Summary

The Zynq® UltraScale+™ MPSoC provides multiple processing units, including four Cortex™-A53 cores, two Cortex-R5 cores, a platform management unit (PMU), configuration security unit (CSU), and a user-specified number of MicroBlaze™ processors in the programmable logic (PL). When multiple software teams are involved in system development, cores can potentially interfere with each other. Isolation prevents interference between the subsystems that use the cores. Isolation is a requirement in security and functional safety applications.

The Zynq UltraScale+ MPSoC provides the Xilinx® memory protection unit (XMPU) and the Xilinx peripheral protection unit (XPPU) for hardware protection of memory and peripherals. These protection units complement the isolation provided by TrustZone (TZ) and by the Zynq UltraScale+ MPSoC memory management units (MMUs). The Isolation Flow allows a system to be built using a structured isolation methodology. This application note describes how to isolate the subsystems in a Zynq UltraScale+ MPSoC system using XMPU, XPPU, and TZ.

The reference design files for this application note can be downloaded from the Xilinx website. For detailed information about the design files, see Reference Design, page 40.

Introduction

Zynq UltraScale+ MPSoC designs use multiple subsystems. The subsystems include one or more central processing units (CPUs) or other masters, memories, and peripherals. Interference occurs when a master in one subsystem accesses a memory region or peripheral that it is not intended to access. Interference can result from software bugs or from a malicious actor creating interference.

In this application note, the Isolation Flow in the Vivado® design suite is used to define a system that uses isolated subsystems. The subsystems are the application processing unit (APU), real-time processing unit (RPU), and platform management unit (PMU). The objective of the Isolation Flow is to ensure that each subsystem executes with freedom from interference (FFI) from other subsystems. The Isolation Flow uses the Vivado design suite and the Software Development Kit (SDK) to create systems with FFI. The Isolation Flow configures protection units and TZ for subsystem isolation. The system hardware generated in the Vivado design suite is exported to SDK, which is used to create system software. In addition to the basic software that runs on subsystems, SDK can be used to control and monitor the protection unit and TZ functionality. The software allows the developer to include an error reaction to interference of a subsystem.
After a system is defined and implemented, it needs to be validated for basic functionality of the subsystems, including any subsystem intercommunication. The isolation between subsystems can be verified by injecting faults that invoke protection unit and TZ isolation functionality. This includes testing of the error reaction to the interference defined by the system architect.

This application note uses the Isolation Flow on a bare-metal system. The methodology provides a framework for Isolation Flow development in systems that use operating systems. This application note includes:

- UltraScale MPSoC Architecture overview with a focus on protection units
- XMPU Overview
- XPPU Overview
- TrustZone Overview
- Isolation Flow
- Using SDK for System Development

**Hardware and Software Requirements**

The hardware and software requirements for the reference design system include:

- Xilinx ZCU102 evaluation platform or Avnet UltraZed-EG board
- Two USB type-A to USB mini-B cables (for UART, JTAG communication)
- Micro secure digital multimedia card (uSD) flash card (16 GB)
- Xilinx Software Development Kit 2017.1
- Xilinx Vivado Design Suite 2017.1
- Tera Term or equivalent communication terminal software
UltraScale MPSoC Architecture

This section discusses the hardware components in the UltraScale MPSoC architecture that are used to create subsystems and the protection units used for FFI. At a high level, Zynq UltraScale+ MPSoCs consist of a processing system (PS) and programmable logic (PL). Zynq UltraScale+ MPSoC regions are also defined by power domains, including the full power domain (FPD) and low power domain (LPD) regions.

Figure 1 shows the high-level architecture of the Zynq UltraScale+ MPSoC, including the PS and PL, and the interfaces between the regions. The memory regions are double data rate (DDR) memory, on-chip memory (OCM), tightly-coupled memory (TCM), and advanced eXtensible interface (AXI) block RAM in the PL. Access to memory is controlled by the memory controllers, direct memory access controllers (DMACs), memory management units (MMUs), and the XMPUs. The peripherals, mostly in the LPD, include devices in the input/output unit (IOU), and other devices such as the gigabit Ethernet MAC (GEM). The GEM and USB peripherals function as both master and slave. Access to the peripherals can be dedicated or shared. Isolation of the peripherals is provided using the XPPU.
Figure 1: Zynq UltraScale+ Architecture and Power Domains
Figure 2 shows the location of the XMPUs and the XPPU in the Zynq UltraScale+ MPSoC. There are eight XMPUs. Six of the XMPUs protect transactions into the DDR, one protects the OCM, and one protects transactions into the FPD. There is one XPPU, which is located at the input to the LPD.

The hardware provides other components used in isolation. The system memory management unit (SMMU) provides memory management for non-CPU masters such as direct memory access controllers (DMACs). The SMMU is used in systems that use a hypervisor. The AXI timeout blocks (ATBs) ensure that AXI transactions for which there is not a slave response do not halt. The AXI isolation blocks (AIBs) facilitate the transition to a powered down state for regions that are powered down. Powering down unused regions is important in isolation.

Several types of access control need to be used in conjunction with the protection units and TZ. AXI bus transactions can be read, write, or read/write. A section with code or data constants should not be writeable. The memory region’s read/write permissions can be defined with the Isolation Flow. TZ uses the AXI bus AxProt[] signals, while the protection units use the master identifiers (IDs) to implement isolation.
Supporting least privilege, the Zynq UltraScale+ MPSoC has four exception levels used to control privileges based on the code that is run. The ARMv8 architecture exception levels (ELs) are exclusively for the APU. The RPU and PMU do not use ELs.

Reference System Description

A first step in developing a system on the Zynq UltraScale+ MPSoC is defining the functionality in terms of the architecture. This means defining the tasks performed in the APU, RPU, and the PL. In most systems, there is a joint requirement that the subsystems are isolated to perform their tasks without interference. There should also be communication between the subsystems. Two methods of inter-subsystem communication are defined in this application note. In one method, the inter-processor interconnect (IPI) is used to communicate using message buffers. In the second, shared memory (OCM and DDR) is used for inter-subsystem communication.

In some systems, the resources provided by the Zynq UltraScale+ device allow dedicated peripherals to be used by the different subsystems. For example, because there are two UARTs in the PS, UART0 can be assigned to the APU subsystem, and UART1 can be assigned to the RPU subsystem. If a MicroBlaze CPU in the PL needs an independent UART, an AXI UART Lite or microprocessor debug module (MDM) can be used. Dedicated resources have an advantage in isolation because the data in the peripheral cannot be corrupted by another subsystem.

In some cases, there are reasons for subsystems to share resources. An example is that non-volatile memory (NVM) is used to store the boot image for all subsystems loaded at power-up. A second example is the DDR controller (DDRC). While it is possible to add an AXI DDRC in the PL, this increases the resources used, which increases the cost. The isolation is increased, but the effect of using the added DDRC resources on reliability is less clear.

When using development boards such as the ZCU102 or UltraZed-EG, there might be constraints in the resources used by each subsystem. The multiplexed I/O (MIO) and device board interfaces might present resource limitations that do not exist on a custom board.

Figure 3 shows the basic reference system consisting of three subsystems. The APU subsystem is comprised of the A53-0, UART0, SPI0, TTC0, SWDT0, and shares the GPIO. The SWDT0 and TTC0 are internal peripherals, and the UART0 and SPI0 connect to external device I/Os. The A53-0 is the APU master. The 32 MB DDR region starting at address \texttt{0x0} is included in the APU subsystem. The OCM region from 192K – 256k and the 1 MB DDR region starting at address \texttt{0x60000000} are shared with the RPU subsystem.
The PMU is included in this reference design to address a silicon issue. The XMPU/XPPU configuration can be locked. If the XMPU/XPPU configuration is locked and the XMPU/XPPU interrupts are enabled, the XMPU/XPPU interrupts cannot be cleared. As the default, the XMPU/XPPU configuration is not locked.

To lock the XMPU/XPPU configuration, psu_protection_lock can be called, the XMPU/XPPU lock bit can be set, or the FPD_XMPU protection can be used. In systems in which the XMPU/XPPU configuration is not locked, the XMPU/XPPU isolation of memory and peripherals and the use of XMPU/XPPU interrupts is unaffected.

In systems in which the XMPU/XPPU configuration is not locked, an adversary can gain access to the system software and change the XMPU/XPPU configuration to cause system problems. An adversary who accesses the software can cause a comparable set of problems modifying other software. An option is to lock the XMPU/XPPU configuration and not use interrupts. In some fail safe applications, the requirement might be to reset the system and not clear the interrupt.

The reference system uses the PMU as the XMPU/XPPU master in the pmu_subsystem. The XMPU/XPPU configuration is not locked. The reference system provides the framework code for an interrupt handler. This allows the addition of code based on the error reaction requirements.

The reference system focuses on isolation between APU and RPU subsystems. The R5-0 is the master in the RPU subsystem. The RPU subsystem peripherals are UART1, I2C1, TTC1, SWDT1, and GPIO. The R5-0 is the only master in the OCM region from 0 – 192K. The R5 shares access with the OCM region 192K – 256 K with the APU master. The R5 owns the 16 MB regions starting at 0x40000000. The R5 shares the 1 MB region starting at 0x60000000.
The XMPU provides protection of memory regions assigned to each subsystem. Figure 4 shows a functional diagram of the XMPU.

There are six XMPUs at the input to the DDRC interface and one XMPU at the input to the OCM. There is also an XMPU at the input of the FPD interconnect for protection of FPD controllers (SATA and PCIe). The XMPU configuration using the Vivado Isolation Configuration is exported to the first stage boot loader (FSBL). The XMPUs can be reconfigured during runtime by secure masters, but this is not generally recommended. If secure boot is used, the FSBL is authenticated by the device, and consequently the configuration of the XMPU/XPPU is trusted. The XMPU registers can be locked so that they cannot be written. The registers can be read to verify their state. As part of the Zynq UltraScale+ Software Test Library (STL) targeted principally at functional safety applications, Xilinx provides the capability to run self tests on the XMPU. Each XMPU protects up to 16 regions, with regions aligned on either 1 MB (DDR) or 4 KB (OCM) boundaries. For each region, the memory protection is based on two checks:

- The address of the transaction is within the region defined by START_ADDR and END_ADDR
- The master ID of the incoming transaction is allowed

The START_ADDR, END_ADDR, and master ID (MID) are defined in the system setup, and the values are readable in the XMPU register space. The APU has a single master ID. In the split mode, each R5 has a master ID. When the RPU is configured in lock step mode, a single Cortex-R5 ID is used. In lockstep mode, a single master ID is used. If an access does not match any of the regions, the XMPU action is to allow or poison the transaction. When an error occurs, the XMPU poisons the request, records the address and master ID of the first transaction that failed the check, and flags the violation. Optionally, the XMPU generates an interrupt, and an error reaction can be included in the interrupt handler.

See the “System Protection Unit” chapter in the Zynq UltraScale+ MPSoC Technical Reference Manual (UG1085) [Ref 1] for additional information on the XMPU.
The XMPU and XPPU have a silicon issue related to the locking of the registers and use of interrupts. The issue is discussed in Xilinx Answer 66183. The solution used in the reference design is to use the PMU subsystem for handling XMPU and XPPU events.

**XPPU Overview**

The XPPU protects the system-level control registers (SLCRs), peripherals, message buffers such as those used in interprocessor interrupts, and Quad SPI flash memory. A master ID list is used to define the masters that are allowed to access peripherals. Eight of the 20 master IDs are predefined. An aperture permission list (APL) specifies permissions on peripheral addresses that masters can access. Permissions are based on master ID.

In Figure 5, the AXI master's ID must be in the aperture permission list. The AXI slave's entry comes from a corresponding bit in the permission field. If the requirements are not met, the transaction is poisoned, and an interrupt is optionally generated.

In transaction poisoning, a sink module detects an illegal transaction and returns an error response. This causes either a data abort or an interrupt.

The Zynq UltraScale+ MPSoC Processing System LogiCORE IP Product Guide (PG201) [Ref 2] provides additional information on the master ID and aperture permission lists, permission checking, and error handling.
TrustZone Overview

The Isolation Flow facilitates the use of protection units and TZ. Protection units provide isolation using AXI bus transactions. Xilinx distinguishes isolation provided by the protection units and TZ using the terms transaction isolation and state isolation, respectively. TZ state isolation can be more comprehensive than protection unit transaction isolation. With TZ, the processor, IP, memory, and interrupts in subsystems are assigned secure world (SW) or non-secure world (NSW) settings. The subsystem can context switch between SW and NSW states. This improves device utilization at the expense of software complexity.

The reference design provides a critical framework to start using TZ. To realize all of the advantages of TZ state isolation generally requires support from a Xilinx ecosystem partner. Figure 6 shows a system that uses TZ. TZ defines subsystems as SW and NSW. Also, in TZ, CPUs, peripherals, and memories are world defined.

In the general ARM® use case, TZ uses hardware and software functionality to provide isolation. TZ defines SW and NSW operational states. Because functional safety (FS) applications sometimes have isolation requirements analogous to security applications, some FS applications distinguish between safety critical and non-safety critical functions.
The intent is to ensure that critical functions in the SW cannot be corrupted by functions that operate in the NSW. In most, but not all TZ systems, the same CPU time multiplexes between the SW and NSW. This is an efficient use of resources. This usually requires a relatively complex context switch. In the general case, trusted software runs in the SW using a standalone board support package (BSP) or a small operating system in the SW. NSW software runs on a rich operating system, often Linux, which generally has a wider attack surface.

As an example of SW functionality, secure boot, secure firmware update, key management, reset, power management, and the critical function of the system are performed in the SW. Non-critical applications such as status reporting, non-essential analytics, and performance monitoring are performed in the NSW as a Linux application. The isolation provided by TZ minimizes the probability that a cyber attack or software bug in the NSW affects code or data in the SW.

Because the context switch between code running in a SW and a NSW is complex, it is easier if one CPU is statically configured to operate in the SW, and a second CPU is statically configured to operate in the NSW. With the number of CPUs provided in the Zynq UltraScale+ MPSoC, this might be an option. As an example, the R5-0 can operate statically in the SW, and the A53-0 statically in the NSW.

In TZ, masters, slaves, and memory are designated to function in either the SW or the NSW. Unlike masters, the “world” designation of slaves and memory is almost always static. A master in the SW has access to slave and memory belonging to the SW or the NSW (i.e., everything). A master in the NSW has access to slave and memory belonging to the NSW only. An access attempt by a NSW master to a SW peripheral or memory is not allowed. The access is rejected by the slave, generally with a SLVERR or DECERR response.

The TZ hardware isolation on the Zynq UltraScale+ MPSoC uses the AxPROT[1] signal on the AXI bus as the filtering mechanism to determine if an access is legal. This is used on the ARM advanced high-performance bus (AHB) and the advanced peripheral bus (APB) in the PS. The AXI interconnect IP used in the PL also supports AxPROT[1], allowing relatively straightforward TZ isolation in the PL. While the MicroBlaze processor and AXI IP do not support SW/NSW operating states, the custom logic to add the AxPROT signals is relatively simple.

In the Zynq UltraScale+ MPSoC, the APU, RPU, PMU, CSU, and PL have separate interrupt controllers. While this provides isolation, interrupts can be routed between subsystems.
Isolation Flow

In the 2017.1 release, the Vivado design suite provides the Isolation Flow to automate the setup of the subsystems including the XMPU, XPPU, and TZ configuration settings. In the Vivado tools, this is accomplished by adding ZynqMP to the Diagram pane, double-clicking the device, and selecting **Switch to Advanced Mode > Isolation Configuration.** The GUI provides an interface that allows system definition of isolation to be performed at a high level.

**Table 1** lists the IP cores in the reference system using the Isolation Flow, which defines the subsystems masters, memories, and peripherals. The start address, size, TZ settings, and access settings are defined for all memories and peripherals. Many of the IP cores are configured as part of the board preset. Changing the board preset might lead to an MIO conflict. Slaves need to be enabled before they appear in the Isolation Flow.

<table>
<thead>
<tr>
<th>Subsystem Definition</th>
<th>Start Address</th>
<th>Size</th>
<th>TrustZone</th>
<th>Access</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>APU Subsystem</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>A53-0</td>
<td></td>
<td></td>
<td>NSW</td>
<td></td>
</tr>
<tr>
<td>OCM</td>
<td>0xFFFF0000</td>
<td>64K</td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>DDR_LOW</td>
<td>0x0</td>
<td>32 MB</td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>DDR_LOW</td>
<td>0x60000000</td>
<td>1 MB</td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>UART0</td>
<td></td>
<td></td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>GPIO</td>
<td></td>
<td></td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>SWDT0</td>
<td></td>
<td></td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>TTC0</td>
<td></td>
<td></td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>SPI0</td>
<td></td>
<td></td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td><strong>RPU Subsystem</strong></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>R5-0</td>
<td></td>
<td></td>
<td>SW</td>
<td></td>
</tr>
<tr>
<td>OCM</td>
<td>0xFFFFC0000</td>
<td>192K</td>
<td>SW</td>
<td>R/W</td>
</tr>
<tr>
<td>OCM</td>
<td>0xFFFF0000</td>
<td>64K</td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>DDR_LOW</td>
<td>0x40000000</td>
<td>16 MB</td>
<td>SW</td>
<td>R/W</td>
</tr>
<tr>
<td>DDR_LOW</td>
<td>0x60000000</td>
<td>1 MSB</td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>UART1</td>
<td></td>
<td></td>
<td>SW</td>
<td>R/W</td>
</tr>
<tr>
<td>GPIO</td>
<td></td>
<td></td>
<td>NSW</td>
<td>R/W</td>
</tr>
<tr>
<td>SWDT1</td>
<td></td>
<td></td>
<td>SW</td>
<td>R/W</td>
</tr>
<tr>
<td>TTC1</td>
<td></td>
<td></td>
<td>SW</td>
<td>R/W</td>
</tr>
<tr>
<td>I2C1</td>
<td></td>
<td></td>
<td>SW</td>
<td>R/W</td>
</tr>
</tbody>
</table>
When constructing the isolated system, the Vivado design suite Isolation Flow design rule checks (DRCs) provide an early indication of system-level conflicts. This facilitates developing a system architecture with isolation.

Use these steps to create the IPI design in the $ISOLATION_FLOW directory.

1. Start the Vivado tools in the $ISOLATION_FLOW directory.
2. Click Quick Start > Create Project.
3. Click Next five times until the Boards selection pane is visible. Select either the ZCU102 or the Avnet UltraZed-EG IO Carrier Card board. The Avnet board files need to be installed in \Xilinx\Vivado\2017.1\data\boards\board_files. In some shared Linux environments, it is necessary to use MYVIVADO to install the Avnet board files.
4. Click Next. Click Finish.
5. Click Flow Navigator > IPI Integrator and select Create Block Design. Accept the defaults in the dialog box and click OK.
6. Undock the Diagram pane.
7. Click +. Enter zy in the search box. Add the Zynq UltraScale+ MPSoC to the IPI design.
8. From the Block Design pane (usually to the left of the Diagram pane), select the Board tab. Add the DIP switches and LEDs on the board to the IPI design. This provides basic use of the PL using AXI GPIOs. When targeting the ZCU102 or UltraZed-EG board, the DIP switches and LEDs are assigned to MIO pins, not PL I/O pins.
9. Double-click DIP Switches. Under AXI GPIO, select GPIO. Click OK.
10. Double-click LED. Under Create New IP, select AXI GPIO > GPIO. Do not use GPIO2. Click OK.
11. Click Run Connect Automation. Click All Automation in the dialog box. Click OK.
12. Click Run Block Automation. Use the defaults provided in the dialog box. Click OK.
13. In the Diagram pane, double-click Zynq UltraScale+ MPSoC.
14. Specify system content without defining isolation requirements. Ensure that there are no MIO conflicts.
   a. Select I/O Configuration > Low Speed > Memory Interfaces. Select QSPI, SD 0, SD 1.
   b. Select I/O Configuration > Low Speed > Peripherals. Select I2C1, SPI0, UART0, UART1, GPIO0. For SPI0, change MIO to use 38..43.
   c. Click I/O Configuration > Low Speed > Processing Unit. Select SWDT0, SWDT1, TTC0, and TTC1.
   d. Deselect PCIe IP.
15. Click I/O Configuration > Peripheral > High Speed. Deselect Display Port and SATA.
16. The following steps create the isolated subsystems defined in Table 1. Check Switch to Advanced Mode.
17. As shown in Figure 7, click **Isolation Configuration**. Isolation Configuration is the last entry under the Page Navigator.

18. Check the **Enable Isolation** box.

19. Click the + icon. Select **Add New Subsystem**. Enter **RPU** in the dialog box, followed by the Enter key. Do not click OK until all of the steps in the configuration are complete.

20. Click the + icon. Select **Add New Subsystem**. Enter **APU** in the dialog box, followed by the Enter key.

21. Right-click **RPU**. Select **Add Master**. Under Processors, select **RPU0**.
22. Right-click on **APU**. Select **Add Master**. Under Processors, select **APU**.

23. Right-click on **RPU**. Select **Add Slaves** > **Memory** > **OCM**. Change the OCM size to 192 KB. Change the TZ Settings to Secure. Set the start address to **0xFFFFC0000**.

24. Right-click on **RPU**. Select **Add Slaves** > **Memory** > **OCM**. Change the OCM size to 64 KB. Change the TZ Settings to NonSecure. Set the start address to **0xFFFFF0000**. Entries are colored red until pressing Enter causes conflicts to be resolved.

25. Right-click on **RPU**. Select **Add Slaves** > **Memory** > **DDR_Low**. Change the DDR size to 16 MB. Change the TZ Settings to Secure. Set the start address to **0x40000000**.

26. Right-click on **RPU**. Select **Add Slaves** > **Memory** > **DDR_Low**. Change the DDR size to 1 MB. Change the TZ Settings to NonSecure. Set the start address to **0x60000000**.

27. Right-click on **RPU**. Select **Add Slaves** > **Peripherals** > **UART1**. Change the TZ Settings to Secure.

28. Right-click on **RPU**. Select **Add Slaves** > **Peripherals** > **GPIO**. Change the TZ Settings to NonSecure.

29. Right-click on **RPU**. Select **Add Slaves** > **Peripherals** > **I2C1**. Change the TZ Settings to Secure.

30. Right-click on **RPU**. Select **Add Slaves** > **Peripherals** > **SWDT1**. Change the TZ Settings to Secure.

31. Right-click on **RPU**. Select **Add Slaves** > **Peripherals** > **TTC1**. Change the TZ Settings to Secure.
Figure 8 shows the contents of the RPU subsystem.

32. Right-click on APU. Select Add Slaves > Memory > OCM. Change the OCM size to 64 KB. Change the TZ Settings to NonSecure. Set the start address to 0xFFFF0000.

33. Right-click on APU. Select Add Slaves > Memory > DDR_Low. Change the DDR size to 32 MB. Change the TZ Settings to Secure. Set the start address to 0x0.

34. Right-click on APU. Select Add Slaves > Memory > DDR_Low. Change the DDR size to 1 MB. Change the TZ Settings to NonSecure. Set the start address to 0x60000000.

35. Right-click on APU. Select Add Slaves > Peripherals > UART0. Change the TZ Settings to NonSecure.

36. Right-click on APU. Select Add Slaves > Peripherals > GPIO. Change the TZ Settings to NonSecure.

37. Right-click on APU. Select Add Slaves > Peripherals > SPI0. Change the TZ Settings to NonSecure.

38. Right-click on APU. Select Add Slaves > Peripherals > SWDT0. Change the TZ Settings to NonSecure.
39. Right-click on APU. Select Add Slaves > Peripherals > TTC0. Change the TZ Settings to NonSecure. 

Figure 9 shows the contents of the APU subsystem.

![Figure 9: APU Subsystem](image)

40. At the top of the pane, select Presets > Save Configuration. Enter zynqmp_isolation_flow as the Preset Name and the File Name. This saves a Tcl file that allows the configuration to be saved and optionally reloaded in subsequent projects. It also allows the configuration to be verified, as discussed in Zynqmp_isolation_flow.tcl File. Check Enable Isolation to activate Isolation Configuration functionality.

When Enable Isolation is checked, the isolation defined in the Isolation Configuration is enabled. When Enable Secure Debug is checked, the debug master “DAP” has access to all isolated memory regions. This ensures that the debug master has access to the whole PS address space. When Lock Unused Memory is unchecked, the memory regions that are not isolated have open access to all masters in the system. When this option is checked, all unprotected memory regions are blocked for all masters.

41. Click OK.
42. The PMU subsystem is required as a method to handle the silicon issues described in Xilinx Answer 66183. Expand the PMU subsystem and verify the predefined contents are as shown in Figure 10.

**Note:** Xilinx recommends editing the PMU firmware only if necessary.

![Figure 10: PMU Subsystem](image)

43. Click **Run Block Automation**. Click **OK** to use the default settings.
44. Verify that the IPI design in the Diagram pane resembles that shown in Figure 11.

![Diagram pane with IPI design](image)

**Figure 11:** IPI Design

45. Enter F6 to validate the design. If necessary, deselect two HPM FPD masters set by the 2017.1 Run Block Automation command.

46. Re-dock the Diagram pane.

47. Right-click **Sources** > **design_1 (design_1.bd)** > **Create HDL Wrapper**. Click OK to allow the Vivado tools to manage the wrapper.

48. Under **Flow Navigator** > **IPI Integrator** > **Program and Debug**, click **Generate Bitstream**. Click OK in the next two dialog boxes.

49. When bitstream generation is complete, click **File** > **Export Hardware**. Check **Include Bitstream**.

50. Click **File** > **Launch SDK**.
Zynqmp_isolation_flow..tcl File

The zynqmp_isolation_flow..tcl file and the isolation configuration design rule check are used to verify that the Isolation Flow produces the expected results. Figure 12 shows an excerpt from the zynqmp_isolation_flow..tcl file. The PSU_PROTECTION_DDR_SEGMENTS and PSU_PROTECTION_OCM_SEGMENTS define the ownership, start address, and TZ settings for the DDR.

```
CONFIG.PSU_PROTECTION__SUBSYSTEMS {rpu:RFU0|apu:APU|PMU Firmware:PMU} \
CONFIG.PSU_PROTECTION__Masters_TZ {RFU0:Secure} \nCONFIG.PSU_PROTECTION__Masters
{USB1:NonSecure;0|USB0:NonSecure;1|S_AXI_LD:NA;0|S_AXI_HPC1_FPD:NA;0|S_AXI_HPC0_FPD:NA;0|S_AXI_HPC3_FPD:NA;0|S_AXI_HPC2_FPD:NA;0|S_AXI_HPC1_FPD:NA;0|S_AXI_HP0_FPD:NA;0|S_AXI_ACP:NA;0|S_AXI_ACP:NA;0|SD1:NonSecure;1|SD0:NonSecure;1|SATA1:NonSecure;0|SATA0:NonSecure;0|RPU0:Secure;1|QSPI:NonSecure;1|PMU:NA;1|PCIe:NonSecure;0|NAND:NonSecure;0|LDM A:NonSecure;1|GPU:NonSecure;1|GEM3:NonSecure;1|GEM2:NonSecure;0|GEM1:NonSecure;0|GEM0:NonSecure;0|FDMA:NonSecure;1|DAP:NonSecure;0|Cor esight:NA;1|CSU:NA;1|APU:NA;1} \nCONFIG.PSU_PROTECTION__DDR_SEGMENTS {SA:0x40000000; SIZE:16; UNIT:MB; RegionT2:Secure; WrAllowed:Read/Write; subsystemId:rpu| SA:0x60000000; SIZE:1; UNIT:MB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu| SA:0x0; SIZE:32; UNIT:MB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu| SA:0x0; SIZE:1; UNIT:MB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu} \nCONFIG.PSU_PROTECTION__OCM_SEGMENTS {SA:0xFFFFC0000; SIZE:192; UNIT:KB; RegionT2:Secure; WrAllowed:Read/Write; subsystemId:rpu| SA:0xFFFF0000; SIZE:64; UNIT:KB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu| SA:0xFFFF0000; SIZE:64; UNIT:KB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu} \nCONFIG.PSU_PROTECTION__LPD_SEGMENTS {SA:0xFF010000; SIZE:64; UNIT:KB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu| SA:0xFF00A000; SIZE:64; UNIT:KB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu| SA:0xFF00A000; SIZE:64; UNIT:KB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu| SA:0xFF030000; SIZE:64; UNIT:KB; RegionT2:NonSecure; WrAllowed:Read/Write; subsystemId:apu} 
```

Figure 12: Zynqmp_isolation_flow..tcl
Using SDK for System Development

The Vivado design suite hardware design flow (HDF) is exported to SDK using the hardware software interface (HSI). The memory protection and TZ settings defined in the Isolation Flow are exported with Tcl protection scripts that are used by the SDK FSBL. After the FSBL is run, the setup of the registers can be verified in *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 1]. Figure 13 is a functional diagram of the Isolation flow. The subsystem definition uses the Isolation Configuration function in the Vivado System Configuration Wizard. The system is exported to SDK from the Vivado tools. In SDK, the FSBL, memory, and peripheral tests are developed.

**Figure 13:**  *Vivado Design Suite > SDK Flow*
Software Projects

This section describes how to use SDK to create software for the design created with the Vivado design suite. This flow starts from SDK generated by the Launch SDK command run in the Vivado tools. Table 2 lists the two systems, memory and peripheral, created in this section. The memory test system uses the rpu_mem_test and apu_mem_test projects. The peripheral test system uses the rpu_peripheral_test and apu_peripheral_test projects. The startup r5_fsbl and pmu_fw projects provide boot and startup operations in both systems.

Table 2: Software Projects

<table>
<thead>
<tr>
<th>Project</th>
<th>Function</th>
</tr>
</thead>
<tbody>
<tr>
<td>r5_fsbl</td>
<td>First stage boot loader running on R5-0</td>
</tr>
<tr>
<td>pmu_fw</td>
<td>Platform management unit firmware</td>
</tr>
<tr>
<td>Memory System</td>
<td></td>
</tr>
<tr>
<td>rpu_mem_test</td>
<td>Memory test run on R5-0</td>
</tr>
<tr>
<td>apu_mem_test</td>
<td>Memory test run on A53-0</td>
</tr>
<tr>
<td>Peripheral System</td>
<td></td>
</tr>
<tr>
<td>rpu_peripheral_test</td>
<td>Peripheral test run on R5-0</td>
</tr>
<tr>
<td>apu_peripheral_test</td>
<td>Peripheral test run on A53-0</td>
</tr>
</tbody>
</table>

After creation, linker scripts are generated to ensure the code is executed from regions distinct from other subsystems or software projects. Code used for a memory test that writes and reads test patterns cannot run the test in the data (code) section.

Creating the r5_fsbl Project

The FSBL loads software projects, and optionally the bitstream, at boot. This section provides the steps to create the FSBL.

1. Create the r5_fsbl project. Select File > New > Application Project.
2. Enter r5_fsbl in the Project Name dialog box. Figure 14 shows the GUI for creating the r5_fsbl software project. The software application’s GUI provides a consistent interface for creating software projects.
Figure 14: Creating FSBL Software Project
3. Change the Processor to psu_cortexr5_0.

**Note:** If the processor is left as psu_cortexa53_0, and the Zynq MP FSBL is selected in step 5, SDK displays "This application requires at least 168 KB of OCM".

The CONFIG.PSU__PROTECTION_OCM_SEGMENTS in the zynqmp_isolation_flow..tcl file confirms that 64 KB is assigned to the APU subsystem.

4. Click **Next**.
5. Select **Zynq MP FSBL**. Click **Finish**.
6. In the Project Explorer pane, right-click **r5_fsbl**, and select **C/C++ Build Settings**.
7. Click **ARM R5 gcc compiler > Symbols**. In the Defined symbols pane, click the green + icon, and add FSBL_DEBUG_INFO in the dialog box. Click **OK** twice.

Generally, the Isolation Flow produces expected results. If not, the first step in debugging is to review the zynqmp_isolation..tcl preset produced by the Vivado tools to verify that the system configuration is correct. A second step is to analyze register settings. One method to test register values is to add code to xfsbl_hooks.c as shown in Figure 15.

![Figure 15: FSBL Code Edits for Displaying Register Settings](image)

**Creating the apu_mem_test Project**

This section provides the steps to create the memory test software that runs on the A53-0.

1. Enter **File > New > Application Project**.
2. Enter **apu_mem_test** in the Project Name dialog box. Click **Next**.
3. Select **Memory Tests**. Click **Finish**.
4. In the Project Explorer pane, right-click **apu_mem_test**. Click **Generate Linker Script**.
5. Under the Basic tab, select `psu_ocm_ram_0_MEM_0` from the pull-down menu for the Place Code Sections in:, Place Data Sections in:, and Place Heap and Stack in: fields. This 64K segment starts at 0xFFFF0000 and is shared by the APU and RPU subsystems.

**Figure 16** shows the interface to generate the linker script. In some cases, the linker script can segment the code and data section such that the R/W attribute in the Isolation Flow can be set to Read Only for code sections.

![Generating the Linker Script](image)

**Figure 16:** Generating the Linker Script

6. Click **Generate**. Click **Yes** to overwrite.
The memory test is configurable by editing `n_memory_ranges` in `apu_mem_test/src/memorytest.c` or by editing the memory ranges defined in `memory_config_g.c`. Figure 17 shows the code to edit.

![Image of memory configurations](image-url)

**Figure 17:** Memory Test

**Creating the `apu_peripheral_test` Project**

This section provides the steps to create the peripheral software that runs on the A53-0. There are peripheral tests for all peripherals in the subsystem.

1. Enter **File > New > Application Project**.
2. Enter **apu_peripheral_test** in the Project Name dialog box. Click **Next**.
3. Select **Peripheral Test**. Click **Finish**.
4. In the Project Explorer pane, right-click **apu_peripheral_test**. Click **Generate Linker Script**. Under the Basic tab, select **psu_ddr_0_MEM_4** to place code and data sections, and stack and heap, at \(0x2000000\).

The peripheral test provides code for testing the peripherals defined in the subsystem. For the APU subsystem, code is auto-generated for the SCU generic interrupt controller (GIC), PS serial peripheral interface (SPI), watchdog timer, triple timer counter, and general purpose input output (GPIO) devices. It is straightforward to edit tests in **apu_peripheral_test/src**.

**Figure 18** shows the code used to test peripherals.

*Figure 18: Peripheral Test Source Code*
Creating the rpu_mem_test Project

This section provides the steps used to develop RPU memory tests.

1. Select **File > New > Application Project**.
2. Enter **rpu_mem_test** in the Project Name dialog box.
3. Change the Processor to **psu_cortexr5_0**. Click **Next**.
4. Select **Memory Test**. Click **Finish**.
5. In the Project Explorer pane, right-click **rpu_mem_test**. Click **Generate Linker Script**. Under the Basic tab, select **psu_r5_0_atcm_MEM_0** for code and data sections, and for stack and heap.
6. Click **Generate**. Click **Yes** to overwrite.

Creating the rpuPeripheral_test Project

This section defines the steps used to develop RPU peripheral tests.

1. Select **File > New > Application Project**.
2. Enter **rpuPeripheral_test** in the Project Name dialog box.
3. Change the Processor to **psu_coretexr5_0**. Click **Next**.
4. Select **Peripheral Test**. Click **Finish**.
5. In the Project Explorer pane, right-click **rpuPeripheral_test**. Click **Generate Linker Script**. Under the Basic tab, select **psu_r5_0_atcm_MEM_0** for the code and data sections, and for the stack and heap.
6. Click **Generate**. Click **Yes** to overwrite.

Creating Boot Images for APU and RPU Subsystem Functionality

In this section, two images are generated and run to show the expected functionality of the APU and RPU subsystems. In the first boot image, the APU and RPU memory test systems are combined into an image and run on the ZCU102 or UltraZed-EG board. In the second boot image, the APU and RPU peripheral tests are combined into an image and run on the board. Two communication terminals are set up to display results from the APU and RPU subsystems.

1. In SDK, click **Xilinx Tools > Create Boot Image**.
2. Under Architecture, select **Zynq MP**. Click the **Create New BIF File** radio button.
3. Browse to $ISOLATION_FLOW/project_1/project_1.sdk in the Output BIF file path dialog box.
4. Browse to $ISOLATION_FLOW/project_1/project_1.sdk/BOOT.bin in the Output path dialog box.
5. In Boot image partitions, click **Add**. As shown in **Figure 19**, browse to the `r5_fsbl.elf` file. Select bootloader as **Partition Type**. Under Destination CPU, select **R5 Single**. Click **OK**.

![Add new boot image partition](image)

**Figure 19**: Adding the `r5_fsbl.elf` Partition

6. Click **Add**. Browse to `design_1_wrapper_hw_platform_0/design_1_wrapper.bit`. Click **OK**.

7. Click **Add**. Browse to `apu_mem_test/Debug/apu_mem_test.elf`. Click **OK**.

8. Click **Add**. Browse to `rpu_mem_test/Debug/rpu_mem_test.elf`. Change the Destination CPU settings to **R5 Single**. Click **OK**.
9. Verify that the partition list is as shown in Figure 20. Click **Create Boot Image**.

![Create Boot Image](image)

*Figure 20: Creating Boot Image*

10. Insert an SD card into the PC SD card slot. The ZCU102 uses the standard SD card. The UltraZed-EG uses a micro SD card, so an adapter might be required.

11. Copy `$ISOLATION_FLOW/project_1/project_1.sdk/BOOT.bin` to the SD card.

12. Take the SD card out of the PC SD card slot and insert it into the board SD card slot.

13. Set the mode switch settings on the board to SD mode. The mode select switch is SW 2 on the UltraZed-EG board. The mode select switch is SW 6 on the ZCU102 board. Set the switch to OFF, ON, OFF, ON.

14. For APU subsystem output, invoke a communication terminal such as Tera Term. Specify 115200 baud rate, 8 bit data no parity, 1 stop bit, and no flow control.

15. For RPU subsystem output, invoke a second communication terminal such as Tera Term. Specify 115200 baud rate, 8 bit data no parity, 1 stop bit, and no flow control.

16. Use SW8 to power cycle the UltraZed-EG board. Use SW1 to power cycle the ZCU102 board.
17. Verify that the memory test outputs for the APU and RPU subsystems is as shown in Figure 21.

![Figure 21: Memory Test Results](image1.png)

18. After verifying that isolation performs as expected, it is useful to verify that regions shared by the APU and RPU subsystems function as expected. The 64K region in the OCM address range 0xFFFF0000 – 0xFFFFFFFF is shared by the two subsystems. The 1 MB region at DDR address 0x60000000 is also shared.

19. Repeat step 1 to step 12 to create a boot image that runs the APU and RPU peripheral tests. When adding the last two partitions, add `apu_peripheral_test.elf` and `rpu_peripheral_test.elf` rather than the memory tests.

20. Repeat step 13 to step 18 to copy the `BOOT.bin` image to the SD card. Insert the card into the board SD card slot.
21. Verify that the peripheral tests produce output similar to that shown in Figure 22.

Figure 22: Peripheral Test Results
Creating PMU Firmware Software Project

Create the PMU firmware software project. This project is used in Error Reaction in Interrupt Handler.

1. In SDK, select **File > New > Application Project**.
2. In the Project Name dialog box, enter **pmu_fw**.
3. Select **Processor > psu_pmu_0**. Click **Next**.
4. Select **ZynqMP PMU Firmware** under Available Template. Click **Finish**.

Subsystem Communication

Most systems use inter-subsystem communication. As an example, the APU subsystem can support an Ethernet interface that receives and transmits data from or to a server for both the APU and RPU subsystems. The RPU subsystem can generate log files and transmit them to the APU subsystem, which then transfers them to the server using Ethernet. Similarly, a server can send commands to the RPU subsystem using the APU subsystem Ethernet, which then uses the inter-subsystem communication mechanism to transfer the command to the RPU subsystem. A second inter-subsystem communication use case is a cross-channel monitor.

The Zynq UltraScale+ MPSoC provides IPI buffers to support inter-channel communication between the APU, RPU, and PMU subsystems. The exchange between the APU, RPU, and PMU subsystems uses 32-byte request and response buffers. Figure 23 shows interprocessor communication using IPI in the reference design. See the *Zynq UltraScale+ MPSoC Technical Reference Manual* (UG1085) [Ref 1] for more information on using IPI.

![Figure 23: Interprocessor Communication Using IPI](image)

The protocol for the message exchange is for the requesting subsystem APU to trigger an interrupt to the receiving subsystem (RPU). The interrupt triggering subsystem fills the data in the IPI channel's request buffer. The RPU interrupt receiving master reads the content of the request buffer. If the interrupt receiving master needs to provide response data to the interrupt triggering master, the response buffer is used. The response buffer is read by the triggering master (APU).
The Zynq UltraScale+ MPSoC provides eleven IPI channels for inter-subsystem communication. Of the eleven, channels 3 – 6 are dedicated to the PMU, and the remaining are configurable as masters. The inter-subsystem communication is supported in hardware and uses the xipipsu device driver. Each channel provides six registers, used principally to trigger the interrupt and check status. Figure 24 shows the hardware support for interprocessor interrupts in the Vivado design suite.

Follow these steps to communicate between the APU and RPU subsystems.

1. Enter `xsdk -workspace . &.`
2. Select **File > New > Application Project**.
3. Enter **apu_ipi** in the Project Name dialog box. Click **Next**.
4. Select **Empty Application**.
5. Right-click **apu_ipi > src**.
6. Click **Import > File System > Next**.
7. In the From Directory: dialog box, browse to the **apu_ipi.c** file in the reference design **src** directory. Click **Finish**.
8. Select **File > New > Application Project**.
9. Enter **rpu_ipi** in the Project Name dialog box. Click **Next**.
10. Select **Empty Application**.
11. Right-click **rpu_ipi > src**.
12. Click **Import > File System > Next**.
13. In the From Directory: dialog box, browse to the **rpu_ipi.c** file in the reference design **src** directory. Click **Finish**.
14. Select **Xilinx Tools > Create Boot Image**.
15. Click the **Create New BIF file** radio button.
16. In the Output BIF file path: dialog box, browse to the **project_1/project_1.sdk** directory.
17. In the Output path: dialog box, browse to the **project_1/project_1.sdk** directory.
18. Click **Add**. In the File Path dialog box, browse to select the **r5_fsbl/Debug/r5_fsbl.elf** file. Select the Partition Type as bootloader and run on the R5 0 processor.
19. Click **Add**. Browse to the **apu_ipi/Debug/apu_ipi.elf** file. Click **OK**.
20. Insert the SD card into the PC SD card slot.
21. Copy **BOOT.bin** to the SD card.
22. Take the SD card out of the PC SD card slot and insert it into the ZC102 or UltraZed-EG SD card slot.
23. Set up two communication terminals.
24. Power-cycle the board.
25. Verify that the expected output is as shown in Figure 25.

Figure 25: Inter-Subsystem Communication
Fault Injection

After the memory and peripheral functionality in the subsystems is verified, the isolation of the subsystems should be tested using fault injection.

The system used to verify functional operation of the APU and RPU subsystems provides the framework for fault injection. The fault types injected include XMPU, XPPU, and TZ detected faults involving illegal memory and peripheral accesses.

A TZ fault occurs when a non-secure master attempts either a memory or a peripheral access to a memory region or a peripheral defined as secure. One method for injecting a fault is to add this code to be executed by A53-0 in the non-secure APU subsystem:

```
Xil_Out32(Secure_Memory_Address, 0x12345678);
```

The code is added to the source code in `apu_mem_test` or `apu_peripheral_test`. In the basic reference design, secure memory regions in the OCM include any address in the `0xFFFFC0000 – 0xFFFFEFFFF` range. The secure memory in DDR is the 1 MB address range with the `0x40000000` start address.

The fault injection methodology is similar for peripherals. The access must be from a non-secure master (i.e., A53-0 in the APU subsystem) to a secure peripheral. The secure peripherals in the reference design belong to the RPU subsystem, and include the triple timer counter (TTC0), system watchdog timer (SWDT0), universal asynchronous receiver transmitter (UART0), and the serial peripheral interface (SPI0). The GPIO is non-secure, allowing it to be shared by the APU and RPU subsystems.

In addition to faults due to TZ settings, memory access faults are detected by the XMPU, and peripheral access faults are detected by the XPPU. The XMPU detects accesses by the A53-0 or R5-0 to a memory region that is not defined by the Isolation Flow. Adding this instruction to either the APU or the RPU subsystem code is detected by the XMPU:

```
Xil_Out32(0x80000000, 0x12346578);
```

A peripheral access fault is injected by writing to a peripheral that is not included in the system. The SPI1 is not included in either the APU or RPU subsystem. A fault is injected by writing to SPI1.

This code is inserted into either the `apu_peripheral_test` or `rpu_peripheral_test` code. Because the RPU subsystem is able to access SPI0, a straightforward modification to the existing `rpu_peripheral_test` uses `xparameters.h` to change the SPI0 test to target SPI1.
Error Reaction in Interrupt Handler

The XMPU and XPPU optionally generate an interrupt when there is a memory or peripheral access violation. The system can be set up so that the interrupt is connected to either the APU GIC, RPU GIC, or AXI INTC, allowing an interrupt handler to be implemented by the APU, RPU, or PMU.

The error reaction in the interrupt handler is defined by the system requirements. For example, in one system requirement, the reaction might be to power down the system and require intervention to restart the system. In another system, in which availability is a prevalent requirement, the system might remain functional. In this case, the error reaction might be to log the error, notify a server for possible scheduled maintenance, and continue operation.

Figure 26 shows the PMU FW code for handling an error. The file shown is xpfw_xpu.c. The message (shown later) is printed from the XPfw_Xpu_IntrHandler function. This handler can be modified to implement the required error reaction.

![Figure 26: PMU FW Code for Error Handling](image-url)
Figure 27 shows the results of the PMU handling an error in the reference design. In this example, the error is generated by the Xilinx system debugger (XSDB) memory read by the APU of the OCM owned by the RPU. While there is no error reaction in this example, the code discussed in Fault Injection provides a framework for error handling.

![Terminal Console Output](image)

Figure 27: Interrupt Handler for Invalid Transaction

Conclusion

With the large number of processors on the Zynq UltraScale+ MPSoC, designers need to ensure that code running on any processor or master is unable to interfere with or corrupt memory regions or peripherals that are not part of the master's subsystem. This application note describes how to use the hardware and software mechanisms provided by the XMPPU, XPPU, and TZ for the isolation of subsystems. This functionality complements other isolation methods, such as least privilege and hypervisors.
Reference Design

Download the reference design files for this application note from the Xilinx website.

Table 3 shows the reference design matrix.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>General</strong></td>
<td></td>
</tr>
<tr>
<td>Developer name</td>
<td>Lester Sanders</td>
</tr>
<tr>
<td>Target devices</td>
<td>Zynq UltraScale+ MPSoCs</td>
</tr>
<tr>
<td>Source code provided</td>
<td>Yes</td>
</tr>
<tr>
<td>Source code format</td>
<td>C</td>
</tr>
<tr>
<td>Design uses code and IP from existing Xilinx application note and reference designs or third party</td>
<td>No</td>
</tr>
<tr>
<td>Static code analysis/MISRA C</td>
<td>Yes</td>
</tr>
<tr>
<td><strong>Simulation</strong></td>
<td></td>
</tr>
<tr>
<td>Functional simulation performed</td>
<td>N/A</td>
</tr>
<tr>
<td>Timing simulation performed</td>
<td>N/A</td>
</tr>
<tr>
<td>Test bench used for functional and timing simulations</td>
<td>N/A</td>
</tr>
<tr>
<td>Test bench format</td>
<td>N/A</td>
</tr>
<tr>
<td>Simulator software/version used</td>
<td>N/A</td>
</tr>
<tr>
<td>SPICE/IBIS simulations</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Implementation</strong></td>
<td></td>
</tr>
<tr>
<td>Synthesis software tools/versions used</td>
<td>N/A</td>
</tr>
<tr>
<td>Implementation software tools/versions used</td>
<td>N/A</td>
</tr>
<tr>
<td>Static timing analysis performed</td>
<td>N/A</td>
</tr>
<tr>
<td><strong>Hardware Verification</strong></td>
<td></td>
</tr>
<tr>
<td>Hardware verified</td>
<td>Yes</td>
</tr>
<tr>
<td>Hardware platform used for verification</td>
<td>Avnet UltraZed-EG and ZCU102 evaluation board</td>
</tr>
</tbody>
</table>
Documentation Navigator and Design Hubs

Xilinx Documentation Navigator provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open the Xilinx Documentation Navigator (DocNav):

- From the Vivado IDE, select Help > Documentation and Tutorials.
- On Windows, select Start > All Programs > Xilinx Design Tools > DocNav.
- At the Linux command prompt, enter docnav.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In the Xilinx Documentation Navigator, click the Design Hubs View tab.
- On the Xilinx website, see the Design Hubs page.

*Note:* For more information on Documentation Navigator, see the Documentation Navigator page on the Xilinx website.

References

2. *Zynq UltraScale+ MPSoC Processing System LogiCORE IP Product Guide (PG201)*
3. *Zynq UltraScale+ MPSoC Register Reference (UG1087)*
4. *ARM TrustZone*

Revision History

The following table shows the revision history for this document.

<table>
<thead>
<tr>
<th>Date</th>
<th>Version</th>
<th>Revision</th>
</tr>
</thead>
<tbody>
<tr>
<td>07/26/2017</td>
<td>1.0</td>
<td>Initial Xilinx release.</td>
</tr>
</tbody>
</table>
The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx’s limited warranty, please refer to Xilinx’s Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx’s Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

AUTOMOTIVE APPLICATIONS DISCLAIMER
AUTOMOTIVE PRODUCTS (IDENTIFIED AS “XA” IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE (“SAFETY APPLICATION”) UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD (“SAFETY DESIGN”). CUSTOMER SHALL, PRIOR TO USING OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

© Copyright 2017 Xilinx, Inc. Xilinx, the Xilinx logo, Artix, ISE, Kintex, Spartan, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. AMBA, AMBA Designer, ARM, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, and MPCore are trademarks of ARM in the EU and other countries. All other trademarks are the property of their respective owners.