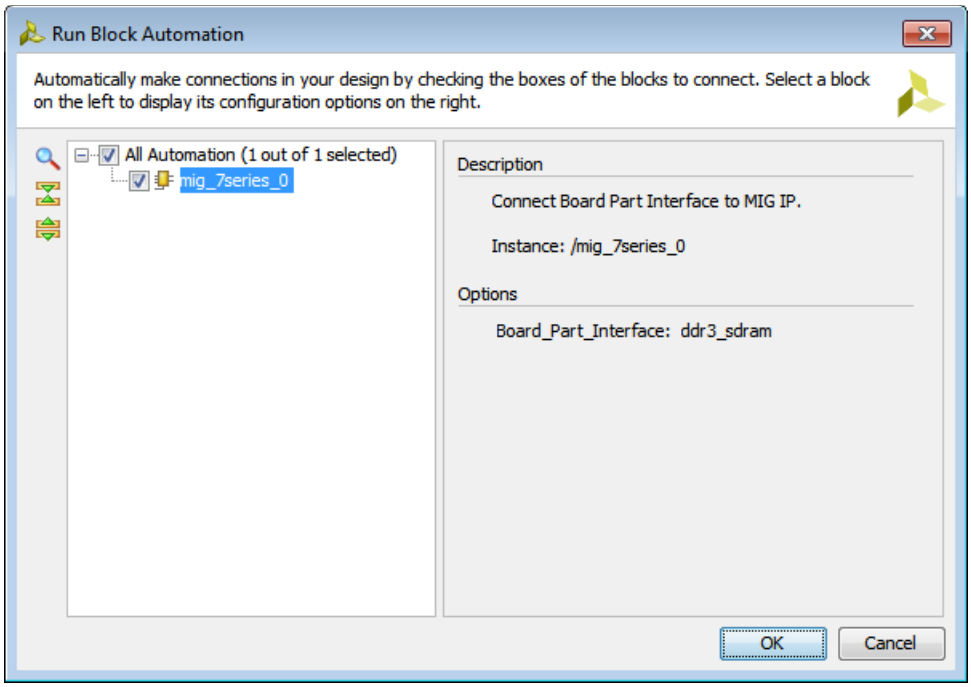


Figure 4-10: **Run Block Automation dialog box**

The Run Block Automation dialog box shows the available IP. In this case, the block design only has the MIG IP you previously added. Ensure the MIG is selected and click **OK**.

The SYS_CLK and DDR interfaces of the MIG IP are connected to the DDR memory components on the platform board. The MIG core is configured for 400 MHz operation with the correct pins selected to interface to the KC705 board.

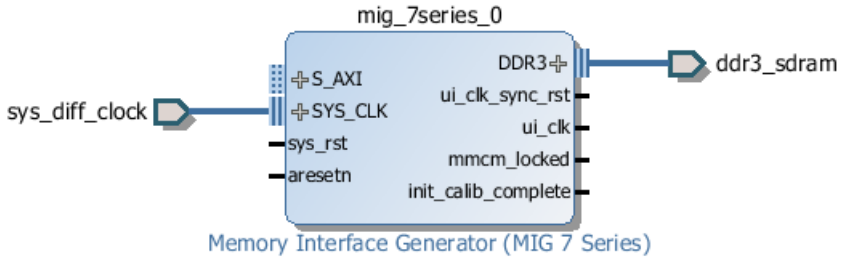


Figure 4-11: **MIG Core in Block Design After Running Block Automation**

Adding a Clocking Wizard

If the design requires clocking in addition to the clock generated by the MIG core, you need to add a Clocking wizard IP into the block design.

Select the **Add IP** command, type Clock into the search field, and select the Clocking Wizard IP.

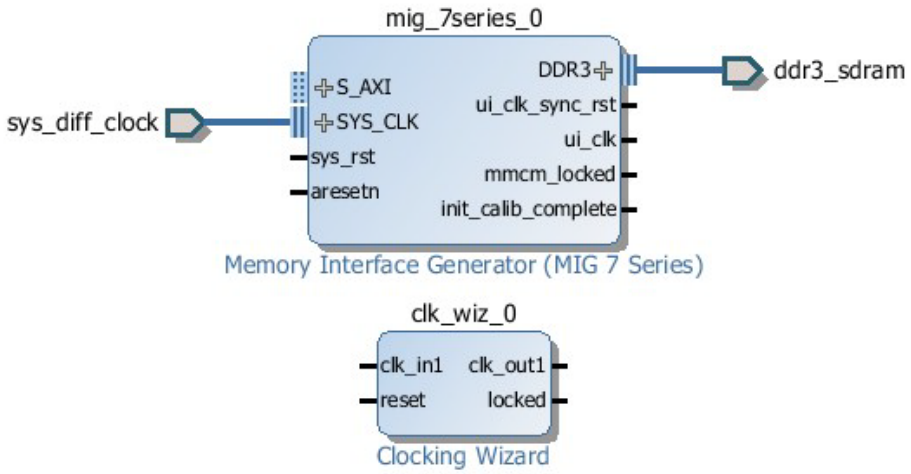


Figure 4-12: **Clocking Wizard**

Follow these steps to connect the Clocking Wizard to the MIG core.

1. Connect the `ui_clk` output of the MIG IP, as well as any other clocks generated, to the `clk_in1` input of the Clocking wizard, as shown in Figure 4-13, page 94.

   Note: For the UltraScale MIG IP, connect the `c0_ddr4_ui_clk` pin to the Clocking Wizard.

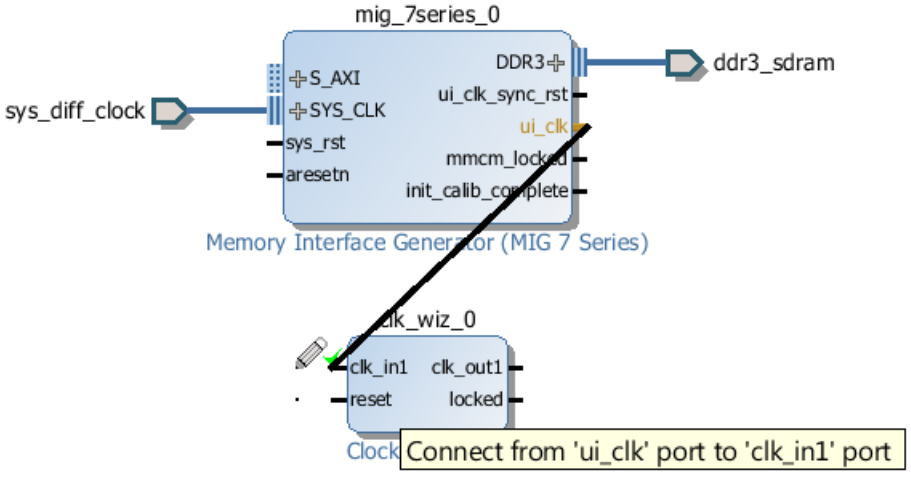


Figure 4-13: **Connect ui_clk to clk_in1**

2. Connect the `ui_clk_sync_rst` pin of the MIG core to the `reset` pin of the Clocking wizard, as shown below.

   Note: For the UltraScale MIG IP, connect the `c0_ddr4_ui_clk_sync_rst` pin to the Clocking Wizard.

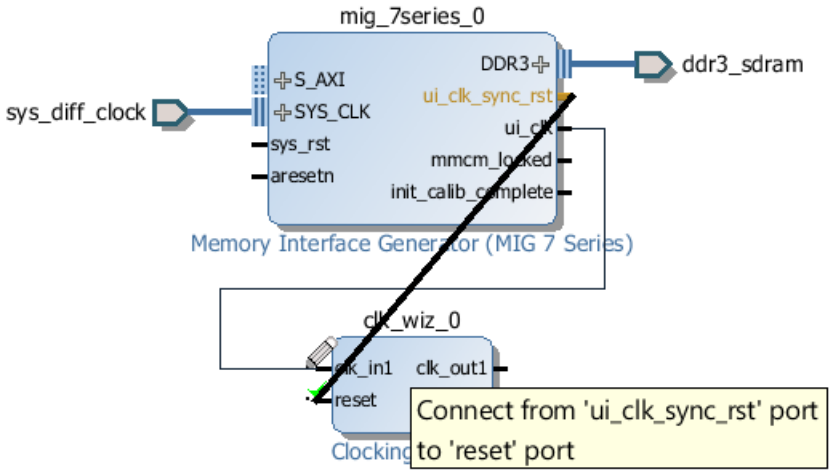


Figure 4-14: **Connect ui_clk_sync_rst to the Reset Port**

3. Configure the Clocking wizard to generate any required clocks for the design, by double-clicking on the IP.

Adding an AXI Master

To complete the MIG design, an AXI master such as a Zynq or MicroBlaze embedded processor, or an external processor is required. The following procedure shows you how to instantiate a MicroBlaze processor into the block design.

1. Select the **Add IP** command, type Micro into the search field, and select the MicroBlaze processor to add it to the design.
2. Click **Run Block Automation** to construct a basic MicroBlaze system, and configure the settings in the dialog box as follows:
   - Local Memory: Select the required amount of local memory from pull-down menu.
   - Local Memory ECC: Turn on ECC if desired.
   - Cache Configuration: Select the required amount of Cache memory.
   - Debug Module: Specify the type of debug module from the pull-down menu.
   - Peripheral AXI Interconnect: This option must be enabled.
   - Interrupt Controller: Optional.
   - Clock Connection: Select the clock source from the pull-down menu.

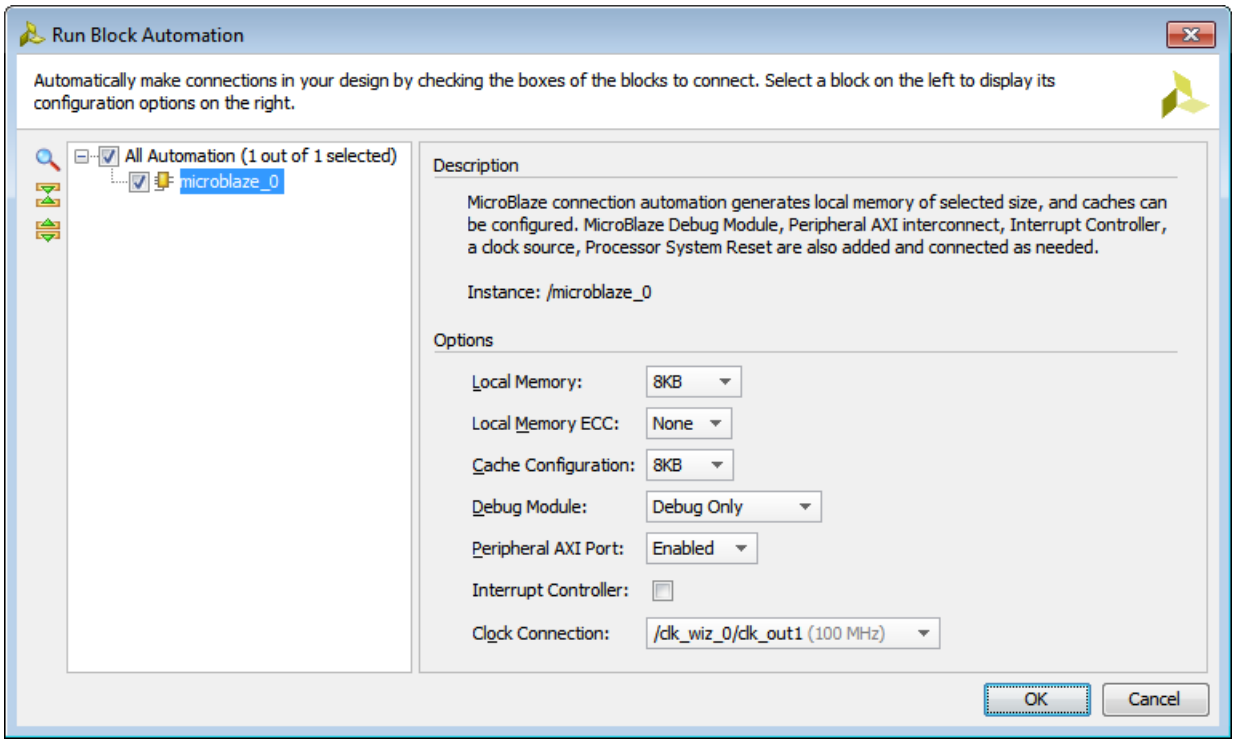


Figure 4-15: **Run Block Automation Settings**

3. Click **OK**.

   The Run Block Automation has added and connected IP needed to support the MicroBlaze processor into the block design. The block design should look similar to Figure 4-16. However, notice that the MIG core is not yet connected to the MicroBlaze processor.

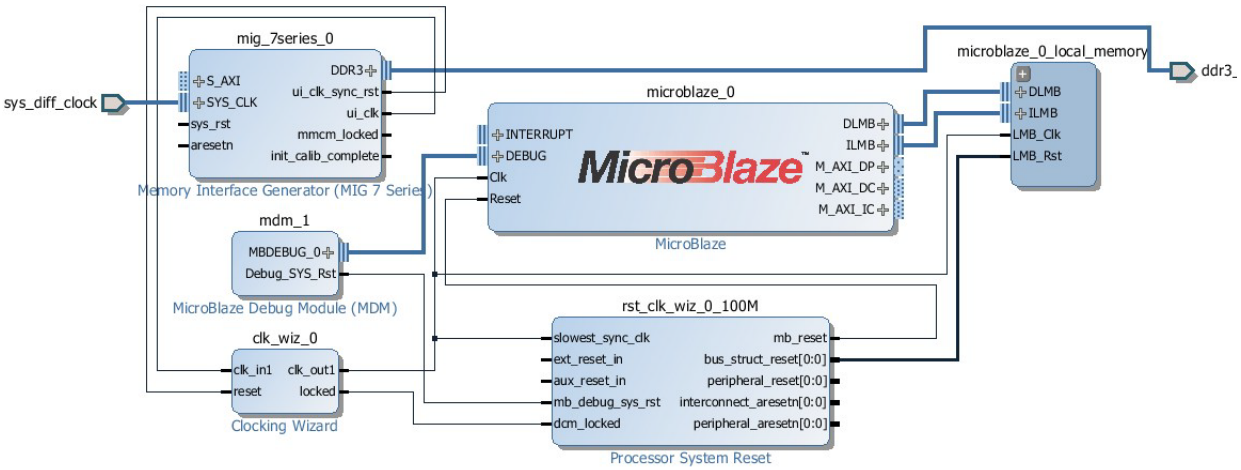


Figure 4-16: **Block Design After Running Block Automation for MicroBlaze**

4. At the top of the design canvas, click **Run Connection Automation** to connect the MIG core to the MicroBlaze processor. The Run Connection Automation dialog box opens as shown below.

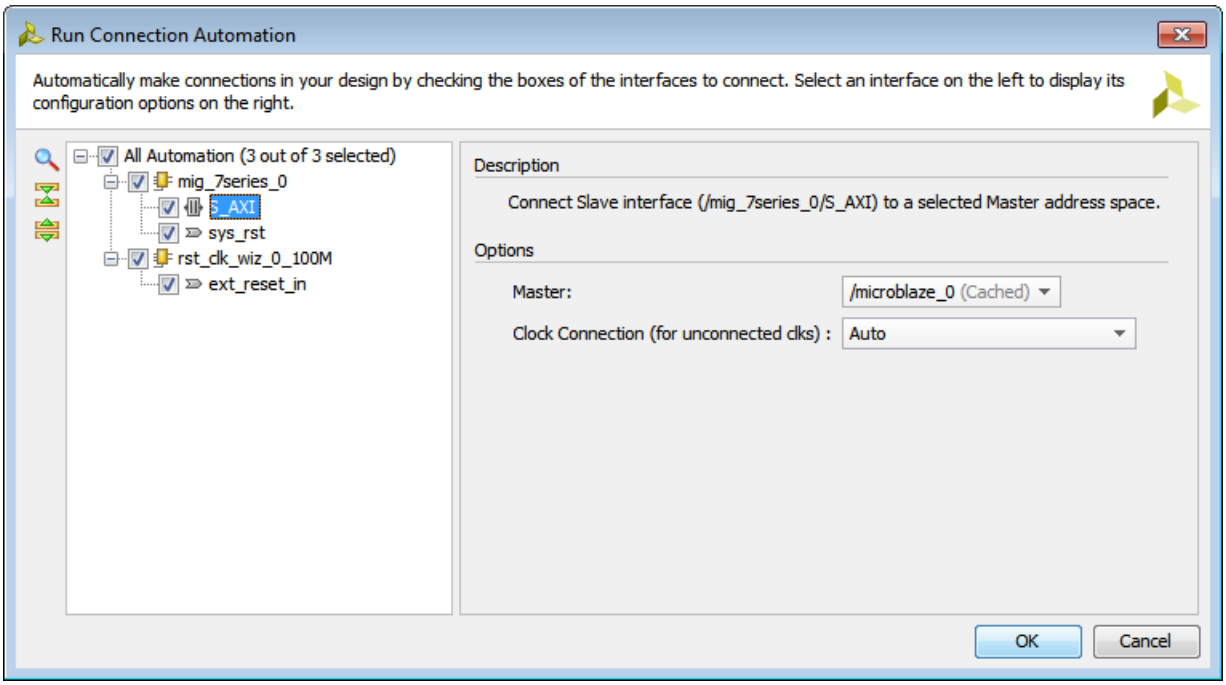


Figure 4-17: **Run Connection Automation Dialog Box to Connect MIG to MicroBlaze**

5. Select the `S_AXI` interface of the `mig_7series_0`.

   Note: For the UltraScale MIG IP, select the `C0_DDR4_S_AXI` interface of the `mig_0`.

6. Select the `/microblaze_0 (Cached)` option from the drop-down menu on the right.
7. Click **OK**.

   This instantiates an AXI Interconnect and makes the required connection between the MIG core and the MicroBlaze processor, as shown in the diagram below.

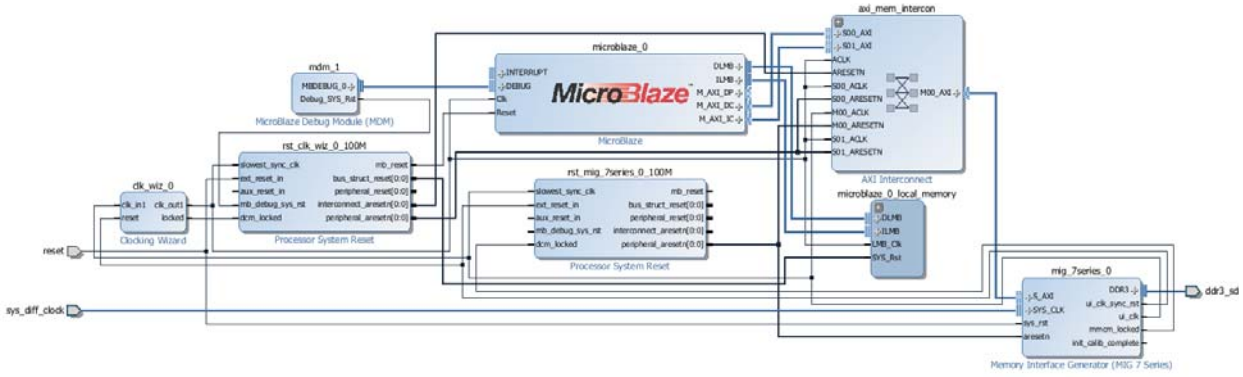


Figure 4-18: **MIG/MicroBlaze Connections**

From here you can complete any remaining connections to the design, such as connecting to an external reset source, or connecting any interrupt sources through a concat IP to the MicroBlaze processor.

Creating a Memory Map

To generate the address map for this design, click the **Address Editor** tab above the diagram. The memory map is automatically created as IP and added to the design. You can set the addresses manually by entering values in the **Offset Address** and **Range** columns. You can manually set addresses by entering values in **Offset Address** and **Range** columns. Refer to this link in the *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator (UG994)* [Ref 3] for more information. .

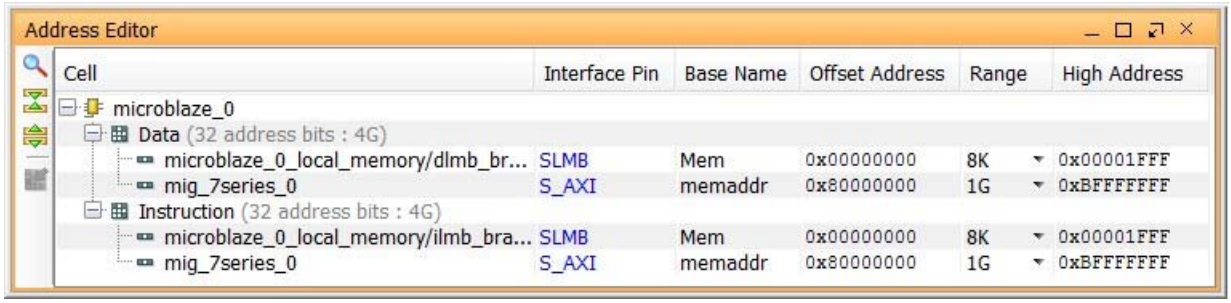


Figure 4-19: **Address Editor**

TIP: *The Address Editor tab only appears if the diagram contains an IP block that functions as a bus master, such as the MicroBlaze processor in the following diagram*

Running Design Rule Checks

The Vivado IP integrator runs basic design rule checks in real time as you create the design. However, problems can occur during design creation. For example, the frequency on a clock pin might not be set correctly. To run a comprehensive design check, click the **Validate Design** button .

If the design is free of warnings and errors, a successful validation dialog box displays.

Implementing the Design

Now you can implement the design, generate the bitstream, and create the software application in SDK.

XILINX
ALL PROGRAMMABLE™

Chapter 5

Reset and Clock Topologies in IP Integrator

Overview

To create designs with IP integrator that function correctly on the target hardware, you must understand reset and clocking considerations. This chapter provides information about clock and reset connectivity at the system level. In the Vivado® IP integrator, you can use the Xilinx platform board flow, which enables you to configure IP in your design to connect to board components using signal interfaces in an automated manner. You can also make all the connections manually. The examples and overall flow described in this chapter use the platform board flow, but the considerations are valid for all block designs.

For designs using the Memory Interface Generator (MIG) core, the IP core provides the clock source, and the primary clock from the board oscillator must be connected directly to the MIG core. Refer to Chapter 4, Designing with the MIG Core for more information.

The MIG core can generate up to five additional clocks (MIG core for UltraScale devices can generate only 4 additional clocks), which you can use for resetting the design as needed. For designs that contain a MIG core, ensure that the primary onboard clock is connected to memory controller, and then use the user clock (`ui_clock`) as additional clock sources for the rest of the design.

For IP integrator designs with platform board flow, specific IP (for example, MIG and Clocking wizard) support board-level clock configuration. For the rest of the system, clocking can be derived from the supported IP. Similarly, for driving reset signals, board-level reset configuration is supported by a specific reset IP (for example, proc_sys_reset). You can use other IP that also require external reset but are not currently supported by the platform board flow.

The following sections illustrate the reset topologies for different types of designs.

MicroBlaze Design without a MIG Core

For any design that uses a MicroBlaze™ processor without a MIG core, you can instantiate a Clocking Wizard IP to generate the clocks required. For the platform board flow, you can configure the connection as follows:

1. After instantiating a MicroBlaze processor in the design, run Block Automation for MicroBlaze. This creates the MicroBlaze sub-system, as shown in Figure 5-1.

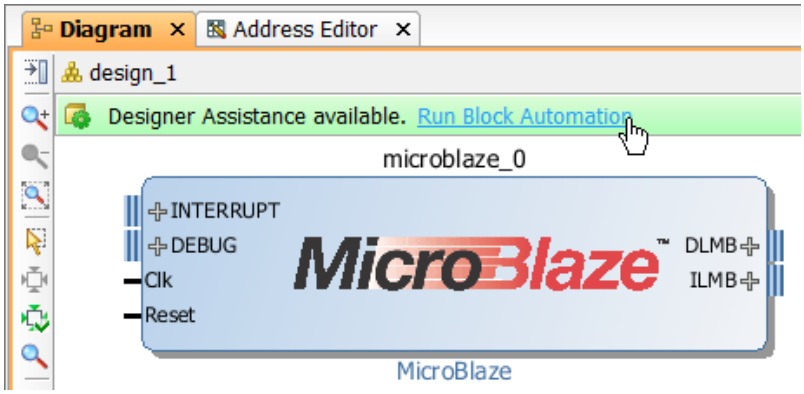


Figure 5-1: **Run Block Automation on the MicroBlaze**

2. In the Run Block Automation dialog box, select the **New Clocking Wizard** option to instantiate the Clocking Wizard IP, and click **OK**.

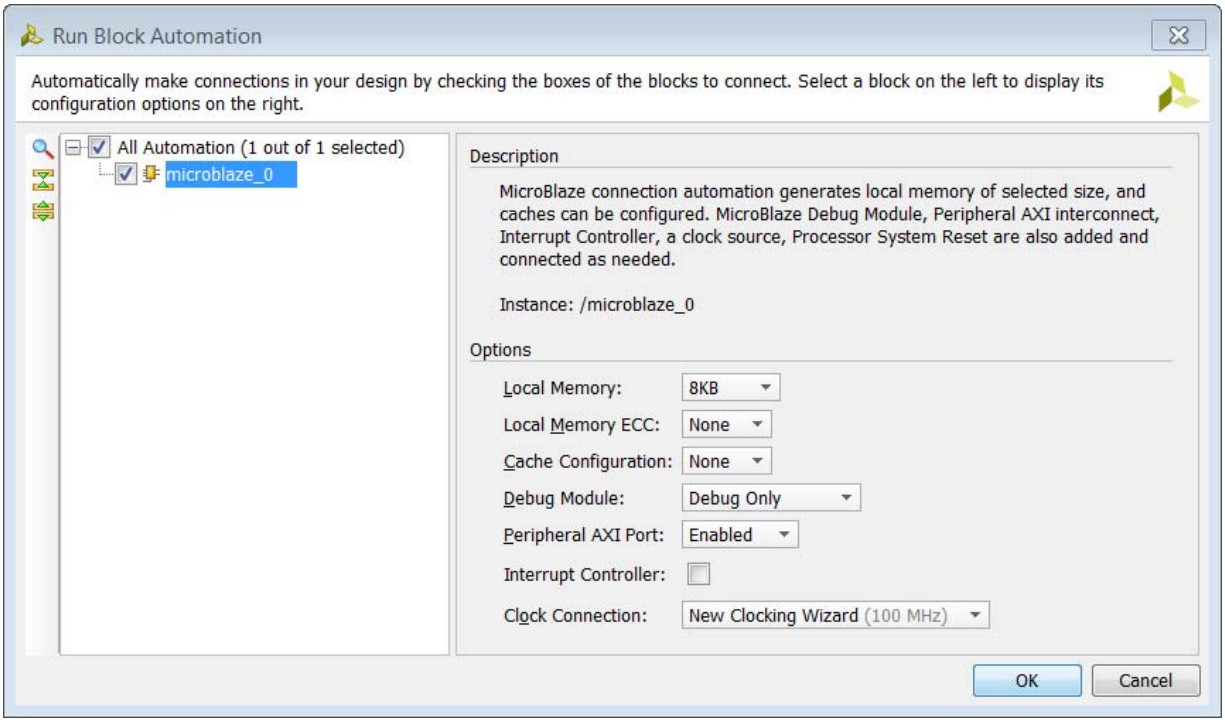


Figure 5-2: **Run Block Automation Dialog Box for the MicroBlaze**

Running Block Automation also instantiates and connects the Proc Sys Reset IP to the various blocks in the design. The IP integrator canvas looks like Figure 5-3.

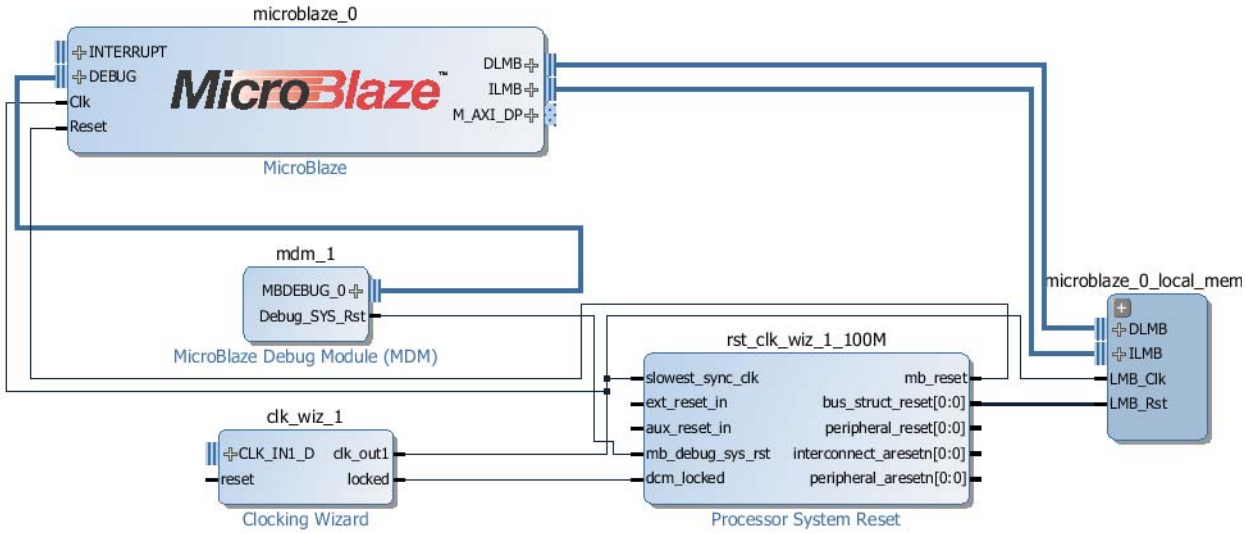


Figure 5-3: **Effect of Running Block Automation**

3. Click **Run Connection Automation** and select **/clk_wiz_1/CLK_IN1_D** to connect the on-board clock to the input of the Clocking Wizard IP, according to the board definition.

   Note: You can customize the Clocking Wizard to generate the various clocks required by the design.

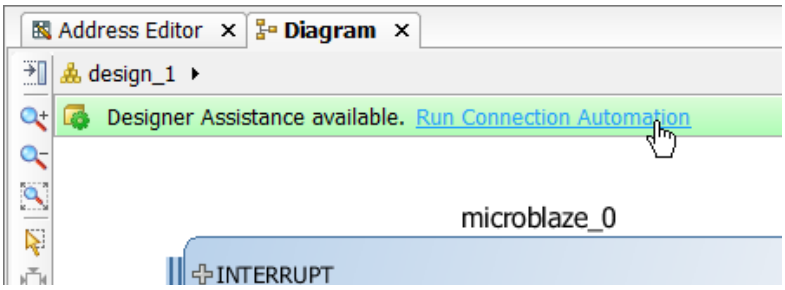


Figure 5-4: **Running Connection Automation on the Clocking Wizard**

4. In the Run Connection Automation dialog box, select **sys_diff_clock** to select the board interface for the target board, or select **Custom** to tie a different input clock source to the Clocking Wizard IP, and then click **OK**.

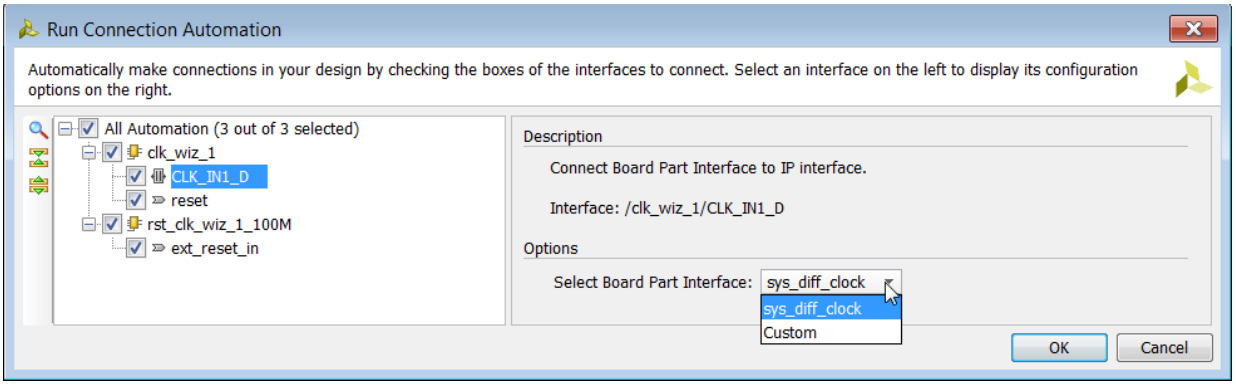


Figure 5-5: **Connecting the On-board sys_diff_clock to the CLK_IN1_D pin of Clocking Wizard**

This creates a `sys_diff_clock` input port on the IP integrator canvas and then connects the port to the CLK_IN1_D input of the Clocking Wizard.

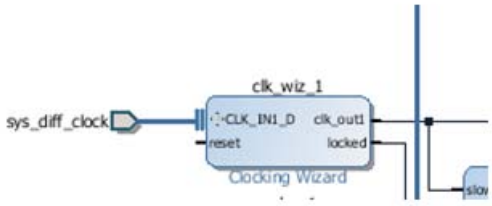


Figure 5-6: **Connecting sys_diff_clock Input as the Input Clock Source to the Clocking Wizard**

5. For the reset pin of the Clocking Wizard, select the dedicated reset interface on the target board or a Custom reset input source.

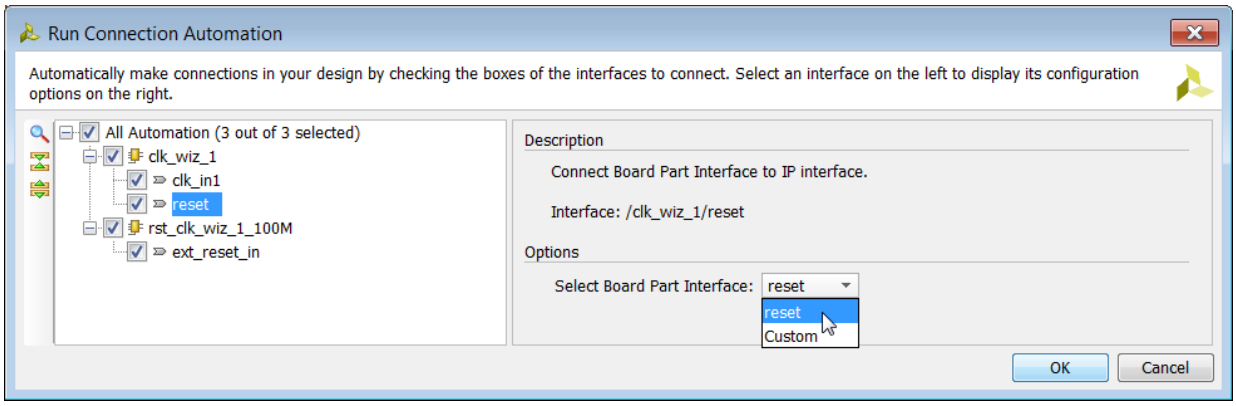


Figure 5-7: **Connect the On-board Reset**

Note: Steps 4 and 5 above can also be done by dragging and dropping the System Differential Clock under the Clock Sources Folder and FPGA Reset from the Reset folder in the Board tab.

6. For the ext_reset_in pin for the Processor System Reset block choose the same reset source as chosen for the Clocking Wizard in the step above or a Custom reset source.

   After you make your choice and click **OK**, the IP integrator canvas looks like Figure 5-8.

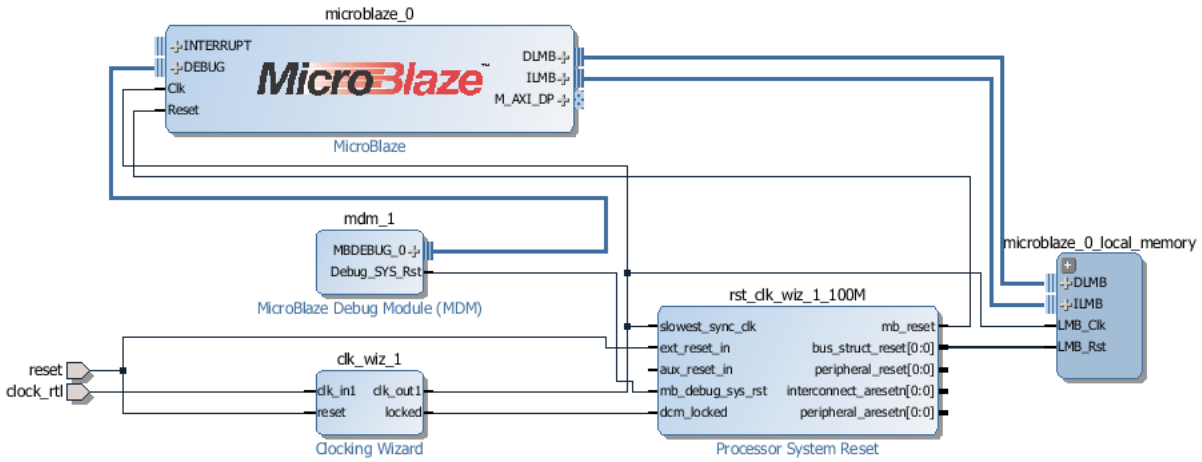


Figure 5-8: **On-Board Reset Connected to the Proc Sys Reset IP**

CAUTION! *If the platform board flow is not used, ensure that the "locked" output of the Clocking Wizard is connected to the "dcm_locked" input of Proc_Sys_Reset.*

MicroBlaze Design with a MIG Core

RECOMMENDED: *As mentioned in the introduction, the MIG IP is a clock source, and Xilinx recommends that you connect the on-board clock directly to the MIG core.*

The MIG core provides a user clock (ui_clock) and up to five additional clocks (four in case of UltraScale MIG) that can be used in the rest of the design. You can configure the connection as follows:

1. When using the platform board flow automation in a design that contains the MIG IP, Xilinx recommends that you add the MIG IP first (or drag and drop the ddr3_sdram interface from the Board window which instantiates the MIG core and configures it for the board), and then run Block Automation. This connects the on-board clock to the MIG core.

You can then customize MIG to generate additional clocks, if required.

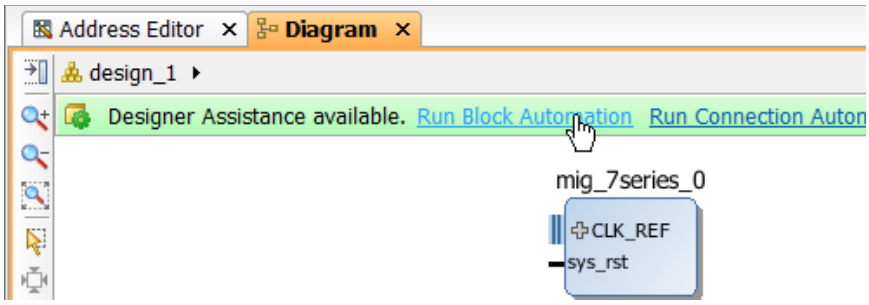


Figure 5-9: **Running Block Automation on the MIG Core**

2. The Run Connection Automation dialog box states that the ddr3_sdram interface is available. Click **OK**.

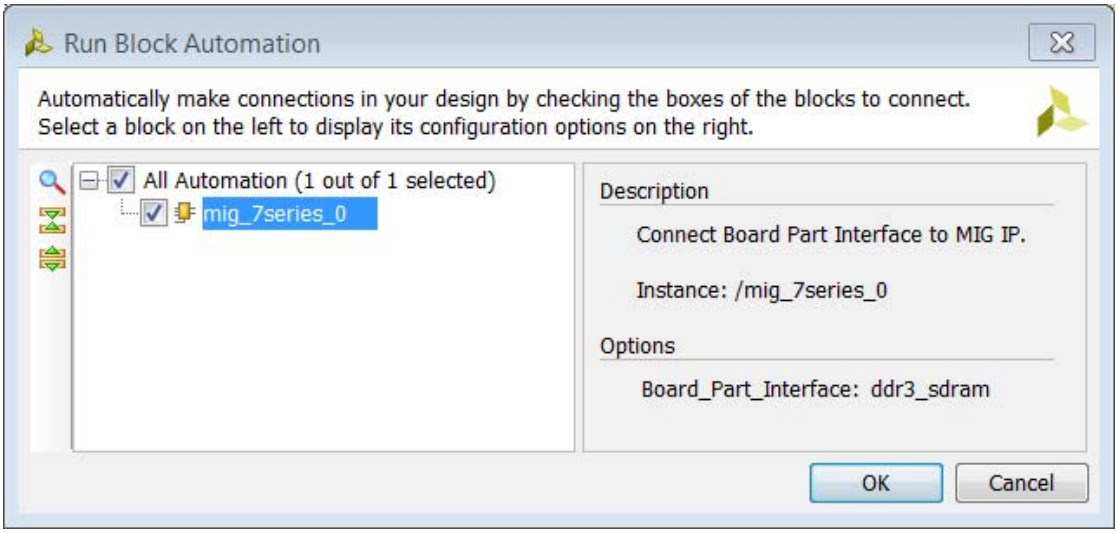


Figure 5-10: **Running Block Automation on the MIG Core**

This connects the interface ports to the MIG as shown in Figure 5-11.

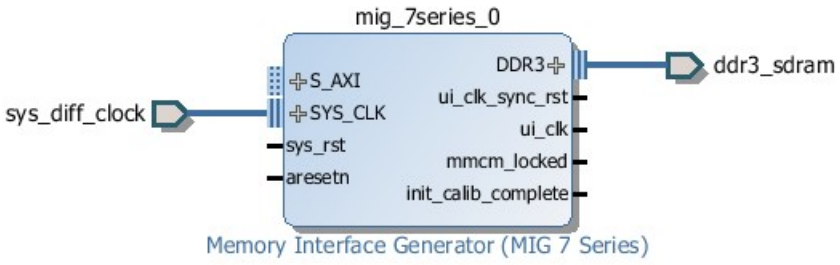


Figure 5-11: **Block Automation Creating the DDR3_SDRAM**

3. Add the MicroBlaze processor to the design and run Block Automation.

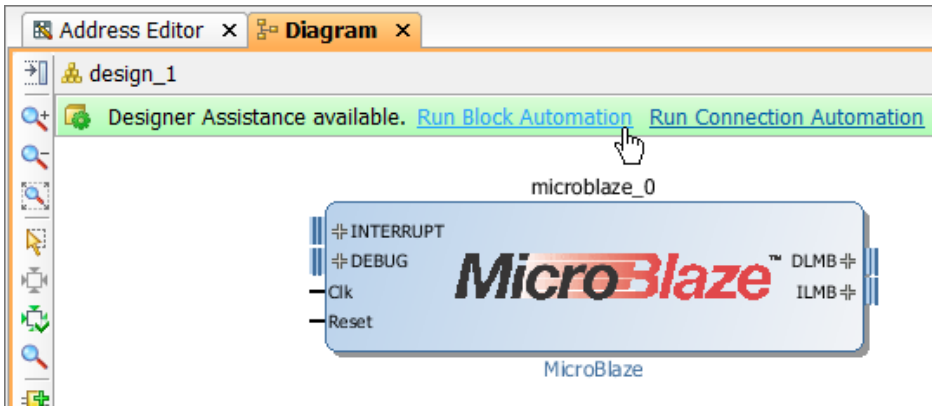


Figure 5-12: **Instantiating and Running Block Automation on the MicroBlaze**

4. In the **Clock Connection** field of the Run Block Automation dialog box, select the MIG ui_clk (**/mig_7series_1/ui_clk**) as the clock source for the MicroBlaze processor, and then click **OK**.

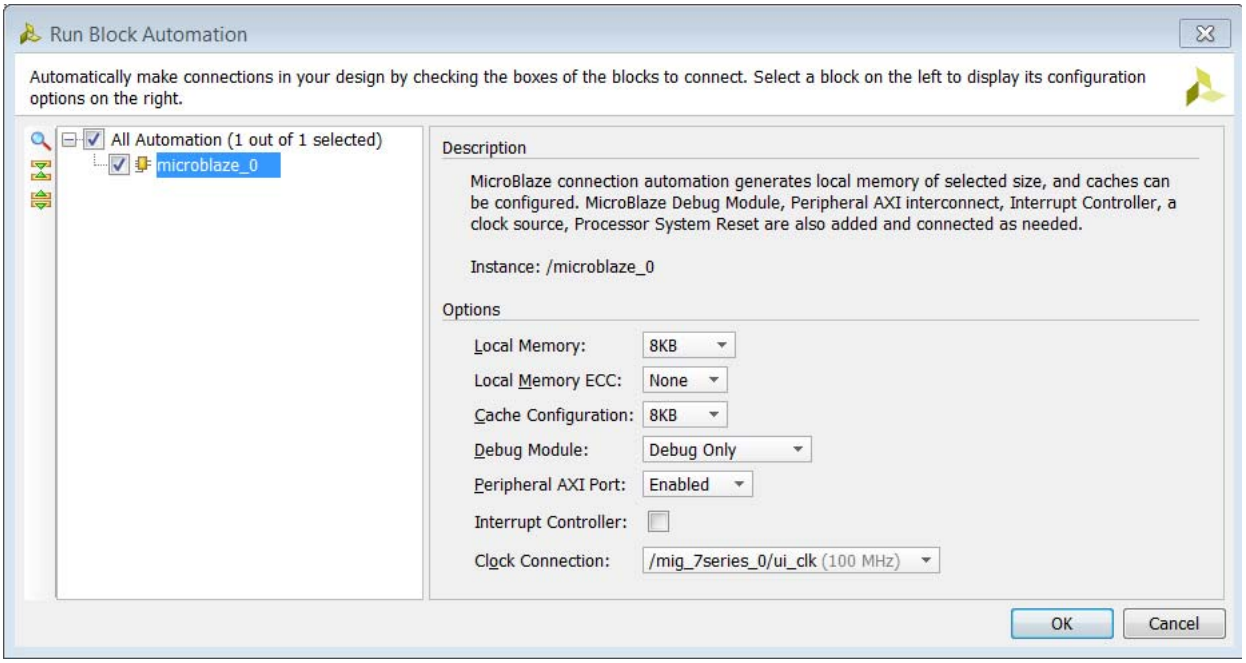


Figure 5-13: **Run Block Automation Options for the MicroBlaze Processor**

This creates a MicroBlaze subsystem and connects the `ui_clk` as the input source clock to the subsystem, as shown by the highlighted net in Figure 5-14.

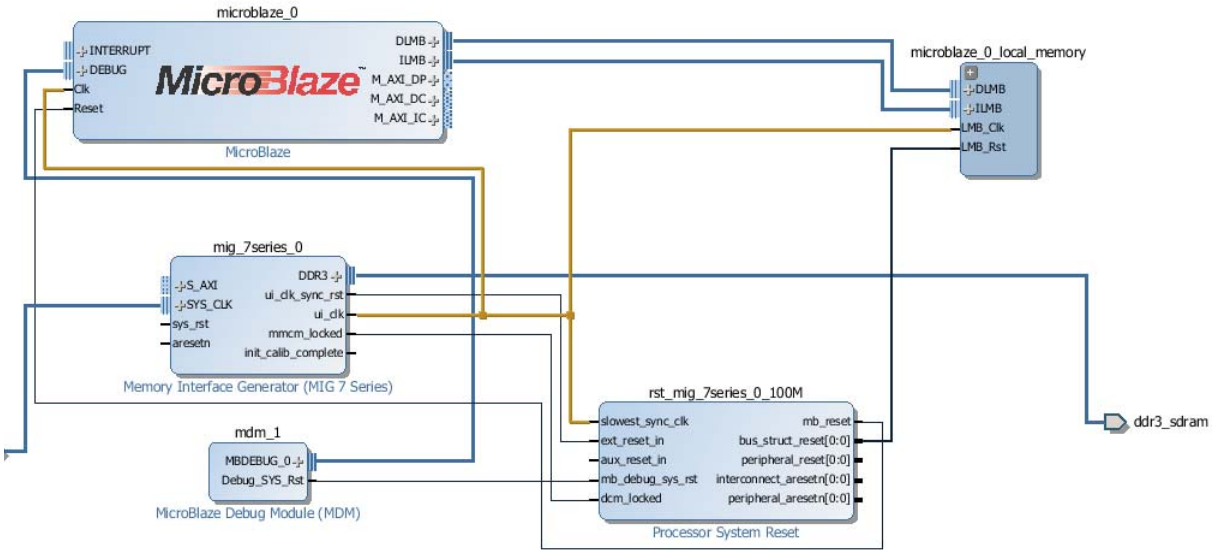


Figure 5-14: **Connect the Output Clock from the MIG Core to Clock the Design**

5. Make the following additional connections:

   a. Click **Connection Automation** and select `/mig_7series/S_AXI` to connect the MIG to MicroBlaze.

   b. In the Run Connection Automation dialog box select `/microblaze_0` (Cached) option for the S_AXI interface.

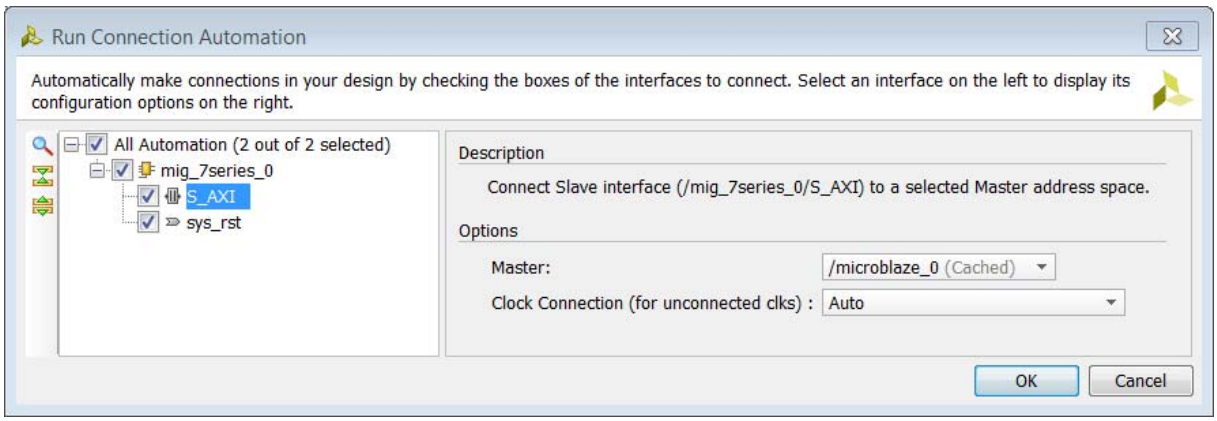


Figure 5-15: **Run Connection Automation Dialog Box**

   c. Connect the on-board reset to the `sys_rst` input of the MIG IP.

d. Click **OK**.

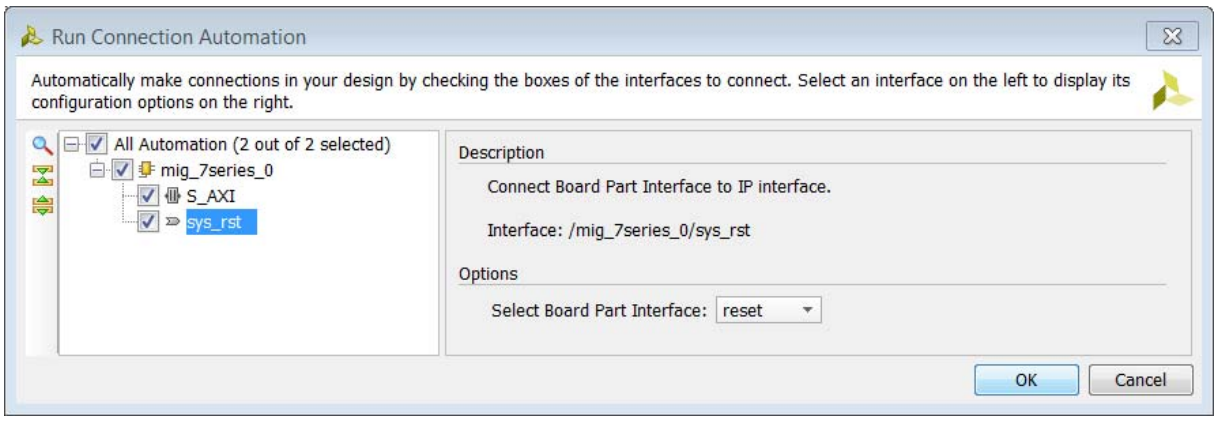


Figure 5-16: **Connecting sys_rst pin of the MIG to on-board reset**

Figure 5-17 shows the completed connection for MB-MIG with Designer assistance.

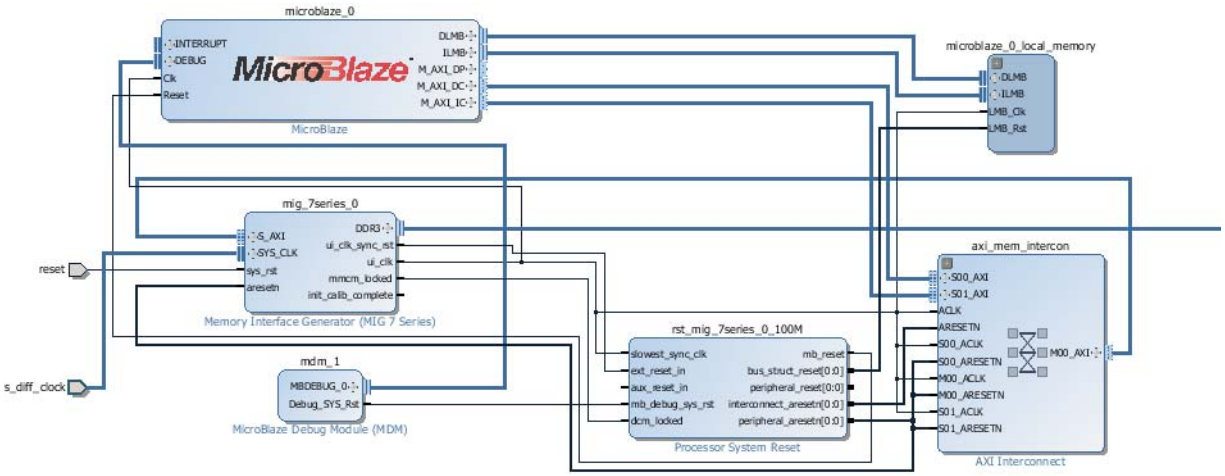


Figure 5-17: **Connect reset and mmcp_locked Pins**

Zynq Design without PL Logic

For Zynq designs without programmable logic (PL), all the clocks are contained in the ZYNQ7 Processing System IP. Use the following steps to add a Zynq design without PL.

1. After adding the ZYNQ7 Processing System IP, click **Run Block Automation** and select **/processing_system7_0**.

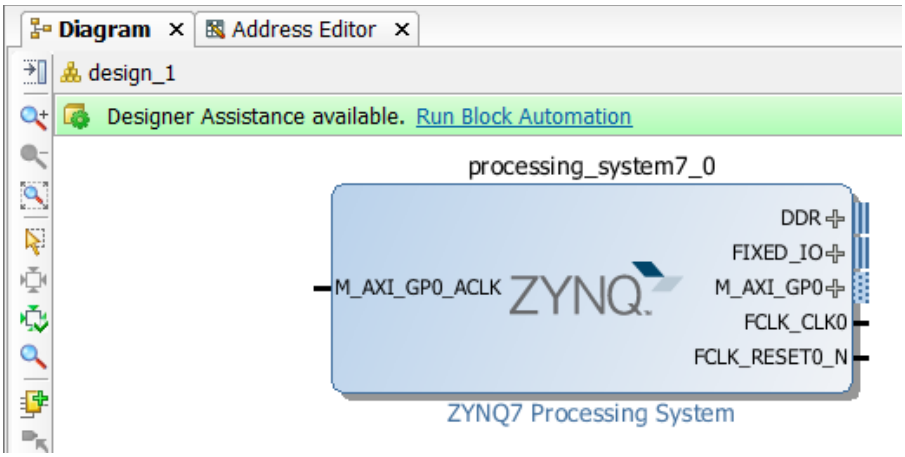


Figure 5-18: **Run Block Automation on Zynq**

2. The Run Block Automation states that the FIXED_IO and the DDR interfaces will be connected to external ports.
3. Click **OK**.

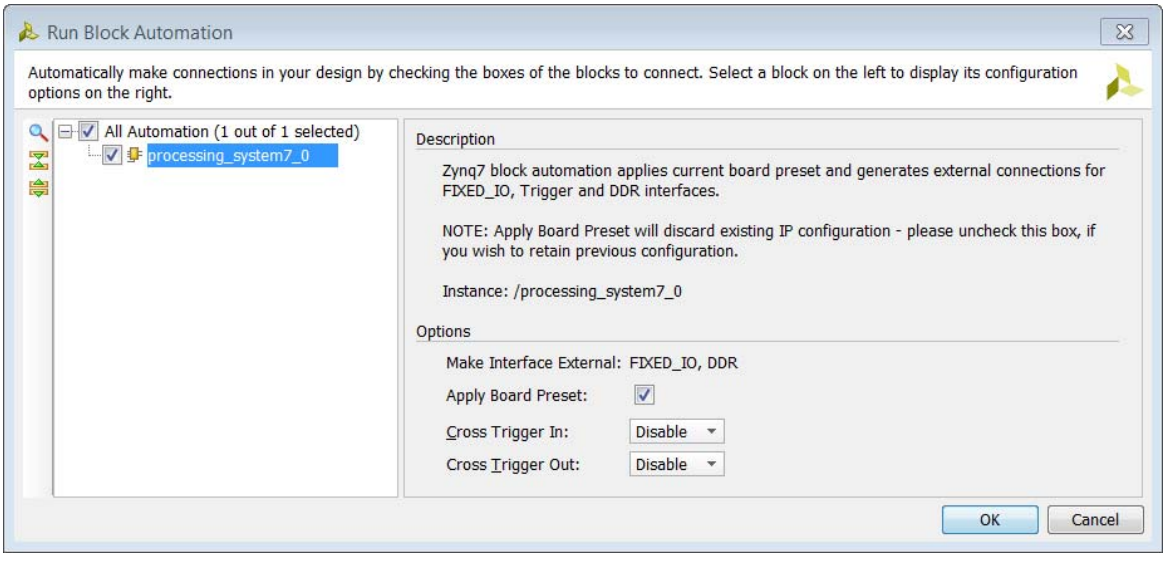


Figure 5-19: **Run Block Automation on the ZYNQ7 Processor**

4. Double-click the **ZYNQ7 Processing System** to re-customize the IP.

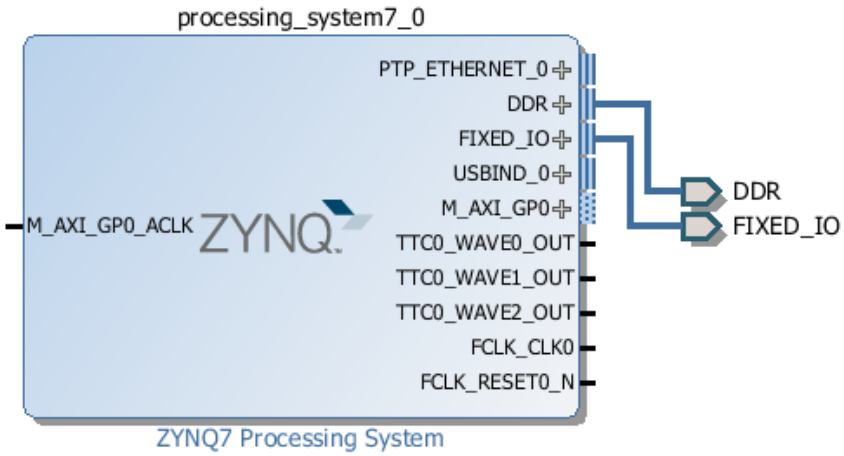


Figure 5-20: **Re-Customizing the ZYNQ7 IP**

5. Set the specific clocks in the Re-Customize_IP dialog box Clocking Configuration page.

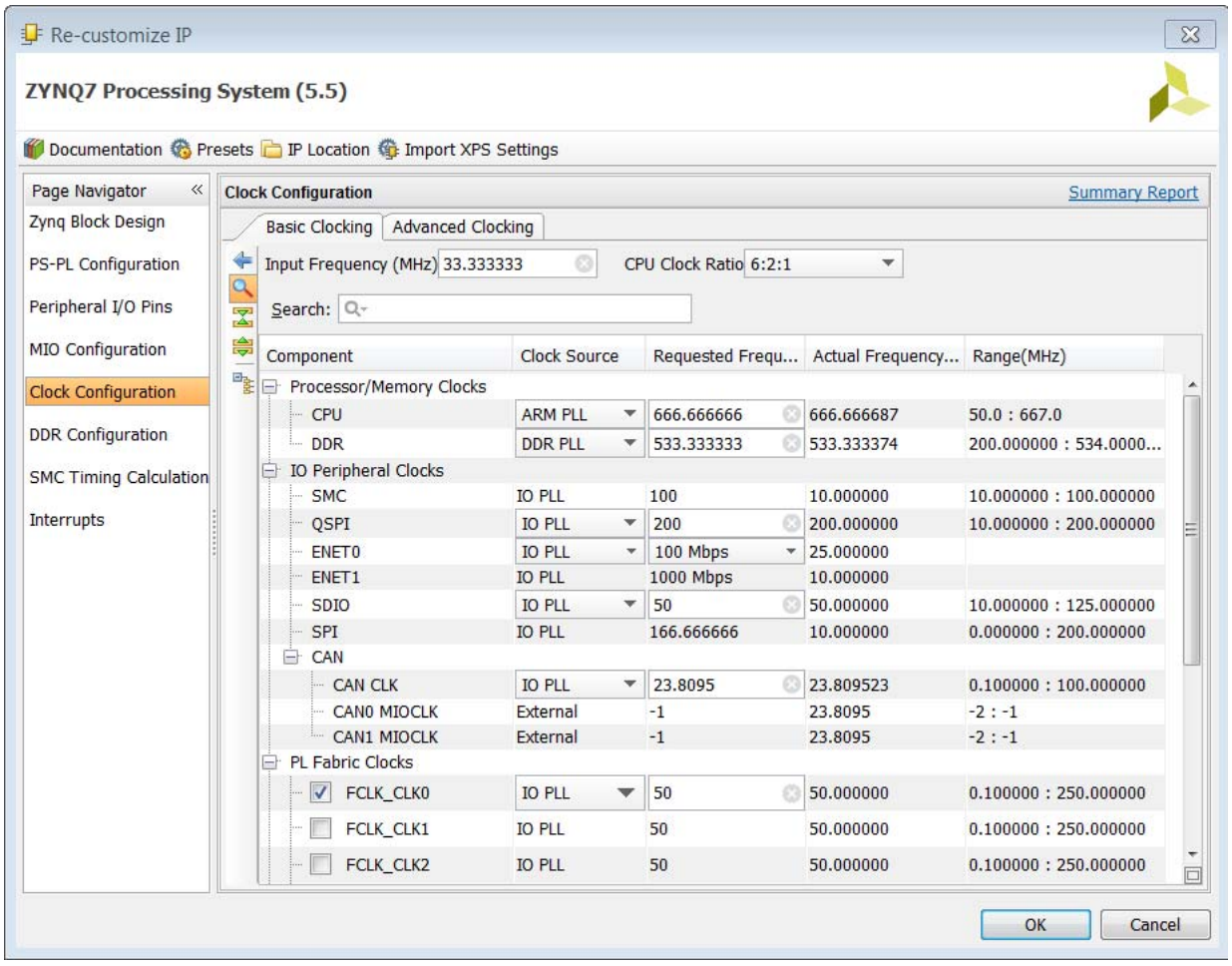


Figure 5-21: **Clock Configuration Options for the ZYNQ7 Processing System**

Zynq-7000 Design with PL Logic

RECOMMENDED: *For designs with a Zynq-7000 processor that contain custom logic in the PL fabric (but without MIG IP), it is recommended that the clocking and reset for the PL portion of the design be sourced from the PS. Any one of the PL Fabric Clocks-FCLK_CLK0, FCLK_CLK1, FCLK_CLK2 and FCLK_CLK3-can be used for the clock source. The associated resets for each of these clocks-FCLK_RESET0_N, FCLK_RESET1_N, FCLK_RESET2_N,and FCLK_RESET3_N-can be used for resetting the PL.*

Use the following steps to add a Zynq-7000 design with PL.

1. After adding the ZYNQ7 Processing System IP, click **Run Block Automation** and select **/processing_system7_0**.

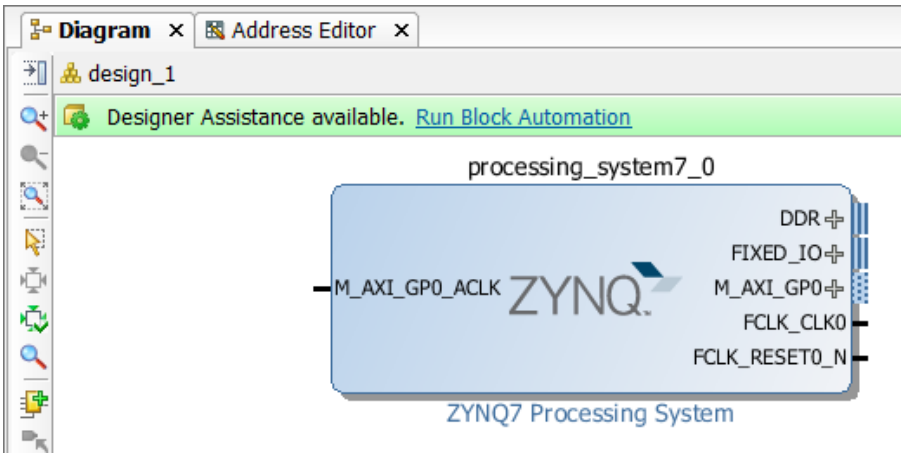


Figure 5-22: **Run Block Automation on the ZYNQ7 Processing System**

The Run Block Automation dialog box states that the FIXED_IO and the DDR interfaces will be connected to external ports.

2. Click **OK**.

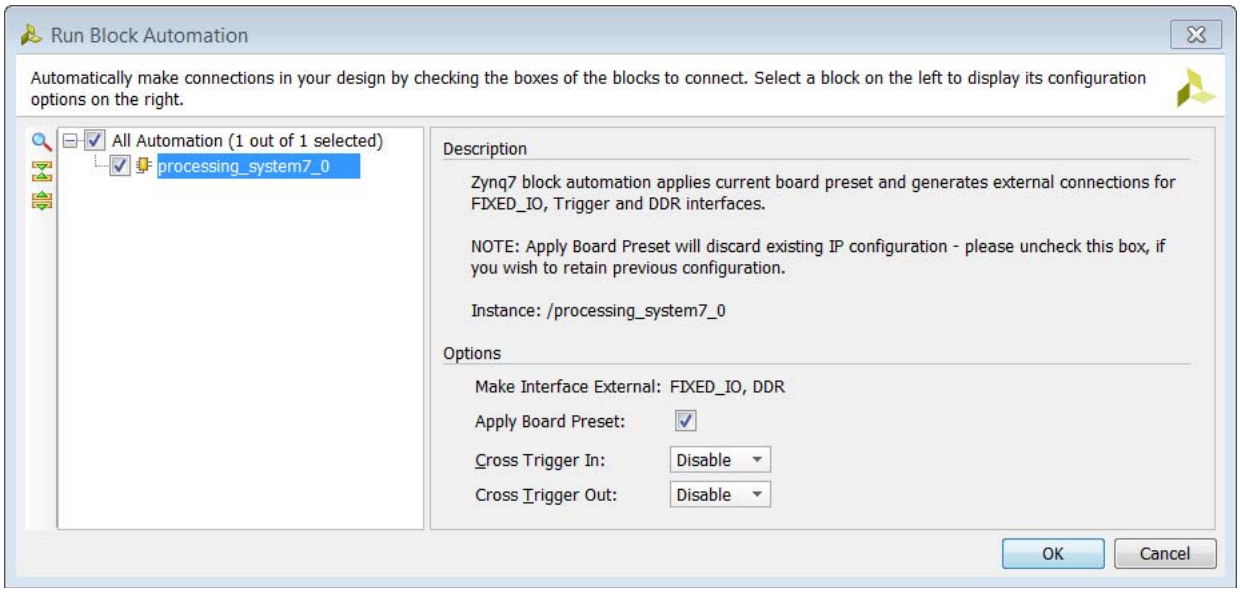


Figure 5-23: **Run Block Automation Dialog Box for the ZYNQ7 Processing System**

3. Double-click the ZYNQ7 Processing System to re-customize the IP.

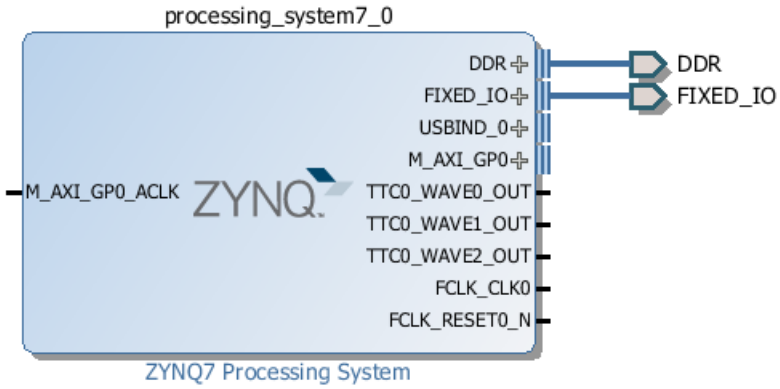


Figure 5-24: **Re-Customize the ZYNQ7 Processing System**

4. In the Re-customize IP dialog box, click **Clock Configuration** in the Page Navigator and then expand **PL Fabric Clocks**.

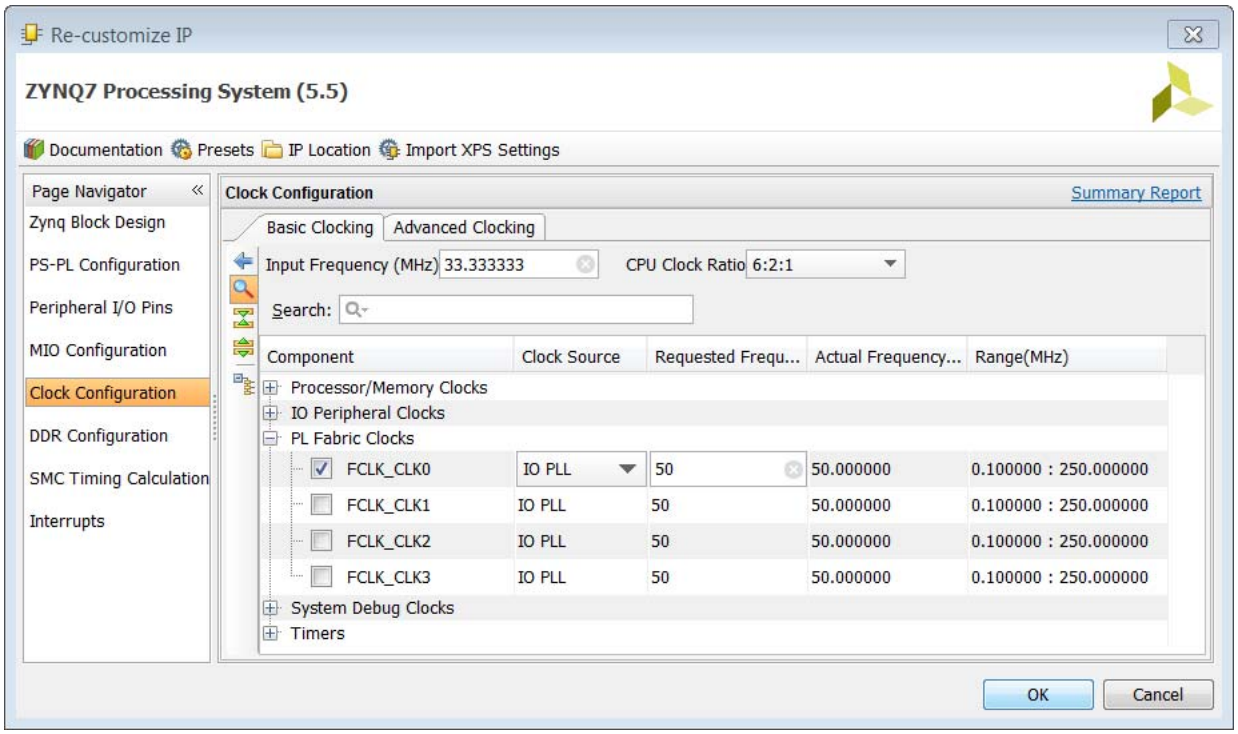


Figure 5-25: **Specify the Frequency of the Fabric Clock**

5. Click **PS-PL Configuration** in the Page Navigator and expand **General**.

6. Expand **Enable Clock Resets** and select the appropriate resets for the PL fabric.

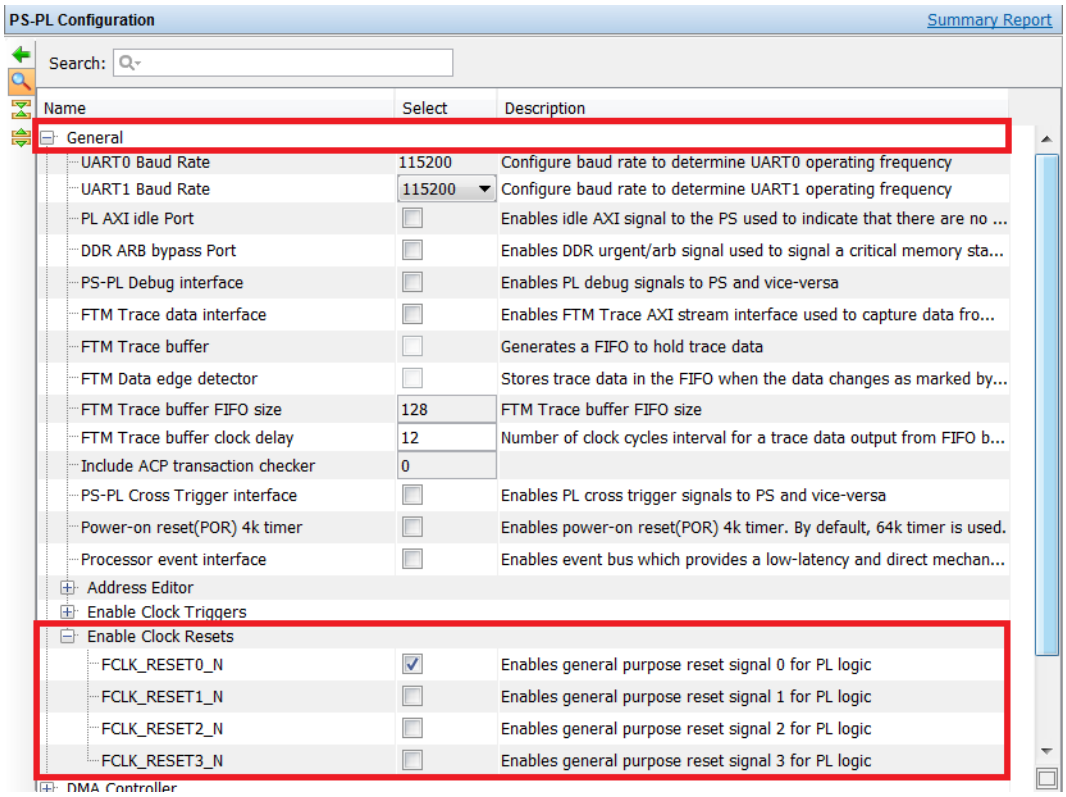


Figure 5-26: **Specify the Output Clock to the PL Fabric**

7. Instantiate an IP such as AXI GPIO in the PL fabric. Then, click **Run Connection Automation**.

   The Run Connection Automation dialog box states that the s_axi port of the GPIO will be connected to the ZYNQ7 Processing System master.

8. Click **OK**.

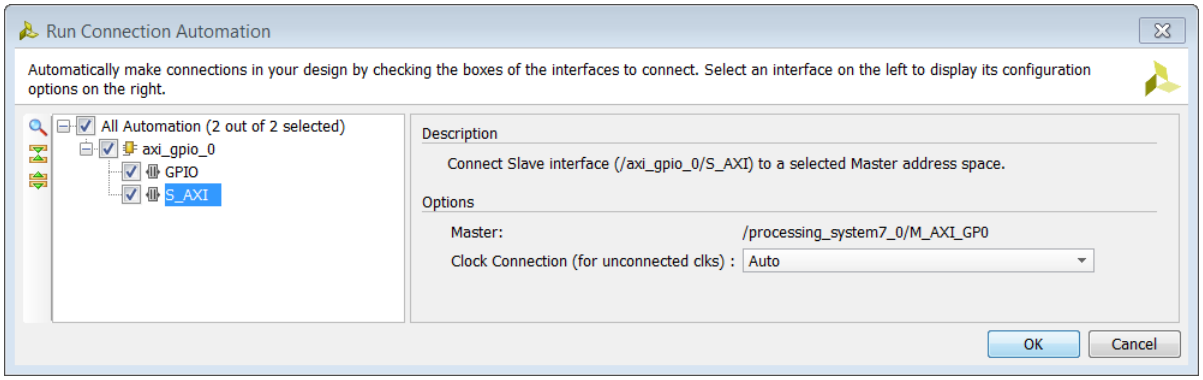


Figure 5-27: **Run Connection Automation Dialog Box to Connect GPIO**

The clock and resets in the IP integrator design should look as shown in the highlighted nets in Figure 5-28.

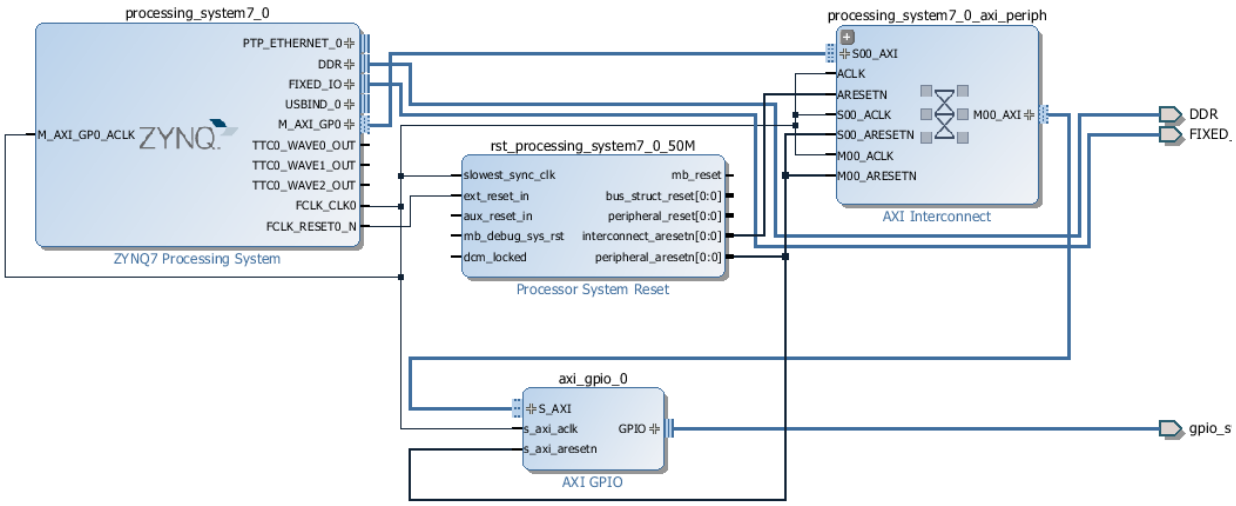


Figure 5-28: **Using the Output Clock from the ZYNQ PS7 IP to Clock the Design**

Zynq Design with a MIG core in the PL

RECOMMENDED: *For Zynq designs that include a MIG core in the PL, it is recommended that the input clock to the MIG core use an external clock source instead of the PS Fabric clock. The external clock from an on-board oscillator would be cleaner in terms of jitter when compared to clocks from the PS. You can use PS Fabric clocks for other portions of the PL design if required.*

1. Add the MIG IP and configure according to design requirements.
2. Then, connect the input clock source to the `SYS_CLK` input of the MIG core by right-clicking `SYS_CLK` in the block design and selecting **Create Interface Port**.
3. In the Create Interface Port dialog box, specify the options as shown in Figure 5-29.

4. Click **OK**.

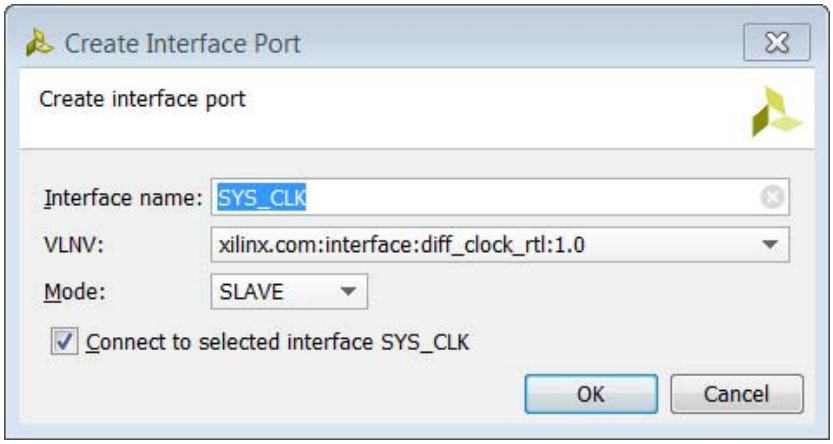


Figure 5-29: **Connecting the On-Board Clock Source to the MIG Core**

5. If the design uses a MicroBlaze processor, add it to the design and run **Block Automation**.
6. In this case, you select the MIG `ui_clk` as the clock connection.

   The Run Block Automation dialog box opens.

7. Specify **/mig_7series_1/ui_clk** as the input clock.

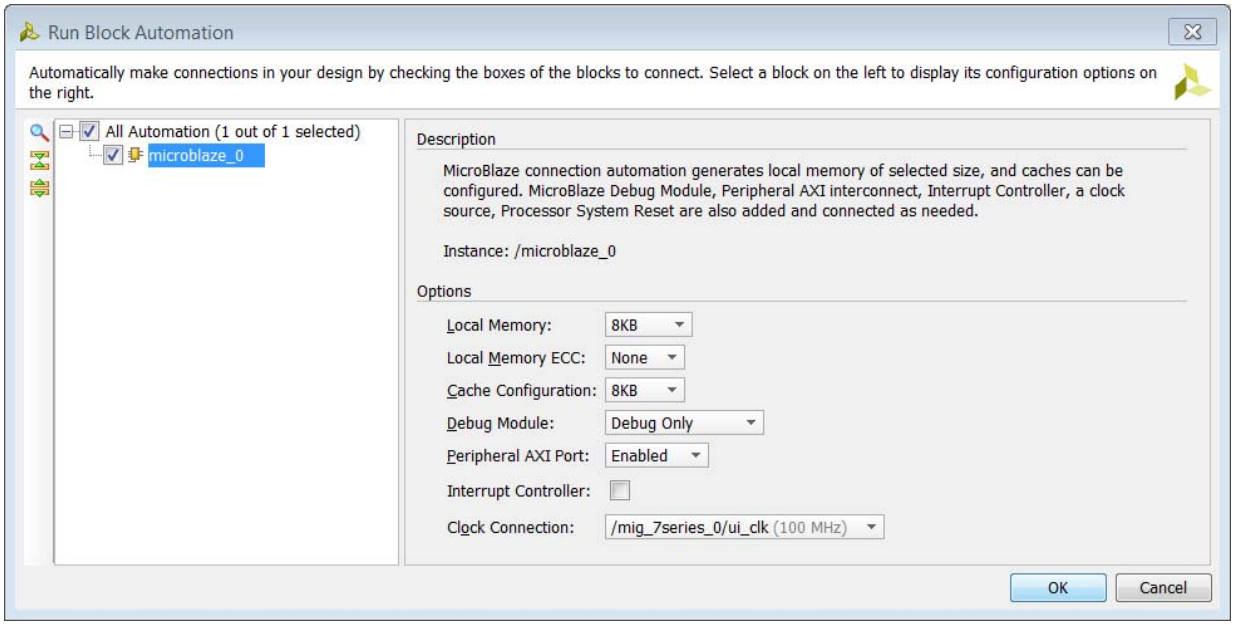


Figure 5-30: **Specifying MicroBlaze Options**

8. Click **OK**.

The block design looks like Figure 5-31.

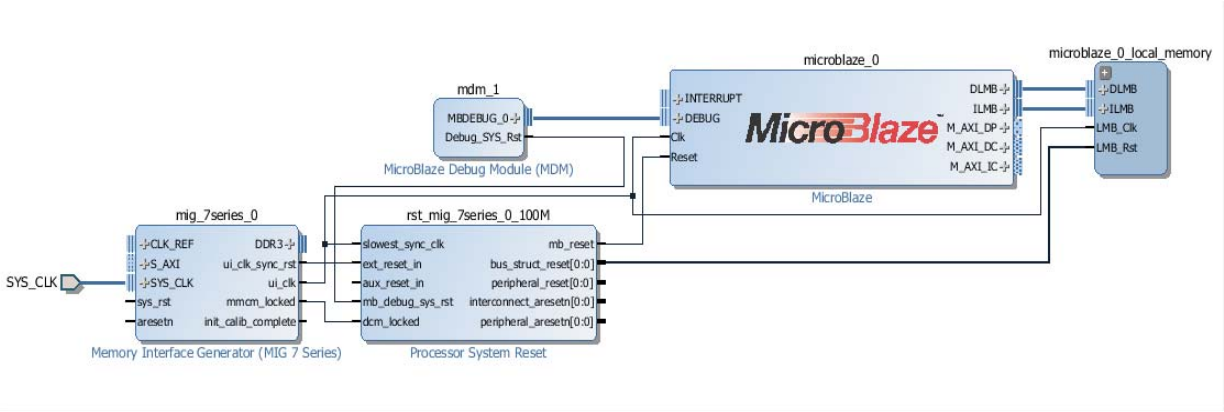


Figure 5-31: **Block Design after Running Block Automation on the MicroBlaze**

9. Connect the sys_rst pin of the MIG to an external reset source by selecting the sys_rst pin, right-clicking and selecting **Make External** from the menu.

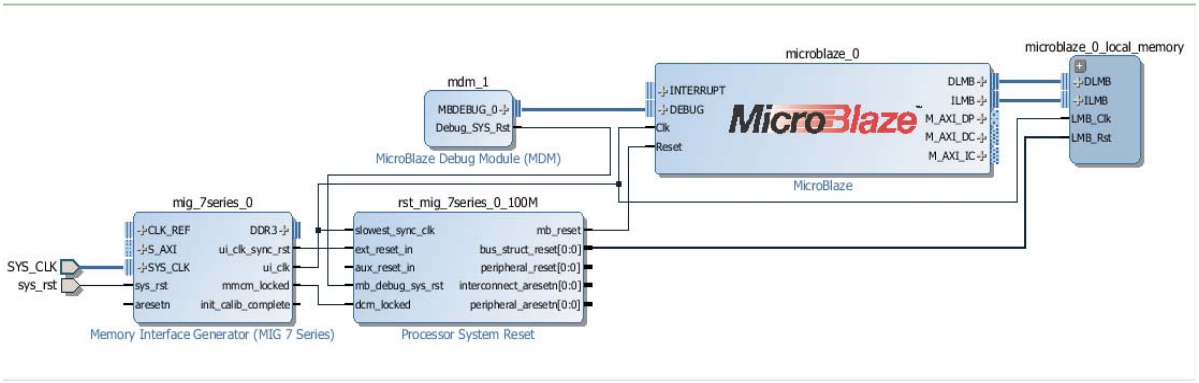


Figure 5-32: **Complete the Block Design**

Designs with MIG and the Clocking Wizard

For designs that require specific clock frequencies not generated by the MIG core, you can instantiate a Clocking Wizard IP and use the `ui_clock` output of the MIG IP as the clock input for the IP Clocking wizard.

You also need to make the following additional connections:

1. Connect the onboard reset to the Clocking Wizard reset input in addition to the MIG IP.
2. Connect the `mmcm_locked` pin of the MIG and locked pin of Clocking wizard to the `Util_Vector_Logic` IP configured to the AND operation. Then, connect the output of the `Util_Vector_Logic` to the `dcm_locked` input of `Proc_Sys_Reset`.

XILINX
ALL PROGRAMMABLE™

Chapter 6

Using UpdateMEM to Update BIT files with MMI and ELF Data

Overview

A single device, with one or more embedded processors, as well as programmable logic, needs a single boot image, which must contain the merged CPU software and FPGA bitstream images. UpdateMEM is a data translation tool to map contiguous blocks of data across multiple block RAMs which constitute a contiguous logical address space.

With the combination of Zynq®-7000 AP SoC device, or Microblaze embedded processors, on UltraScale architecture or 7 series devices, UpdateMEM merges the CPU software image of an Executable and Linkable Format (ELF) file into the FPGA bitstream created by the Vivado Design Suite and the write_bitstream command, by mapping the ELF data onto the memory map information (MMI) for the Block RAMs in the design. As a result, the software for an embedded processor can be initialized from block RAM-built address spaces within an FPGA bitstream. This provides a powerful and flexible means of merging parts of CPU software and FPGA design tool flows.

The Vivado Design Suite will automatically merge an associated ELF file for an embedded processor design when generating the device bitstream. If you have associated the ELF file using the **Tools > Associate ELF Files** command from the Vivado IDE the Vivado Design Suite will merge the data as needed.

Using the **Associate ELF Files** command will add the SCOPED_TO_REF and SCOPED_TO_CELLS properties to the associated ELF files. SCOPED_TO_REF associates the ELF file with all instances of the specified hierarchical module, or block design, and SCOPED_TO_CELLS associates the ELF file with specified instances of the specified embedded processor cells.

You can also run the UpdateMEM command at any time to manually associate the ELF file and MMI file with the BIT file of the implemented design.

IMPORTANT: *UpdateMEM can only be used to update un-encrypted bitstream files.*

UpdateMEM

For embedded processor based designs the UpdateMEM command merges CPU software images into bitstream files, to initialize the block RAM memory within the target Xilinx device.

The UpdateMEM command takes the following inputs:

- A bitstream (BIT) file is initially generated by the Vivado Design Suite implementation tools. You can create a bitstream file from an implemented design using the **write_bitstream** Tcl command. A bitstream (BIT) file is a binary data file that contains a bit image of the design, to be downloaded to a Xilinx device. The UpdateMEM command reads a BIT file as an input, and writes a BIT file as its output.
- The memory map information (MMI) file is a text file that describes how individual Block RAMs on the Xilinx device are grouped together to form a contiguous address space called an Address Block. The MMI file is automatically written by the Vivado Design Suite and placed into the `<project>.runs/impl_1` folder when generating the bistream, or can be manually written by the **write_mem_info** command. The UpdateMEM command uses the MMI file to identify the physical BRAM resources that map to a specific address range. For more information on the MMI file, refer to BRAM Memory Map Info (MMI) File, page 121.
- An Executable and Linkable Format (ELF) file, which is a product of the software development kit (SDK), is a binary data file that contains an executable program image ready for running on an embedded processor. The ELF file contains the data to be mapped by UpdateMEM into the address ranges of the BRAMs.
- Optionally, a memory (MEM) file is a manually created text file that describes contiguous blocks of data to initialize or populate a specified address space. The UpdateMEM command can use the MEM file in place of an ELF file. See Memory (MEM) Files, page 120 for more information.
- The instance ID of the embedded processor in the design, in order to associate the ELF or MEM file with the processor.

The UpdateMEM command populates contiguous blocks of data defined in ELF or MEM files, across multiple block RAMs of a Xilinx device mapped by the MMI file. The UpdateMEM command merges the memory information into a bitstream file for configuring and programming the target Xilinx device.

The UpdateMEM command also lets you merge multiple data files for multiple processors in designs that have more than one embedded processor. In this case, the **-data** and **-proc** options must be specified in pairs, with the first **-data** file providing the software image or memory content for the first **-proc** specified. The second **-data** applies to the second **-proc**., and so on.

This command returns the name of the bitstream file created from the inputs, or returns an error if it fails.

Arguments:

-meminfo <arg> - (Required) Name of the memory mapping information (MMI) file for the implemented design.

-data <arg> - (Required) Name of the Executable and Linkable Format (ELF) file, or a MEM file to map into BRAM addresses.

-bit <arg> - (Required) Name of the input bitstream (BIT) file. If the file extension is missing, an extension of .bit is assumed.

Note: The UpdateMEM command can only be used with un-encrypted bitstream files.

-proc <arg> - (Required) Instance path of the embedded processor.

TIP: *You can specify multiple processors for the UpdateMEM command in cases where a design has multiple embedded processors. In this case the* ***-data*** *and* ***-proc*** *options must be specified in pairs, with the first* ***-data*** *argument applying to the first* ***-proc*** *argument. However, the UpdateMEM command can take either ELF file or MEM file in a single run, but cannot use both* ***-data*** *formats at the same time even when specifying multiple processors.*

-out <arg> - (Required) Specify the name of output file, without suffix. The file will have a suffix of **.bit** applied automatically.

-force - (Optional) Overwrite the specified output file if it already exists.

Examples:

The following example reads the specified MEM info file, ELF file, and bitstream file, and generates the merged bitstream file:

```
updatemem -meminfo top.mmi -data hello_world.elf -bit top.bit -proc
design_1_i/microblaze_1 -out top_meminfo.bit
```

The following example shows the use of UpdateMEM in a block design that has two embedded microblaze processors, one with an associated ELF file, and the other using a MEM file. Notice this requires two passes of the updatemem command, with the output bitstream of the first acting as the input bitstream of the second:

```
updatemem -bit top.bit -meminfo top.mmi -data top1.elf \
-proc system_i/microblaze_1 -out first_out.bit

updatemem -bit first_out.bit -meminfo top.mmi -data top2.mem  \
-proc system_i/microblaze_2 -out top_out.bit
```

Memory (MEM) Files

A Memory (MEM) file is a manually edited text file that describes contiguous blocks of data. that can be used in place of the ELF file. The format of MEM files is an industry standard, consisting of two basic elements:

- Hexadecimal address specifier - An address specifier is indicated by an @ character followed by the hexadecimal address value. There are no spaces between the @ character and the first hexadecimal character.
- Hexadecimal data values - Hexadecimal data values follow the hexadecimal address value, separated by spaces, tabs, or carriage-return characters.
- Because the MEM file is in hex format, each character represents four bits, or a nibble, in the memory.

Hexadecimal data values can consist of as many hexadecimal characters as desired. However, when a value has an odd number of hexadecimal characters, the first hexadecimal character is assumed to be a zero. For example, hexadecimal values A, C74, and 84F21 are interpreted as the values 0A, 0C74, and 084F21 respectively.

IMPORTANT: *The common **0x** hexadecimal prefix is not allowed. Using this prefix on MEM file hexadecimal values is flagged as a syntax error.*

There must be at least one data value following an address, up to as many data values that belong to the previous address value. Following is an example of the most common MEM file format:

```
@0000 3A @0001 7B @0002 C4 @0003 56 @0004 02
@0005 6F @0006 89...
```

UpdateMEM requires a less redundant format. An address specifier is used only once at the beginning of a contiguous block of data. The previous example is rewritten as:

```
@0000 3A 7B C4 56 02 6F 89...
```

The address for each successive data value is derived according to its distance from the previous address specifier. A MEM file can have as many contiguous data blocks as required. While the gap of address ranges between data blocks can be any size, no two data blocks can overlap an address range.

TIP: *UpdateMEM allows the free-form use of both // and /*...*/ commenting styles in the MEM file.*

The Vivado Design Suite also supports a MEM File format for memory initialization as described at this link in the *Vivado Design Suite User Guide: Synthesis* (UG901) [Ref 2]. The MEM File format supported by the Vivado Design Suite is different from the file format supported by UpdateMEM.

You should define the MEM file structure for Vivado tools to match the synthesis view of the memory as an array, which adheres to the Verilog language specification. The MEM file used for UpdateMEM should include spaces to match the `<Datawidth>` tag as defined in the memory map info (MMI) file. See MMI File Syntax, page 124 for more information.

According to the Verilog language specification, the memory is treated as an array, so for Vivado synthesis the MEM file for a 64k memory (256x256 array) should look as follows:

```
@00000000
aaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbbb
```

Note: White space and/or comments are used to separate the numbers.

However, for the UpdateMem command, which has a post implementation physical view of the memory, the MEM file should look as follows:

```
@00000000
aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa aaaaaaaaaaaaaaaa
bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb bbbbbbbbbbbbbbbb
```

Note: For UpdateMEM, the spaces that separate the words are determined by the MSB and LSB attributes of the <Datawidth> tag defined in the MMI file.

BRAM Memory Map Info (MMI) File

The following are design considerations for block RAM-implemented address spaces, and the definition of memory map info files:

- The block RAMs come in fixed-size widths and depths, where CPU address spaces might need to be much larger in width and depth than a single block RAM. Consequently, multiple block RAMs must be logically grouped together to form a single CPU address space as seen in Figure 6-1, page 122.
- A single CPU bus access is often multiple bytes wide of data, for example, 32 or 64 bits (4 or 8 bytes) at a time.
- CPU bus accesses of multiple data bytes might also access multiple block RAMs to obtain that data. Therefore, byte-linear CPU data must be interleaved by the bit width of each block RAM and by the number of block RAMs in a single bus access. However, the relationship of CPU addresses to block RAM locations must be regular and easily calculable.
- CPU data must be located in a block RAM-constructed memory space relative to the CPU linear addressing scheme, and not to the logical grouping of multiple block RAMs.
- Address space must be contiguous, and in whole multiples of the CPU bus width. Bus bit lane interleaving is allowed only in the sizes supported by Virtex® device block RAM port sizes.

- Addressing must account for the differences in instruction and data memory space. Because instruction space is not writable, there are no address width restrictions. However, data space is writable and usually requires the ability to write individual bytes. For this reason, each bus bit lane must be addressable.
- The size of the memory map and the location of the individual block RAMs affect the access time. Evaluate the access time after implementation to verify that it meets the design specifications.

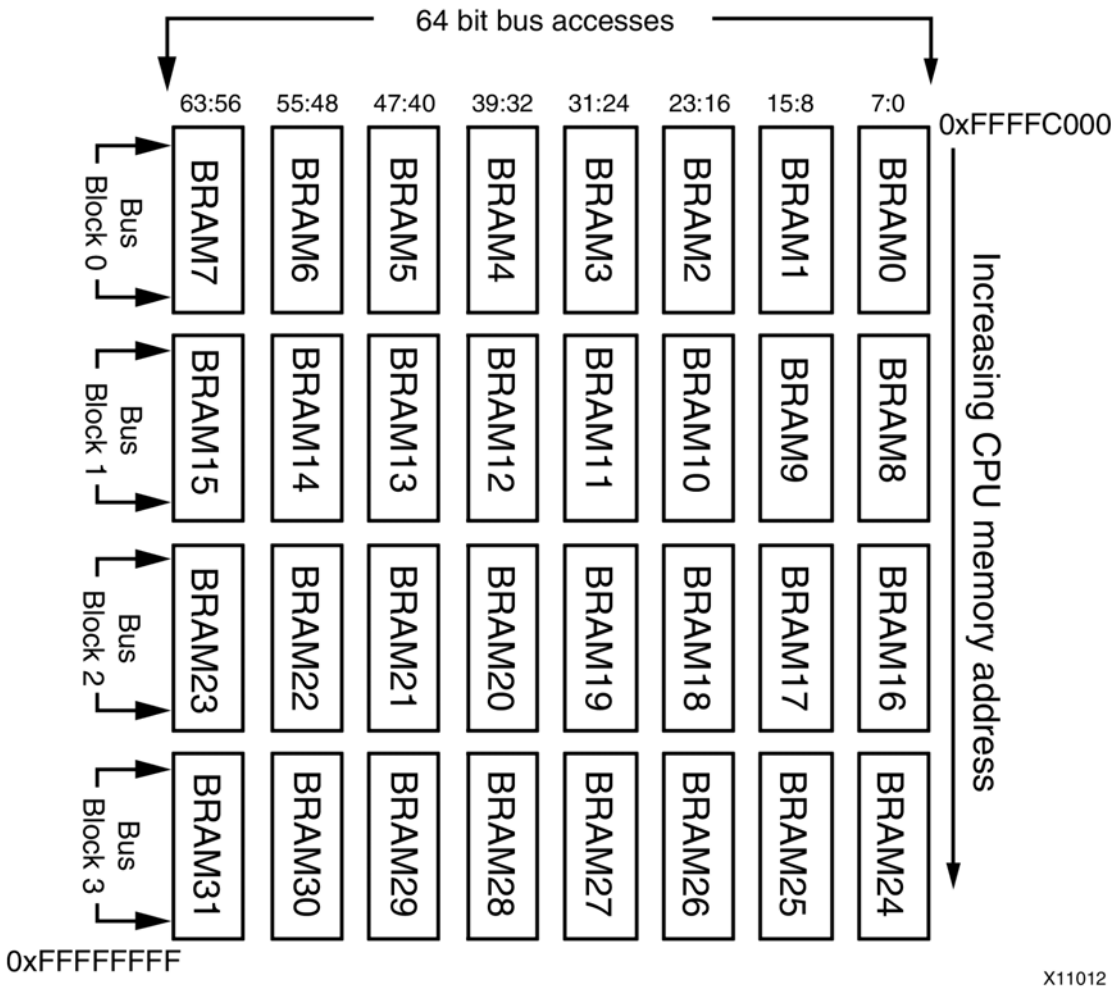


Figure 6-1: **Block RAM Address Space**

The address space in the figure above consists of four bus blocks: Bus Block 0 through 3.

- CPU bus accesses are 8 block RAMs (64 bits) wide, with each column of block RAMs occupying an 8-bit wide slice of a CPU bus access called a Bit Lane.
- Each row of 8 block RAMs in a bus access are grouped together in a Bus Block. Hence, each Bus Block is 64 bits wide and 4096 bytes in size.

- The entire collection of block RAMs is grouped together into a contiguous address space called an Address Block.

The upper right corner address is 0xFFFFC000, and the lower left corner address is 0xFFFFFFFF. Because a bus access obtains 8 data bytes across 8 block RAMs, byte-linear CPU data must be interleaved by 8 bytes in the block RAMs.

In this example using a 64 bit data word indexed by bytes from left to right as [0:7], [8:15]:

- Byte 0 goes into the first byte location of bit lane block RAM7, byte 1 goes into the first byte location of Bit Lane block RAM6; and so forth, to byte 7.
- CPU data byte 8 goes into the second byte location of Bit Lane block RAM7, byte 9 goes into the second byte location of Bit Lane block RAM6 and so forth, repeating until CPU data byte 15.
- This interleave pattern repeats until every block RAM in the first bus block is filled.
- This process repeats for each successive bus block until the entire memory space is filled, or the input data is exhausted.

As described in MMI File Syntax, page 124 the order in which bit lanes and bus blocks are defined controls the filling order. For the sake of this example, assume that bit lanes are defined from left to right, and bus blocks are defined from top to bottom.

This process is called Bit Lane Mapping, because these formulas are not restricted to byte-wide data. This is similar, but not identical, to the process embedded software programmers use when programmed CPU code is placed into the banks of fixed-size EPROM devices.

The important distinctions to note between the two processes are:

- Embedded system developers generally use a custom software tool for byte lane mapping for a fixed number and organization of byte-wide storage devices. Because the number and organization of the devices cannot change, these tools assume a specific device arrangement. Consequently, little or no configuration options are provided. By contrast, the number and organization of FPGA block RAMs are completely configurable (within FPGA limits). Any tool for byte lane mapping for block RAMs must support a large set of device arrangements.
- Existing byte lane mapping tools assume an ascending order of the physical addressing of byte-wide devices because that is how board-level hardware is built. By contrast, FPGA block RAMs have no fixed usage constraints and can be grouped together with block RAMs anywhere within the FPGA fabric. Although this example displays block RAMs in ascending order, block RAMs can be configured in any order.

Memory Map Info (MMI) Features

A Memory Map Info (MMI) file is an XML file designed for computer parsing. It is similar to high-level computer programming languages in using the following features:

- Block structures by XML keyword tags or directives

  MMI maintains similar structures in groups or blocks of data. MMI creates blocks to delineate address space, bus access groupings, and comments.

- Symbolic name usage

  MMI uses names and keywords to refer to groups or entities (improving readability), and uses names to refer to address space groupings and Block RAMs.

MMI observes the following conventions:

- Keywords are case-sensitive
- Indenting is for clarity only.
- White space is ignored except where it delineates items or keywords.
- Line endings are ignored. You can have as many items as you want on a single line.
- Numbers can be entered as decimal or hexadecimal. Hexadecimal numbers use the 0xXXX notation form.

MMI File Syntax

The Memory Map Info (MMI) file is an XML file that syntactically describes how individual block RAMs make up a contiguous logical data space. You can create an MMI file from an open implemented design in the Vivado Design Suite using the **write_mem_info** Tcl command. The implemented design provides the needed placement information of the Block RAM resources.

UpdateMEM uses the MMI file as input to direct the translation of data into the proper initialization form. The Example MMI file below shows the XML-based syntax used to describe the organization of block RAM usage.

```
<?xml version="1.0" encoding="UTF-8"?>
<MemInfo Version="1" Minor="0">
  <Processor Endianness="Little" InstPath="design_1_i/microblaze_0">
    <AddressSpace
Name="design_1_i_microblaze_0.design_1_i_microblaze_0_local_memory_dlmb_bram_if_cnt
lr" Begin="0" End="8191">
      <BusBlock>
        <BitLane MemType="RAMB32" Placement="X2Y17">
          <DataWidth MSB="15" LSB="0"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
```

```
        <BitLane MemType="RAMB32" Placement="X3Y17">
          <DataWidth MSB="31" LSB="16"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
      </BusBlock>
    </AddressSpace>
  </Processor>
  <Processor Endianness="Little" InstPath="design_1_i/microblaze_1">
    <AddressSpace
Name="design_1_i_microblaze_1.design_1_i_microblaze_1_local_memory_dlmb_bram_if_cnt
lr" Begin="0" End="8191">
      <BusBlock>
        <BitLane MemType="RAMB32" Placement="X4Y13">
          <DataWidth MSB="15" LSB="0"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
        <BitLane MemType="RAMB32" Placement="X4Y14">
          <DataWidth MSB="31" LSB="16"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
      </BusBlock>
    </AddressSpace>
  </Processor>
  <Processor Endianness="Little" InstPath="design_1_i/processing_system7_0">
    <AddressSpace Name="design_1_i_processing_system7_0.design_1_i_axi_bram_ctrl_0"
Begin="1073741824" End="1073750015">
      <BusBlock>
        <BitLane MemType="RAMB32" Placement="X2Y18">
          <DataWidth MSB="15" LSB="0"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
        <BitLane MemType="RAMB32" Placement="X2Y19">
          <DataWidth MSB="31" LSB="16"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
      </BusBlock>
    </AddressSpace>
  </Processor>
  <Config>
    <Option Name="Part" Val="xc7z020clg484-1"/>
  </Config>
</MemInfo>
```

Address Map Definitions (Multiple Processor Support)

UpdateMEM supports multiple processors using the following XML tags:

```
<Processor Endianness="Little" InstPath="design_1_i/processing_system7_0">
</Processor>
```

Address Space Definitions

The outermost definition of an address space is composed of the following components:

```
<AddressSpace Name="design_1_i_processing_system7_0.design_1_i_axi_bram_ctrl_0"
Begin="1073741824" End="1073750015">
</AddressSpace>
```

The ADDRESS_SPACE and /ADDRESS_SPACE tags define a single contiguous address space. The mandatory **Name=** following the ADDRESS_SPACE tag provides a symbolic name for the entire address space. Referring to the address space name is the same as referring to the entire contents of the address space.

An MMI file can contain multiple ADDRESS_SPACE definitions, even for the same address space, as long as each ADDRESS_SPACE name is unique.

Next is the beginning and ending address values that the Address Space occupies by using the **Begin=** and **End=** pair.

Bus Block Definitions (Bus Accesses)

Inside an ADDRESS_SPACE definition are a variable number of sub-block definitions called Bus Blocks.

```
<BusBlock>
</BusBlock>
```

Each Bus Block contains block RAM Bit Lane definitions that are accessed by a parallel CPU bus access.

The order in which the Bus Blocks are specified defines which part of the address space a Bus Block occupies. The lowest addressed Bus Block is defined first, and the highest addressed Bus Block is defined last. The top-to-bottom order in which Bus Blocks are defined also controls the order in which UpdateMEM fills those Bus Blocks with data.

Bit Lane Definitions (Memory Device Usage)

A Bit Lane definition determines which bits in a CPU bus access are assigned to particular block RAMs. Each definition takes the form of MemType with Placement data, followed by the bit numbers and AddressRange the Bit Lane occupies. The syntax is as follows:

```
<BitLane MemType="RAMB32" Placement="X2Y19">
          <DataWidth MSB="31" LSB="16"/>
          <AddressRange Begin="0" End="2047"/>
          <Parity ON="false" NumBits="0"/>
        </BitLane>
```

Note: Although Parity is defined in the MMI file, it is not supported by UpdateMEM at this time.

Normally the bit numbers are given in the following order:

```
<DataWidth MSB=bit_num LSB=bit_num>
```

If the order is reversed to have the Least Significant Bit (LSB) first and the Most Significant Bit (MSB) second, UpdateMEM bit-reverses the Bit Lane value before placing it into the block RAM.

As with Bus Blocks, the order in which Bit Lanes are defined is important. But in the case of Bit Lanes, the order infers which part of Bus Block CPU access a Bit Lane occupies. The first Bit Lane defined is inferred to be the most significant Bit Lane value, and the last defined is the least significant Bit Lane value. In the figure below, the most significant Bit Lane is BRAM7, and the least significant Bit Lane is BRAM0. As seen in Example Block RAM Address Space Layout, this corresponds with the order in which the Bit Lanes are defined.

When UpdateMEM inputs data, it takes data from data input files in Bit Lane sized chunks, from the most right value first to the left most. For example, if the first 64 bits of input data are 0xB47DDE02826A8419 then the value 0xB4 is the first value to be set into a Block RAM.

Given the Bit Lane order, BRAM7 is set to 0xB4, BRAM6 to 0x7D, and so on until BRAM0 is set to 0x19. This process repeats for each successive Bus Block access BRAM set until the memory space is filled or until the input data is exhausted. The figure below expands the first Bus Block to illustrate this process.

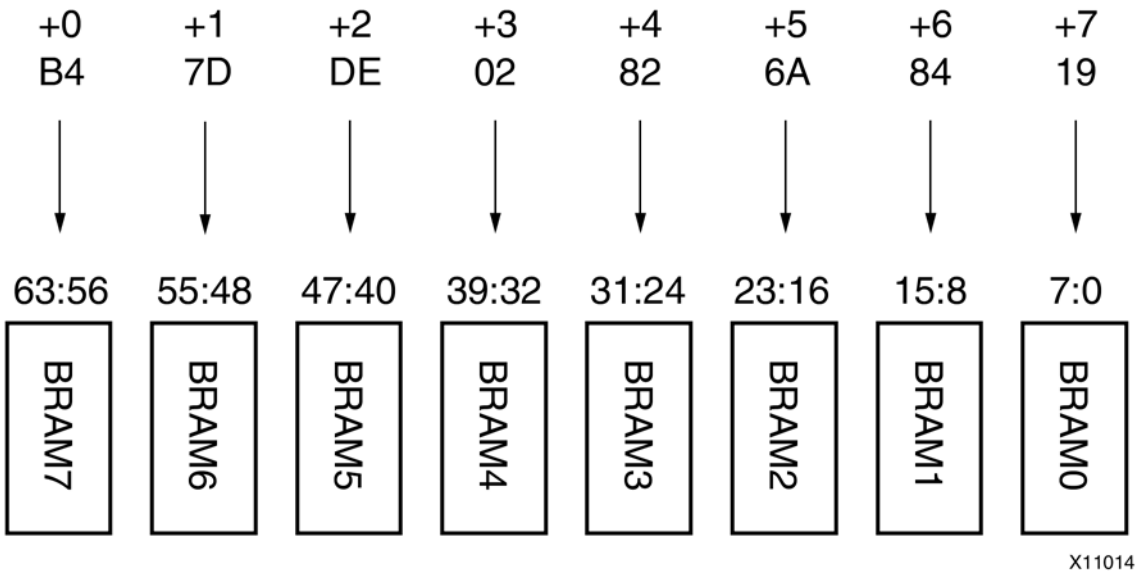


Figure 6-2: **Bit Lane Fill Order**

The Bit Lane definitions must match the hardware configuration. If the MMI is defined differently from the way the hardware actually works, the data retrieved from the memory components will be incorrect.

Bit Lane definitions also have some optional syntax, depending on what device type keyword is used in the Address Block definition.

When specifying Block RAM cells, the physical row and column location within the FPGA device can be indicated. Following are examples of the physical row and column location:

```
Placement="X3Y5"
```

Use the **Placement=** keyword to assign the corresponding block RAM to a specific resource location in the FPGA device. In this case the Block RAM is placed at column 3 and row 5 in the FPGA device.

In addition to using correct syntax for Bit Lane and Bus Block definitions, you must take into account the following limitations:

- While the examples in this document use only byte-wide data widths for clarity, the same principles apply to any data width for which a block RAM is configured.
- There cannot be any gaps or overlaps in Bit Lane numbering. All Bit Lanes in an Address Block must be the same number of bits wide.
- The Bit Lane widths are valid for the memory device specified by the device type keyword.
- The amount of byte storage occupied by the Bit Lane block RAMs in a Bus Block must equal the range of addresses inferred by the start and end addresses for a Bus Block.
    - All Bus Blocks must be the same number of bytes in size.
    - A block RAM instance name can be specified only once.
    - A Bus Block must contain one or more valid Bit Lane definitions.
    - An Address Block must contain one or more valid Bus Block definitions.

UpdateMEM checks for all these conditions and transmits an error message if it detects a violation.

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

See the Xilinx Solution Centers for support on devices, software tools, and intellectual property at all stages of the design cycle. Topics include design assistance, advisories, and troubleshooting tips.

References

These documents provide supplemental material useful with this guide:

1. *Zynq-7000 All Programmable SoC Software Developers Guide* (UG821)
2. *Vivado Design Suite User Guide: Synthesis* (UG901)
3. *Vivado Design Suite User Guide: Designing IP Subsystems Using IP Integrator* (UG994)
4. *Zynq-7000 AP SoC Technical Reference Manual* (UG585)
5. *Vivado Design Suite Tutorial: Designing IP Subsystems Using IP Integrator* (UG995)
6. *Zynq-7000 All Programmable SoC PCB Design Guide* (UG933)
7. *MicroBlaze Processor Reference Guide* (UG984)
8. *MicroBlaze Debug Module (MDM) v3.1 Product Guide* (PG115)
9. *ISE to Vivado Design Suite Migration Guide* (UG911)
10. *Vivado Design Suite User Guide: Using Constraints* (UG903)
11. *7 Series FPGAs Memory Interface Solutions User Guide* (UG586)
12. *UltraScale Architecture-Based FPGAs Memory Interface Solutions* (PG150)

Training Resources

Xilinx provides a variety of training courses and QuickTake videos to help you learn more about the concepts presented in this document. Use these links to explore related training resources:

1. Vivado Design Suite QuickTake Video: Designing with Vivado IP Integrator
2. Vivado Design Suite QuickTake Video: Targeting Zynq Devices Using Vivado IP Integrator
3. Essentials of FPGA Design Training Course
4. Vivado Design Suite Embedded Systems Design
5. Vivado Design Suite Advanced Embedded Systems Design

Please Read: Important Legal Notices