



Memory and Peripheral Protection Unit for PL Isolation in Zynq UltraScale+ Devices

XAPP1353 (v1.0) January 14, 2021

Summary

Isolation design methods help protect the system from erroneous application software and misbehaving hardware interfaces. Erroneous software may include malicious or unintentional code behavior that may corrupt system memory or cause system failures. Misbehaving hardware includes incorrect device configuration, malicious functionality, or unintentional design. The Zynq® UltraScale+™ devices includes TrustZone (TZ) technology to facilitate system design isolation.

The Zynq UltraScale+ MPSoCs and Zynq UltraScale+ RFSocS incorporate many features for design security that includes Arm® TrustZone (TZ) technology, Xilinx® peripheral protection units (XPPU), Xilinx memory protection units (XMPU), a system memory management unit (SMMU), AXI translation buffer units (TBU), and TZ control registers for protection within the PS AXI infrastructure.

For more information, on TrustZone, Security, and Anti-Tamper measures, refer to the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*. *Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)* provides a detailed example of implementing design isolation for the PS sub-systems.

This application note extends the isolation methods, described in [XAPP1320](#), into the programmable Logic (PL) sub-system of the example design, by introducing a VHDL based XMPU PL softcore, to bridge the gap between PS and PL isolation methods including PS-to-PL interfaces.

Note: It is strongly recommended that you complete the isolation design tutorial in *Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)* prior to proceeding with the tutorials in this document. While the reference design in this application note specifically targets Zynq UltraScale+ MPSoC, all isolation methods apply to the Zynq UltraScale+ devices as well.

Introduction

This application note includes all of the design concepts, functional descriptions, and specifications. If you need to fast-track the use of the XMPU in a PL design, you may skip ahead to the tutorials provided in the [Isolation Example Design](#) section, and refer to the [Overview](#), [Functional Description](#), and [XMPU_PL Usage Examples](#) sections as needed.

The reference design provided with this application note (`zupl_xmpu_v1_0`) implements an XMPU_PL for Zynq PL designs. It is a functionally tested reference IP that includes software driver support for bare-metal standalone OS applications, but it is not a part of the Xilinx LogiCORE Library. This application note provides a detailed functional description of the XMPU_PL module with implementation and usage tutorials.

Hardware and Software Requirements

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

- Xilinx ZCU102 evaluation platform
- Two USB type-A to USB mini-B cables (for UART, JTAG communication)
- Secure Digital (SD) memory card
- Xilinx Software Development Kit (SDK) 2019.1 or Vitis™ 2019.2/2020.1
- Xilinx Vivado® Design Suite
- Serial communication terminal software (such as Tera Term or PuTTY)

Overview

The Processor System (PS) of the Zynq UltraScale+ devices have eight XMPUs to protect the memory and FPD slaves (XMPU_OCM, XMPU_DDR (6) and XMPU_FPD), and one XPPU to protect the LPD peripherals. However, the PL AXI interfaces are not protected by any of these protection units.

The reference design implements the XMPU_PL function for Zynq UltraScale+ devices. It serves as both a memory and peripheral protection unit for the PL and utilizes a functional interface, similar to the XMPUs in the PS. Multiple XMPU_PL(s) may be used within the PL design to selectively monitor AXI transactions. XMPU_PL(s) may be used to provide protection to PL slaves from the PS masters, PS slaves from PL masters (such as MicroBlaze, PicoBlaze processors, DMAs, or custom PL masters), or anywhere within the user's PL AXI network design.

The Zynq UltraScale+ MPSoCs and Zynq UltraScale+ RFSocS have three (PS->PL) AXI4-master I/Fs that may transmit AXI transactions originating from any one of fifty (50) PS masters. See [Appendix A: Master ID List](#) for a list of PS masters. The PS->PL master I/Fs are:

- M_AXI_HPM0_LPD
- M_AXI_HPM0_FPD
- M_AXI_HPM1_FPD

The XMPU_PL verifies that a system master has access to an address and poisons unauthorized transactions. The XMPU_PL IP Integrator (IPI) symbol is shown in the following figure and provides the following features:

- Slave AXI4-Full (32 bit) port for XMPU run-time configuration
- Slave AXI4-Full (32, 64, 128 bit) port for incoming AXI transactions to be monitored
- Master AXI4-Full port for transferred incoming transactions
- A single IRQ interrupt output for access attempt violations
- Up to 16 (sixteen) individually configurable address regions
- Supports Secure, Non-Secure, and Strictly Non-Secure regions

- Supports IP integrator with `busif` ports
- Supports static configuration through Customization GUI
- Detects the originating AXI master ID of incoming PS AXI transactions
- Detects the security level of transactions
- Supports *poison-by-address* and *poison-by-attribute*
- Supports both internal and external AXI Sink

Figure 1: `zupl_xmpu_v1_0` (XMPU_PL) IPI Symbol

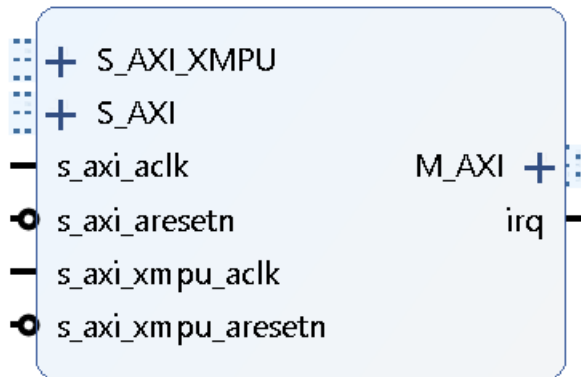
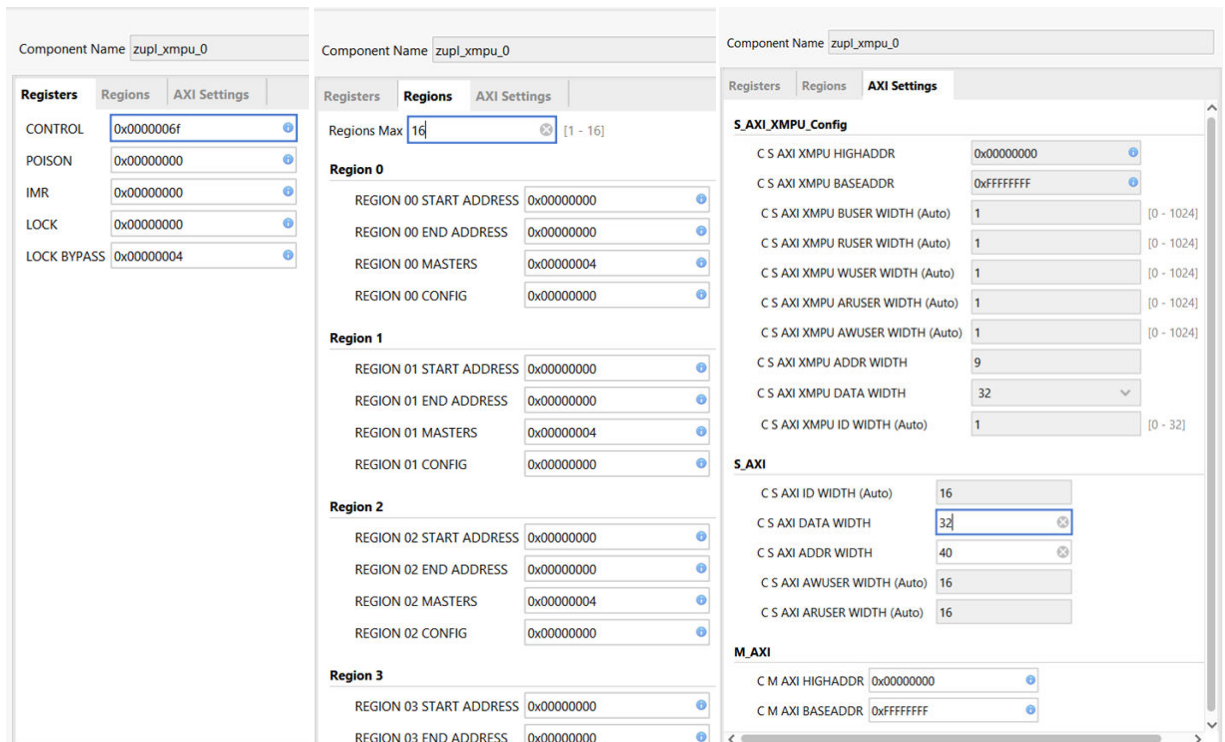


Figure 2: `zupl_xmpu_v1_0` (XMPU_PL) IPI Customization Window



XMPU_PL Configuration

The XMPU_PL may be statically configured from the customization window in the IP Integrator. Refer to the [Functional Description](#) section for a detailed description of all the configuration registers. Alternatively, the XMPU_PL may be dynamically configured at run-time through the S_AXI_XMPU AXI4 slave port. While some of the run-time interface registers are read-only, their initialization values may be controlled through the static interface of the customization GUI.

S_AXI_XMPU has been implemented as an AXI4-Full I/F to ensure the Master ID of the originating AXI master is available within the transaction, via the AxUser bus. AxUser is collectively AWUSER and ARUSER for write and read transactions, respectively.

The Regions Max, S_AXI_DATA_WIDTH, M_AXI_BASEADDR, and M_AXI_HIGHADDR values are VHDL parameters only and not available through the run-time interface.

Regions Max sets the number of AXI Monitors to be synthesized in the core. The SW cannot define more regions than this setting. The absolute maximum value is sixteen (16). Reducing this number decreases the utilized PL resources by ~130 LUTs per region. This parameter is exported to `xparameters.h`. *Region* configuration and Master IDs are explained in the following section.

S AXI DATA WIDTH sets the width of the AXI data bus to be protected. This must be selected by the user to match the upstream master. Available options are: 32, 64, 128-bit.

M AXI BASEADDR and *M AXI HIGHADDR* are not required to be set, and have no impact on the core's functionality. Their presence is for the user's convenience and they provide the address range mapped to M_AXI. These values are exported to the `xparameters.h`

Configuration Lock

The LOCK register, when set, locks out changes to all configuration registers (except interrupt status and control) by making the configuration registers read only. The lock can only be bypassed by those Master IDs enabled in the LOCK_BYPASS register. However, any master with a mapped address to the S_AXI_XMPU port can enable, disable, or respond to XMPU_PL interrupts.

Note: If LOCK is statically set and no Master IDs are enabled in the LOCK_BYPASS, then run-time configuration changes will not be possible. Refer to [Isolating the XMPU_PL Configuration](#) on how to restrict read access to the configuration registers.

Regions

Each XMPU_PL provides up to sixteen (16) regions, numbered from zero (0) to fifteen (15). Each region is defined by a start address and an end address. Regions are 256B address aligned. The start and ending address registers hold the upper 32 bits of a 40 bit address[39:8].

When a memory space is included in more than one XMPU_PL region configuration, if any of the corresponding regions trigger a violation, then the transaction is poisoned in accordance with the REGION CONFIG register option settings. Refer to [Functional Description](#) for a detailed description.

Each region can be independently enabled or disabled. If a region is disabled, it is not used for protection checking. Each region is assigned a list of masters that are authorized to access the region and has an independent security and check type selection.

- *Secure*: Secure transactions from authorized masters.
- *Non-Secure*: Secure and non-secure transactions from authorized masters.
- *Non-Secure Strict Check Type*: Non-secure transactions from authorized masters.

Note: Non-secure transactions from unauthorized masters will be poisoned.

If the address requested does not match any of the regions, then the XMPU_PL takes the default action (allow or poison) as specified in the control register options. There are three ways to poison a request:

- **Poison by address - internally**
Divert the transaction to a sink that resides inside the core.
- **Poison by address - externally**
Forward the transaction replacing the address with the value in the poison register.
- **Poison by attribute**
Forward the transaction with a poison attribute (`AxProt[1]=1`)

Master IDs

Each XMPU_PL Region and Lock_Bypass monitors use the Master ID in each AXI transaction to validate the transaction. The REGION MASTERS register selects specific Masters. Refer to the [Functional Description](#) section for a detailed register description. All the Master IDs and associated Masks are stored in the `zupl_xmpu` reference design vhdl package. The Master ID is masked by a [MIDM] bit field and then compared against a [MID] bit field.

Depending on AXI Security Permission checks, the transaction is allowed when the following equation is satisfied:

$$[MID] \text{ and } [MIDM] == AXI_MasterID \text{ and } [MIDM]$$

For more information on Master ID, refer to the *Zynq UltraScale+ Device Technical Reference Manual (UG1085)*. There are fifty masters with unique IDs in the Zynq UltraScale+ MPSoCs. These are summarized in the [Appendix A: Master ID List](#).

Note: The user need not know the specific MasterID values to configure the XMPU_PL Region and Lock_Bypass. As described in the [Functional Description](#) section, each bit position within those registers corresponds to a particular master (master-pairs for DMA channels) that are enabled or disabled.

AXI Permissions

The `AxProt[2:0]` (ARPROT and AWPROT, collectively) holds the permission levels for the AXI transaction. `AxProt[0]` indicates the Privilege level, `AxProt[1]` indicates Security level, and `AxProt[2]` indicates whether it's an instruction or data type transaction. The definitions and values are shown in Table 1.

ARPROT: Read Transaction Permissions

AWPROT: Write Transaction Permissions

Table 1: AXI Protection Permissions

AxPROT[2:0]	AXI Protection Level
000	Data Access, Secure , Unprivileged
001	Data Access, Secure , Privileged
010	Data Access, Non-secure , Unprivileged
011	Data Access, Non-secure , Privileged
100	Instruction Access, Secure , Unprivileged
101	Instruction Access, Secure , Privileged
110	Instruction Access, Non-secure , Unprivileged
111	Instruction Access, Non-secure , Privileged

`AxProt[1]` holds the security level for the AXI transaction. In the Processing System (PS), the TrustZone (TZ) setting for an AXI master is transferred over the AXI3 infrastructure using `AxUser[10]`, but this information is not transferred to the AXI4 PL interfaces. Unfortunately, `AxProt[1]` does not directly reflect the TZ setting for all masters.

PS masters having a TZ NONSECURE register setting, such as DMAs, use `AxProt[1]` to communicate the AXI Permission security level in accordance with its TZ setting. Therefore, regardless of whether isolation is enabled in the design, the DMA may be dynamically configured to make AXI transfers with either secure or non-secure AXI Permissions.

The APU sets `AxProt` bits in accordance with the exception level of the thread requesting the AXI transfer. Bare-metal standalone OS applications always execute at EL3 (`AxProt[1]=0`) which is AXI secure. Therefore, even if an APU application may be considered non-secure in the Isolated System, its AXI Permissions indicate it as being secure. This is why you must use Master IDs to control region access authorization. However, APU applications running from a Linux kernel execute at EL0 (`AxProt[1]=1`) which is non-secure and may be elevated by the OS or hypervisor.

The RPU and PMU do not support multiple exception levels and always operate at EL3. Therefore, you must use the MasterIDs to block their access to a region.



TIP: Non-Secure Strict Check Type Regions will only allow transactions from authorized masters with a Non-Secure TZ setting, like DMAs, or with multiple exception level settings, such as a Linux app in the APU. Otherwise, simply define the region as secure and specify which masters should have access in the region configuration.

Poison By Address

Poison-by-Address is enabled by default in the XMPU_PL CTRL control register. This causes a poisoned transaction to be redirected to either an internal or external sink. If external sink is selected, then the poisoned transaction is redirected to the address specified in the *POISON* register. As with the region start and end registers, the poison register is 256B aligned and specifies the upper 32 bits of the 40-bit address[39:8].

Internal Sink is enabled by default and causes the poisoned transaction to be redirected to a hidden peripheral inside the core.

Note: The internal sink is not visible to, or address mapped, in the system.

DECERR (decode error) is the default setting in the CTRL register. The DECERR will likely result in an EXCEPTION in the processor that receives the response. Exception Handling should be installed in the application to avoid hanging the processor.

The data that is written to the internal sink is not stored and gets lost. The external sink option exists in the event that the designer wishes to construct their own SINK peripheral in order to capture additional information from the transaction.

Table 2: SINK AXI Response

AXI Response Encoding		
RRESP[1:0] BRESP[1:0]	Response	Description
0	OKAY	OK
1	EXOKAY	Exclusive OK
2	SLVERR	Slave Error
3	DECERR ¹	Decode Error

Notes:

1. Default

Poison by Attribute

The Poison-by-Attribute is enabled by default in the CTRL register. This results in any poisoned transaction that is transferred to the M_AXI port to have non-secure privilege set ($AxProt[1]=1$).

There are only two conditions when this occurs:

- Poison by address is not enabled
- Poison by address is enabled with external sink



TIP: Using the Poison-by-Attribute while disabling Poison-by-Address can also be used with the secure option in the AXI interconnect advanced settings. The method is demonstrated in the [XMPU_PL Usage Examples](#) section, [Isolating Secure Slaves](#).

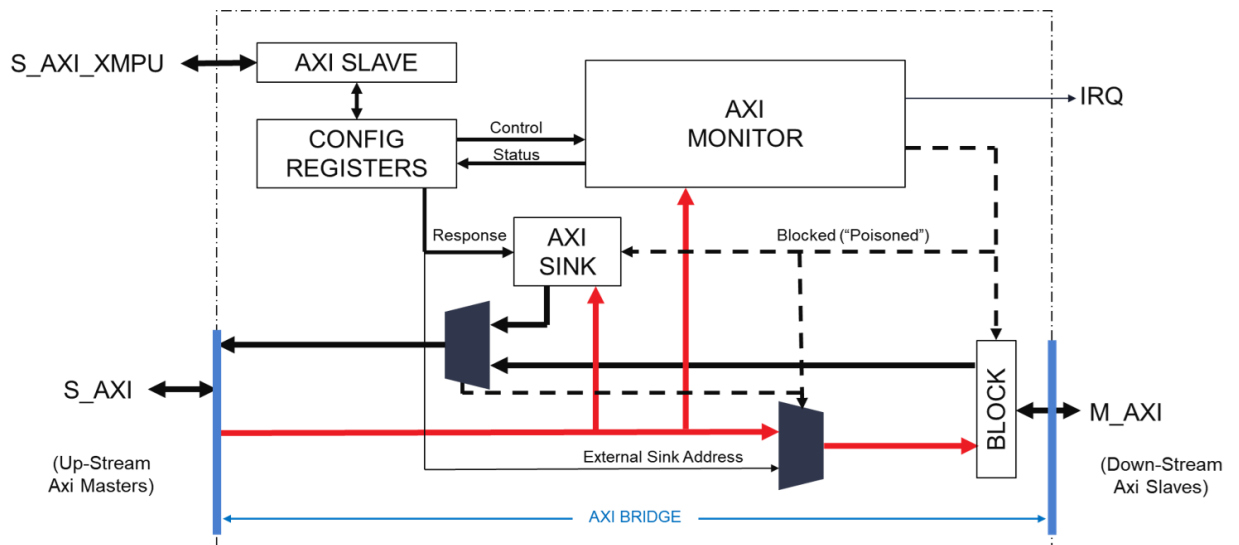
Functional Description

This section provides further details on the core's architecture, functionality, and the configuration register module.

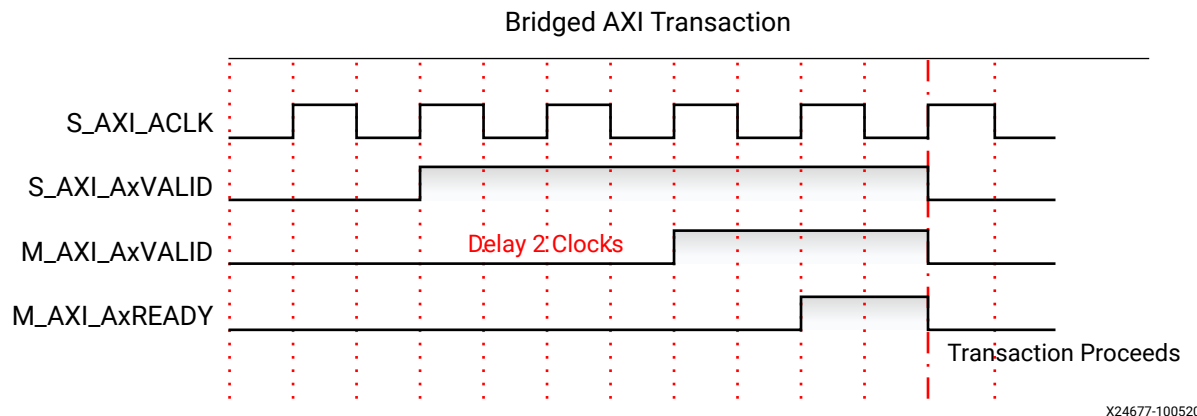
XMPU_PL Architecture

The XMPU_PL block diagram is shown in the following figure. S_AXI (slave) and M_AXI (master) AXI4 ports form an AXI Bridge that passes through authorized transactions and blocks unauthorized transactions. AXI Read and Write channels are completely independent of each other. If one channel is blocked for a violation, the other proceeds; if it does not trigger a violation.

Figure 3: XMPU_PL Block Diagram



The bridge relationship makes the XMPU_PL transparent to the system address mapping. Up-stream masters still map directly to down-stream slaves. Incoming transactions are subject to a two clock-cycle delay while the AXI-Monitor determines whether to allow or block. An example timing diagram is shown in the following figure.

Figure 4: AXI Bridge Monitoring Delay Timing Diagram


Transactions between the upstream master and downstream slave are initiated by the master with the VALID signal. The XMPU_PL initially delays the transmission of the VALID signal to evaluate the transaction. If a transaction is not to be blocked (not poisoned) it proceeds without any additional or accumulative clock cycle latency. This results in all following transitions of signals are not delayed.

Each region in the XMPU_PL is independently activated and monitored. If a region is enabled and the requested transaction address is within its range, then the MasterID is compared to the enabled masters, and the AXI permissions are compared against the region's configuration settings to determine if a violation has been triggered. If any region triggers a violation, then the transaction is blocked in accordance with the poisoning type configuration settings.

When a violation occurs, the status is communicated back to the Configuration Registers Module to capture the transaction's target address and originating MasterID into the error status registers. If the violation corresponds to an enabled interrupt flag, then the ISR register is updated and the IRQ output is asserted.

Module Registers Summary

The XMPU_PL module registers and address offsets are shown in the following table. The following sections provide the bit field definitions for each module register.

Table 3: XMPU_PL Module Registers

Register Name	Address Offset	Type	Description
Control and Status			
CTRL	0x000	mixed	Control and Implementation
ERR_STATUS1	0x004	ro	Error Status, Violation Address
ERR_STATUS2	0x008	ro	Error Status, Violation Master ID
POISON	0x00C	rw	External Sink Address
ISR	0x010	mixed	Interrupt Status and Clear
IMR	0x014	ro	Interrupt Mask

Table 3: XMPU_PL Module Registers (cont'd)

Register Name	Address Offset	Type	Description
IEN	0x018	wo	Interrupt Enable
IDS	0x01C	wo	Interrupt Disable
LOCK	0x020	rw	Register Write Lock
LOCK_BYPASS	0x024	mixed	Enable Master Access
REGIONS	0x028	ro	Number of Active Regions
Region Control			
R{00:15}_START	0x100+	mixed	Region starting base address
R{00:15}_END	0x104+	ro	Region ending address
R{00:15}_MASTERS	0x108+	ro	Select authorized PS Masters
R{00:15}_CONFIG	0x10C+	rw	Enable and Configure

CTRL Control Register

The CTRL register is shown in the following table.

Table 4: XMPU_PL CTRL Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:7	ro	0x0	Reserved
PoisonAxiResp	6:5	rw	0x3	Select AXI response to poisoned transactions. <ul style="list-style-type: none"> 0x0: OKAY 0x1: EXOKAY 0x2: SLVERR 0x3: DECERR <p>Note: If ExternalSinkEn is enabled, then the peripheral at the address specified in the POISON register transmits the response.</p>
ExternalSinkEn	4	rw	0x0	0: Transactions poisoned by address terminate in the XMPU_PL 1: Transactions poisoned by address are routed to a sink specified by POISON[PL_SINK_ADDR]
PoisonAttributeEn	3	rw	0x1	0: Transaction is not poisoned. AxProt[1] remains at original value. 1: Enables Poison by Address. Transaction routed to internal or external sink address. See CTRL[ExternalSinkEn]
PoisonAddressEn	2	rw	0x1	0: Transaction is not poisoned. Transaction proceeds to original address. 1: Enables Poison by Address. Transaction routed to internal or external sink address. See CTRL[ExternalSinkEn]
DefWrAllowed	0	rw	0x1	Default Write Allowed. Ensure the following steps are implemented if a write transaction address and master ID miss in the Region List: 0: poison the transaction with a Write Permission Violation 1: transaction allowed, regardless of security level

Table 4: XMPU_PL CTRL Register Bit Field Summary (cont'd)

Field Name	Bits	Type	Reset Value	Description
DefRdAllowed	0	rw	0x1	Default Read Allowed. If a read transaction address and master ID miss in the Region List, then: 0: poison the transaction with a Read Permission Violation 1: transaction allowed, regardless of security level

Error Status 1 Register

The ERR_STATUS1 register is shown in the following table. The first AXI violation is recorded. Once an ISR[3:1] status bit is set, subsequent violations are not recorded, but their transactions are poisoned. The status bits are cleared by a system reset and can be cleared by software.

Table 5: ERR_STATUS1 (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
AXI_ADDR	31:0	ro	0x0	Address bits of a poisoned read or write transaction. Read-only.

Error Status 2 Register

The ERR_STATUS2 register is shown in the following table. The first AXI violation is recorded. Once an ISR[3:1] status bit is set, subsequent violations are not recorded, but their transactions are poisoned. The status bits are cleared by a system reset and can be cleared by a software.

Table 6: ERR_STATUS2 (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:10	ro	0x0	Reserved
AXI_ID	9:0	ro	0x0	Master ID from a poisoned read or write transaction. Read-only.

Poison Address Register

The POISON register is shown in the following table.

Table 7: POISON (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
PL_SINK_ADDR	31:0	rw	0x00800000	The value is set by user for poison base address, determined by PL Address Mapping. The XMPU replaces the incoming AXI address (39 down to 8) with the PL_SINK_ADDR. Address (7 down to 0) is retained from the originating address for alignment. Downstream, the XMPU_PL_Sink unit responds to the transaction.

ISR Interrupt Status Register

The ISR register interrupts are shown in the following table. The bits in the status register are sticky and remain asserted until cleared by writing a 1 to the asserted bit.

Reading AXI Access Violations:

- 0: no interrupt request
- 1: interrupt requested

Writing AXI Access Violations:

- 0: no effect
- 1: clear bit to 0

If a Status bit is 1 and its Mask is 0, then the IRQ interrupt signal is activated to the interrupt controller. The first AXI violation is recorded. Once an ISR[3:1] status bit is set, subsequent AXI violations are not recorded, but their transactions are poisoned. The status bits are cleared by a system reset and can be cleared by a software

Table 8: ISR (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:4	ro	0x0	Reserved
SecurityVIO	3	wtc	0x0	Security violation by AXI Master: A non-secure master tries to access a secure memory space.
WrPermVIO	2	wtc	0x0	Write Permission violation by AXI Master. Write access attempted to enabled region with WrAllowed = 0. Or the transaction missed in the region list and CNTRL [DefWrAllowed] = 0.
RdPermVIO	1	wtc	0x0	Read Permission violation by AXI Master. Read access attempted to enabled region with RdAllowed = 0. The transaction missed in the region list and CNTRL [DefRdAllowed] = 0.
Reserved	0	ro	0x0	Reserved

IMR Interrupt Mask Register

The IMR register is shown in the following table. For each violation interrupt mask bit:

- 0: enabled.
- 1: masked (disabled). If the ISR bit = 1 (asserted interrupt) and the IMR bit = 0 (not masked), then the IRQ to the interrupt controller is asserted.

Software checks the ISR to determine the cause of the interrupt. Read only.

Table 9: IMR (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:4	ro	0x0	Reserved
SecurityVIO	3	ro	0x1	Security violation by AXI master
WrPermVIO	2	ro	0x1	Write Permission violation by AXI Master
RdPermVIO	1	ro	0x1	Read Permission violation by AXI Master
Reserved	0	ro	0x0	Reserved

IEN Interrupt Enable Register

The IEN register is shown in the following table.

- 0: no effect.
- 1: enable interrupt (sets mask = 0). Write-only.

Table 10: IEN (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:4	ro	0x0	Reserved
SecurityVIO	3	wo	0x0	Security violation by AXI Master
WrPermVIO3	2	wo	0x0	Write Permission violation
RdPermVIO1	1	wo	0x0	Read Permission violation
Reserved	0	wo	0x0	Reserved

IDS Interrupt Disable Register

The IDS register is shown in the following table.

- 0: no effect.
- 1: disable interrupt (sets mask = 1). Write-only.

Table 11: IDS (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:4	ro	0x0	Reserved
SecurityVIO	3	wo	0x0	Security violation by AXI Master

Table 11: IDS (XMPU_PL) Register Bit Field Summary (cont'd)

Field Name	Bits	Type	Reset Value	Description
WrPermVIO	2	wo	0x0	Write Permission violation
RdPermVIO	1	wo	0x0	Read Permission violation
Reserved	0	wo	0x0	Reserved

LOCK Register

The LOCK register is shown in the following table.

Register writes to ZUP_XMPU_PL may be done by any bus masters when LOCK [RegWrDis] = 0. When LOCK [RegWrDis] = 1, all register writes may only be done by secure bus masters enabled in LOCK_BYPASS register. The write lock prevents all other masters from writing to all registers except the interrupt status registers: ISR, IMR, IEN and IDS.

Note: All ZUP_XMPU_PL registers are readable by secure or non-secure bus masters.

Note: Regardless of the LOCK [RegWrDis] setting, the status registers are always writable by secure and non-secure bus masters.

Table 12: LOCK (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
RegWrDis	0	rw	0x0	Register Write Disable. Applies to all registers except ISR, IMR, IEN and IDS. 0: read/write allowed 1: read-only Once this bit is set, it can only be cleared by a master enabled in the LOCK_BYPASS register.

BYPASS Register

The BYPASS register is shown in the following table.

Register writes to ZUP_XMPU_PL may be done by any bus masters when LOCK [RegWrDis] = 0. When LOCK [RegWrDis] = 1, all register writes may only be done by secure bus masters enabled in LOCK_BYPASS register. The write lock prevents all other masters from writing to all registers except the status registers: ISR, IMR, IEN, and IDS.

Note: All ZUP_XMPU_PL registers are readable by secure or non-secure bus masters.

Note: Regardless of the LOCK [RegWrDis] setting, the status registers are always writable by secure and non-secure bus masters.

Table 13: LOCK_BYPASS (XMPU_PL) Register Bit-Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31	ro	0x0	Reserved
MID_FPD_DMA[6:7]	30	rw	0x0	Enable FPD DMA [ch 6:7]
MID_FPD_DMA[4:5]	29	rw	0x0	Enable FPD DMA [ch 4:5]
MID_FPD_DMA[2:3]	28	rw	0x0	Enable FPD DMA [ch 2:3]
MID_FPD_DMA[0:1]	27	rw	0x0	Enable FPD DMA [ch 0:1]
MID_DP_DMA[4:5]	26	rw	0x0	Enable DisplayPort DMA [ch 4:5]
MID_DP_DMA[2:3]	25	rw	0x0	Enable DisplayPort DMA [ch 2:3]
MID_DP_DMA[0:1]	24	rw	0x0	Enable DisplayPort DMA [ch 0:1]
MID_PCIE	23	rw	0x0	Enable PCIe
MID_DAP_AXI	22	rw	0x0	Enable Debug Access Port AXI
MID_GPU	21	rw	0x0	Enable GPU
MID_SATA1	20	rw	0x0	Enable SATA1
MID_SATA0	19	rw	0x0	Enable SATA0
MID_APU	18	rw	0x0	Enable APU. Note: Requires that <code>AxProt[1]=0</code>
MID_GEM3	17	rw	0x0	Enable GEM3
MID_GEM2	16	rw	0x0	Enable GEM2
MID_GEM1	15	rw	0x0	Enable GE1
MID_GEM0	14	rw	0x0	Enable GEM0
MID_QSPI	13	rw	0x0	Enable QSPI
MID_NAND	12	rw	0x0	Enable NAND
MID_SD1	11	rw	0x0	Enable SD1
MID_SD0	10	rw	0x0	Enable SD0
MID_LPD_DMA[6:7]	9	rw	0x0	Enable LPD DMA [ch 6:7]
MID_LPD_DMA[4:5]	8	rw	0x0	Enable LPD DMA [ch 4:5]
MID_LPD_DMA[2:3]	7	rw	0x0	Enable LPD DMA [ch 2:3]
MID_LPD_DMA[0:1]	6	rw	0x0	Enable LPD DMA [ch 0:1]
MID_DAP_APB	5	rw	0x0	Enable Debug Access Port APB
MID_USB1	4	rw	0x0	Enable USB1
MID_USB0	3	rw	0x0	Enable USB0
MID_PMU	2	rw	0x1	Enable PMU
MID_RPU1	1	rw	0x0	Enable RPU1
MID_RPU0	0	rw	0x0	Enable RPU0

Regions Register

The regions register is shown in the following table. The table displays the number of secure regions enabled. It is a *read only* register.

Table 14: Regions (XMPU_PL) Register Bit-Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:5	ro	0x0	Reserved
ENABLED	4:0	ro	0x0	Number of active regions Note: There are 16 available regions that are independently enabled in the R[region]_CONFIG registers.

Rxx_START Region Starting Address Register

The R[n]_START register is shown in the following table. Each region is defined by a start and end address base addresses mapped to the PL.

Note: Address Offset: 0x00000[n]00

Table 15: R[n]_START (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
ADDR	31:0	rw	0x0	AXI address within the PL. Note: Bits [31:0] correspond to address bits [39:8].

Rxx_END Region Ending Address Register

The R[n]_END register is shown in the following table. Each region is defined by a start and end address base addresses mapped to the PL.

Note: Address Offset: 0x00000[n]04

Table 16: R[n]_END (XMPU_PL) Register Bit Field Summary

Field name	Bits	Type	Reset Value	Description
ADDR	31:0	rw	0x0	AXI address within the PL. Note: Bits [31:0] correspond to address bits [39:8].

Rxx_MASTERS Region Masters Register

The AXI_MasterID from the requester is compared with all authorized secure MasterIDs for the region addressed. If the originating master is authorized: *False*, transaction is poisoned; if it is *True*, transaction is forwarded downstream.

Note: Address Offset: 0x00000[n]08

Note: PMU is always authorized by default.

The R[n]_MASTERS register is shown in the following table.

Table 17: R[n]_MASTERS (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31	ro	0x0	Reserved
MID_FPD_DMA[6:7]	30	rw	0x0	Enable FPD DMA [ch 6:7]
MID_FPD_DMA[4:5]	29	rw	0x0	Enable FPD DMA [ch 4:5]
MID_FPD_DMA[2:3]	28	rw	0x0	Enable FPD DMA [ch 2:3]
MID_FPD_DMA[0:1]	27	rw	0x0	Enable FPD DMA [ch 0:1]
MID_DP_DMA[4:5]	26	rw	0x0	Enable DisplayPort DMA [ch 4:5]
MID_DP_DMA[2:3]	25	rw	0x0	Enable DisplayPort DMA [ch 2:3]
MID_DP_DMA[0:1]	24	rw	0x0	Enable DisplayPort DMA [ch 0:1]
MID_PCIE	23	rw	0x0	Enable PCIe
MID_DAP_AXI	22	rw	0x0	Enable Debug Access Port AXI
MID_GPU	21	rw	0x0	Enable GPU
MID_SATA1	20	rw	0x0	Enable SATA1
MID_SATA0	19	rw	0x0	Enable SATA0
MID_APU	18	rw	0x0	Enable APU. Note: Requires that AxProt[1]=0.
MID_GEM3	17	rw	0x0	Enable GEM3
MID_GEM2	16	rw	0x0	Enable GEM2
MID_GEM1	15	rw	0x0	Enable GEM1
MID_GEM0	14	rw	0x0	Enable GEM0
MID_QSPI	13	rw	0x0	Enable QSPI
MID_NAND	12	rw	0x0	Enable NAND
MID_SD1	11	rw	0x0	Enable SD1
MID_SD0	10	rw	0x0	Enable SD0
MID_LPD_DMA[6:7]	9	rw	0x0	Enable LPD DMA [ch 6:7]
MID_LPD_DMA[4:5]	8	rw	0x0	Enable LPD DMA [ch 4:5]
MID_LPD_DMA[2:3]	7	rw	0x0	Enable LPD DMA [ch 2:3]
MID_LPD_DMA[0:1]	6	rw	0x0	Enable LPD DMA [ch 0:1]
MID_DAP_APB	5	rw	0x0	Enable Debug Access Port APB
MID_USB1	4	rw	0x0	Enable USB1
MID_USB0	3	rw	0x0	Enable USB0
MID_PMU	2	rw	0x1	Enable PMU
MID_RPU1	1	rw	0x0	Enable RPU1
MID_RPU0	0	rw	0x0	Enable RPU0

Rxx_CONFIG Region Configuration Register

The R[n]_CONFIG register is shown in the following table. If a transaction address is within an enabled region's start and end addresses, then the [WrAllowed]/[RdAllowed] condition is checked. If the transaction R/W type is allowed, then the security Master ID check is performed. When more than one address region includes the transaction address (regions overlap) or if any region poisons the transaction, then it takes precedence.

Note: Address Offset: 0x00000[n]0C

Table 18: R[n]_CONFIG (XMPU_PL) Register Bit Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:6	ro	0x0	Reserved
MidCheckDisable	5	rw	0x0	0: [default] Master ID is checked. Transactions are only considered secure when MasterID aligns with R00_MASTERS[] Register. 1: Disables Master ID check during security check. Any transaction with AxProt[1] = 0 will be considered Secure. Note: PL_Masters such as MicroBlaze™ do not propagate a MasterID. Setting MidCheckDisable = 1 allows WrAllow and RdAllow to define the permissions for the region.
NSCheckType	4	rw	0x0	Non-secure Region Check Type. Secure masters may or may not be allowed to access Non-Secure (NS) memory regions. 0: relaxed checking; secure requests may access a non-secure (NS) region. 1: strict checking; secure requests may only access a secure region. A non-secure access request can only access non-secure regions regardless of bit setting.
RegionNS	3	rw	0x0	Select security level of region: 0: secure. 1: non-secure (NS).
WrAllowed	2	rw	0x1	Allow writers to region: 0: not allowed; write transaction poisoned. 1: allowed.
RdAllowed	1	rw	0x1	Allow readers to region: 0: not allowed; read transaction poisoned. 1: allowed.
Enable	0	rw	0x0	Enable region: 0: disabled. 1: enabled.

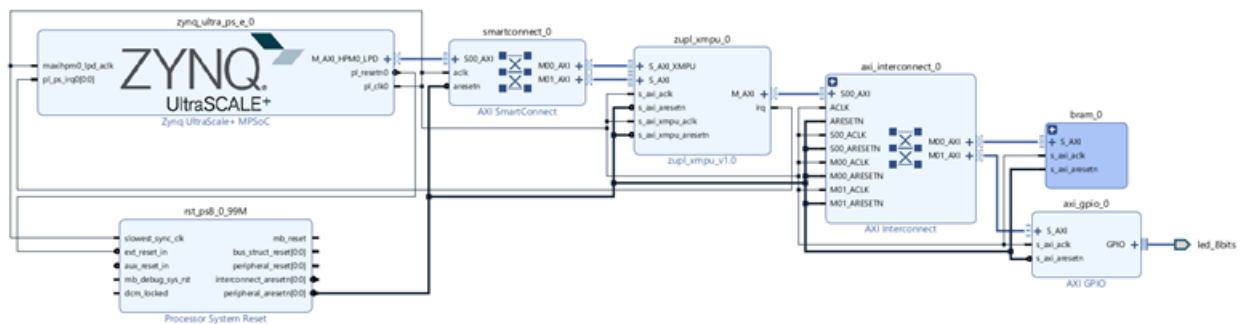
XMPU_PL Usage Examples

The Programmable Logic (PL) of the Zynq UltraScale+ devices allows the designer to create a fully custom system. This section provides some guidance on various design scenarios.

AXI SmartConnect

The XMPU_PL functionality relies on access to the AXI MasterID contained in transactions from PS masters. The S_AXI and S_AXI_XMPU have been implemented as AXI4 full interfaces to maintain the AxUser port connections which carries the MasterID values. If an inter-connect block is needed between the PS and the XMPU_PL, use the AXI SmartConnect, as shown in the following figure, instead of AXI Interconnect. AXI Interconnect blocks do not pass the AxUser bus and block the transmission of the MasterIDs. However, AXI Interconnect blocks may be used to connect to PL Masters or multiple end-point slaves, as MasterIDs are not utilized in those connections.

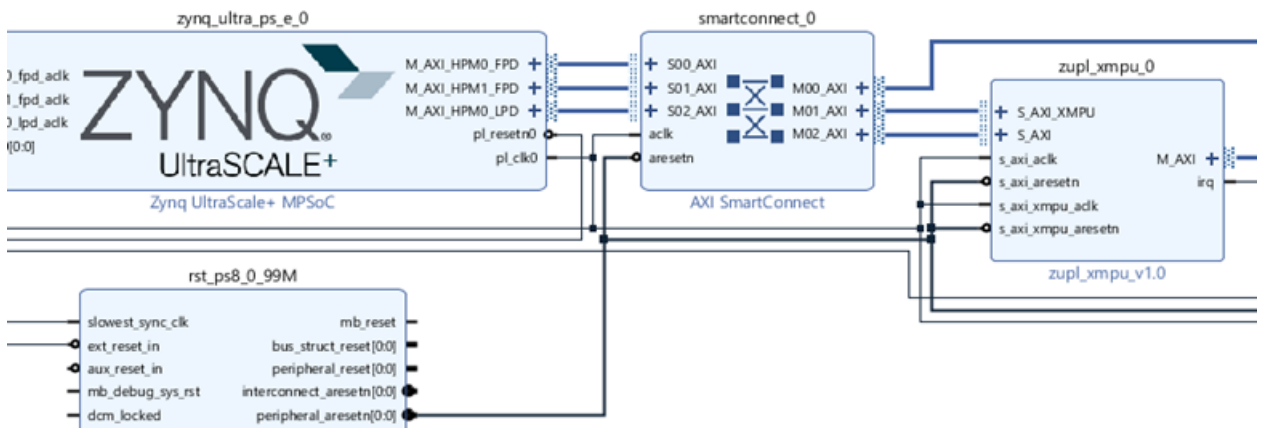
Figure 5: Using AXI SmartConnect



Connecting to Multiple PS Master I/Fs

The SmartConnect combines multiple PS Master I/Fs into a single or multiple XMPU_PLs, as evidenced from the following figure. Use SmartConnect instead of AXI-Interconnect to maintain access to PS MasterIDs.

Figure 6: Connecting Multiple PS Masters I/Fs

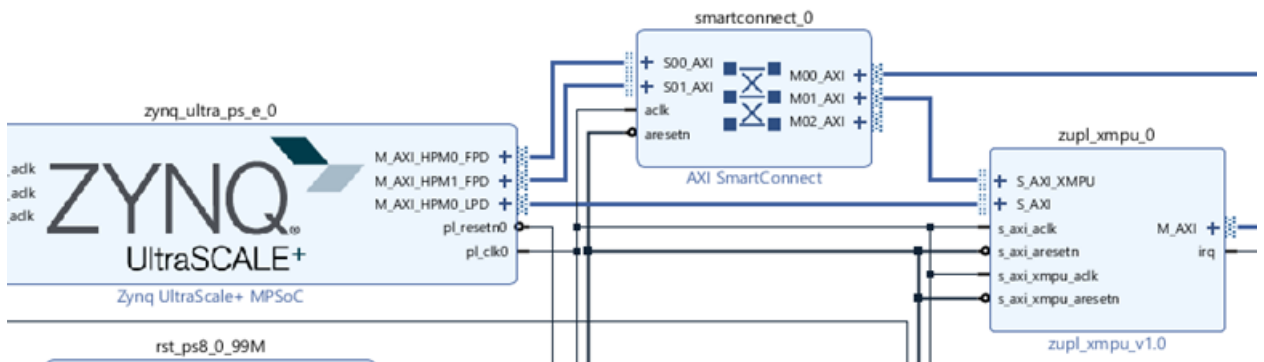


Note: The XMPU_PL will not provide any AXI data width conversion. Use SmartConnect upstream, and/or AXI-Interconnect downstream, to provide any needed data or clock conversions between the PS Master and end-point PL-Slaves.

Connecting Directly to PS Master I/Fs

The following figure shows the S_AXI port of the XMPU_PL may be directly connected to a PS master I/F. The data widths of both interfaces are selectable in their respective IP customization settings in the IP integrator. It is the responsibility of the user to ascertain that both are set to the same value.

Figure 7: PS Master I/F Direct Connection

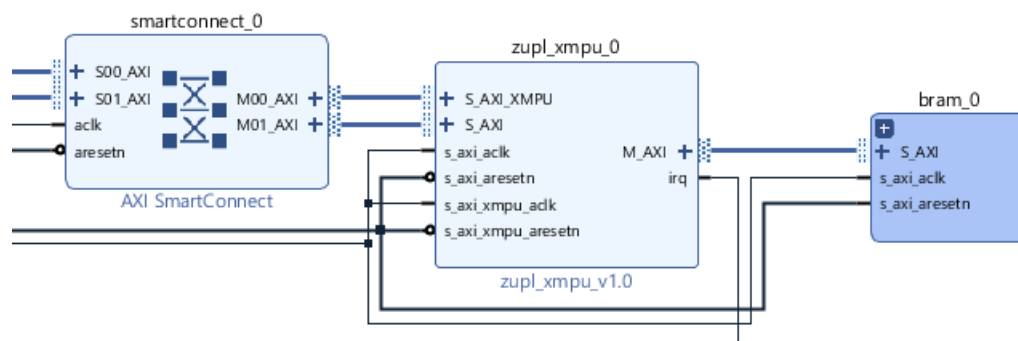


Note: The XMPU_PL will not provide any AXI data width conversion. Use SmartConnect upstream, and/or AXI-Interconnect downstream, to provide any needed data or clock conversions between the PS Master and end-point PL-Slaves.

Connecting Directly to PL Slave I/Fs

An XMPU_PL can be dedicated to a specific PL Slave and directly connected to the slave I/F without an interconnect stage. The XMPU_PL AXI Data Width must be set in accordance with the slave's data width (typically, 32-bits).

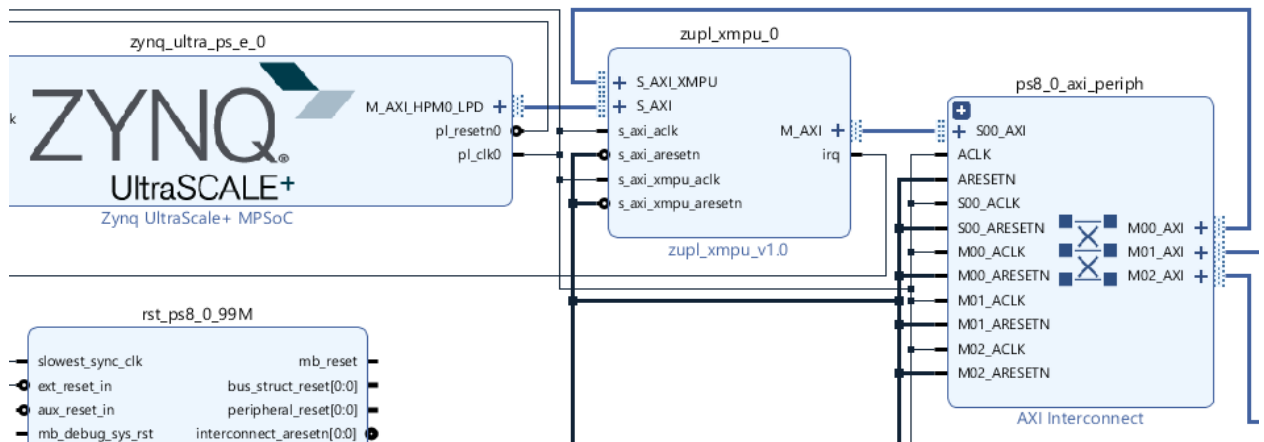
Figure 8: Slave I/F Direct Connection



Isolating the XMPU_PL Configuration

As described in [Configuration Lock](#), from the [Overview](#) section, the XMPU_PL configuration registers can be write protected from unauthorized masters, but are still readable. The following figure demonstrates one way to completely isolate the configuration I/F.

Figure 9: Configuration I/F Isolation



Map the S_AXI_XMPU configuration slave port to the M_AXI of the `zupl_xmpu` instead of using the Configuration Lock. Either the static or run-time configuration can define a region to protect the XMPU_PL configuration from both read and write accesses.

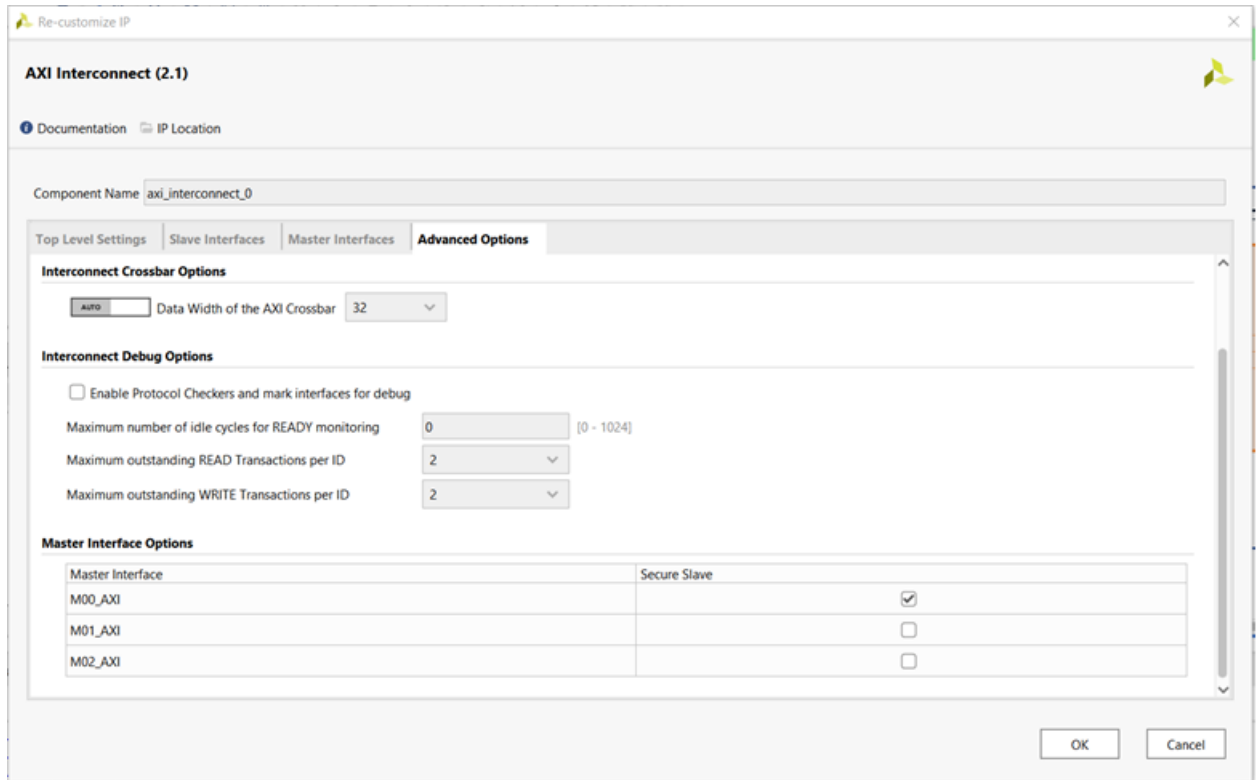


TIP: If using a run-time application to define the XMPU_PL configuration protection region, ensure that the `DefRdAllowed` and `DefWrAllowed` settings in the CTRL register are set. Otherwise, the run-time application may not have access to load the region parameters. `DefRdAllowed` and `DefWrAllowed` are set by default.

Isolating Secure Slaves

Enabling the Advanced Configuration Options in the AXI-Interconnect IPI customization window reveals Master Interface Options to select AXI Master output ports as being connected to Secure Slaves. The AXI-Interconnect customization window is shown in the following figure.

Figure 10: AXI-Interconnect Secure Slaves



Applying this setting causes the AXI-Interconnect to poison any transaction targeting a secure slave with an unsecure protection level ($AxProt[1]=1$).

This feature can be used in conjunction with the XMPU_PL Poison-by-Attribute setting. By disabling Poison-by-Address setting in the XMPU_PL, a poisoned transaction gets forwarded with non-secure protection level ($AxProt[1]=1$) causing the AXI-Interconnect to block the transaction.

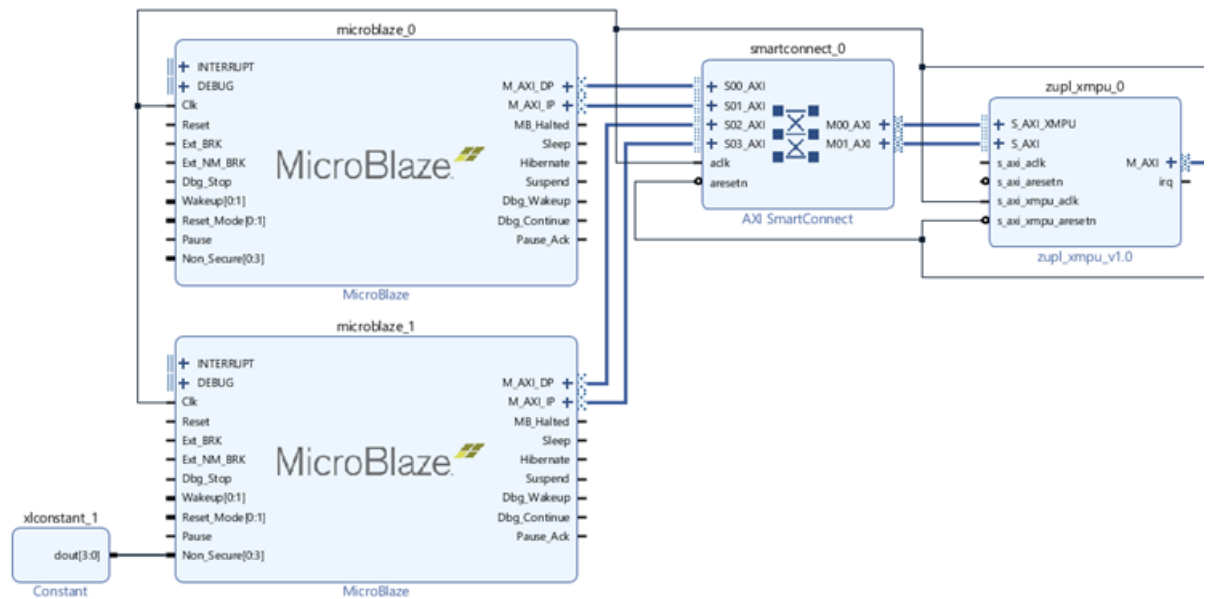
Note: The SmartConnect does not have this feature.



TIP: The AXI-Interconnect Secure Slave feature may also be used to isolate secure slaves from Non-secure PL masters without the use of an XMPU_PL.

Isolating PL Masters

PL masters, such as MicroBlaze or AXI DMA, do not output a MasterID, nor do they utilize the AxUser side-channel. Therefore, such masters cannot be differentiated from each other on that basis. The following figure shows MicroBlaze processors that supports a Non_Secure operating mode.

Figure 11: Secure and Non_Secure MicroBlazes


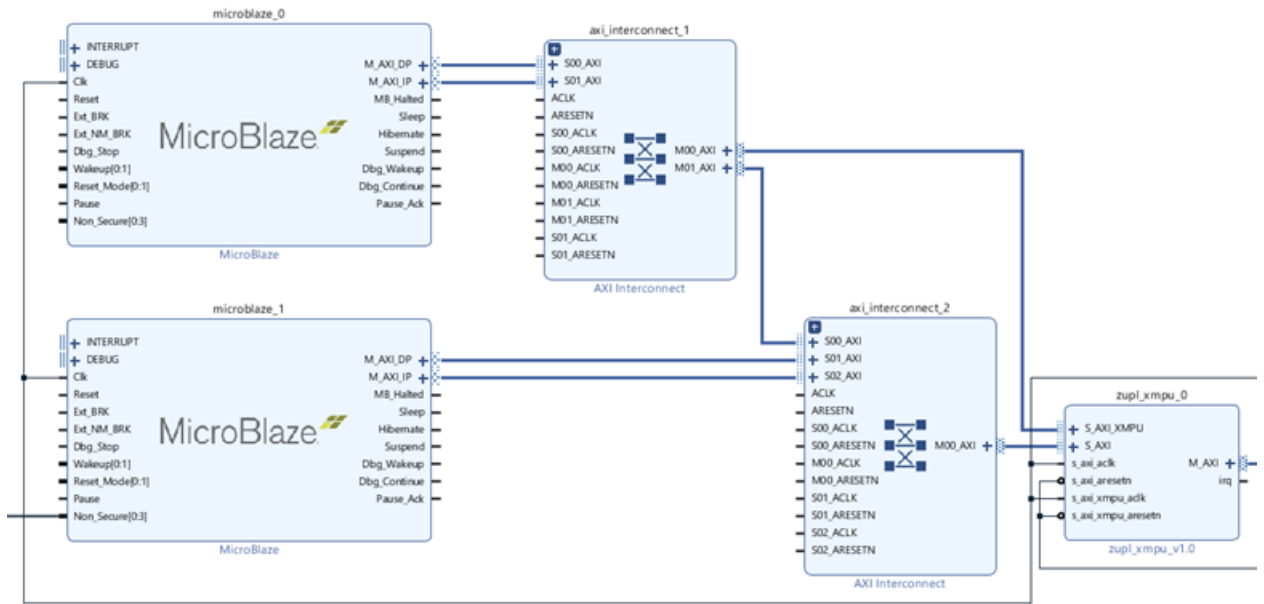
The Non_Secure[0:3] inputs may be asserted by a constant in the IPI block design. Each of the four bits control the Security level (AxProt[1]) for each of the AXI master ports (M_AXI_DP, M_AXI_IP, M_AXI_DC, M_AXI_IC).

For the configuration above, it is recommended to disable the MasterID checks in the region configuration, Rxx_CONFIG[MidCheckDisable], and rely on the security level to differentiate between the processors.



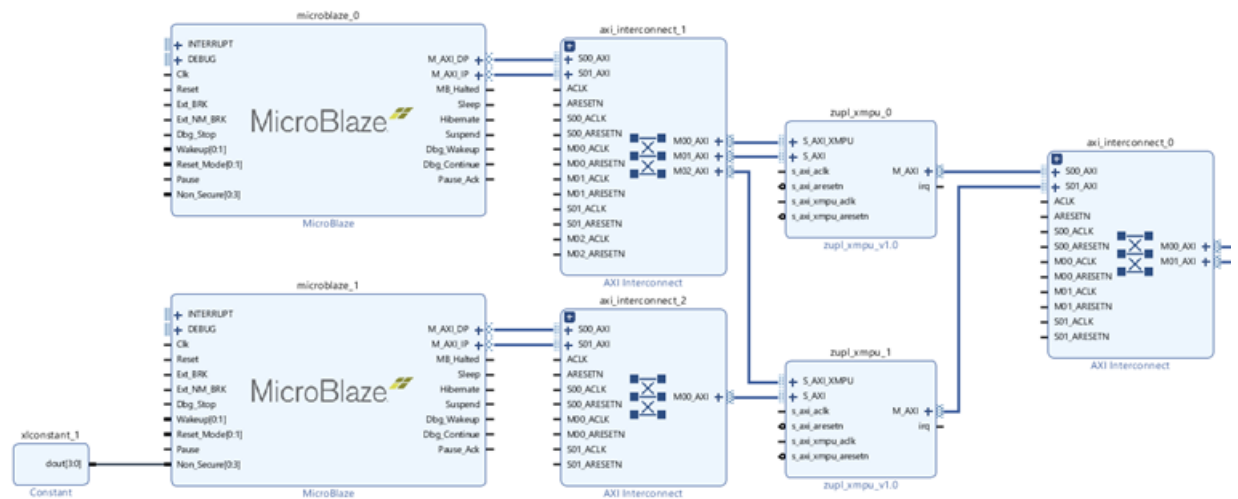
TIP: Using security level controls on the PL master enables the capability of using NonSecure with Strict Check Type regions.

Figure 12: S_AXI_XMPU Isolated to Secure MicroBlaze



The previous figure shows an example of isolating the S_AXI_XMPU configuration port to the secure MicroBlaze. Additional protections are not required as only the secure MicroBlaze has a physical connection. Similarly, the designer can establish a path to any secure processor, in the PL or PS, of their choosing to configure and manage any XMPU_PL in the system.

Figure 13: MicroBlaze with Dedicated XMPU



The previous example exhibits that each and every MicroBlaze processor has a dedicated XMPU_PL. There is no need to differentiate between masters in this configuration.



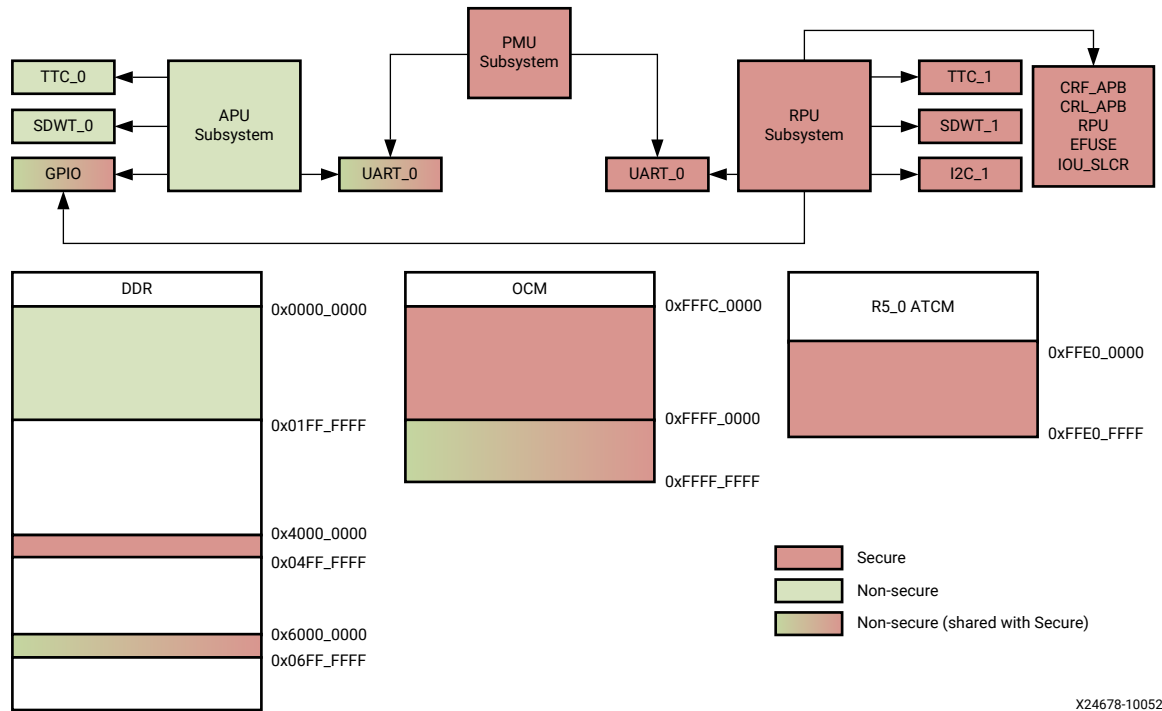
TIP: If the run-time configuration access is not needed for system operation, the example in the previous figure could have alternatively been implemented with the AXI MMU IP which also provides address decoding, read and write access control, and is only statically configured.

Isolation Example Design

System Isolation

The isolation reference design created in *Isolation Methods in Zynq UltraScale+ MPSoCs* (XAPP1320) is the starting point for building the PL isolation example design. The TrustZone (TZ) settings for the Processing System (PS) are shown in the following figure.

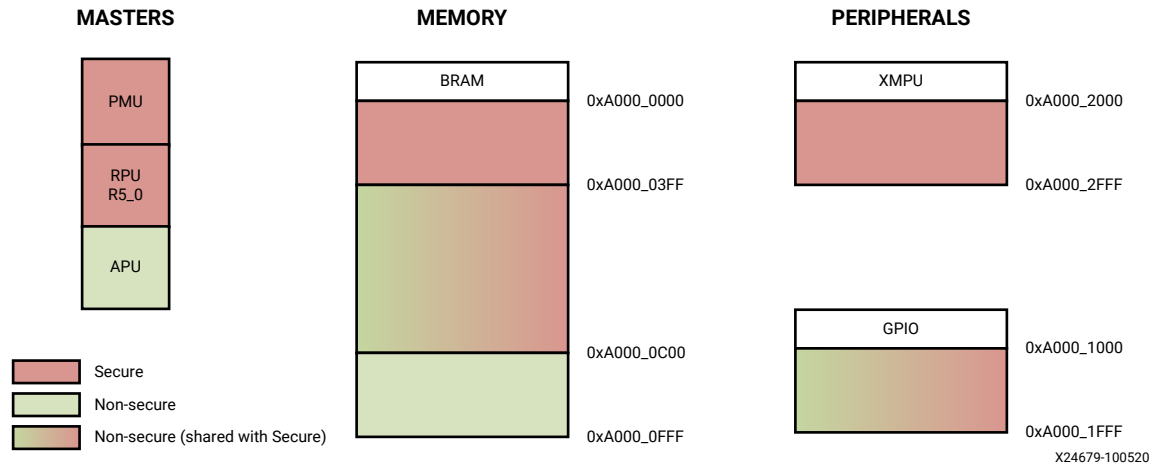
Figure 14: Isolation Reference Design TrustZone Settings



The system contains three active PS masters. The PMU and RPU (r5_0) are designated Secure, and the APU is designated Non-Secure. All three masters execute as bare metal standalone OS.

The PL isolation example design adds the PL memory and peripheral elements shown in the following figure.

Figure 15: PL Isolation Security Settings



Secure PL Memory

The first KB of the PL BRAM will be designated as secure, which means that it must only be accessible by the secure masters, PMU and RPU (R5_0).

Secure PL Peripherals

The S_AXI_XMPU configuration port of the XMPU_PL will be designated as secure, which means that it will only be accessible by the Secure masters, PMU and RPU (R5_0). If you use the LOCK registers in the XMPU, the configuration port becomes writable to only the designated masters, but still is readable by other masters.

Non Secure PL Memory

PL BRAM's last KB is designated as non-secure and is accessible only by the APU. Configure the associated XMPU region as non-secure with Strict Check Type. The APU has to set the AXI protection security level to non-secure ($AxProt[1]=1$) to access the region. Since bare metal standalone applications are being run in this example, all transactions originating from the APU enters the PL as secure ($AxProt[1]=0$). Therefore, to isolate the region to the APU, the region is configured as secure, but only the APU is authorized to access it.

Non-Secure Shared Memory & Peripherals

The middle two KBs of the PL BRAM and the AXI GPIO are designated as non-secure shared. They must be accessible by both secure and non-secure masters. One way to accomplish this is to designate regions to cover their respective address ranges and list all the masters as authorized. Alternatively, omit defining a region and instead utilize the default CTRL register settings to allow read and write access to undefined ranges.

The following table shows the XMPU_PL configuration for the example design. The MACRO definitions can be found in the `zupl_xmpu` BSP SW driver (`zupl_xmpu_hw.h`).

Table 19: XMPU PL Region Definitions

CONTROL	MACROS	Description
CTRL	XMPU_PL_CTRL_DEFRD + XMPU_PL_CTRL_DEFWR + XMPU_PL_CTRL_PSNATTREN + XMPU_PL_CTRL_PSNADDREN + XMPU_PL_CTRL_ARSP_DEC	Default Read Default Write Poison by Attribute Poison by Address Poison Response DECERR
LOCK	1	enable
LOCK_BYPASS	XMPU_PL_MID_RPU0 + XMPU_PL_MID_PMU	RPU0 PMU
REGION 0		
R00_START	BRAM BASEADDR	BRAM Base Address
R00_END	BRAM BASEADDR + 0x03FF	Size 1KB
R00_MASTERS	XMPU_PL_MID_RPU0 + XMPU_PL_MID_PMU	RPU0 PMU
R00_CONFIG	XMPU_PL_REGION_WR_ALLOW + XMPU_PL_REGION_RD_ALLOW + XMPU_PL_REGION_ENABLE	Region Write Allow Region Read Allow Region Enable
REGION 1		
R01_START	BRAM BASEADDR + 0x0C00	BRAM Base Address + 3 KB
R01_END	BRAM BASEADDR + 0x0FFF	Size 1 KB
R01_MASTERS	XMPU_PL_MID_APU0	APU
R01_CONFIG	XMPU_PL_REGION_WR_ALLOW + XMPU_PL_REGION_RD_ALLOW + XMPU_PL_REGION_ENABLE	Region Write Allow Region Read Allow Region Enable

Reference Design

Download the reference design files for this application note from the [Xilinx website](#).

Reference Design Matrix

The following checklist indicates the procedures used for the provided reference design.

Table 20: Reference Design Matrix

Parameter	Description
General	
Developer name	Carl Carmichael
Target devices	Zynq UltraScale+ Devices
Source code provided?	Yes
Source code format (if provided)	C, VHDL
Design uses code or IP from existing reference design, application note, 3rd party or Vivado software? If yes, list.	<i>Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)</i>
Simulation	
Functional simulation performed	Yes
Timing simulation performed?	No

Table 20: Reference Design Matrix (cont'd)

Parameter	Description
Test bench provided for functional and timing simulation?	No
Test bench format	No
Simulator software and version	Yes
SPICE/IBIS simulations	N/A
Implementation	
Synthesis software tools/versions used	N/A
Implementation software tool(s) and version	N/A
Static timing analysis performed?	Yes
Hardware Verification	
Hardware verified?	Yes
Platform used for verification	ZCU102 Evaluation Board

Reference Design Zip File

The xapp1353-pl-isolation.zip file (download from [Xilinx website](#)) contains a Vivado packaged IP with example design support files. A description of the zip archive is provided in the following table.

Table 21: Contents of Reference Design Archive

Directory/File Name	Description
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0	IP Repository Package
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/component.xml	This IP-XACT file defines the contents of the IP to Vivado.
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/bd/bd.tcl	The Tcl script used by Vivado IP Integrator supports integration of the IP in the Block Design.
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/drivers/zupl_xmpu_v1_0	This is the directory of the low-level software drivers for the zupl_xmpu PL peripheral. When the Vivado project's hardware is exported to SDK/Vitis, these drivers are included in the export, and will be included in any board support package (BSP) created within the SDK/Vitis workspace that uses the exported hardware.
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/xapp1320_isolation	This directory contains files to build the isolation example reference design from <i>Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)</i> .
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/xcu102_example	This directory contains files to build the PL isolation example reference design.
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/gui/zupl_xmpu_v1_0.gtc./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/xgui/zupl_xmpu_v1_0.tcl	The Tcl script used by Vivado IP Integrator creates the configuration GUI for the PL instance IP.
./XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/hdl	Reference core (vhdl) source files

Build HW Design in Vivado (2019.1 – 2020.1)

Once you have obtained and extracted the design files for this tutorial, you have the option to either manually add the `zupl_xmpu` reference core to the isolation reference design, or use an automated script to build the completed HW platform.

If you wish to make the modifications manually, proceed to [Start with the XAPP1320 Isolation Reference Design](#) . If you wish to build the HW platform with an automated script, proceed to [Build with the Automated Design Script](#) section.

Start with the XAPP1320 Isolation Reference Design

Isolation Reference Design

The next section provides a step-by-step instruction to manually add the `zupl_xmpu` reference core to isolation reference design. Reconstruct the reference design from *Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)*.

An automated script is provided to build the design. If you wish to review the procedures for creating an isolated design, refer to *Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)*.

Following are the steps to build the isolation reference design:

1. Unzip the `zupl_xmpu` archive: `zupl_xmpu_v1_0[revision].zip`.
2. a. If running on Linux: Browse to the `./zupl_xmpu_v1_0/example_designs/xapp1320_isolation` directory and run Vivado.
- b. If running Vivado on Windows, use the Tcl Console to navigate to the `zupl_xmpu_v1_0/example_designs/xapp1320_isolation` directory:

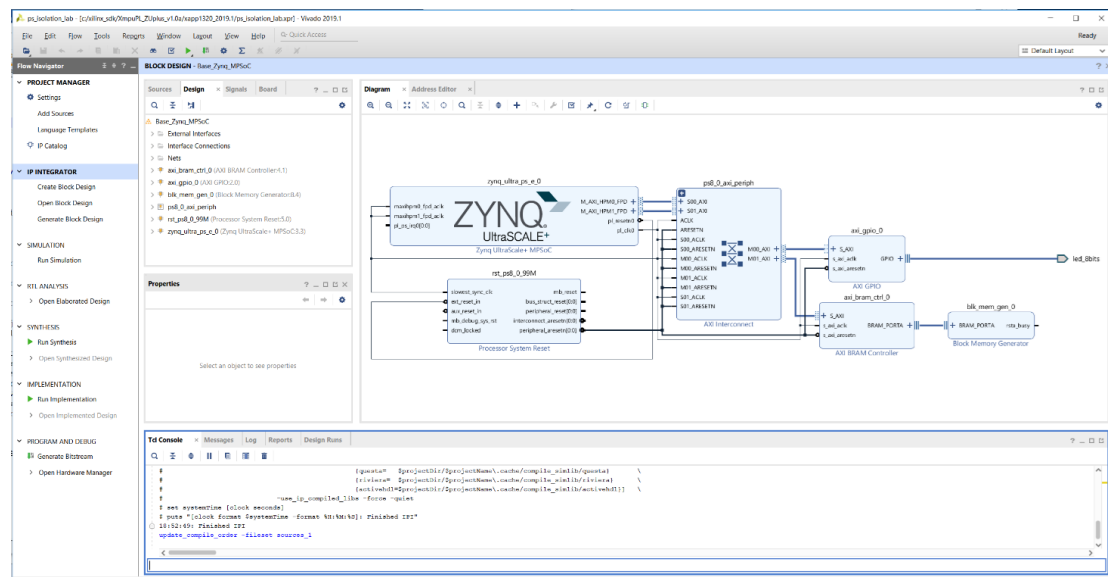
```
cd[<your_path>/XmpuPL_ZUplus_v1.0[revision]/zupl_xmpu_v1_0/example_designs/xapp1320_isolation}
```

3. Run the `example_design.tcl` script:

```
source ./example_design.tcl
```

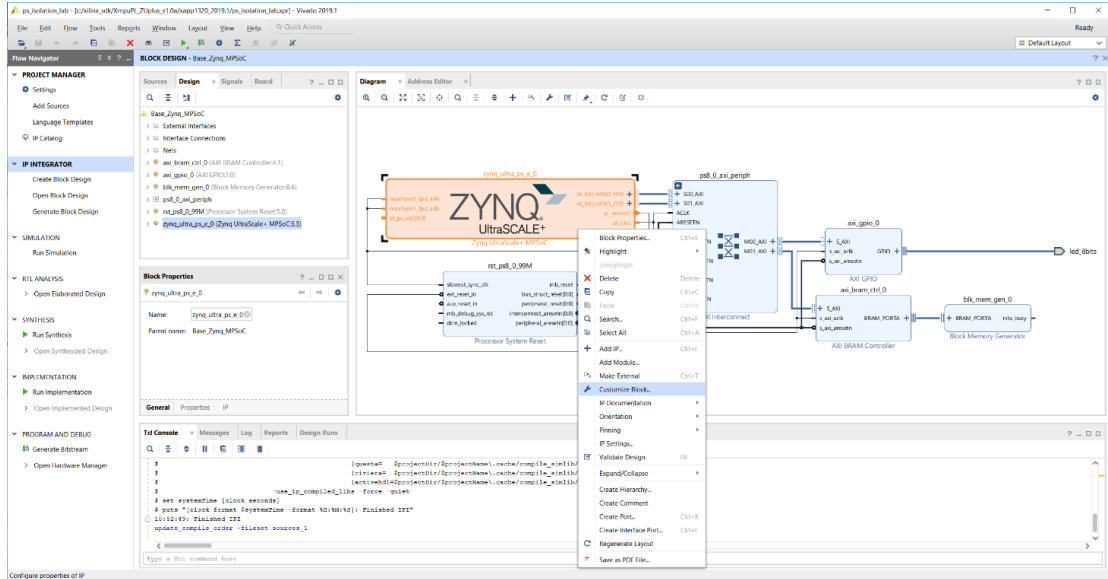
When the IP Integrator Block Design is complete, it looks like the following figure.

Figure 16: Isolation Example Block Design



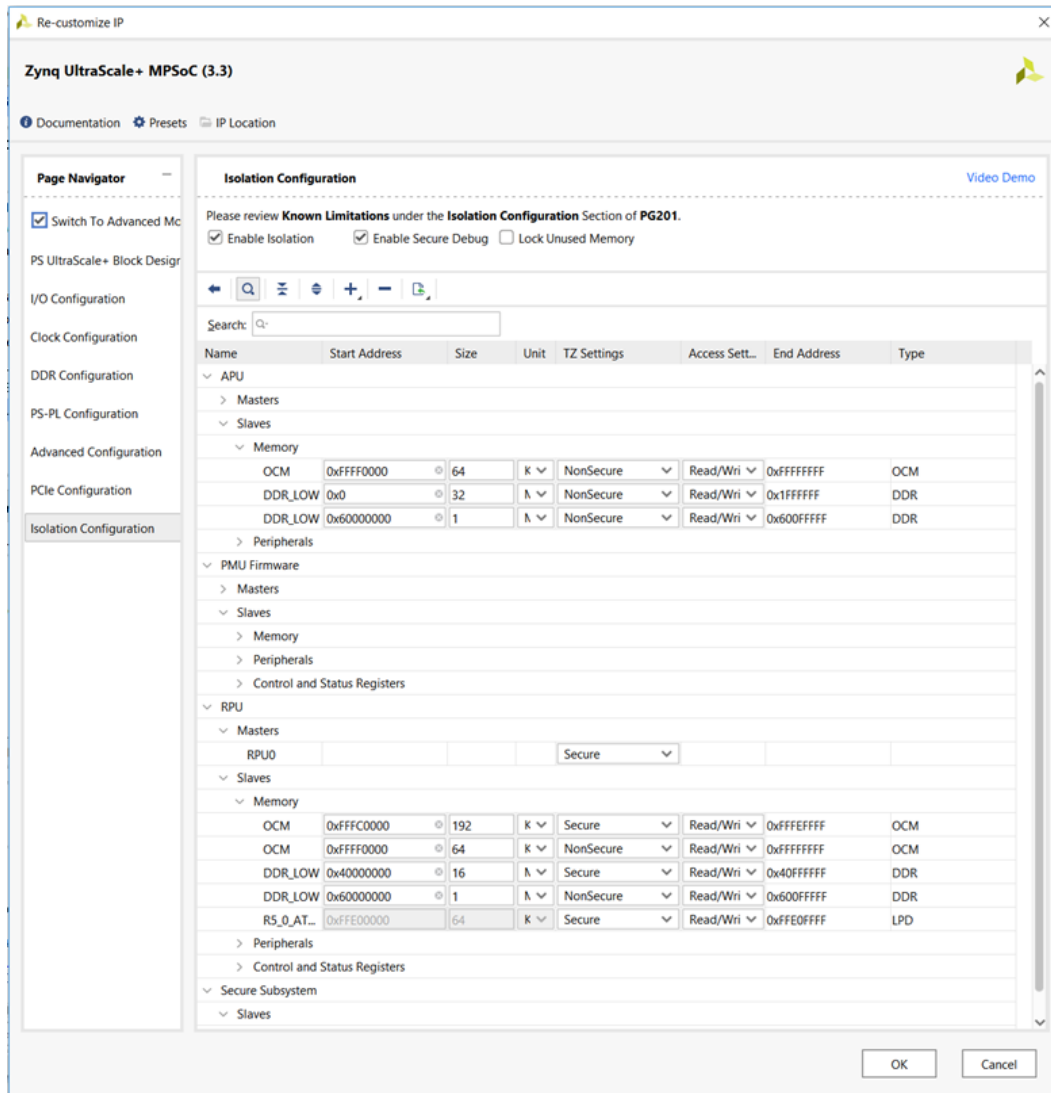
4. Right-click `zynq_ultra_ps_e_0` and select **Customize Block...**

Figure 17: Customize Zynq_Ultra_PS



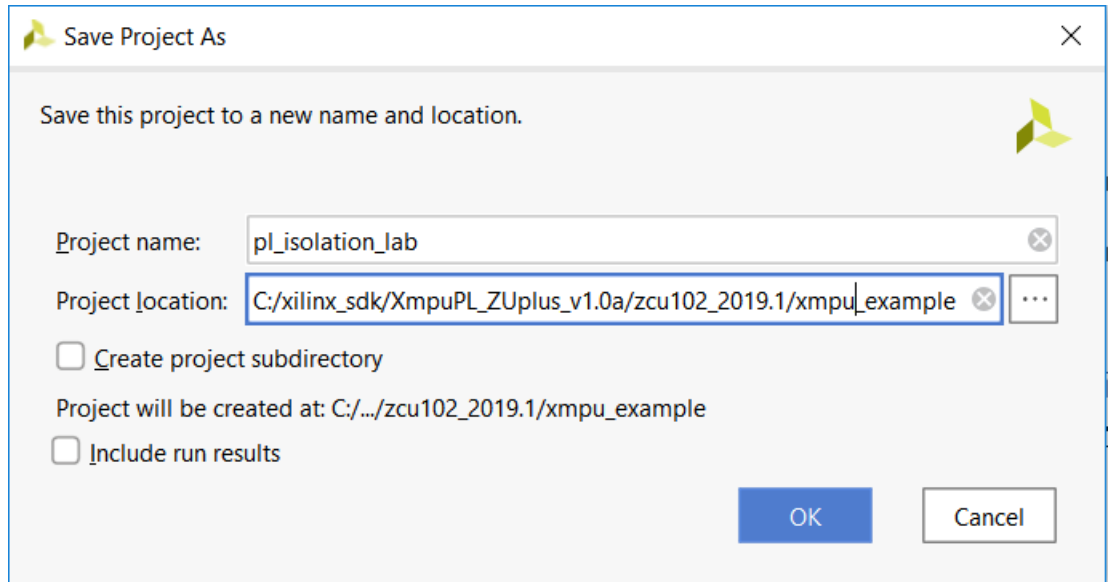
5. Click **Switch to Advanced Mode**, then click **Isolation Configuration**, and you can see that the isolation parameters as described in *Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)* have been implemented. The following figure shows a sample of the settings.

Figure 18: Isolation Configuration Parameters



- a. Click **Cancel** to close the customization window.
6. Save the project to a new name before making modifications.
 - a. **File-> Project->Save As**
 - b. Fill in the required information as shown in the following figure:
 - i. Project name: `pl_isolation_lab`
 - ii. Project Location: `<your_path>/XmpuPL_ZUpplus_v1.0a/zcu102_<version>/xmpu_example`

Figure 19: Save Project As



- iii. Do not create project sub directory. Do not include run results. Click **OK**.

Manual Insertion of the XMPU_PL in the IP Integrator

Manual Insertion in the IP Integrator

Isolation reference design gets created in the previous section, [Start with the XAPP1320 Isolation Reference Design](#), and is saved to the following location:

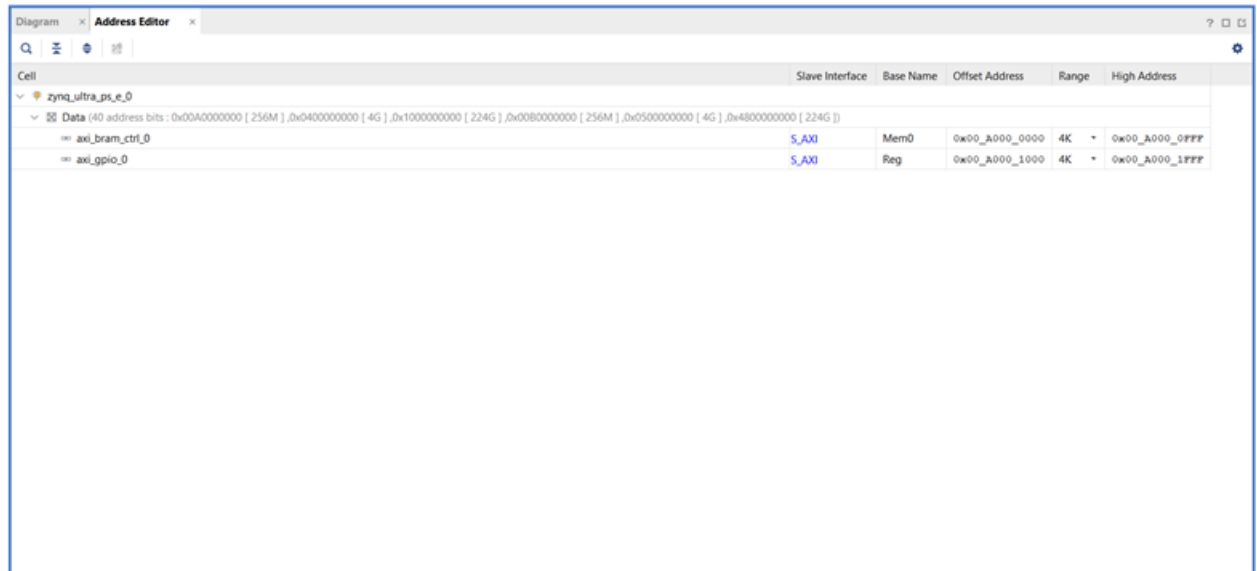
```
/XmpuPL_ZUplus_v1.0a/xcu102_<version>/xmpu_example/pl_isolation_lab.xpr
```

Open the project in Vivado, and click **Open Block Design** if you have it closed.

You will go through the following steps to add a XMPU_PL module to the block design.

1. Click the **Address Editor** and note the current mappings in the following pane.
 - a. `axi_bram_ctrl_0` is mapped to `0x00_A000_0000 (4K)` and `axi_gpio_0` is mapped to `0x00_A000_1000 (4K)` in the **Address Editor** window. Return to the diagram.

Figure 20: Address Editor



2. **Add** the `zupl_xmpu_v1_0` core to your repository.
 - a. Click **Settings** beneath Project Manager. This is located in the Flow Manager.
 - b. Under Project Settings, expand > IP, and click **Repository**.
 - c. Click the + symbol in the IP Repositories.
 - d. Browse to the `zupl_xmpu_v1_0` directory and click **Select**.
 - e. One (1) repository must be added to the project. Click **OK** to clear the **Add Repository** window.
 - f. Click **OK** to clear the **Settings** window.
3. Add the `zupl_xmpu_v1_0` core to the block design.
 - a. Click the + symbol in the Block Diagram window.
 - b. **Type** `zupl` in the Search field type and **double-click** `zupl_xmpu_v1_0` or press **enter**.
4. Add a **SmartConnect** IP core.
 - a. Click the + symbol in the Block Diagram window.
 - b. Type `smart` in the search field type.
 - c. Double-click **AXI SmartConnect** or just press **enter**.
 - d. Right-click the `smartconnect_0` instance and select **Customize Block**.
 - e. Change the Number of Master Interfaces to **2** and click **OK**.
5. **Disconnect** the AXI Interconnect block from the Zynqzynq PS block.
 - a. **Select** and **delete** the bus signals between `zynq_ultra_ps_e_0` and `ps8_0_axi_periph`.
 - b. Right-click `ps8_0_axi_periph` and **Customize Block**.
 - c. Reduce the Number of Slave Interfaces to **1**. Click **OK**.


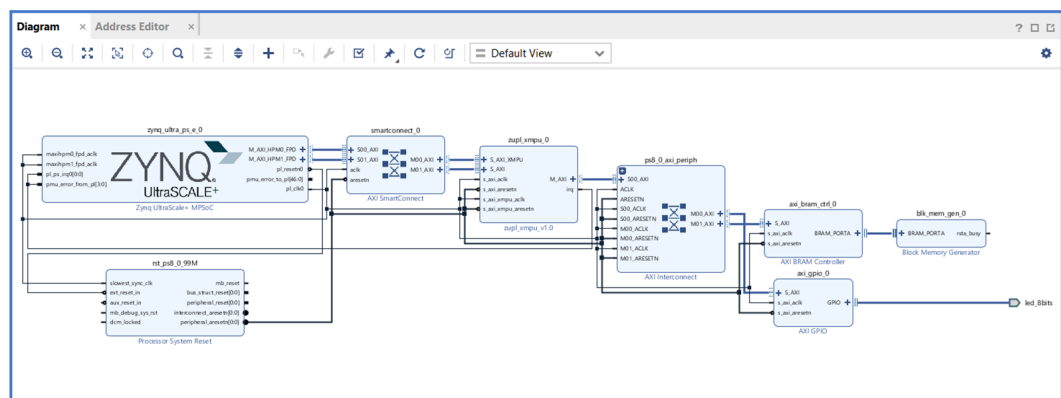
6. Connect the Zynq PS M_AXI_ ports.
 - a. Connect `zynq_ultra_ps_e_0/M_AXI_HPM0_FPD` to `smartconnect_0/S00_AXI`.
 - b. Connect `zynq_ultra_ps_e_0/M_AXI_HPM1_FPD` to `smartconnect_0/S01_AXI`.
7. Connect the XMPU AXI ports.
 - a. Connect `zupl_xmpu_0/S_AXI_XMPU` to `smartconnect_0/M00_AXI`.
 - b. Connect `zupl_xmpu_0/S_AXI` to `smartconnect_0/M01_AXI`.
 - c. Connect `zupl_xmpu_0/M_AXI` to `ps8_0_axi_periph/S00_AXI`.
 - d. Regenerate Layout. Click **OK**.
8. Connect the AXI clock and reset ports.
 - a. Click **Run Connection Automation**.
 - b. Select **All Automation**. Click the Regenerate button.
 
 - c. Manually connect any unconnected `aclk` or `aresetn` ports.
9. Connect the IRQ signal.
 - a. This example design demonstrates the usage of PMU and RPU to receive interrupts from the XMPU so the `pmu_error_from_pl` port needs to be enabled. Right-click `zynq_ultra_ps_e_0` and select **Customize Block**.
 - b. Click **PS-PL Configuration**. Expand > **General**. Expand > **Others**.
 - c. Select the check box for **Errors to and from PMU**. Click **OK**.
 - d. Connect `zupl_xmpu_0/irq` port to both `pl_ps_irq0[0:0]` and `pmu_error_from_pl[3:0]` ports on `zynq_ultra_ps_e_0`.
 - e. Regenerate Layout. The diagram resembles the following.

Figure 21: xmpu_pl Example Block Diagram



10. Map the Address segments.
 - a. Click **Address Editor**.

- b. If using 2019.1 or 2019.2:
 - i. Expand > `zupl_xmpu_0` > **M_AXI** > **Unmapped Slaves (2)**.
 - ii. Right-click `axi_bram_ctrl_0` and `axi_gpio_0` and select **Assign Address**.
 - iii. If the new address mappings do not match the size and base address shown in `>zynq_ultra_ps_e_0>Data<`, then fix the values to match. Fix the range value first and then the offset address.
 - iv. Expand > `zynq_ultra_ps_e_0` > **Data** > **Unmapped Slaves (1)**.
 - v. Right-click `zupl_xmpu_0` and select **Assign Address**.
 - vi. `S_AXI_XMPU` must be mapped to `0x00_A000_2000 (4K)`.
 - vii. The final configuration is shown in the following diagram.

Note: The `zupl_xmpu_0/S_AXI` still shows as an Unmapped Slave. This is due to the AXI Bridge in the core. Do not map this segment.

Figure 22: `xmpu_pl` Example Address Map

Cell	Slave Interface	Slave Segment	Offset Address	Range	High Address
zupl_xmpu_0					
M_AXI (40 address bits : 1T)					
axi_bram_ctrl_0	S_AXI	Mem0	0x00_A000_0000	4K	0x00_A000_0FFF
axi_gpio_0	S_AXI	Reg	0x00_A000_1000	4K	0x00_A000_1FFF
zynq_ultra_ps_e_0					
Data (40 address bits : 0x00A0000000 [256M] ,0x0400000000 [4G] ,0x1000000000 [224G] ,0x0800000000 [256M] ,0x0500000000 [4G] ,0x4800000000 [224G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x00_A000_0000	4K	0x00_A000_0FFF
axi_gpio_0	S_AXI	Reg	0x00_A000_1000	4K	0x00_A000_1FFF
zupl_xmpu_0	S_AXI_XMPU	S_AXI_XMPU_Config	0x00_A000_2000	4K	0x00_A000_2FFF
Unconnected Slaves					
zupl_xmpu_0	S_AXI	S_AXI			

- c. If using 2020.1:
 - i. Expand > **Network 0** > `zynq_ultra_ps_e_0` > **Data** > **Unassigned (4)**.
 - ii. Right-click `zupl_xmpu_0: S_AXI_XMPU (S_AXI_XMPU_Config)` and select **Assign**.
 - iii. Change the range of `S_AXI_XMPU` to **4K**.
 - iv. Change the Master Base Address of `S_AXI_XMPU` to `0x00_A000_2000`.
 - v. Select **File** > **Save Block Design**.
 - vi. Select **Tools** > **Validate Design**.
 - vii. Ignore warnings about unmapped slaves. Click **OK**.
 - viii. Right-click **Unconnected Slaves/zupl_xmpu_0: S_AXI (S_AXI)** and select **Exclude**.
 - ix. Select **File** > **Save Block Design**.

11. Customize the `zupl_xmpu_0` block.

- a. Return to the previous diagram and right-click `zupl_xmpu_0` and **Customize Block**.

- b. Select **AXI Settings**.
- c. The C_S_AXI_DATA_WIDTH is set to the default value of 32. Leave it at default setting. The AXI infrastructure blocks adjusts for the PS M_AXI_ bus widths.
- d. The M_AXI_BASEADDR and M_AXI_HIGHADDR will not have any functional effect. However they are provided as a means to communicate to the SW Driver the address range that the XMPU monitor. These values will be exported to the `xparameters.h` file and be included in the peripheral's instance configuration data.
- e. (Optional) Set these values to correspond with the address ranges shown in the previous figure.
 - i. HIGHADDR: 0xA0001FFF
 - ii. BASEADDR: 0xA0000000

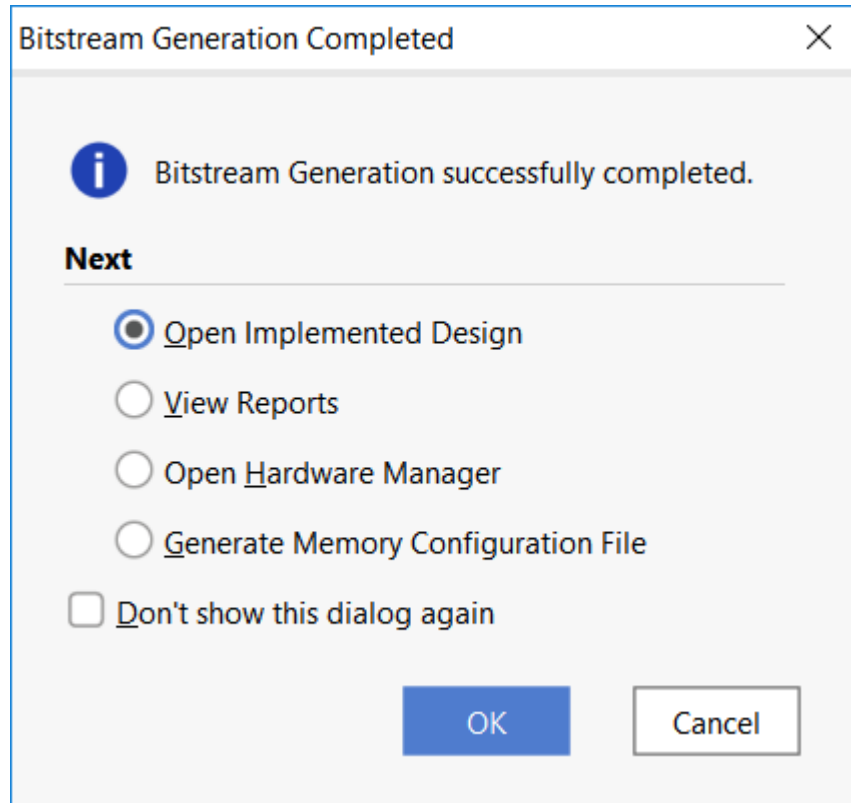


TIP: Use the upper 32 bits if you need to specify a 40 bit address..

- f. Select the Regions Tab and note the value for Regions Max. The default is the absolute maximum setting at 16. If the HW designer knows exactly how many regions the SW designer needs, they could select a lower number to conserve the PL resources. The setting can be kept to default for the time being.
 - g. Click **OK**.
12. (Optional) Set **Project Synthesis Language**.
- a. The top level synthesis language for the project may optionally be set to either VHDL or Verilog. You can choose either one of them for this demonstration.
 - b. Click **Settings** in the Flow Manager under Project Settings.
 - c. Click **General** under Project Settings.
 - d. Select the Target Language: VHDL or Verilog. Click **OK**.
13. Create the top level wrapper.
- a. Right-click Base_Zynq_MPSoC and select **Create HDL Wrapper** in the Block Design Sources window.
 - b. Let Vivado manage wrapper. Click **OK**.
 - c. Click **OK** if using 2020.1 and a Critical Messages window appears.
14. Implement design.
- a. Click **Generate Block Design** under IP Integrator.
 - i. Select **Out of context** per IP and click **Generate**.
 - b. If a **Generate Output Products** dialogue appears when the module runs have launched:
 - i. Click **OK**.
 - ii. If using 2020.1, a **Critical Messages** window appears; click **OK**.
 - c. Wait for all the block runs to complete.
 - i. View the status in the upper right corner or monitor the Out-of-Context Module Runs on the Design Runs tab below.

- d. Click **Generate Bitstream** in the Flow Navigator, click **OK** or **Yes** and then **OK**.
- e. When the **Bitstream Generation Completed** window appears (see the following figure), click **OK** to open implemented design.

Figure 23: **Bitstream Generation Completed**

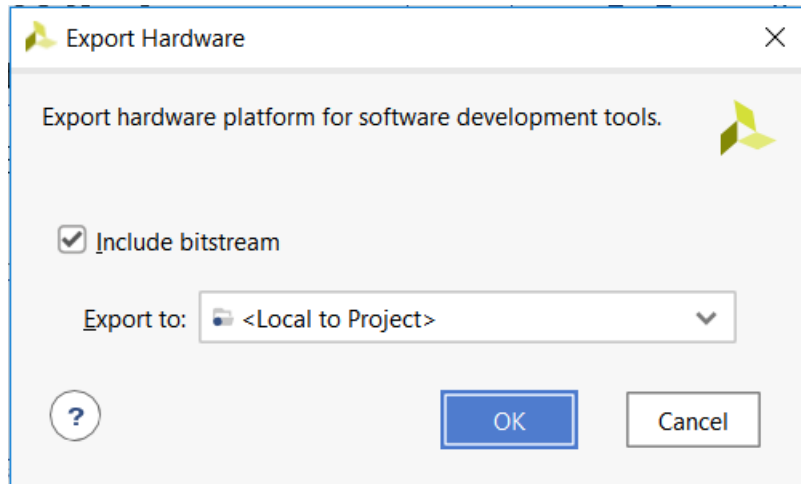


Note: For the Vitis flow, it is recommended to have the implemented design open when exporting hardware to Vitis. For SDK, it is not used.

15. Export hardware.

- a. Select **File->Export->Export Hardware** if using Vivado 2019.1.
 - i. Check **Include bitstream** and click **OK**.

Figure 24: Export Hardware in SDK

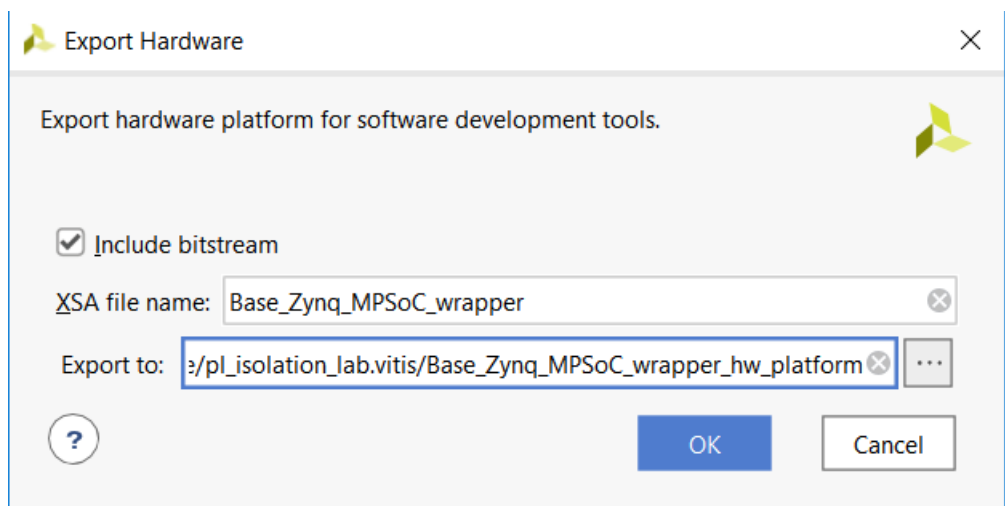


- b. Open the implemented design if using Vivado 2019.2 or 2020.1.
 - i. Select **File**-> **Export**->**Export Hardware**
 - ii. Select **Fixed**, and click **Next** if using 2020.1.
 - iii. Check **Include bistream**. If using 2020.1, click **Next**.
 - iv. XSA file name: `Bas_Zynq_MPSoC_wrapper`
 - v. Export to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_<version>/xmpu_example/  
pl_isolation_lab.vitis/Base_Zynq_MPSoC_wrapper_hw_platform
```

If prompted, click **OK** to **Create Directory**.

Figure 25: Export Hardware in Vitis



- vi. Click **OK** or **Next** > then **Finish**.

The hardware design is now complete. If using Vivado 2019.1, then proceed to [Creating the Isolation Test SW Applications in SDK 2019.1](#). If using Vivado 2019.2 or newer, proceed to [Creating the Isolation Test SW Applications in Vitis 2019.2](#) or [Creating the Isolation Test SW Applications in Vitis 2020.1](#), accordingly.

Build with the Automated Design Script

The previous section provided step-by-step instructions for manually creating the isolation example design from the isolation reference design provided in *Isolation Methods in Zynq UltraScale+ MPSoCs (XAPP1320)*. The following steps have been included in a script for an automated design build.



CAUTION! Running this script overwrites any existing build of the `xmpu_example` design.

Run the following steps to build the `xmpu` example design:

1. Unzip the `zupl_xmpu` archive: `zupl_xmpu_v1_0[revision].zip`
2. Start Vivado 2019.1, 2019.2, or 2020.1
 - a. If running Linux, browse to the `./zupl_xmpu_v1_0/example_designs/zcu102_example` directory and run Vivado.
 - b. If running Vivado on Windows, use the Tcl Console to navigate to the `./zupl_xmpu_v1_0/example_designs/zcu102_example` directory:

```
cd{<your_path>/XmpuPL-ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/zcu102_example}
```

3. Run the `xmpu_example_design.tcl` script:

```
source./xmpu_example_design.tcl
```

4. Click **Cancel** when the **Bitstream Generation successfully completed** window appears.

Note: For details on the address mapping and `xmpu` configuration for the design, refer to [step 10](#) and [step 11](#) in the previous chapter: [Manual Insertion of the XMPU_PL in the IP Integrator](#).

The hardware design is now complete. The automated script has already exported the hardware. If using Vivado 2019.1, proceed to [Creating the Isolation Test SW Applications in SDK 2019.1](#).

If using version 2019.2 or newer, proceed to [Creating the Isolation Test SW Applications in Vitis 2019.2](#) or [Creating the Isolation Test SW Applications in Vitis 2020.1](#).

Creating the Isolation Test SW Applications in SDK 2019.1

This section describes how to use SDK to create software that runs on the isolated system, created in the previous section. The following sections demonstrate five software projects that are created to test the features previously discussed. These projects and their functions are listed in the following table.

Table 22: Isolation Test Application Projects

Project	Description
r5_fsbl	FSBL running on R5_0
pmu_fw_u0	PMU firmware: event handler (prints to uart0)
pmu_fw_u1	PMU firmware: event handler (prints to uart1)
rpu_fault_injection	Fault Injection code running on R5_0
apu_fault_injection	Fault Injection code running on APU_0

Launch SDK

1. Open the `pl_isolation_lab` project in Vivado 2019.1 and select **File > Launch SDK**.
2. Click **OK** on the Launch Software Development Kit (SDK) tool dialog box.
3. Take a look at the `system.hdf` displayed in the middle screen, after SDK has been initialized and finished importing its hardware specification.
4. Note that the address map for each processor now contains an entry for `zupl_xmpu_0`.
5. In the project explorer frame, expand:

```
Base_Zynq_MPSoC_wrapper_hw_platform_0>drivers>zupl_xmpu_v1_0>src
```

You will find the `zupl_xmpu` SW driver files. These will be included in the board support packages (BSP) created for each project.

First Stage Boot Loader (FSBL)

The FSBL runs at boot time and loads the PS software projects and the PL bitstream. This section provides the steps necessary for creating this project.

1. Select **File > New > Application Project**.
 - a. Project Name: `r5_fsbl`
 - b. OS Platform: `<default>`
 - c. Hardware Platform: `<default>`
 - d. Processor: `psu_cortexr5_0`
 - e. Language: `<default>`
 - f. Compiler: `<default>`
 - g. Hypervisor Guest: `<default>`
 - h. Board Support Package: `<default>`
2. Click **Next**.
 - a. Available Templates: Zynq MP FSBL
3. Click **Finish**.

PMU Error Manager

The PMU firmware is required for any MPSoC processor system. The isolation reference design additionally utilized the error manager to handle and report violations detected by the PS XMPU/XPPU. Therefore, in this lab the PMU is also used for the initialization, configuration, and interrupt handling from the XMPU_PL.

The PMU Scheduler is used for XMPU_PL initialization and configuration, while the PMU Event Manager responds to interrupts and reports access violations from the core. The necessary code modifications are provided, and will be imported into the project. For a more detailed view of the implementation, refer to [A closer Look at the Platform Management Unit \(PMU\)](#).

Note: Some safety critical applications may prohibit the use of any modifications to the PMU firmware. For this reason, an additional lab, [Creating the Simple XMPU_PL \(RPU\) Example in SDK 2019.1](#), has been provided to demonstrate XMPU_PL initialization, configuration, and handling entirely from within the RPU.

For the isolation test, the APU can only print to UART0, and the RPU can only print to UART1. Therefore, two versions of the PMU firmware application is created. One prints to the UART0, and one to the UART1, to complement the RPU and APU fault injection applications.

To create the PMU firmware for UART0:

1. Select **File > New > Application Project**.
 - a. Project Name: `pmu_fw_u0`
 - b. OS Platform: `<default>standalone`
 - c. Hardware Platform: `<default> Base_Zynq_MPSoC_wrapper_hw_platform_0`
 - d. Processor: `psu_pmu_0`
 - e. Language: `<default>C`
 - f. Board Support Package: `<default>Create New pmu_fw_u0_bsp`
2. Click **Next**.
 - a. Available Templates: Zynq PMU Firmware
3. Click **Finish**.
4. Click to expand `pmu_fw_u0` in the project explorer.
5. Right-click `src` and select **Import...**
6. Select **General > File System** and click **Next**.
7. Browse and navigate to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/  
zcu102_example/sources/src/pmu_fw_2019.1
```
8. Click **Select Folder**.
9. Click **Select All**.
10. Click **Overwrite existing sources**.

11. Click Finish.

To create the PMU firmware for UART1:

1. Select File > New > Application Project.

- a. Project Name: `pmu_fw_u1`
- b. OS Platform: `<default>standalone`
- c. Hardware Platform: `<default> Base_Zynq_MPSoC_wrapper_hw_platform_0`
- d. Processor: `psu_pmu_0`
- e. Language: `<default>C`
- f. Board Support Package: `<default>Create New pmu_fw_u1_bsp`

2. Click Next.

- a. Available Templates: Zynq PMU Firmware

3. Click Finish.**4. Click to expand `pmu_fw_u1` in the project explorer.****5. Right-click `src` and select **Import...******6. Select **General > File System** and click **Next**.****7. Browse and navigate to :**

```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/  
zcu102_example/sources/src/pmu_fw_2019.1
```

8. Click **Select Folder.****9. Click **Select All**.****10. Click **Overwrite existing sources**.****11. Click **Finish**.****12. Right-click `pmu_fw_u1_bsp` and select **Board Support Package Settings**.****13. Select **standalone**.****14. Change `stdin` and `stdout` to `psu_uart_1`.****15. Click **OK**.**

APU Fault Injection Application in SDK

The fault injection applications from *Isolation Methods in Zynq UltraScale+ MPSoCs* ([XAPP1320](#)) performs read and write tests on various memory and peripheral locations to demonstrate the application of the TrustZone settings and response to the identity of the requesting master. These applications have been expanded to include the PL memory and peripheral locations shown in the following table.

Table 23: Fault Injection PL Memory Addresses

PL Memory Addresses	
PL_BRAM_S_BASE	0xA0000000
PL_BRAM_NS_SHARED_BASE	0xA0000400
PL_BRAM_NS_BASE	0xA0000C00

Table 24: Fault Injection PL Peripheral Addresses

PL Peripheral Addresses	
PL_XMPU_S_START	0xA0002000
PL_XMPU_S_LOCK	0xA0002020
PL_GPIO_NS_SHARED_START	0xA0001000

The following steps show how to create the APU fault injection application:

1. Select **File > New > Application Project**.
 - a. Project Name: `apu_fault_injection`
 - b. OS Platform: `<default>standalone`
 - c. Hardware Platform: `<default>`
 - d. Processor: `psu_cortexa53_0`
 - e. Language: `<default>C`
 - f. Compiler: `<default>64-bit`
 - g. Hypervisor Guest: `<default>No`
 - h. Board Support Package: `<default>Create new apu_fault_injection_bsp`
2. Click **Next**.
 - a. Available Templates: Empty Application
3. Click **Finish**.
4. Click to expand `apu_fault_injection` in the project explorer.
5. Right-click `src` and select **Import...**
6. Select **General > File System** and click **Next**.
7. Browse and navigate to:


```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/zcu102_example/sources/src/apu_fault_injection
```
8. Click **Select Folder**.
9. Click **Select All**.
10. Click **Overwrite existing sources...**
11. Click **Finish**.

Create the APU Fault Injection Boot Image

Note: For the following steps:

```
<build_path>=<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.1/xmpu_example/  
pl_isolation_lab.sdk
```

To create the boot image:

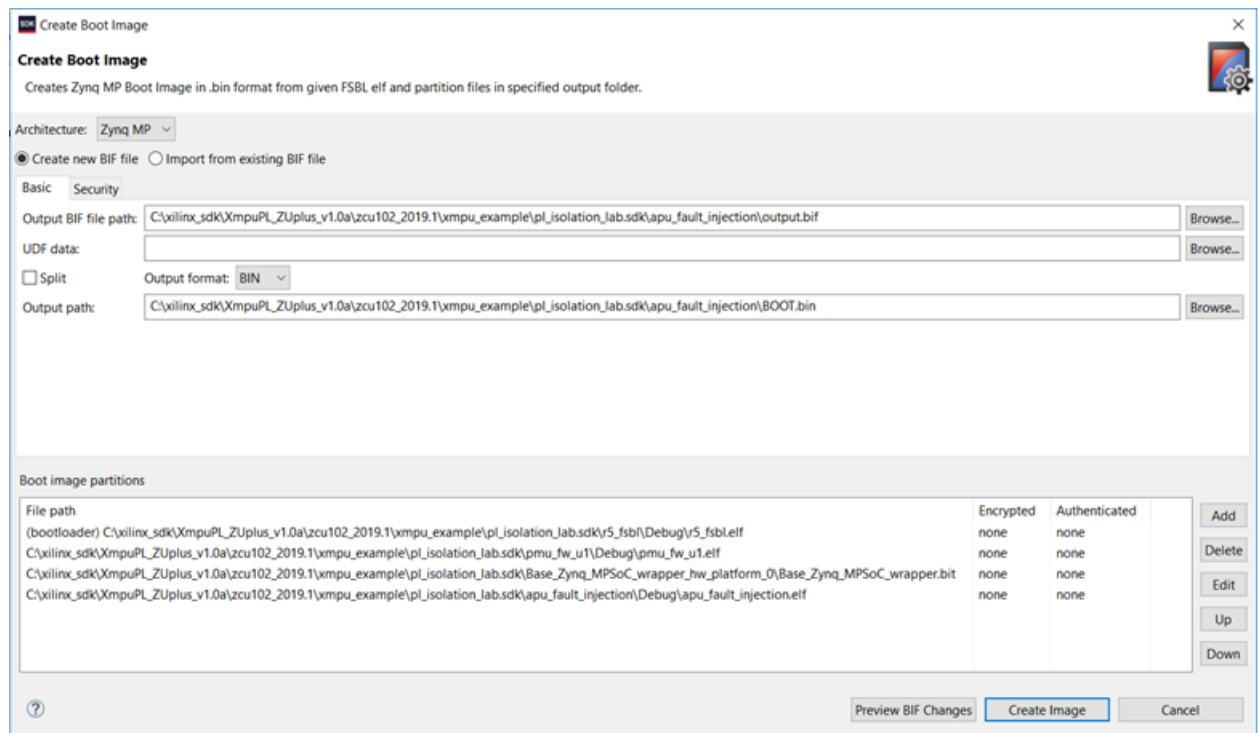
1. Select **Xilinx> Create Boot Image**
 - a. Architecture: Zynq MP
 - b. Check **create new BIF file**
 - c. Output BIF file path: <build_path>/apu_fault_injection/output.bif
 - d. Output path: <build_path>/apu_fault_injection/BOOT.bin
 - e. Continue without clicking create image
2. Click **Add**
 - a. File path: <build_path>/r5_fsbl/Debug/r5_fsbl.elf
 - b. Partition type: bootloader
 - c. Destination device: PS
 - d. Destination CPU: R5 Single
 - e. Click **OK**
3. Click **Add**
 - a. File path: <build_path>/pmu_fw_u1/Debug/pmu_fw_u1.elf
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: PMU
 - e. Click **OK**
4. Click **Add**
 - a. File path:

```
<build_path>/Base_Zynq_MPSoC_wrapper_hw_platform_0/  
Base_Zynq_MPSoC_wrapper.bit
```
 - b. Partition type: datafile
 - c. Destination device: PL
 - d. Click **OK**
5. Click **Add**
 - a. File path:

```
<build_path>/apu_fault_injection/Debug/apu_fault_injection.elf
```

- b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: A53 0
 - e. Exception Level: Enable trust zone
 - f. Click **OK**
6. The create boot image window looks like the following figure:
 7. Click **create image**. If prompted, select **overwrite**.

Figure 26: APU Fault Injection Boot Image



RPU Fault Injection Application in SDK

The following steps show how to create the RPU fault injection application:

1. Select **File > New > Application Project**.
 - a. Project Name: `rpu_fault_injection`
 - b. OS Platform: `<default>standalone`
 - c. Hardware Platform: `<default>`
 - d. Processor: `psu_cortexr5_0`
 - e. Language: `<default>C`
 - f. Compiler: `<default>32-bit`

- g. Hypervisor Guest: <default>No
- h. Board Support Package: <default>Create new rpu_fault_injection_bsp
2. Click **Next**.
 - a. Available Templates: Empty Application
3. Click **Finish**.
4. Click to expand **rpu_fault_injection** in the project explorer.
5. Right-click **src** and select **Import...**
6. Select **General > File System** and click **Next**.
7. Browse and navigate to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/  
zcu102_example/sources/src/rpu_fault_injection
```
8. Click **Select Folder**.
9. Click **Select All**.
10. Click **Overwrite existing sources**.
11. Click **Finish**.
12. Select **Project>Clean** to ensure the applications correctly compiled the imported source.
13. Select **Clean all projects** and click **OK**.
 - a. This rebuilds all the BSPs and re-compiles all the applications so it may take a few minutes to complete the process.

Create the RPU Fault Injection Boot Image

Note: For the following steps:

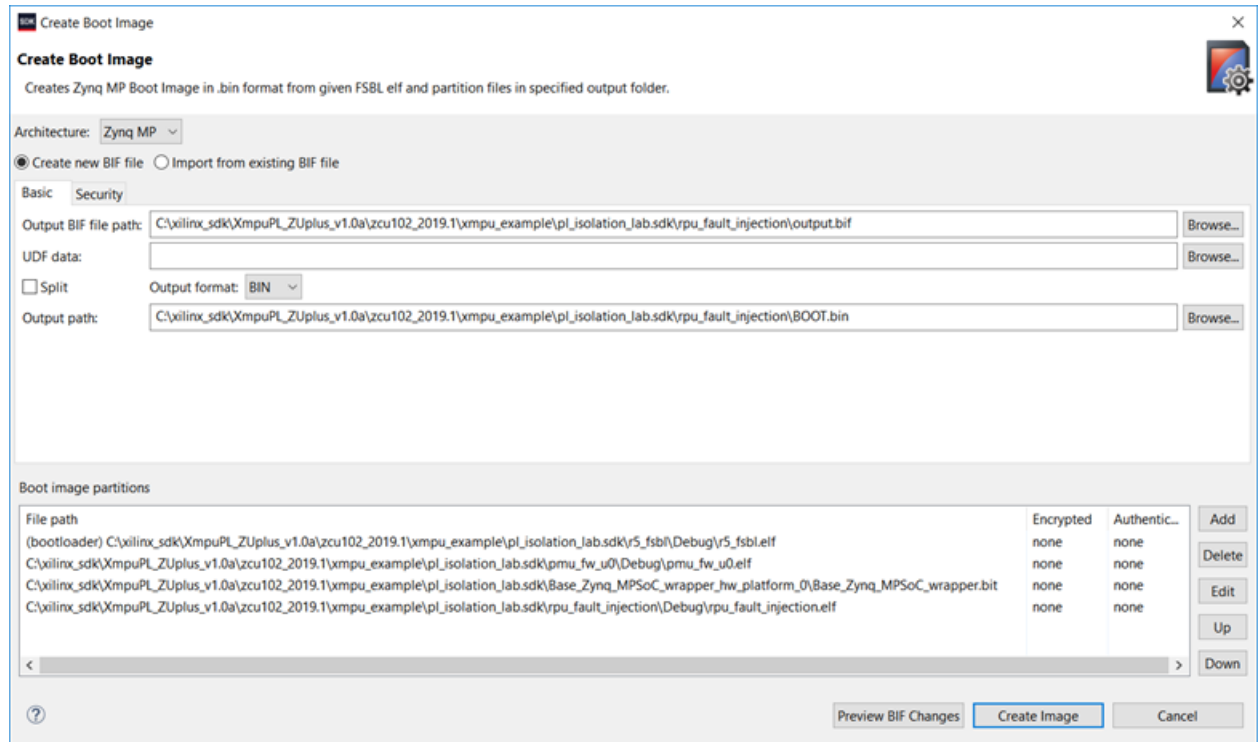
```
<build_path>=<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.1/xmpu_example/  
pl_isolation_lab.sdk
```

To create the boot image:

1. Select **Xilinx> Create Boot Image**
 - a. Architecture: Zynq MP
 - b. Check create new BIF file
 - c. Output BIF file path: <build_path>/rpu_fault_injection/output.bif
 - d. Output path: <build_path>/rpu_fault_injection/BOOT.bin
 - e. Continue without clicking create image
2. Click **Add**
 - a. File path: <build_path>/r5_fsbl/Debug/r5_fsbl.elf
 - b. Partition type: bootloader

- c. Destination device: PS
- d. Destination CPU: R5 Single
- e. Click **OK**
3. Click **Add**
 - a. File path: `<build_path>/pmu_fw_u0/Debug/pmu_fw_u0.elf`
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: PMU
 - e. Click **OK**
4. Click **Add**
 - a. File path: `<build_path>/Base_Zynq_MPSoC_wrapper_hw_platform_0/Base_Zynq_MPSoC_wrapper.bit`
 - b. Partition type: datafile
 - c. Destination device: PL
 - d. Click **OK**
5. Click **Add**
 - a. File path: `<build_path>/apu_fault_injection/Debug/apu_fault_injection.elf`
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: R5 0
 - e. Exception Level: Enable trust zone
 - f. Click **OK**
6. The create boot image window looks like the following figure
7. Click **create image**. If prompted, select **overwrite**.

Figure 27: RPU Fault Injection Boot Image



For further details on how to run the test applications, refer to [Running the Isolation Example on the ZCU102 Board](#).

Creating the Isolation Test SW Applications in Vitis 2019.2

The Build HW Design in Vivado must export the XSA hardware file to:

```
<your_path>XmpuPL_ZUplus_v1.0a/zcu102_2019.2/mpu_example/  
pl_isolation_lab.vitis/Base_Zynq_MPSoC_wrapper_hw_platform/  
Base_Zynq_MPSoC_wrapper.xsa
```

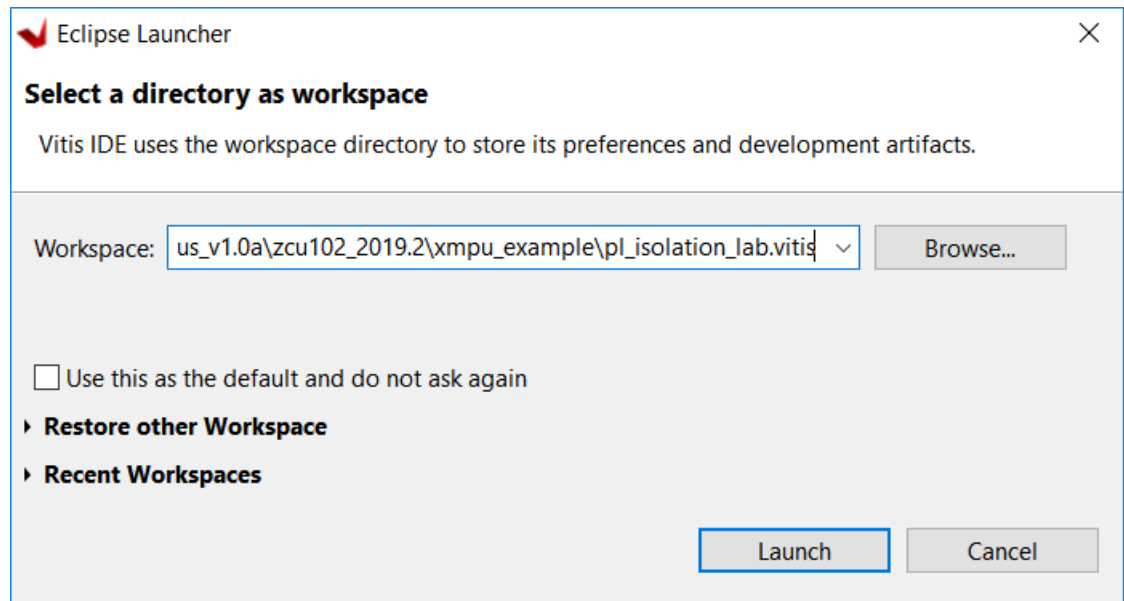
1. Select the workspace in Eclipse Launcher.

a. Workspace:

```
<your_path>\XmpuPL_ZUplus_v1.0a\zcu102_2019.2\mpu_example  
\pl_isolation_lab.vitis
```

b. Click **Launch**

Figure 28: Eclipse Launcher



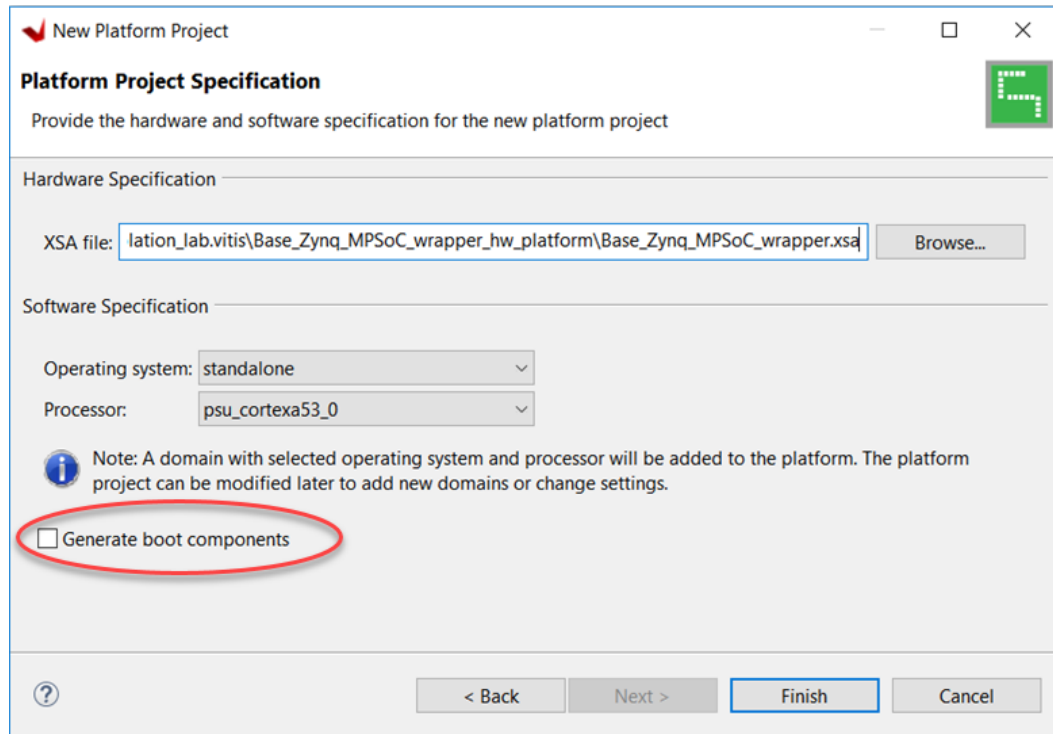
Build the Isolation Test Platform

Create the platform for the isolation test:

1. Select **Create Platform Project**
 - a. Project name: zcu102_isolation_test
 - b. Check Use default location
 - c. Click **Next**
2. Select **Create from hardware specification (XSA)**
 - a. Click **Next**
3. Select XSA file
 - a. Browse to:

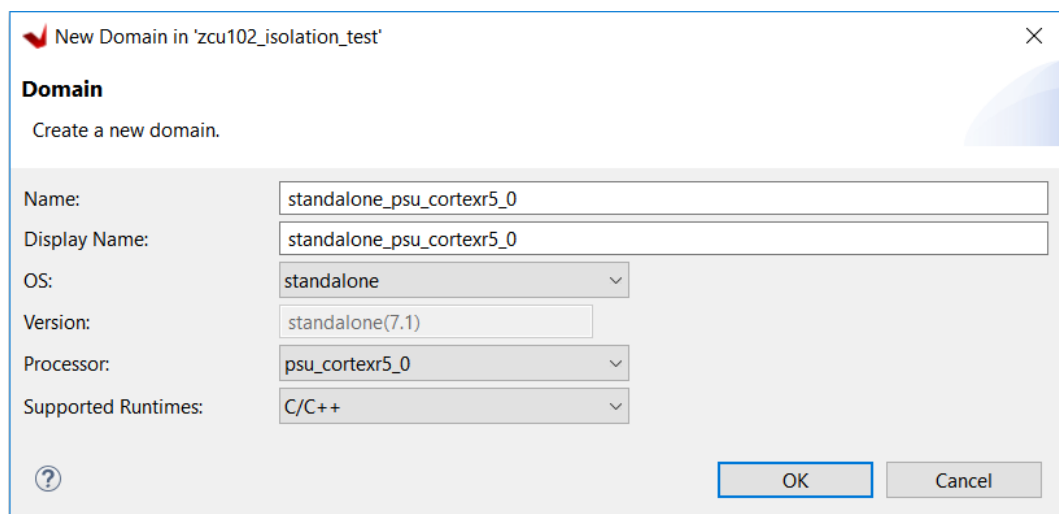
/Base_Zynq_MPSoC_wrapper_hw_platform/Base_Zynq_MPSoC_wrapper.xsa/
 - b. Click **Open**
 - c. Operating system: standalone
 - d. Processor: psu_cortexa53_0
 - e. Do not select **Generate boot components**
 - Uncheck if it is selected.
 - f. Click **Finish**

Figure 29: Platform Project Specification



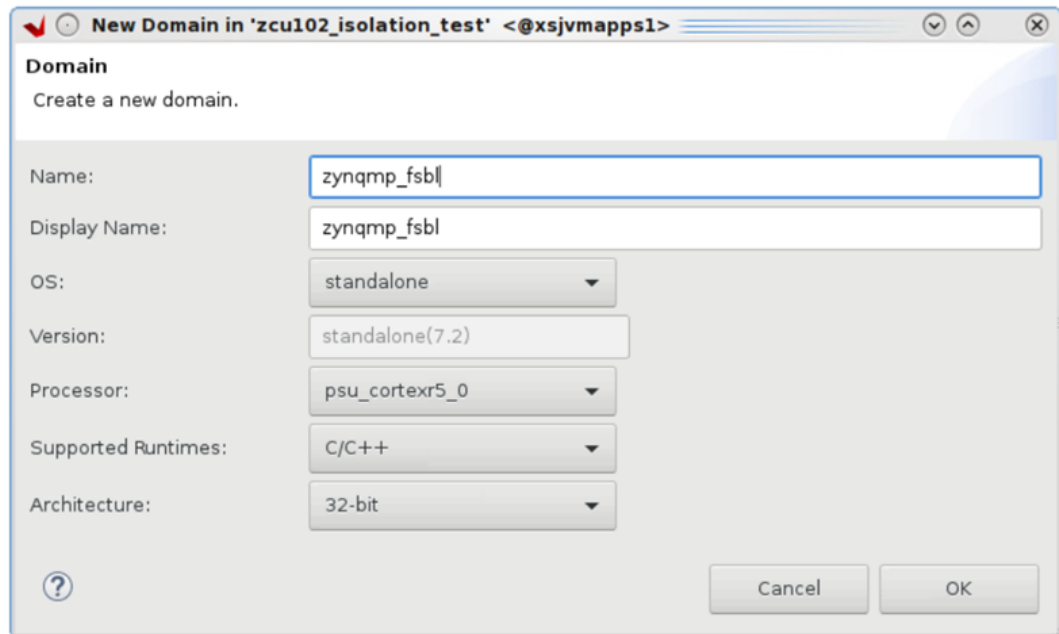
4. In the newly created `zcu102_isolation_test` tab, right-click `psu_cortexa53_0`, and select **Add Domain**:
 - a. Name: `standalone_psu_cortexr5_0`
 - b. Display name: `standalone_psu_cortexr5_0`
 - c. OS: standalone
 - d. Processor: `psu_cortexr5_0`
 - e. Click **OK**

Figure 30: Add Domain R5_0 Standalone



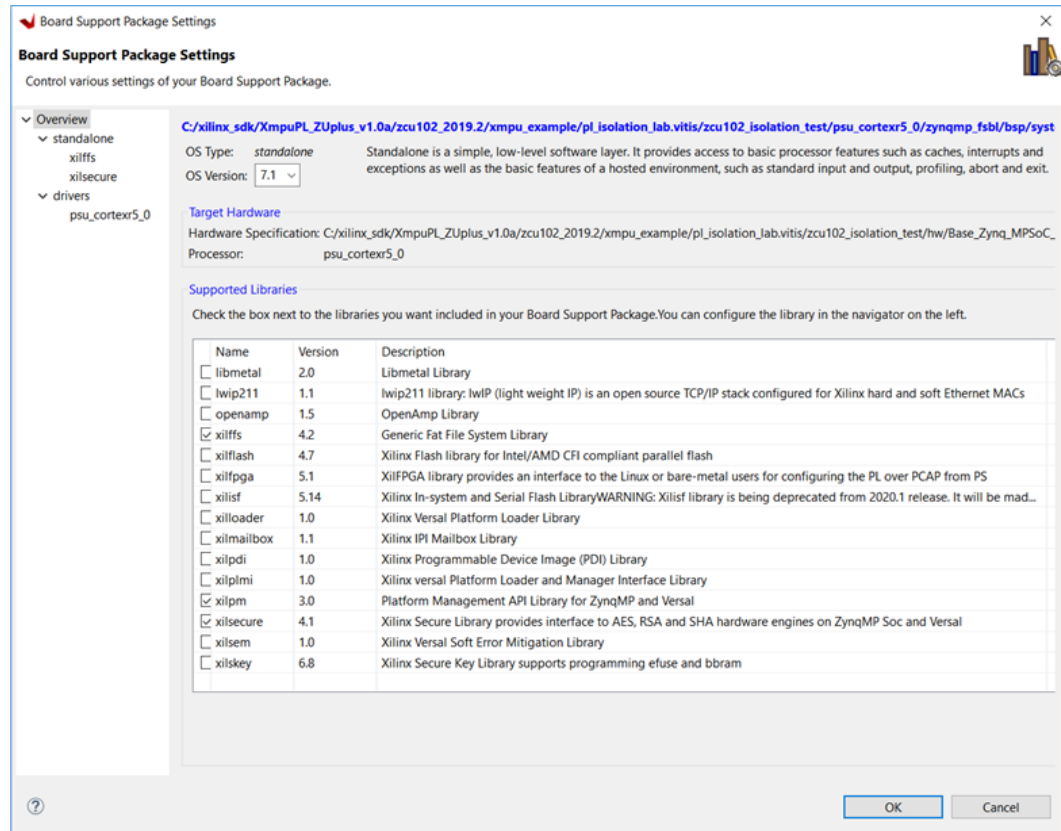
5. In the `zcu102_isolation_test` tab, right-click `psu_cortexr5_0`, and select **Add Domain**:
 - a. Name: `zynqmp_fsbl`
 - b. Display name: `zynqmp_fsbl`
 - c. OS: standalone
 - d. Processor: `psu_cortexr5_0`
 - e. Click **OK**

Figure 31: Add Domain FSBL



6. In the `zcu102_isolation_test` tab, select `psu_cortexr5_0` > `zynqmp_fsbl` > **Board Support Package**
 - a. Click **Modify BSP Settings**
 - b. Select **Overview**
 - c. Check Libraries
 - i. `xilffs` Generic Fat File System Library
 - ii. `xilpm` Platform Management API Library for ZynqMP and Versal
 - iii. `xilsecure` Xilinx Secure Library
 - d. Select **Overview** > **standalone**
 - i. Set `zynqmp_fsbl_bsp` true
 - e. Click **OK**

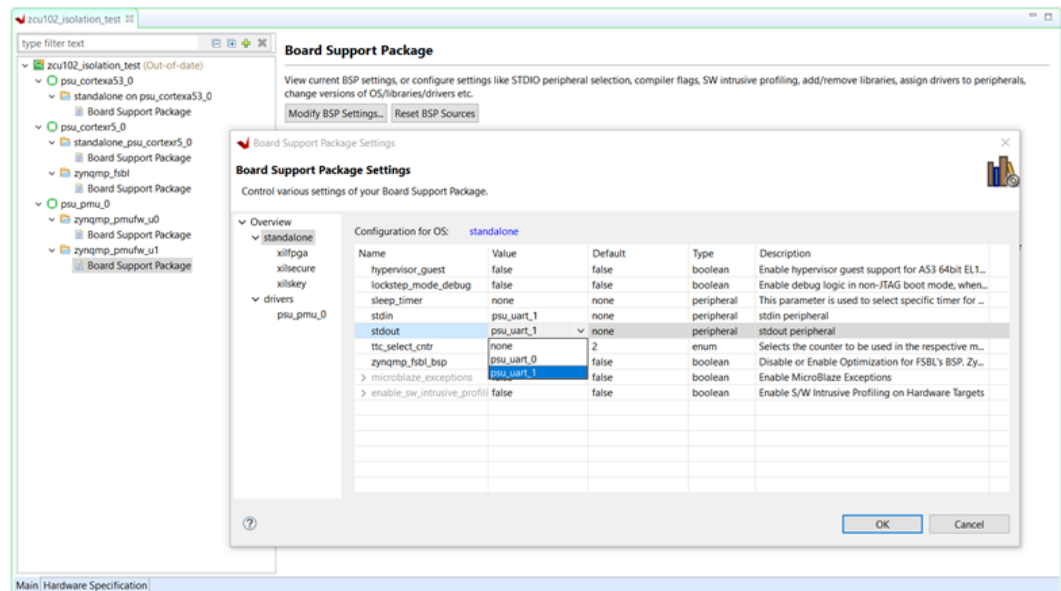
Figure 32: FSBL Board Support Package Settings



7. In the zcu102_isolation_test tab, right-click psu_cortexr5_0, and select **Add Domain**:
 - a. Name: zynqmp_pmufw_u0
 - b. Display name: zynqmp_pmufw_u0
 - c. OS: standalone
 - d. Processor: psu_pmu_0
 - e. Click **OK**
8. In the zcu102_isolation_test tab, right-click psu_pmu_0, and select **Add Domain**:
 - a. Name: zynqmp_pmufw_u1
 - b. Display name: zynqmp_pmufw_u1
 - c. OS: standalone
 - d. Processor: psu_pmu_0
 - e. Click **OK**
9. In the zcu102_isolation_test tab, select psu_pmu_0 > zynqmp_pmufw_u1 > **Board Support Package**
 - a. Click **Modify BSP Settings**
 - b. Select **Overview > standalone**
 - c. Change **stdin** and **stdout** to : **psu_uart_1**

- d. Processor: psu_pmu_0
- e. Click OK

Figure 33: zynqmp_pmufw_u1 Board Support Package Settings



10. Right-click `zcu102_isolation_test` and select **Build Project**

APU Isolation Test System

The APU isolation test system is a container of the necessary applications to run the APU fault injection application to test the isolated system.

1. Select **File>New>Application Project**
 - a. Check **Use default location**
 - b. Project name: `apu_fault_injection`
 - c. System project: **Create New > apu_fault_injection_system**
 - d. Click **Next**
 - e. Select **platform from repository: zcu102_isolation_test**
 - f. Click **Next**
 - g. Domain: `standalone psu_cortexa53_0`
 - h. Click **Next**
 - i. Select **Empty Application**
 - j. Click **Finish**
2. Right-click `apu_fault_injection_system > apu_fault_injection > src` and select **Import sources**

- a. Browse and navigate to:


```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/zcu102_example/sources/src/apu_fault_injection
```
 - b. Click **Select Folder**.
 - c. Click **Select All**
 - d. Click **Overwrite existing sources**
 - e. Click **Finish**
3. Right-click **apu_fault_injection_system** and select **Add.>Application Project**
 - a. Check **Use default location**
 - b. Project name: r5_fsbl
 - c. System project: Select **apu_fault_injection_system** from a drop-down menu
 - d. Click **Next**
 - e. Select **zynqmp_fsbl**
 - f. Click **Next**
 - g. Select **Zynq MP FSBL**
 - h. Click **Finish**
 4. Right-click **apu_fault_injection_system** and select **Add.>Application Project**
 - a. Check **Use default location**
 - b. Project name: pmu_fw_u1
 - c. System project: Select **apu_fault_injection_system** from a drop-down menu
 - d. Click **Next**
 - e. Select **zynqmp_pmufw_u1**
 - f. Click **Next**
 - g. Select **Zynq MP PMU firmware**
 - h. Click **Finish**
 5. Right-click **apu_fault_injection_system > pmufw_u1 > src** and select **Import sources**
 - a. Browse and navigate to:


```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/zcu102_example/sources/src/pmu_fw_2019.2
```
 - b. Click **Select Folder**.
 - c. Click **Select All**
 - d. Click **Overwrite existing sources**
 - e. Click **Finish**
 6. Click **apu_fault_injection_system** and select **Project>Build Project**

When completed if there is an error: platform file not found, ignore it. This is because default boot components were not used. You will be creating a boot image in the following steps.

Create the APU Fault Injection Boot Image

To create the boot image

For the following steps:

```
build_path>=<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.2/xmpu_example/  
pl_isolation_lab.vitis
```

1. Select **Xilinx**> **Create Boot Image**

- a. Architecture: Zynq MP
- b. Check **Create new BIF file**
- c. Output BIF file path: <build_path>/apu_fault_injection/output.bif
- d. Output path: <build_path>/apu_fault_injection/BOOT.bin
- e. Continue without clicking create image

Note: If the boot image partitions are automatically filled, select each one and delete them so that the next steps are performed from scratch.

2. Click **Add**

- a. File path: <build_path>/r5_fsbl/Debug/r5_fsbl.elf
- b. Partition type: bootloader
- c. Destination device: PS
- d. Destination CPU: R5 0
- e. Click **OK**

3. Click **Add**

- a. File path: <build_path>/pmu_fw_u1/Debug/pmu_fw_u1.elf
- b. Partition type: datafile
- c. Destination device: PS
- d. Destination CPU: PMU
- e. Click **OK**

4. Click **Add**

- a. File path: <build_path>Base_Zynq_MPSoC_wrapper_hw_platform/
BaseZynq_MPSoC_hw_platform/Base_Zynq_MPSoC_wrapper.bit
- b. Partition type: datafile
- c. Destination device: PL
- d. Click **OK**

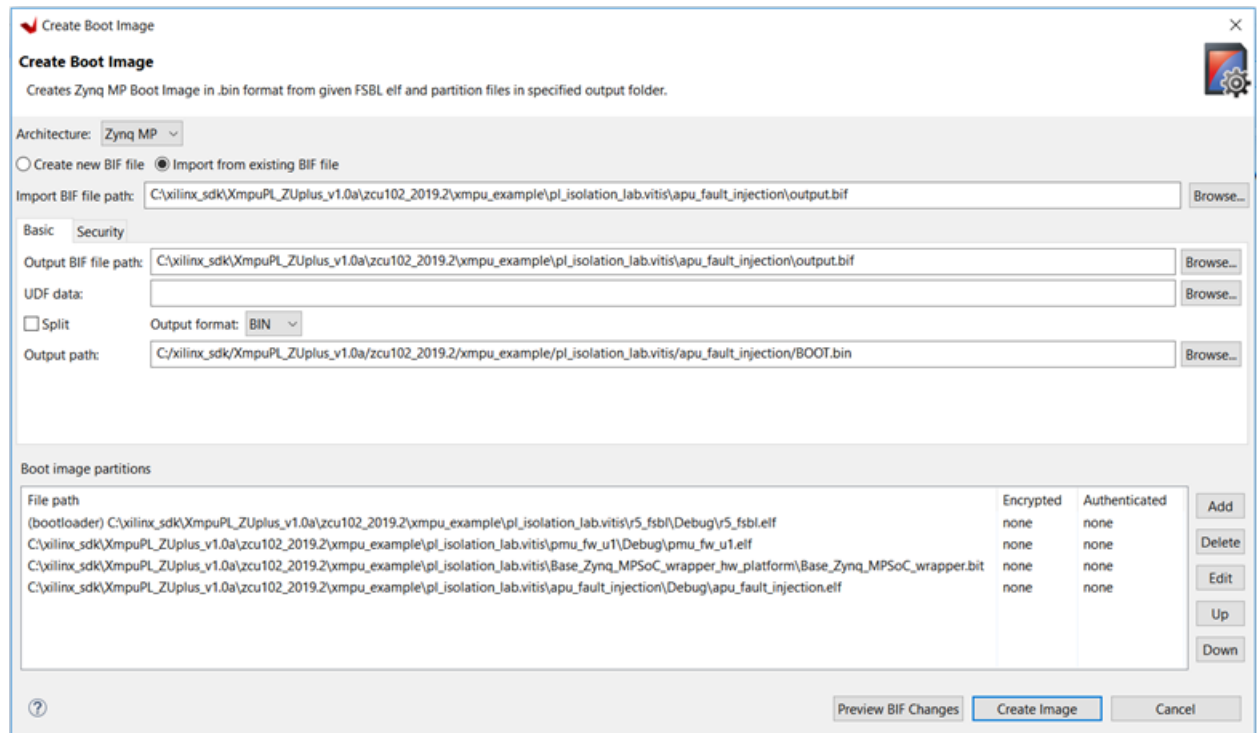
5. Click **Add**
 - a. File path: <build_path>/apu_fault_injection/Debug/apu_fault_injection_elf
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: A53 0
 - e. Click **OK**



CAUTION! The destination CPU defaults to A53_0; but if you do not actually select it from the drop-down, then the parameter may not get written to the BIF file. Use **Preview BIF changes** to verify. If using Linux, try changing the destination CPU and then changing it back to A53 0.

6. The Create Boot Image window looks like the following figure.
7. Click **Create Image** and select **Overwrite** if prompted.

Figure 34: Create Boot Image for APU Fault Injection System



Note: This example is not using secure boot and all applications are standalone OS, hence the exception level and TrustZone settings for BootGen does not matter.

RPU Isolation Test System

The RPU isolation test system is a container of the necessary applications to run the rpu fault injection application to test the isolated system.

1. Select **File>New>Application Project**

- a. Check **Use default location**
 - b. Project name: `rpu_fault_injection`
 - c. System project: **Create New > rpu_fault_injection_system**
 - d. Click **Next**
 - e. Select **platform from repository: zcu102_isolation_test**
 - f. Click **Next**
 - g. Domain: `standalone_psu_cortexr5_0`
 - h. Click **Next**
 - i. Select **Empty Application**
 - j. Click **Finish**
2. Right-click `rpu_fault_injection_system > rpu_fault_injection > src` and select **Import sources**
 - a. Browse and navigate to:


```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/zcu102_example/sources/src/rpu_fault_injection
```
 - b. Click **Select Folder**
 - c. Click **Select All**
 - d. Click **Overwrite existing sources**
 - e. Click **Finish**
 3. Right-click `rpu_fault_injection_system` and select **Add > Application Project**
 - a. Check **Use default location**
 - b. Project name: `pmu_fw_u0`
 - c. System project: Select `rpu_fault_injection_system` from a drop-down menu
 - d. Click **Next**
 - e. Select `zynqmp_pmufw_u0`
 - f. Click **Next**
 - g. Select **Zynq MP PMU firmware**
 - h. Click **Finish**
 4. Right-click `rpu_fault_injection_system > pmufw_u0 > src` and select **Import sources**
 - a. Browse and navigate:


```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/zcu102_example/sources/src/pmu_fw_2019.2
```
 - b. Click **Select Folder**
 - c. Click **Select All**
 - d. Click **Overwrite existing sources**

- e. Click **Finish**
5. Click **rpu_fault_injection_system** and select **Project>Build Project**

Note: When completed if there is an error: platform file not found, ignore it. This is because default boot components were not used. You will be creating a boot image in the following steps.

Create the RPU Fault Injection Boot Image

To create the boot image

For the following steps:

```
build_path> = <your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.2/xmpu_example/  
pl_isolation_lab.Vitis
```

1. Select Xilinx> **Create Boot Image**
 - a. Architecture: Zynq MP
 - b. Check **Create new BIF file**
 - c. Output BIF file path: <build_path>/rpu_fault_injection/output.bif
 - d. Output path: <build_path>/rpu_fault_injection/BOOT.bin
 - e. Continue without clicking create image

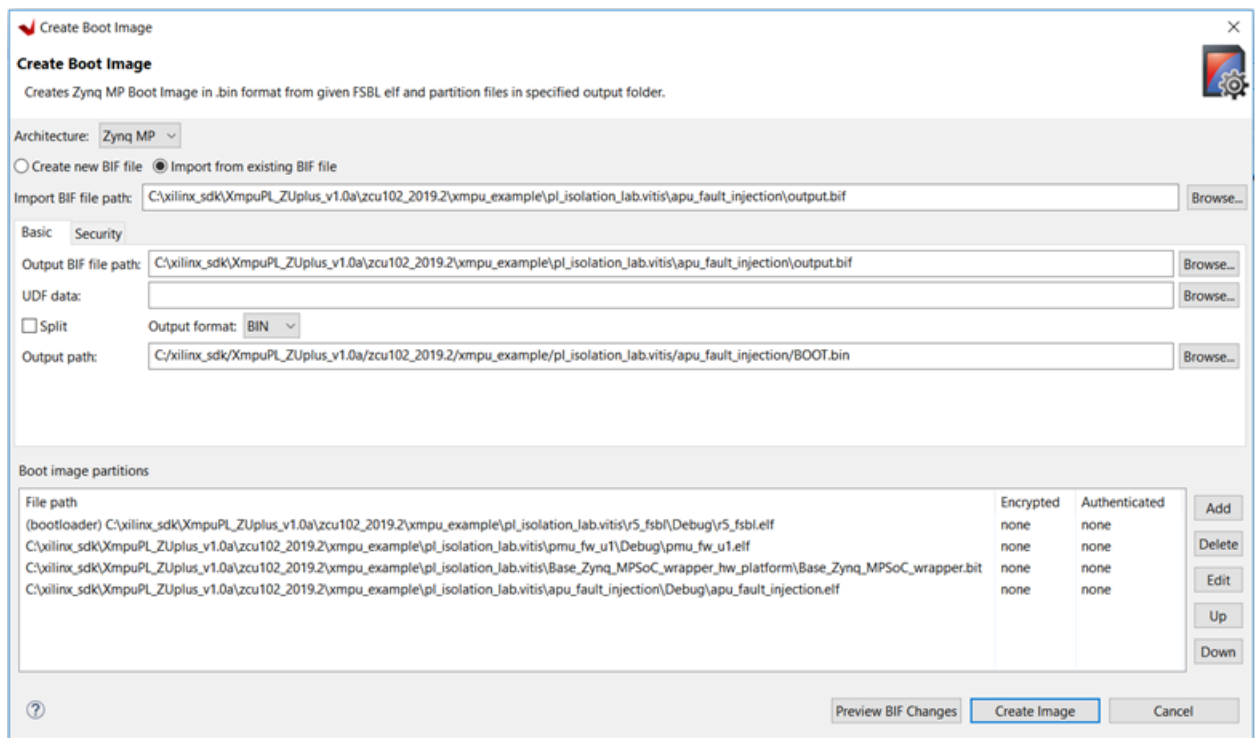
Note: If the boot image partitions are automatically filled, select each one and delete them so that the next steps are performed from scratch.

Note: The RPU isolation test system uses the same r5_fsb that was created in the APU isolation test system.

2. Click **Add**
 - a. File path: <build_path>/r5_fsb1/Debug/r5_fsb1.elf
 - b. Partition type: bootloader
 - c. Destination device: PS
 - d. Destination CPU: R5 0
 - e. Click **OK**
3. Click **Add**
 - a. File path: <build_path>/pmu_fw_u0/Debug/pmu_fw_u0.elf
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: PMU
 - e. Click **OK**
4. Click **Add**

- a. File path: <build_path> Base_Zynq_MPSoC_wrapper_hw_platform/
BaseZynq_MPSoC_hw_platform/Base_Zynq_MPSoC_wrapper.bit
 - b. Partition type: datafile
 - c. Destination device: PL
 - d. Click **OK**
5. Click **Add**
- a. File path: <build_path>/apu_fault_injection/Debug/
rpu_fault_injection_elf
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: R5 0
 - e. Click **OK**
6. The Create Boot Image window looks like the following figure
7. Click **Create Image** and select **Overwrite** if prompted.

Figure 35: Create Boot Image for RPU Fault Injection System



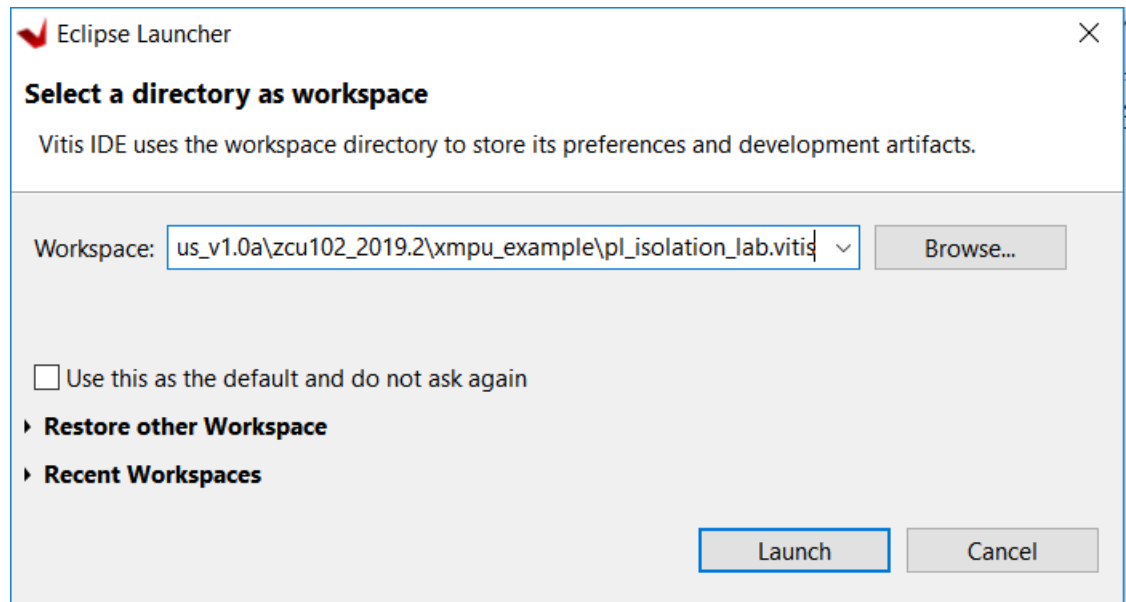
Creating the Isolation Test SW Applications in Vitis 2020.1

The Build HW Design in Vivado section should have exported the XSA hardware file to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2020.1/xmpu_example/  
pl_isolation_lab.vitis/Base_Zynq_MPSoC_wrapper_hw_platform/  
Base_Zynq_MPSoC_wrapper.xsa
```

1. If the pl_isolation_lab project is open in Vivado 2020.1, run the following steps:
2. Select Tools>Launch Vitis IDE
3. Select the workspace in Eclipse Launcher
 - a. Workspace: <your_path>\XmpuPL_ZUplus_v1.0a\zcu102_2020.1\xmpu_example\pl_isolation_lab.Vitis
 - b. Click **Launch**

Figure 36: Eclipse Launcher



Build the Isolation Test Platform

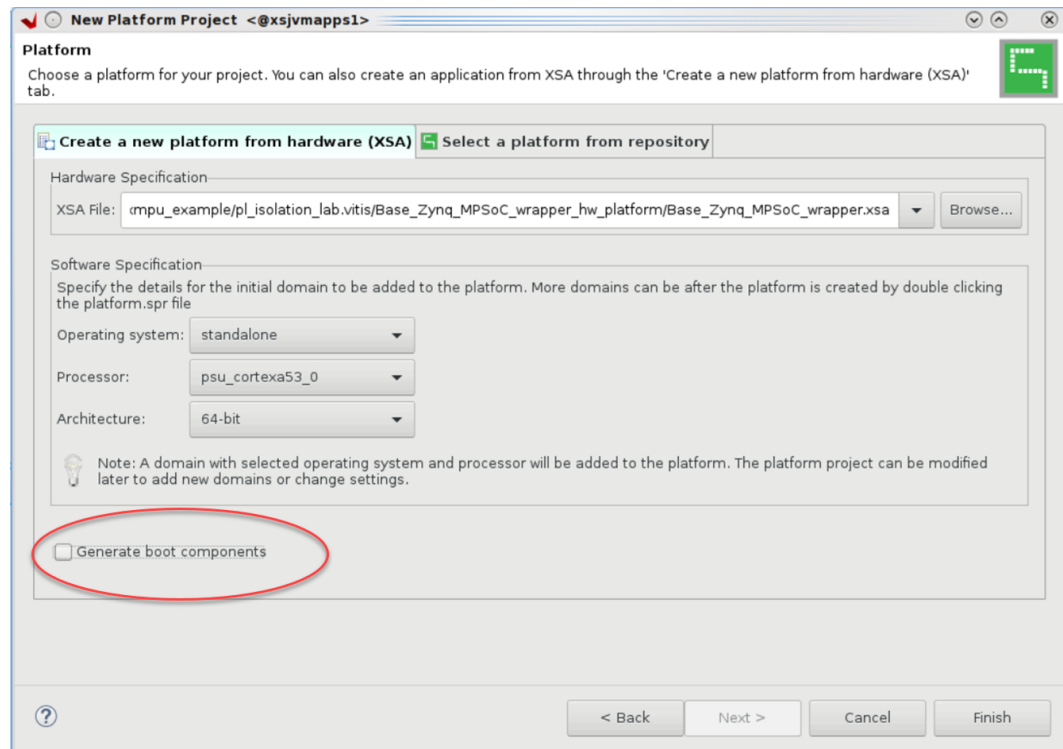
Create the platform for the isolation test:

1. Select **Create Platform Project**
 - a. Project name: zcu102_isolation_test
 - b. Check Use default location
 - c. Click **Next**
2. Select **Create from hardware specification (XSA)**
 - a. Click **Next**
3. Select the XSA file
 - a. Browse to:


```
<Workspace>/Base_Zynq_MPSoC_wrapper_hw_platform/  
Base_Zynq_MPSoC_wrapper.xsa
```
 - b. Click **Open**
 - c. Operating system: standalone

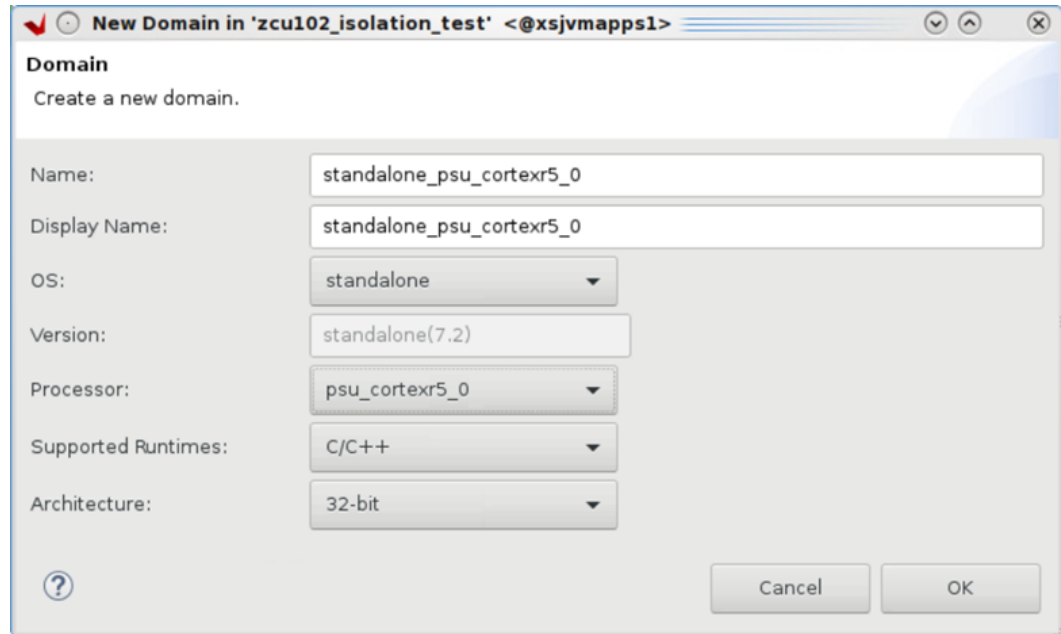
- d. Processor: psu_cortexa53_0
- e. Architecture: 64-bit
- f. Do not select **Generate boot components**
 - Uncheck if it is selected.
- g. Click **Finish**

Figure 37: Platform Project Specification



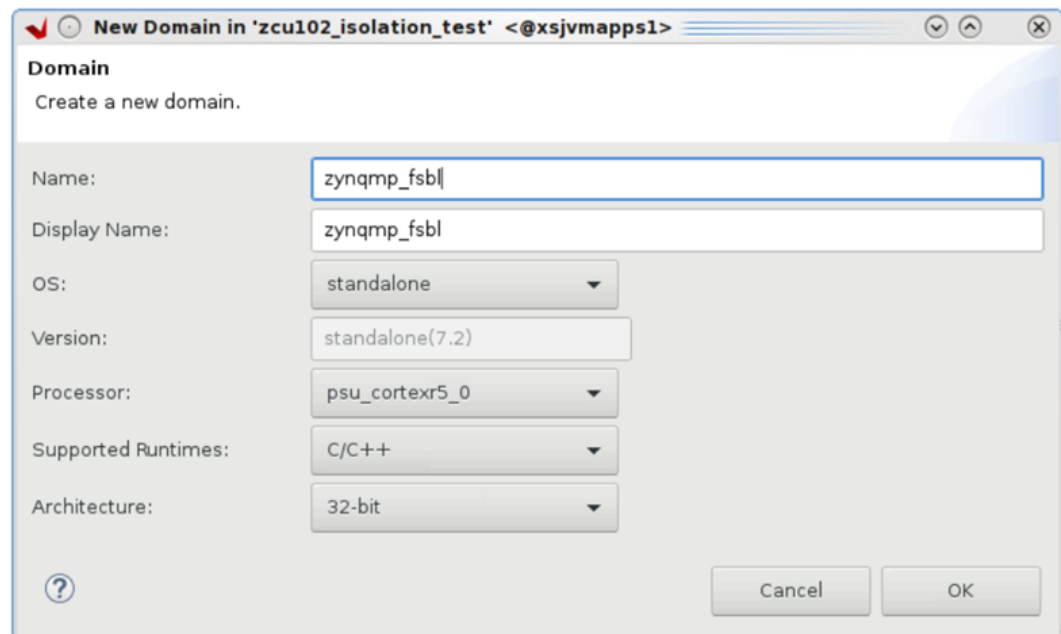
4. In the newly created zcu102_isolation_test tab, right-click psu_cortexa53_0, and select **Add Domain**:
 - a. Name: standalone_psu_cortexr5_0
 - b. Display name: standalone_psu_cortexr5_0
 - c. OS: standalone
 - d. Processor: psu_cortexr5_0
 - e. Click **OK**

Figure 38: Add Domain R5_0 Standalone



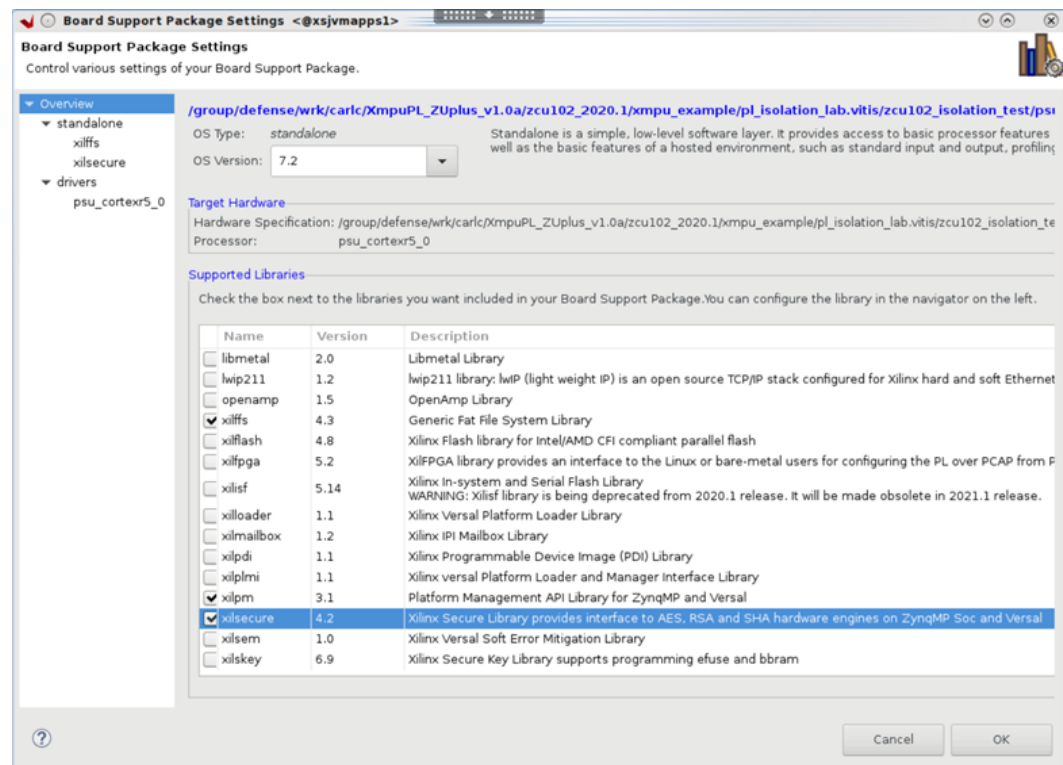
5. In the zcu102_isolation_test tab, right-click psu_cortexr5_0, and select **Add Domain**:
 - a. Name: zynqmp_fsbl
 - b. Display name: zynqmp_fsbl
 - c. OS: standalone
 - d. Processor: psu_cortexr5
 - e. Click **OK**

Figure 39: Add Domain FSBL



6. In the `zcu102_isolation_test` tab, select `psu_cortexr5_0` > `zynqmp_fsbl` > **Board Support Package**
 - a. Click **Modify BSP Settings**
 - b. Select **Overview**
 - c. Check Libraries
 - i. `xilffs` Generic Fat File System Library
 - ii. `xilpm` Platform Management API Library for ZynqMP and Versal
 - iii. `xilsecure` Xilinx Secure Library
 - d. Select **Overview** > **standalone**
 - i. Set `zynqmp_fsbl_bsp` true
 - e. Click **OK**

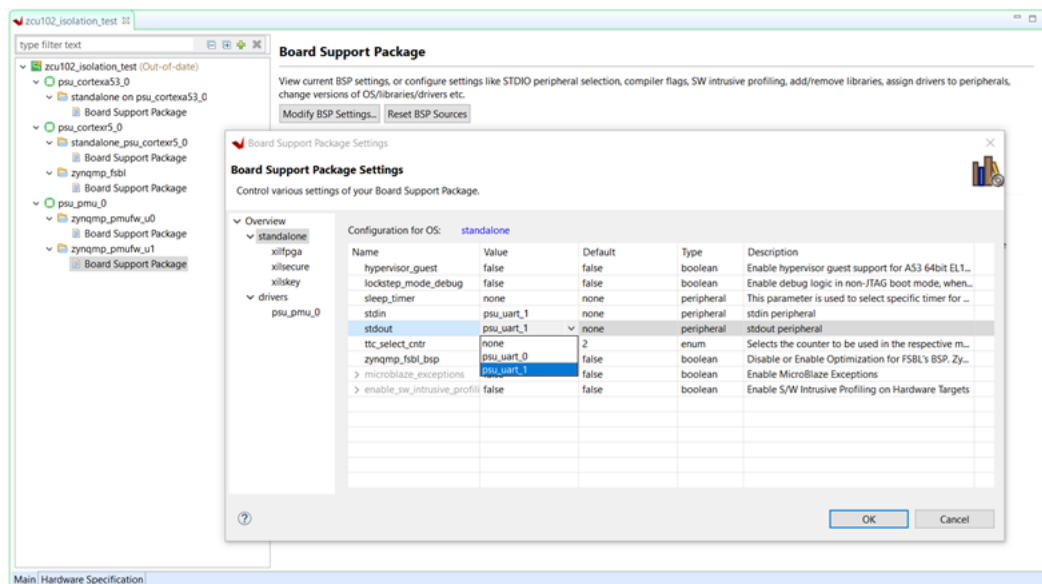
Figure 40: FSBL Board Support Package Settings



7. In the `zcu102_isolation_test` tab, right-click `psu_cortexr5`, and select **Add Domain**:
 - a. Name: `zynqmp_pmufw_u0`
 - b. Display name: `zynqmp_pmufw_u0`
 - c. OS: standalone
 - d. Processor: `psu_pmu_0`
 - e. Click **OK**
8. In the `zcu102_isolation_test` tab, right-click `psu_pmu_0`, and select **Add Domain**:

- a. Name: zynqmp_pmufw_u1
 - b. Display name: zynqmp_pmufw_u1
 - c. OS: standalone
 - d. Processor: psu_pmu_0
 - e. Click **OK**
9. In the zcu102_isolation_test tab, select **psu_pmu_0 > zynqmp_pmufw_u1 > Board Support Package**:
- a. Click **Modify BSP Settings**
 - b. Select **Overview > standalone**
 - c. Change **stdin** and **stdout** to : **psu_uart_1**
 - d. Processor: psu_pmu_0
 - e. Click **OK**

Figure 41: zynqmp_pmufw_u1 Board Support Package Settings



10. Right-click zcu102_isolation_test and select **Build Project**.

APU Isolation Test System

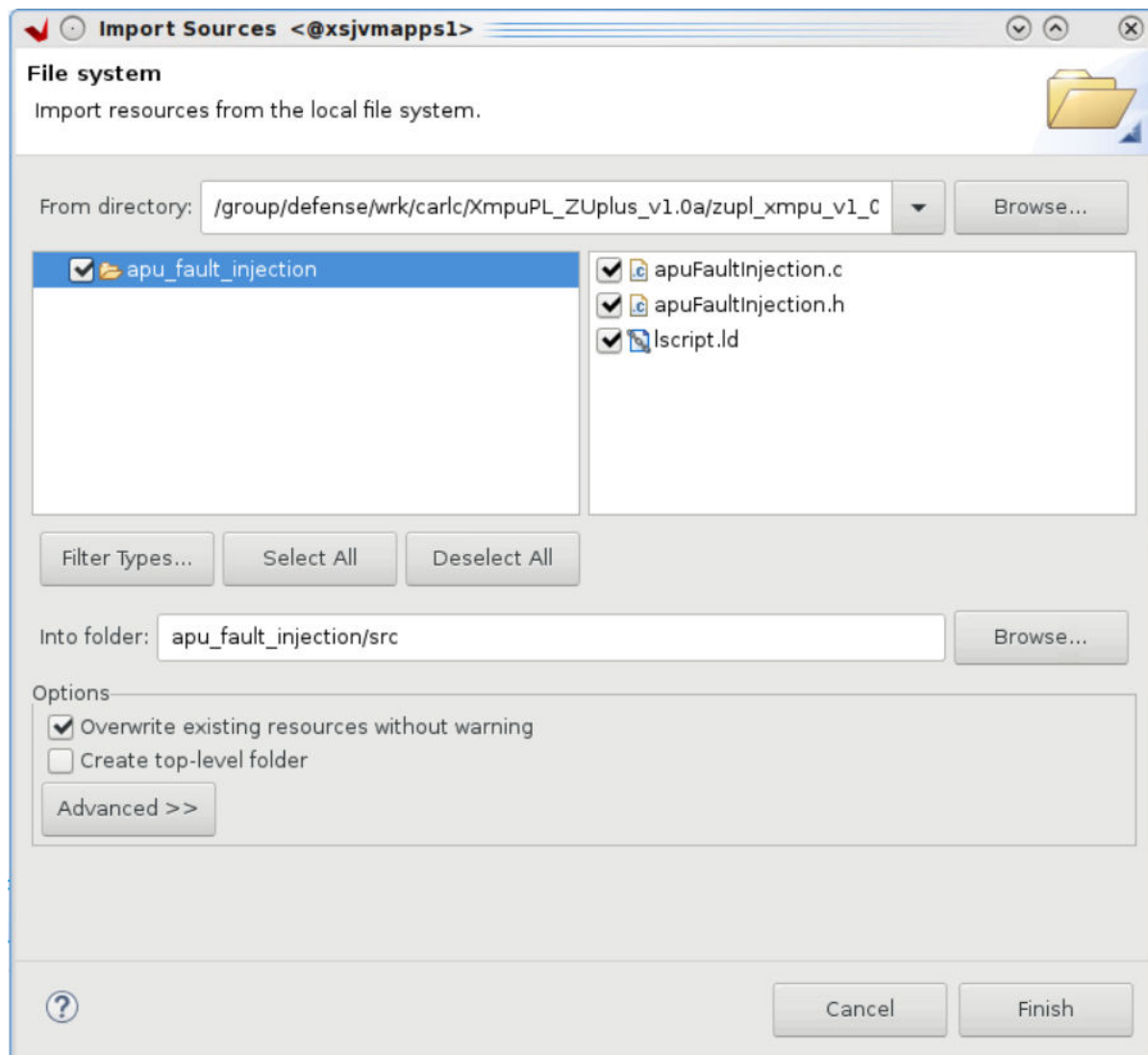
The APU isolation test system is a container of the necessary applications to run the APU fault injection application to test the isolated system.

1. Select **File>New>Application**
 - a. Click **Next** if the welcome page opens
 - b. Project name: apu_fault_injection
 - c. System project: Create New apu_fault_injection_system

- d. Click **Next**
 - e. Select **platform from repository: zcu102_isolation_test**
 - f. Click **Next**
 - g. Domain: standalone psu_cortexa53_0
 - h. Click **Next**
 - i. Select **Empty Application**
 - j. Click **Finish**
2. Right-click **apu_fault_injection_system > apu_fault_injection > src** and select **Import sources...**
- a. Browse and navigate to:

```
<your_path>\XmpuPL_ZUplus_v1.0a\zcu102_2019.2\xmpu_example  
\pl_isolation_lab.vitis
```
 - b. Click **Select Folder.**
 - c. Click **Select All**
 - d. Click **Overwrite existing sources**
 - e. Click **Finish**

Figure 42: Import Sources



3. Right-click **apu_fault_injection_system** and select **Add Application Project...**
 - a. Application project name: r5_fsbl
 - b. Processor: psu_cortexr5_0
 - c. System project: Select **apu_fault_injection_system** from a drop-down menu
 - d. Click **Next**
 - e. Select **zynqmp_fsbl**
 - f. Click **Next**
 - g. Select **Zynq MP FSBL**
 - h. Click **Finish**
4. Right-click **apu_fault_injection_system** and select **Add Application Project...**
 - a. Project name: pmu_fw_u1
 - b. System project: Select **apu_fault_injection_system** from a drop-down menu

- c. Processor: psu_pmu_0
 - d. Click **Next**
 - e. Domain: zynqmp_pmufw_u1
 - f. Click **Next**
 - g. Select **Zynq MP PMU firmware**
 - h. Click **Finish**
5. Right-click **apu_fault_injection_system > pmufw_u1 > src** and select **Import sources...**
- a. Browse and navigate to:


```
your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/zcu102_example/sources/src/pmu_fw_2020.1
```
 - b. Click **Select Folder**
 - c. Click **Select All**
 - d. Click **Overwrite existing sources...**
 - e. Click **Finish**
6. Click **apu_fault_injection_system** and select **Project>Build Project**
- Note:** When completed if there is an error: platform file not found, ignore it. This is because default boot components were not used. You will be creating a boot image in the following steps.

Create the APU Fault Injection Boot Image

To create the boot image

For the following steps:

```
build_path=<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.2/xmpu_example/pl_isolation_lab.vitis
```

1. Select Xilinx> **Create Boot Image**
 - a. Architecture: Zynq MP
 - b. Check **Create new BIF file**
 - c. Output BIF file path: <build_path>/apu_fault_injection/output.bif
 - d. Output path: <build_path>/apu_fault_injection/BOOT.bin
 - e. Continue without clicking create image

Note: If the boot image partitions are automatically filled, select each one and delete, so that the next steps are performed from scratch.

Note: The RPU isolation test system uses the same r5_fsb that was created in the APU isolation test system.

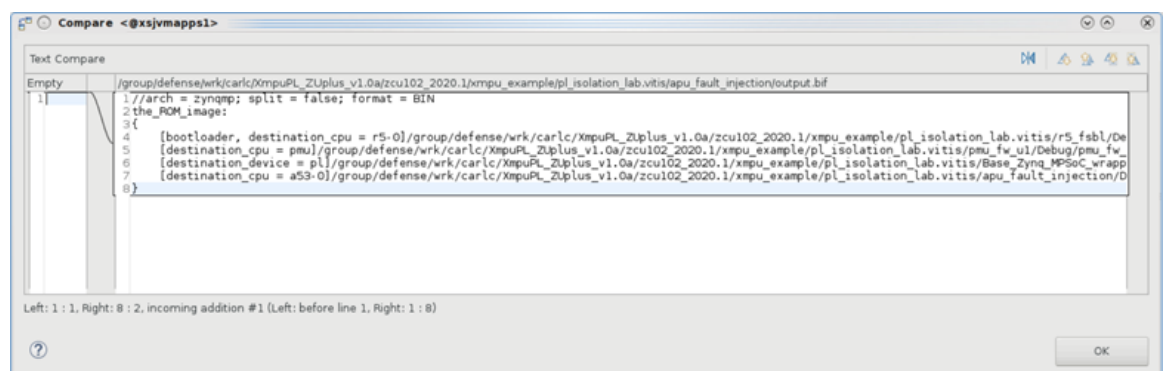
2. Click **Add**

- a. File path: <build_path>/r5_fsbl/Debug/r5_fsbl.elf
 - b. Partition type: bootloader
 - c. Destination device: PS
 - d. Destination CPU: R5 0
 - e. Click **OK**
3. Click **Add**
- a. File path: <build_path>/pmu_fw_u0/Debug/pmu_fw_u1.elf
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: PMU
 - e. Click **OK**
4. Click **Add**
- a. File path: <build_path>Base_Zynq_MPSoC_wrapper_hw_platform/
BaseZynq_MPSoC_hw_platform/Base_Zynq_MPSoC_wrapper.bit
 - b. Partition type: datafile
 - c. Destination device: PL
 - d. Click **OK**
5. Click **Add**
- a. File path: <build_path>/apu_fault_injection/Debug/
rpu_fault_injection_elf
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: A53 0



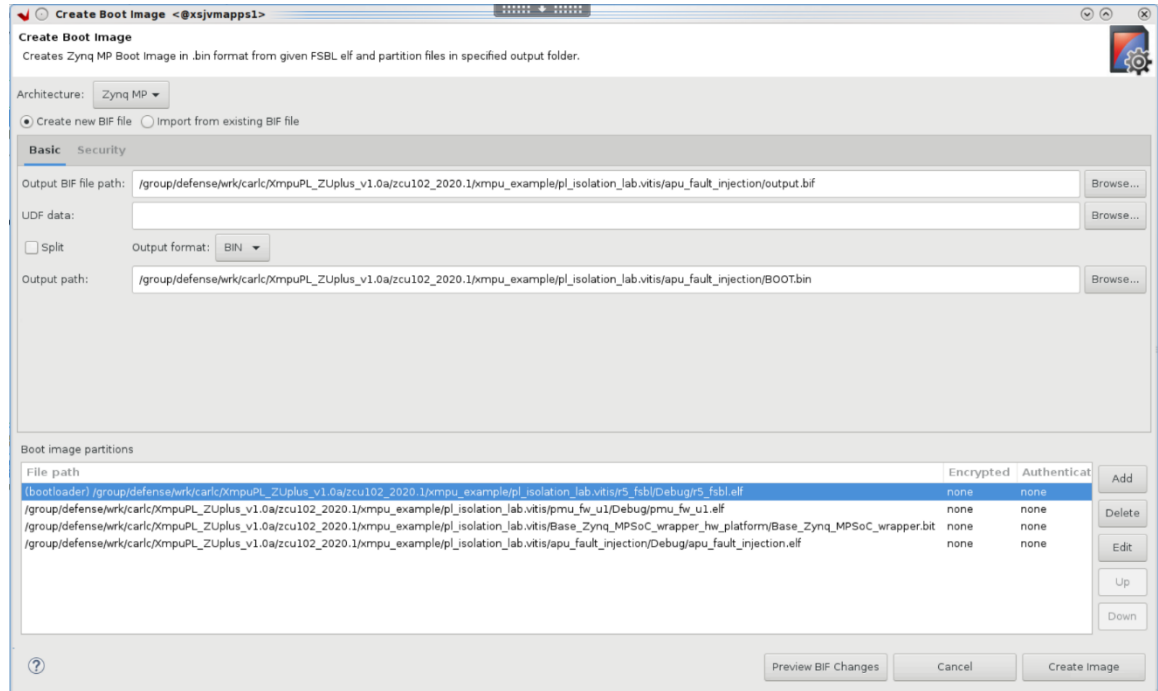
CAUTION! The destination CPU defaults to A53_0, but if you do not actually select it from the drop-down menu, then the parameter may not get written to the BIF file. Use **Preview BIF Changes** to verify.

Figure 43: Preview BIF Changes



- e. Click **OK**
6. The Create Boot Image window looks like the following figure.
7. Click **Create Image** and select **Overwrite** if prompted.

Figure 44: Create Boot Image for RPU Fault Injection System



Note: This example is not using secure boot and all applications are standalone OS, hence the exception level and TrustZone settings for BootGen does not matter.

RPU Isolation Test System

The RPU isolation test system is a container of the necessary applications to run the RPU fault injection application to test the isolated system.

1. Select **File > New > Application Project**.
 - a. Click **Next** if the welcome page opens.
 - b. Select platform from repository: zcu102_isolation_test [custom]
 - c. Click **Next**
 - d. Project name: rpu_fault_injection
 - e. System project: Create New...
 - f. System project details: Project name: rpu_fault_injection_system
 - g. Processor: psu_cortexr5_0
 - h. Click **Next**
 - i. Select a domain: standalone on psu_cortexr5_0

- j. Click **Next**
 - k. Select **Empty Application**
 - l. Click **Finish**
2. Right-click **rpu_fault_injection_system > rpu_fault_injection > src** and select **Import sources**
 - a. Browse and navigate to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/  
zcu102_example/sources/src/rpu_fault_injection
```
 - b. Click **Select Folder**.
 - c. Click **Select All**.
 - d. Click **Overwrite existing sources**
 - e. Click **Finish**
3. Right-click **rpu_fault_injection_system** and select **Add Application Project...**
 - a. Project name: **pmu_fw_u0**
 - b. System project: Select **rpu_fault_injection_system** from a drop-down menu
 - c. Processor: **psu_pmu_0**
 - d. Click **Next**
 - e. Select **zynqmp_pmufw_u0**
 - f. Click **Next**
 - g. Select **Zynq MP PMU firmware**
 - h. Click **Finish**
4. Right-click **rpu_fault_injection_system > pmufw_u0 > src** and select **Import sources**
 - a. Browse and navigate to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/  
zcu102_example/sources/src/pmu_fw_2020.1
```
 - b. Click **Select Folder**.
 - c. Click **Select All**.
 - d. Click **Overwrite existing sources...**
 - e. Click **Finish**.
5. Click **rpu_fault_injection_system** and select **Project > Build Project**

Note: When completed if there is an error: platform file not found, ignore it. This is because default boot components were not used. You will be creating a boot image in the following steps.

Create the RPU Fault Injection Boot Image

To create the boot image

For the following steps:

```
build_path = <your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2020.1/xmpu_example/  
pl_isolation_lab.vitis
```

1. Select > **Create Boot Image**

- a. Architecture: Zynq MP
- b. Check **Create new BIF file**
- c. Output BIF file path: <build_path>/rpu_fault_injection/output.bif
- d. Output path: <build_path>/rpu_fault_injection/BOOT.bin
- e. Continue without clicking create image

Note: If the boot image partitions are automatically filled, select each one and delete, so that the next steps are performed from scratch.

Note: The RPU isolation test system uses the same r5_fsbl that was created in the APU isolation test system.

2. Click **Add**

- a. File path: <build_path>/r5_fsbl/Debug/r5_fsbl.elf
- b. Partition type: bootloader
- c. Destination device: PS
- d. Destination CPU: R5 0
- e. Click **OK**

3. Click **Add**

- a. File path: <build_path>/pmu_fw_u0/Debug/pmu_fw_u0.elf
- b. Partition type: datafile
- c. Destination device: PS
- d. Destination CPU: PMU
- e. Click **OK**

4. Click **Add**

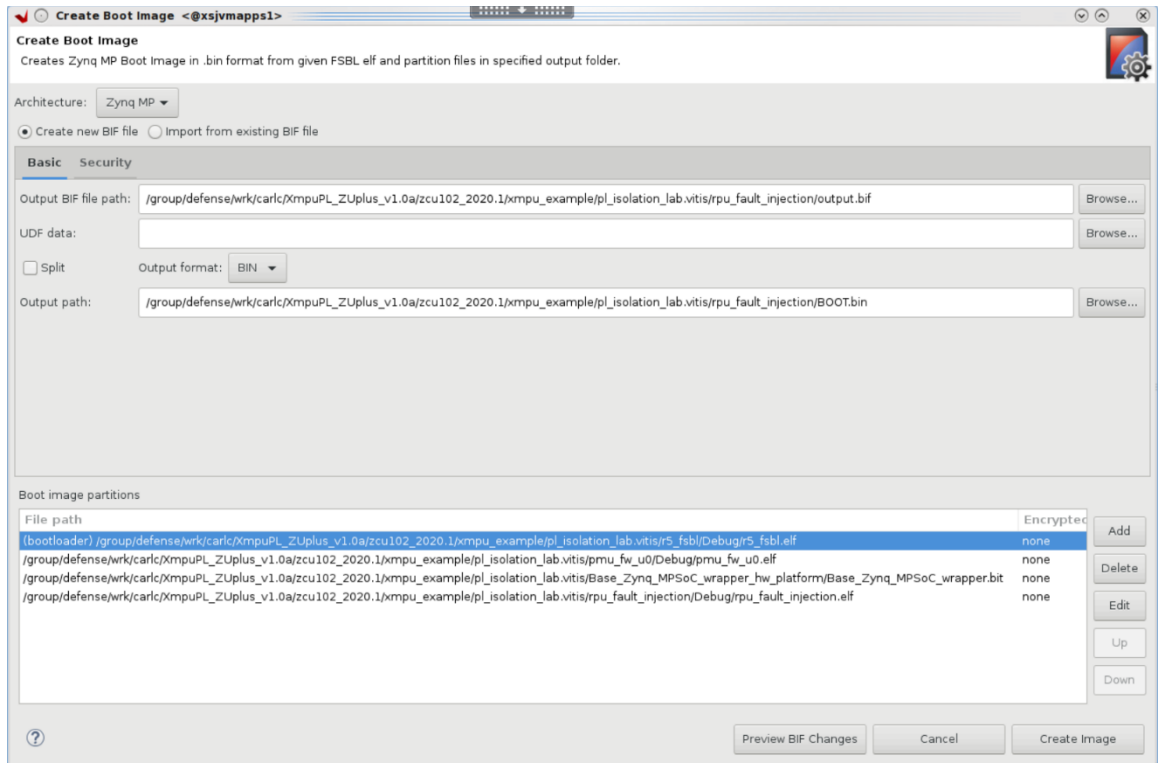
- a. File path: <build_path>/Base_Zynq_MPSoC_wrapper_hw_platform/
BaseZynq_MPSoC_hw_platform/Base_Zynq_MPSoC_wrapper.bit
- b. Partition type: datafile
- c. Destination device: PL
- d. Click **OK**

5. Click **Add**

- a. File path: <build_path>/rpu_fault_injection/Debug/
rpu_fault_injection.elf

- b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: R5 0
 - e. Click **OK**
6. The Create Boot Image window looks like the following figure
7. Click **Create Image** and select **Overwrite** if prompted.

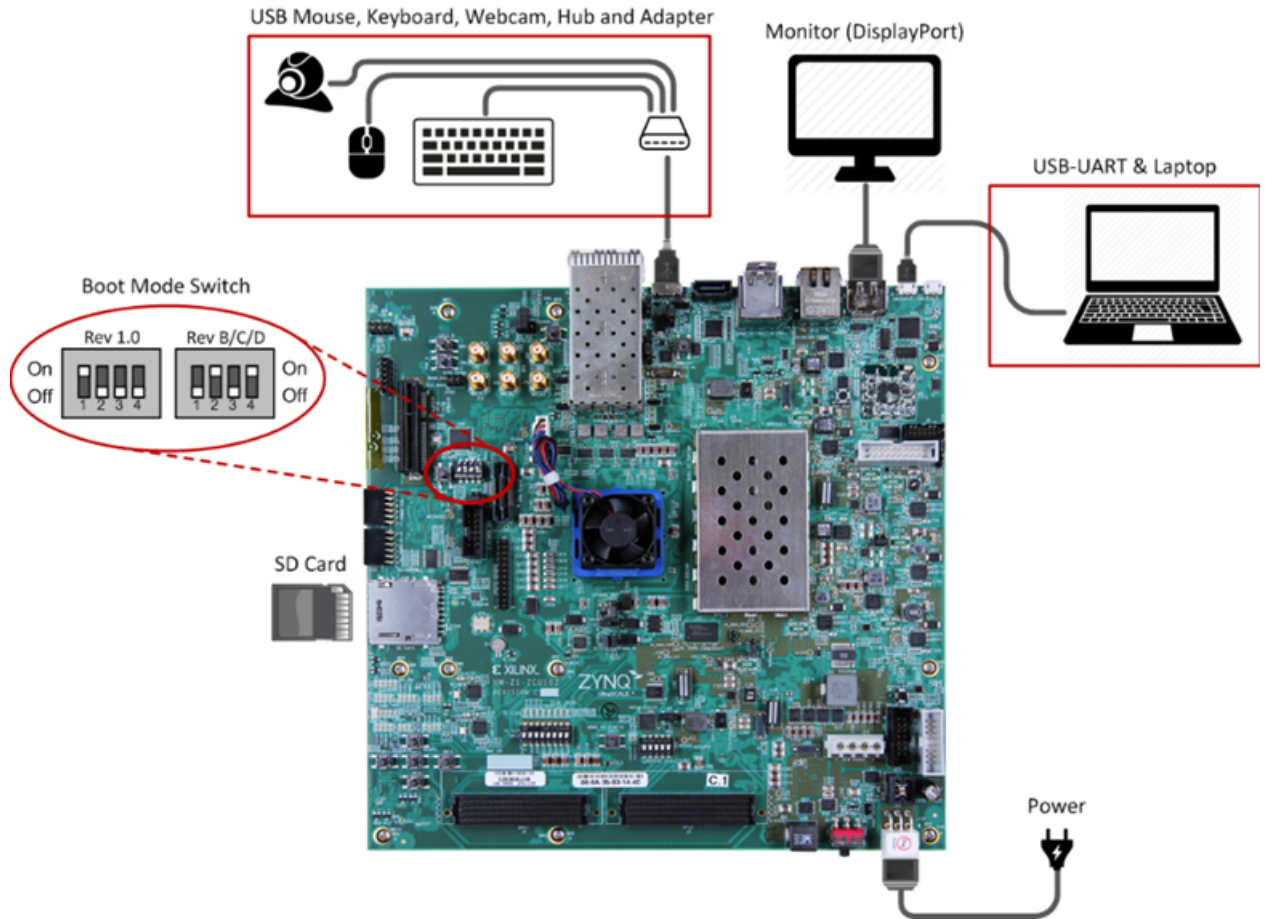
Figure 45: Create Boot Image for RPU Fault Injection System



Running the Isolation Example on the ZCU102 Board

The ZCU102 Evaluation Board is shown in the following figure. For further details, refer to the [ZCU102 Evaluation Board User Guide \(UG1182\)](#).

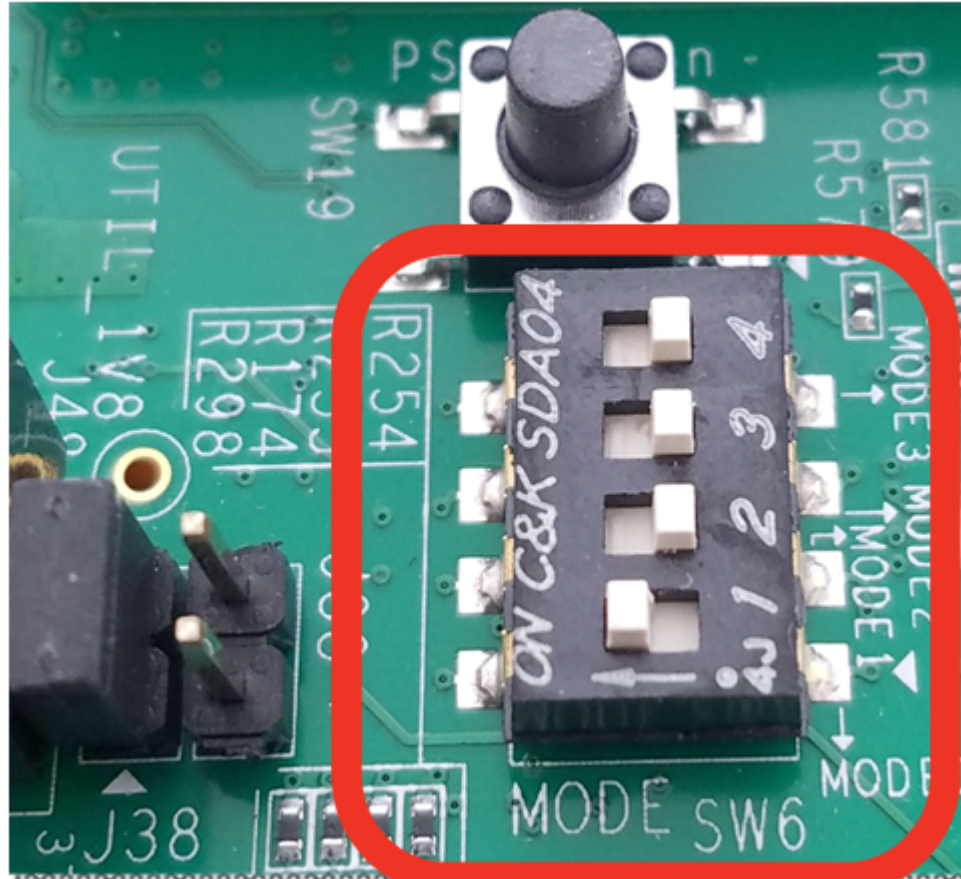
Figure 46: ZCU102 Evaluation Board



ZCU102 Evaluation Board Setup

1. Connect a USB cable to the UART port of the board and identify the COM ports that were mapped to it.
2. (Optional) Connect a USB cable to the JTAG port of the board to utilize the Debugger.
3. Set up two (2) serial communication terminals to observe output on UART0 (APU) and UART1 (RPU).
 - a. Baud rate: 115200
 - b. Data bits: 8
 - c. Parity: None
4. Stop bits: 1
5. Set boot mode:SD (see the following figure for reference)
 - a. $\text{MODE}[3:0] > 1110 > \text{SW6} - [4:1] > \text{OFF} - \text{OFF} - \text{OFF} - \text{ON}$

Figure 47: SW6 Boot Mode



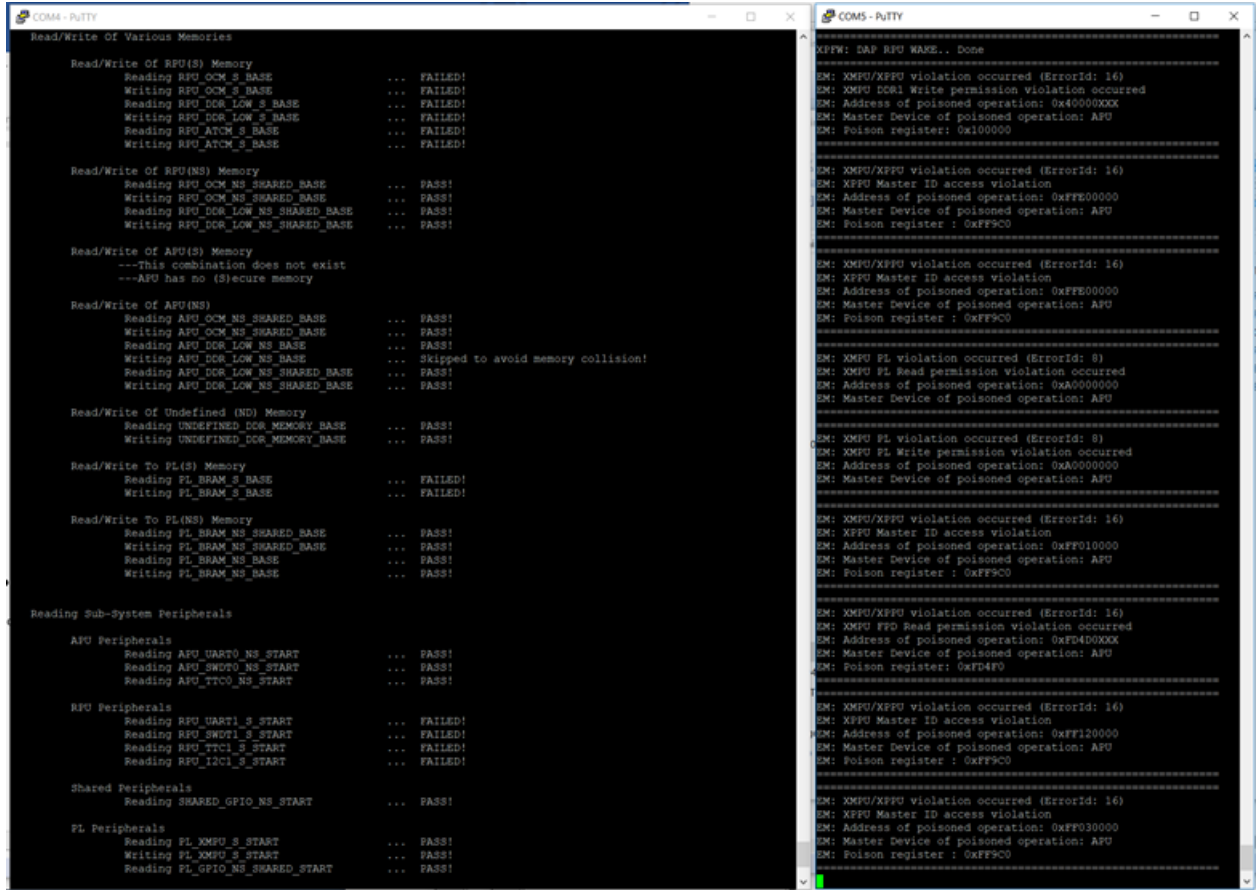
APU Fault Injection Test

- Copy the BOOT.bin file for the APU fault injection application as follows to the SD card:

```
<build_path>/apu_fault_injection/BOOT.bin
```

- Place the SD Card into the socket J100 and power the board
 - After completing initial boot, the fault injection test runs and displays its output to terminals 0 and 1 as shown in the following figure. Term 0 shows the APU output, and term 1 shows the APU output.

Figure 48: APU Fault Injection Output



The read/write address tests shown in term 0 must either *PASS* or *FAIL* in correspondence to the isolation layout of the system. You can refer to [figure 14](#) for further clarity.

The APU is designated non-secure, and hence can successfully read/write to NS (non-secure) and NS_SHARED (non-secure shared with secure) memory and peripherals. Each time a test fails, a violation is reported by the PMU in *term 1*.

Figure 49: PL Memory and Peripheral Test Results (APU)

```

Read/Write To PL(S) Memory
  Reading PL_BRAM_S_BASE           ... FAILED!
  Writing PL_BRAM_S_BASE           ... FAILED!

Read/Write To PL(NS) Memory
  Reading PL_BRAM_NS_SHARED_BASE   ... PASS!
  Writing PL_BRAM_NS_SHARED_BASE   ... PASS!
  Reading PL_BRAM_NS_BASE          ... PASS!
  Writing PL_BRAM_NS_BASE          ... PASS!

...

PL Peripherals
  Reading PL_XMPU_S_START           ... PASS!
  Writing PL_XMPU_S_START           ... PASS!
  Reading PL_GPIO_NS_SHARED_START  ... PASS!
    
```

Examine the term 0 output for the PL memory and peripheral tests, shown in the previous figure. The failed test, on PL_BRAM_S_BASE, violations are reported in term 1, as shown in the following figure.

Figure 50: PL Address Violations in APU

```

=====
EM: XMPU PL violation occurred (ErrorId: 8)
EM: XMPU PL Read permission violation occurred
EM: Address of poisoned operation: 0xA0000000
EM: Master Device of poisoned operation: APU
=====

=====
EM: XMPU PL violation occurred (ErrorId: 8)
EM: XMPU PL Write permission violation occurred
EM: Address of poisoned operation: 0xA0000000
EM: Master Device of poisoned operation: APU
=====
    
```

Note: There is one read permission violation and one write permission violation including the address and originating master ID. `ErrorId:8` corresponds to activity detected on the `pmu_error_from_pl` port used by the `zupl_xmpu_v1_0` irq port, in the PL design, to communicate interrupts to the PMU. The code to respond to this interrupt type has been added to the PMU firmware. You can refer to [A closer Look at the Platform Management Unit \(PMU\)](#) for a detailed understanding of how this was accomplished.

Examining the PL Peripherals tests leads to discovering that the read/write tests to the secure address `PL_XMPU_S_START` did not *FAIL*. This is only because these access attempts did not result in an AXI violation. The XMPU_PL was configured to *Lock Out* configuration changes by any master not authorized by the `LOCK_BYPASS` register.

Although the APU can read the configuration registers, any write attempts are ignored, however, the AXI transaction is processed without error.

The final test from term 0 is to unlock the XMPU_PL Configuration.

Figure 51: **Unlock XMPU_PL (APU)**

```
Attempt Unlock PL XMPU
Read PL_XMPU_S_LOCK 0x00000001
Write PL_XMPU_S_LOCK 0x00000000
Read PL_XMPU_S_LOCK 0x00000001
```

As shown in the previous figure, the `LOCK` register is read and indicates the status as locked. An attempt to clear the register is performed and then re-read. The attempt to write `0x0` to the register is ignored, and the lock remains active.



TIP: The PL design can be altered to completely isolate the `S_AXI_XMPU` slave configuration port of the `zupl_xmpu_v1_0` core and block all read and write access from the unauthorized masters. An example of this is shown in the [Isolating the XMPU_PL Configuration from the XMPU_PL Usage Examples](#) section. This is left as an exercise for the reader.

RPU Fault Injection Test

Copy the `BOOT.bin` file for the RPU fault injection application `<build_path>/rpu_fault_injection/BOOT.bin` to the SD Card, place the SD Card into the socket J100, and power the board.

After completing initial boot, the fault injection test runs and displays its output to terminals 0 and 1 as shown in the following figure. Term 0 shows the APU output, and term 1 shows the RPU output.

Figure 52: RPU Fault Injection Output

```

COM4 - PUTTY
PMU Firmware 2019.1 May 26 2020 17:01:46
PMU ROM Version: xpbs-vs.1.0-0
EM: Enabling XMPU/XPPU interrupts
EM Module (MOD-0): Initialized.
DAE WAKE (MOD-3): Initialized.
LEGACY PWR UP/DN/ISO (MOD-4): Initialized.
XPFW: Calling ROM PWRUP Handler..Done
XPFW: Calling ROM Isolation Handler..Done
XPFW: Calling ROM PWRUP Handler..Done
Copied FSBL image to DDR and image hash checksum calculation s
successful
XMPu1_PmuTask: Initializing PL XMPU
XPFw_SchedulerRemoveTask: Removed 1 tasks
=====
EM: XMPU/XPPU violation occurred (ErrorId: 16)
EM: XMPU DDR0 Read permission violation occurred
EM: Address of poisoned operation: 0x100XXX
EM: Master Device of poisoned operation: RPU0
EM: Poison register: 0x100000
=====
EM: XMPU/XPPU violation occurred (ErrorId: 16)
EM: XMPU DDR0 Write permission violation occurred
EM: Address of poisoned operation: 0x100XXX
EM: Master Device of poisoned operation: RPU0
EM: Poison register: 0x100000
=====
EM: XMPU PL violation occurred (ErrorId: 8)
EM: XMPU PL Read permission violation occurred
EM: Address of poisoned operation: 0xA0000C00
EM: Master Device of poisoned operation: RPU0
=====
EM: XMPU PL violation occurred (ErrorId: 8)
EM: XMPU PL Write permission violation occurred
EM: Address of poisoned operation: 0xA0000C00
EM: Master Device of poisoned operation: RPU0
=====
EM: XMPU/XPPU violation occurred (ErrorId: 16)
EM: XPPU Master ID access violation
EM: Address of poisoned operation: 0xFF000000
EM: Master Device of poisoned operation: RPU0
EM: Poison register : 0xFF9C0
=====
EM: XMPU/XPPU violation occurred (ErrorId: 16)
EM: XPPU Master ID access violation
EM: Address of poisoned operation: 0xFF150000
EM: Master Device of poisoned operation: RPU0
EM: Poison register : 0xFF9C0
=====
EM: XMPU/XPPU violation occurred (ErrorId: 16)
EM: XPPU Master ID access violation
EM: Address of poisoned operation: 0xFF110000
EM: Master Device of poisoned operation: RPU0
EM: Poison register : 0xFF9C0
=====
COM5 - PUTTY
Read/Write Of Various Memories

Read/Write To RPU(S) Memory
Reading RPU_OCM_S_BASE ... PASS!
Writing RPU_OCM_S_BASE ... PASS!
Reading RPU_DDR_LOW_S_BASE ... PASS!
Writing RPU_DDR_LOW_S_BASE ... PASS!
Reading RPU_ATCM_S_BASE ... PASS!
Writing RPU_ATCM_S_BASE ... Skipped to avoid memory collision!

Read/Write Of RPU(NS) Memory
Reading RPU_OCM_NS_SHARED_BASE ... PASS!
Writing RPU_OCM_NS_SHARED_BASE ... PASS!
Reading RPU_DDR_LOW_NS_SHARED_BASE ... PASS!
Writing RPU_DDR_LOW_NS_SHARED_BASE ... PASS!

Read/Write Of APU(S) Memory
---This combination does not exist
---APU has no secure memory

Read/Write Of APU(NS)
Reading APU_OCM_NS_SHARED_BASE ... PASS!
Writing APU_OCM_NS_SHARED_BASE ... PASS!
Reading APU_DDR_LOW_NS_BASE ... FAILED!
Writing APU_DDR_LOW_NS_BASE ... FAILED!
Reading APU_DDR_LOW_NS_SHARED_BASE ... PASS!
Writing APU_DDR_LOW_NS_SHARED_BASE ... PASS!

Read/Write Of ND Memory
Reading UNDEFINED_DDR_MEMORY_BASE ... PASS!
Writing UNDEFINED_DDR_MEMORY_BASE ... PASS!

Read/Write To PL(S) Memory
Reading PL_BRAM_S_BASE ... PASS!
Writing PL_BRAM_S_BASE ... PASS!

Read/Write To PL(NS) Memory
Reading PL_BRAM_NS_SHARED_BASE ... PASS!
Writing PL_BRAM_NS_SHARED_BASE ... PASS!
Reading PL_BRAM_NS_BASE ... FAILED!
Writing PL_BRAM_NS_BASE ... FAILED!

Reading Sub-System Peripherals

APU Peripherals
Reading APU_UART0_NS_START ... FAILED!
Reading APU_SWDT0_NS_START ... FAILED!
Reading APU_TTC0_NS_START ... FAILED!

RPU Peripherals
Reading RPU_UART1_S_START ... PASS!
Reading RPU_SWDT1_S_START ... PASS!
Reading RPU_TTC1_S_START ... PASS!
Reading RPU_I2C1_S_START ... PASS!

Shared Peripherals
Reading SHARED_GPIO_NS_START ... PASS!

PL Peripherals
Reading PL_XMPU_S_START ... PASS!
Writing PL_XMPU_S_START ... PASS!
Reading PL_GPIO_NS_SHARED_START ... PASS!
    
```

The read/write address tests shown in term 1 must either *PASS* or *FAIL* in correspondence to the isolation layout of the system. You can refer to Figure 14 for further clarity.

The RPU is designated non-secure, and hence can successfully read/write to NS (non-secure) and NS_SHARED (non-secure shared with secure) memory and peripherals. Each time a test fails, a violation is reported by the PMU in term 0.

Figure 53: PL Memory and Peripherals Test Results (RPU)

```

Read/Write To PL(S) Memory
  Reading PL_BRAM_S_BASE           ... PASS!
  Writing PL_BRAM_S_BASE           ... PASS!

Read/Write To PL(NS) Memory
  Reading PL_BRAM_NS_SHARED_BASE   ... PASS!
  Writing PL_BRAM_NS_SHARED_BASE   ... PASS!
  Reading PL_BRAM_NS_BASE           ... FAILED!
  Writing PL_BRAM_NS_BASE           ... FAILED!
. . .

PL Peripherals
  Reading PL_XMPU_S_START           ... PASS!
  Writing PL_XMPU_S_START           ... PASS!
  Reading PL_GPIO_NS_SHARED_START   ... PASS!
    
```

Examine the term 0 output for the PL memory and peripheral tests, shown in the previous figure. The failed test, on PL_BRAM_S_BASE, violations are reported in term 1, as shown in the following figure.

Figure 54: PL Address Violations in RPU

```

=====
EM: XMPU PL violation occurred (ErrorId: 8)
EM: XMPU PL Read permission violation occurred
EM: Address of poisoned operation: 0xA0000C00
EM: Master Device of poisoned operation: RPU0
=====

EM: XMPU PL violation occurred (ErrorId: 8)
EM: XMPU PL Write permission violation occurred
EM: Address of poisoned operation: 0xA0000C00
EM: Master Device of poisoned operation: RPU0
=====
    
```

Note: There is one read permission violation and one write permission violation including the address and originating master ID. `ErrorId: 8` corresponds to activity detected on the `pmu_error_from_pl` port used by the `zupl_xmpu_v1_0` irq port, in the PL design, to communicate interrupts to the PMU. The code to respond to this interrupt type has been added to the PMU firmware.

You can refer to [A closer Look at the Platform Management Unit \(PMU\)](#) for a detailed understanding of how this was accomplished.

The final test from term 1 is to unlock the XMPU_PL Configuration.

Figure 55: Unlock XMPU_PL (RPU)

```
Attempt Unlock PL XMPU
Read PL_XMPU_S_LOCK 0x00000001
Write PL_XMPU_S_LOCK 0x00000000
Read PL_XMPU_S_LOCK 0x00000000
```

As shown in the previous figure, the LOCK register is read and indicates the status as *locked*. The register is cleared and then re-read. The RPU is an authorized master in the LOCK_BYPASS registers and retains write privileges to the XMPU_PL configuration registers.

A closer Look at the Platform Management Unit (PMU)

PMU Configuration

To configure the PMU, five source files and a linker script are imported into the `pmufw_src` directory:

1. `xpfw_config.h`
2. `xpfw_mod_sched.c`
3. `xpfw_mod_em.c`
4. `xpfw_pl_xmpu.c`
5. `xpfw_pl_xmpu.h`
6. `lscript.ld`

In the `xpfw_config.h`, you can enable the options for the scheduler, error manager, XMPU/ XPPU (PS) interrupts, and detailed print statements. The following figure shows a code snippet.

Figure 56: xpfw_config.h Snippets

```

/* PMUFW print levels */
#define XPFW_PRINT_VAL (1U)
#define XPFW_DEBUG_ERROR_VAL (1U)
#define XPFW_DEBUG_DETAILED_VAL (1U)
. . .
/* PMU Firmware code include options...
#define ENABLE_PM_VAL (1U)
#define ENABLE_EM_VAL (1U)
#define ENABLE_SCHEDULER_VAL (1U)
. . .
#define XPU_INTR_DEBUG_PRINT_ENABLE_VAL (1U)
. . .
#define USE_DDR_FOR_APU_RESTART_VAL (0U) /* version 2020.1 */

```

This configuration greatly increases the memory footprint of the PMU, mostly due to the detailed debug messaging enabled for this demonstration. The linker script, lscript.ld, reduces the size of the stack from 0x1000 to 0x800 so that pmufw can fit into the allotted 128 KB:

```
_STACK_SIZE = DEFINED(_STACK_SIZE) ? _STACK_SIZE: 0x800;
```

Configuring the XMPU_PL in the PMU Scheduler

The PMU scheduler is used to periodically call a task. In this example, a scheduler task is used to initialize and configure the XMPU_PL. While it only needs to be initialized once, the task needs to wait until PL configuration and start up are complete. A flag indicates whether the XMPU_PL has already been initialized.

The `xpfw_pl_xmpu.h` header file provides API declarations for the following functions:

```
void XmpuPl_PmuTaskInit(const XPfw_Module_t*SchModPtr);
```

Task initialization function registers the Task function in the scheduler.

```
void XmpuPl_PmuTask(void);
```

The Task function is periodically called by the scheduler at a defined interval.

```
void XmpuPl_Interrupt_Handler(u8 ErrorId);
```

The interrupt handler for the XMPU_PL is called by the PMU Error Manager.

To schedule the task, a function call to `XMpuPl_PmuTaskInit` is added to the `SchCfglnit` function in `xpfw_mod_sched.c`, shown in the following figure.

Figure 57: Scheduler Task Initialization in xpfw_mod_sched.c

```

/* Point to the XMpuPL PMU Firmware Library*/
#include "xpfw_pl_xmpu.h"

#ifdef ENABLE_SCHEDULER
static void SchCfgInit(const XPfw_Module_t *ModPtr, const u32 *CfgData, u32 Len)
{
    /* Add in the XMpuPL PMU task */
    XMpuPL_PmuTaskInit(ModPtr);
}
    
```

The source file `xpfw_pl_xmpu.c` and header file `xpfw_pl_xmpu.h` are not a part of the standard PMU source nor is it from the `zupl_xmpu` SW driver set. These are examples of user-created files, created specifically for this demonstration.

The following figure shows the `XMpuPI_PmuTaskInit` function. The `XPfw_CoreScheduleTask` API function schedules the `XMpuPI_PmuTask` task as a callback function in the scheduler. The `XMPUPL_TASK_INTERVAL` sets a callback period of 25 ms.

Figure 58: XMpuPI_PmuTaskInit Function in xpfw_pl_xmpu.c

```

void XMpuPI_PmuTaskInit(const XPfw_Module_t *SchModPtr)
{
    /* schedule the XMpuPl task */
    if (XPfw_CoreScheduleTask(
        SchModPtr, XMPUPL_TASK_INTERVAL, XMpuPl_PmuTask) != XST_SUCCESS) {
        xil_printf("Warning: XMpuPI_PmuTaskInit: Failed to schedule task\r\n");
    }
}
    
```

The `xpfw_pl_xmpu.c` file defines two static variables:

```

static u8 XMpuPl_Initialized = {0U};
    Flag to indicate XMPU_PL initialization status.

    static XmpuPl XmpuInst;
    XMPU_PL instance.
    
```

The `XMpuPI_PmuTask` function is shown in the following figure. First, it checks the `XMpuPI_Initialized` flag to see if the `XMPU_PL` needs to be initialized. Next, it checks the `PCAP Status` to see if the `PL` configuration is `DONE` and has reached the end of start up (`EOS`). Then it calls the `configureXMPU` function. If the `configureXMPU` function completes successfully, then the `XMpuPI_Initialized` flag is set.

Figure 59: XMpuPl_PmuTask function in xpfw_pl_xmpu.c

```

void XMpuPl_PmuTask(void)
{
    /* Initialize and Configure pl_xmpu */
    if (!XMpuPl_Initialized) {
        /* Check that PL configuration is done */
        if ((Xil_In32(CSU_PCAP_STATUS) & PCAP_STAT_DONE_EOS)
            == PCAP_STAT_DONE_EOS) {
            XPfw_Printf(DEBUG_DETAILED,
                "XMpuPl_PmuTask: Initializing PL XMPU\n\r");
            if (0U == configureXMPU(&XmpuInst)) {
                /* Set Initialized flag */
                XMpuPl_Initialized = 1U;
            }
        }
    }
}
    
```



TIP: Though it is not necessary to do so, once the XMPU_PL has been configured, the `XMpuPl_PmuTask` function can be removed from the scheduler, using the `XPfw_CoreRemoveTask`, to avoid continuing to unnecessarily task the PMU. This is left as an exercise for the reader.

The `configureXMPU` function, shown in the previous figure, first initializes the `XmpuPl` instance, and then configures the `XMPU_PL` core. Only one instance is needed for this demonstration design, however, the Simple XMPU_PL (RPU) Example demonstrates initialization for any number of instances.

The SW Driver functions that configures the `XMPU_PL` are shown in [Appendix B: SW Driver Library](#). See the `xpfw_pl_xmpu.h` header file for the macro definitions of the constants used in the `configureXMPU` function.

Figure 60: configureXMPU Function in xpfw_pl_xmpu.c

```

static u32 configureXMPU(XmpuPl *InstancePtr)
{
    u32 Status = {0U};

    /* Initialize XMPU_PL */
    XmpuPl_Config * XmpuPl_ConfigPtr = XmpuPl_LookupConfig(XMPU_DEVICE_ID);
    Status = XmpuPl_CfgInitialize(InstancePtr,
        XmpuPl_ConfigPtr, XmpuPl_ConfigPtr->BaseAddress);
    if (Status != 0U) {
        XPfw_Printf(DEBUG_ERROR, "\n\rXMPU Initialization Failed!\n\r");
    }

    /* Configure XMPU_PL */
    if (Status == 0U) {
        InstWriteReg(InstancePtr, XMPU_PL_CTRL_OFFSET, XMPU_CTRL);
        InstWriteReg(InstancePtr, XMPU_PL_BYPASS_OFFSET, XMPU_LOCK_MASTERS);
        InstWriteReg(InstancePtr, XMPU_PL_LOCK_OFFSET, 1U);

        /* Enable Interrupts */
        XmpuPl_EnableInterrupts(InstancePtr, XMPU_INT_EN);
    }

    /* Add REGION 0 */
    if (Status == 0U) {
        Status = XmpuPl_AddRegion(InstancePtr,
            REGION_0_ADDR, 1U, REGION_0_MASTERS, REGION_0_CFG);
        if (Status != 0U) {
            XPfw_Printf(DEBUG_ERROR, "\n\rXMPU Add Region 0 Failed!\n\r");
        }
    }

    /* Add REGION 1 */
    if (Status == 0U) {
        Status = XmpuPl_AddRegion(InstancePtr,
            REGION_1_ADDR, 1U, REGION_1_MASTERS, REGION_1_CFG);
        if (Status != 0U) {
            XPfw_Printf(DEBUG_ERROR, "\n\rXMPU Add Region 1 Failed!\n\r");
        }
    }

    /* Update XmpuPl Instance */
    if (Status == 0U) {
        Status = XmpuPl_GetConfig(InstancePtr);
        if (Status != 0U) {
            XPfw_Printf(DEBUG_ERROR, "\n\rXMPU Get Config Failed!\n\r");
        }
    }
    return Status;
}
    
```

Handling XMPU_PL Interrupts in the PMU (EM) Error Manager

In the PL design shown in Figure 21, the `zupl_xmpu` reference core interrupt output port, `irq`, is routed to the PS ports `pl_ps_irq[0]` and `pmu_error_from_pl[0]`. The `pl_ps_irq` signal can be used by the global interrupt controller (GIC) to trigger interrupts in the RPU and APU processors. Similarly, the `pmu_error_from_pl` signal triggers an interrupt in the PMU Error Manager.

The PMU Error Manager is customized to respond to system events. The default configuration of the `EmEventHandler`, in `xpfw_mod_em.c`, installs event detection modules for the PMU global registers `ERROR_STATUS_1` and `ERROR_STATUS_2`. The `ERROR_STATUS_2` register provides event triggers for `pmu_error_from_pl[0:3]` on bits `ERROR_STATUS_2[2:5]`. Refer to *Zynq UltraScale+ Device Register Reference (UG1087)* for more details of the PMU global registers.

The *Zynq UltraScale+ MPSoC: Software Developers Guide (UG1137)* provides a detailed description of the PMU firmware and error manager. To enable an event handler for the PL signals, an `XPfw_EmSetAction` function call is added to the `EmCfgInit` function in `xpfw_mod_em.c`, shown in the following figure.

Figure 61: Error Manager Configuration Initialization in `xpfw_mod_em.c`

```

/* CfgInit Handler */
static void EmCfgInit(const XPfw_Module_t *ModPtr, const u32 *CfgData, u32 Len)
{
    u32 ErrId = 0U;
    s32 Status;

    /* Register for Error events from Core */
    (void) XPfw_CoreRegisterEvent(ModPtr, XPFW_EV_ERROR_1);
    (void) XPfw_CoreRegisterEvent(ModPtr, XPFW_EV_ERROR_2);

    /* Init the Error Manager */
    XPfw_EmInit();

    /****** Added for PL XMPU *****/
    /* Set Interrupt action for PL Errors */
    xil_printf("XPfw_EmSetAction Set Interrupt action for PL Errors\n\r");
    Status = XPfw_EmSetAction(EM_ERR_ID_PL, EM_ACTION_CUSTOM, PL_ErrorHandler);
    if (Status != 0) {
        xil_printf("XPfw_EmSetAction PL_ErrorHandler Failed!\n\r");
    }
    /******
    
```

The EM error IDs are defined in `xpfw_error_manager.h`. `EM_ERR_ID_PL` (8U) identifies the PL to PS portion of the `ERROR_STATUS_2` register. The `XPfw_EmSetAction` function call provides the error ID, action type and event handler. Setting the action type to `EM_ACTION_CUSTOM` enables a callback to the event handler.

In the following figure, the event handler, `PL_ErrorHandler`, has been added to `xp_fw_mod_em.c`. This specific example shows the event handler, `PL_ErrorHandler`, calls for the XMPU_PL interrupt handler, `XmpuPl_Interrupt_Handler`, and then clears the event in the `ERROR_STATUS_2` Register.

Note: Only `PL_TO_PS` events are cleared by this handler.

Figure 62: PL Event Handler

```
/* ***** Added for PL XMPU ***** */
void PL_ErrorHandler(u8 ErrorId)
{
    /* Clear the Error Status in XMPU_PL registers */
    XmpuPl_Interrupt_Handler(ErrorId);
    /* Clear the Error Status in PMU registers */
    u32 err_stat = Xil_In32(PMU_GLOBAL_ERROR_STATUS_2);
    Xil_Out32(PMU_GLOBAL_ERROR_STATUS_2,
              (PMU_GLOBAL_ERROR_EN_2_PL_MASK & err_stat));
}
```

The `XmpuPl_Interrupt_Handler` function is shown in the following figure:

Figure 63: XMPU_PL Interrupt Handler

```

void XmpuPl_Interrupt_Handler(u8 ErrorId)
{
    XmpuPl *InstancePtr = &XmpuInst;

    /* Get Interrupt Status */
    u32 xmpu_isr = XMpuPl_GetInterruptStatus(InstancePtr);
    u8 write_err = (xmpu_isr & XMPU_PL_IXR_WRVIO_MSK);
    u8 read_err = (xmpu_isr & XMPU_PL_IXR_RDVIO_MSK);
    u32 xmpu_err1 = InstReadReg(InstancePtr, XMPU_PL_ERRS1_OFFSET);
    u32 xmpu_err2 = InstReadReg(InstancePtr, XMPU_PL_ERRS2_OFFSET);

    /* Display Violation */
    XPfw_Printf(DEBUG_DETAILED,
                "=====\\r\\n");
    XPfw_Printf(DEBUG_DETAILED,
                "EM: XMPU PL violation occurred (ErrorId: %d)\\r\\n", ErrorId);
    if (write_err != 0U) {
        XPfw_Printf(DEBUG_DETAILED,
                    "EM: XMPU PL Write permission violation occurred\\r\\n");
    }
    if (read_err != 0U) {
        XPfw_Printf(DEBUG_DETAILED,
                    "EM: XMPU PL Read permission violation occurred\\r\\n");
    }
    XPfw_Printf(DEBUG_DETAILED,
                "EM: Address of poisoned operation: 0x%X\\r\\n", xmpu_err1);

    /* Identify Master Device */
    u32 MasterID = xmpu_err2 & 0x3FFU;
    for(u32 MasterIdx = 0U; MasterIdx < ARRAYSIZE(XpuMasterIDLUT);
        ++MasterIdx) {

        if ((MasterID >= XpuMasterIDLUT[MasterIdx].MasterID) &&
            (MasterID <= XpuMasterIDLUT[MasterIdx].MasterIDLimit)) {

            XPfw_Printf(DEBUG_DETAILED, "EM: Master Device of poisoned "
                "operation: %s\\r\\n",
                XpuMasterIDLUT[MasterIdx].MasterName);

            break;
        }
    }

    XPfw_Printf(DEBUG_DETAILED,
                "=====\\r\\n");

    /* Clear Interrupt Status */
    XMpuPl_ClearInterruptStatus(InstancePtr, xmpu_isr);
}
    
```

The XmpuPl_Interrupt_Handler function has been specifically designed for the purposes of this demonstration to output XMPU_PL violations in the same format as that used for the XMPU/ XPPU (PS) events handled in xpfw_xpu. For reference, see [figure 49](#) and [figure 53](#).

As with any handler written for XMPU_PL interrupts, you must first get the interrupt status from the ISR register to determine the violation type (read or write). The ERR_STATUS1 and ERR_STATUS2 registers provide the originating AXI address and Master ID, respectively. After printing out the violation data, the interrupt status is cleared from the ISR register. To identify the PS master from the master ID, the static XpuMasterID list has been copied from `xp_fw_xpu.c` and placed into the `xp_fw_pl_xmpu.c` file.

Note: In the Vitis 2019.2 version of the pmufw imported files, the XpuMasterID struct content has been reduced to the APU and RPU0 entries to conserve memory consumption. These are the only masters utilized in this demonstration.

Creating the Simple XMPU_PL (RPU) Example in SDK 2019.1

The previous isolation example utilized the platform management unit (PMU) to handle initialization, configuration, and error handling for the XMPU_PL module. Some applications such as safety critical, may have restrictions on modifying the PMU firmware. Thus, this example provides a simpler demonstration from a single application running in the RPU.

FSBL- First Stage Boot Loader

This example uses the same FSBL created in the previous example. If it does not already exist, follow the steps from [First Stage Boot Loader \(FSBL\)](#). This is located in the [Creating the Isolation Test SW Applications in SDK 2019.1](#).

Default PMU

This example will not use the PMU to manage the XMPU_PL. However, a default PMU is needed for booting from the SD Card.

You must build a new PMU with default settings:

1. **Select File>New>Application Project**
 - a. Project Name: `pmu_fw_default`
 - b. OS Platform: `<default> standalone`
 - c. Hardware Platform: `Base_Zynq_MPSoC_wrapper_hw_platform_0`
 - d. Processor: `psu_pmu_0`
 - e. Language: `C`
 - f. Board Support Package: `<default> Create New pmu_fw_default_bsp`
2. Click **Next**
 - a. Available Templates: `Zynq PMU Firmware`
3. Click **Finish**

RPU XMPU_PL Example Application in SDK

To create the RPU example application:

For the following steps:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.1/xmpu_example/  
pl_isolation_lab.sdk
```

1. Select **File>New>Application Project**
 - a. Project name: `rpu_xmpu_example`
 - b. OS Platform: <default> standalone
 - c. Hardware Platform: <default>
 - d. Processor: `psu_cortexr5_0`
 - e. Language: <default> C
 - f. Compiler: <default> 32-bit
 - g. Hypervisor Guest: <default> No
 - h. Board Support Package: <default> Create New `rpu_xmpu_example_bsp`
2. Click **Next**
 - a. Available templates: *Empty Application*
3. Click **Finish**
4. Click and expand `rpu_xmpu_example` in the Project Explorer window
5. Right-click `src` and select **Import...**
6. Click **General > File System** and click **Next**
7. Browse and navigate to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/  
zcu102_example/sources/src/rpu_xmpu_simple_example
```
8. Click **Select Folder**
9. Click **Select All**
10. Click **Overwrite existing sources...**
11. Click **Finish**

Create the RPU Simple Example Boot Image

To create the boot image

For the following steps:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.1/xmpu_example/  
pl_isolation_lab.sdk
```

1. Select > **Create Boot Image**

- a. Architecture: Zynq MP
 - b. Check **Create new BIF file**
 - c. Output BIF file path: `<build_path>/rpu_fault_injection/output.bif`
 - d. Output path: `<build_path>/rpu_fault_injection/BOOT.bin`
 - e. Continue without clicking create image
2. Click **Add**
 - a. File path: `<build_path>/r5_fsbl/Debug/r5_fsbl.elf`
 - b. Partition type: bootloader
 - c. Destination device: PS
 - d. Destination CPU: RS Single
 - e. Click **OK**
 3. Click **Add**
 - a. File path: `<build_path>/pmu_fw_default/Debug/pmu_fw_default.elf`
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: PMU
 - e. Click **OK**
 4. Click **Add**
 - a. File path: `<build_path>/Base_Zynq_MPSoC_wrapper_hw_platform/
BaseZynq_MPSoC_hw_platform/Base_Zynq_MPSoC_wrapper.bit`
 - b. Partition type: datafile
 - c. Destination device: PL
 - d. Click **OK**
 5. Click **Add**
 - a. File path: `<build_path>/rpu_fault_injection/Debug/
rpu_fault_injection.elf`
 - b. Partition type: datafile
 - c. Destination device: PS
 - d. Destination CPU: R5 0
 - e. Click **OK**
 6. Click **Create Image** and select **Overwrite** if prompted.

Creating the Simple XMPU_PL (RPU) Example in Vitis 2019.2

The previous isolation example utilized the platform management unit (PMU) to handle initialization, configuration, and error handling for the XMPU_PL module. Some applications such as safety critical, may have restrictions on modifying the PMU firmware. Thus, this example provides a simpler demonstration from a single application running in the RPU.

This example uses the same platform created in the previous example. If it does not already exist, follow the steps in [Build the Isolation Test Platform](#) in the [Creating the Isolation Test SW Applications in Vitis 2019.2](#) section.

This example also uses the same R5_FSBL created in the previous example. However, you must create a new default PMU application without any modifications.

RPU Simple XMPU_PL Test System

The RPU simple test system will be a container of the necessary applications to run the simple rpu application to test the XMPU_PL on the isolated system.

1. **Select File>New>Application Project**
 - a. Check Use Default Location
 - b. Project Name: rpu_xmpu_example
 - c. System project: Create New > rpu_xmpu_example_system
 - d. Click **Next**
 - e. Select **platform from repository: zcu102_isolation_test** [custom]
 - f. Click **Next**
 - g. Domain: standalone on psu_cortexr5_0
 - h. Click **Next**
 - i. Select **Empty Application**
 - j. Click **Finish**
2. Right-click **rpu_xmpu_example_system > rpu_xmpu_example > src** and select **Import Sources**
 - a. Browse and navigate to:

```
<your_path>/XmpuPL_ZUplus_v1.0a/zupl_xmpu_v1_0/example_designs/  
zcu102_example/sources/src/rpu_simple_example
```
 - b. Click **Select Folder**.
 - c. Click **Select All**
 - d. Click **Overwrite existing sources...**
 - e. Click **Finish**
3. Right-click **rpu_fault_example_system** and select **Add Application Project**

- a. Check **Use default location**
 - b. Project name: pmufw
 - c. System project: Select **rpu_xmpu_example_system** from a drop-down menu
 - d. Click **Next**
 - e. Domain: zynqmp_pmufw_u0
 - f. Click **Next**
 - g. Select ZynqMP PMU Firmware
 - h. Click **Finish**
4. Click **rpu_fault_example_system** and select **Project>Build project**
 - a. If there is an error when the process is completed and platform file is not found, ignore it.

FSBL

Skip this step if the R5_FSBL from the previous example already exists in the workspace.

To build an FSBL for the R5:

1. Select **File>New>Application Project**
 - a. Check **Use default location**
 - b. Project name: r5_fsbl
 - c. System project: Select **Create New>r5_fsbl_system** from a drop-down menu
 - d. Click **Next**
 - e. Select **platform from repository: zcu102_isolation_test** [custom]
 - f. Click **Next**
 - g. Domain: zynqmp_fsbl
 - h. Click **Next**
 - i. Select Zynq MP FSBL
 - j. Click **Finish**
2. Click **r5_fsbl_system** and select **Project>Build project**

Note: The FSBL for this system had to be created in a separate system because the R5_0 is also being used by the example application. Both the FSBL and the application may utilize the same processor, just not in the same system. They get combined in the boot image.

Create the RPU Simple Example Boot Image

To create the boot image

For the following steps:

```
build_path=<your_path>/XmpuPL_ZUplus_v1.0a/zcu102_2019.2/xmpu_example/  
pl_isolation_lab.vitis
```

1. Select **Xilinx** > **Create Boot Image**

- a. Architecture: Zynq MP
- b. Check **Create new BIF file**
- c. Output BIF file path: <build_path>/rpu_xmpu_example/output.bif
- d. Output path: <build_path>/rpu_xmpu_example/BOOT.bin
- e. Continue without clicking create image

2. Click **Add**

- a. File path: <build_path>/r5_fsbl/Debug/r5_fsbl.elf
- b. Partition type: bootloader
- c. Destination device: PS
- d. Destination CPU: R5 0
- e. Click **OK**

3. Click **Add**

- a. File path: <build_path>/pmufw/Debug/pmufw.elf
- b. Partition type: datafile
- c. Destination device: PS
- d. Destination CPU: PMU
- e. Click **OK**

4. Click **Add**

- a. File path:

```
<build_path>Base/Zynq_MPSoC_wrapper_hw_platform/  
BaseZynq_MPSoC_hw_platform/Base_Zynq_MPSoC_wrapper.bit
```

- b. Partition type: Datafile
- c. Destination device: PL
- d. Click **OK**

5. Click **Add**

- a. File path: <build_path>/rpu_xmpu_example/Debug/rpu_xmpu_example.elf
- b. Partition type: datafile
- c. Destination device: PS
- d. Destination CPU: R5 0

- e. Click **OK**
6. Click **Create Image** and select **Overwrite** if prompted.

Running the Simple Example on the ZCU102 Board

The procedure for setting up the evaluation board is provided in the previous example, [ZCU102 Evaluation Board Setup](#). Copy the BOOT.bin file for the RPU simple example application `<build_path>/rpu_xmpu_example/BOOT.bin` to the SD Card, place the SD card into socket J100, and power the board. If the board is already powered, then cycle PROG_B by pressing SW4.

After completing the initial boot, the PL portion of the fault injection test, also demonstrated in the previous example, runs and displays its output to terminal 1. See the following figure for reference.

Figure 64: Simple XMPU_PL Example Output

```

COM5 - PuTTY
Xilinx Zynq MP First Stage Boot Loader
Release 2019.1 Apr 7 2020 - 10:35:04
---Starting Fault Injection Test (Running on the RPU)---

(S)=Secure, (NS)=Non-Secure, (ND)=Not-Defined
Memories
  PL_BRAM_S_BASE      : BRAM Secure Memory Base Address in PL
  PL_BRAM_NS_SHARED_BASE : BRAM Un-Protected Memory Base Address accessible to ALL Sub-Systems
  PL_BRAM_NS_BASE     : BRAM Non-Secure Memory Base Address only accessible by Un-Secure Sub-Systems
Peripherals
  PL_XMPU_S_LOCK      : Secure XMPU in PL

Read/Write To Memories

  Read/Write To PL(S) Memory
    Reading PL_BRAM_S_BASE      ... PASS!
    Writing PL_BRAM_S_BASE     ... PASS!

  Read/Write To PL(NS) Memory
    Reading PL_BRAM_NS_SHARED_BASE ... PASS!
    Writing PL_BRAM_NS_SHARED_BASE ... PASS!
    Reading PL_BRAM_NS_BASE     ... FAILED!
    Writing PL_BRAM_NS_BASE     ... FAILED!

Reading Sub-System Peripherals

  PL Peripherals
    Reading PL_XMPU_S_START     ... PASS!
    Writing PL_XMPU_S_START     ... PASS!
    Reading PL_GPIO_NS_SHARED_START ... PASS!

  Attempt Unlock PL XMPU
    Read PL_XMPU_S_LOCK        0x00000001
    Write PL_XMPU_S_LOCK       0x00000000
    Read PL_XMPU_S_LOCK        0x00000000

  XMPU PL Interrupts: 2

---Fault Injection Test Complete---
    
```

A Closer Look at the Simple XMPU_PL Example Application

In this example the PL addresses portion of the RPU fault injection test is combined with the initialization, configuration, and management of the XMPU_PL module examples that were previously implemented in the PMU. The `zupl_xmpu` software drivers can be found in the following program:

```
/rpu_example_bsp/psu_cortexr5_0/libsrc/zupl_xmpu_v1_0(SDK)/
zcu102_isolation_test/psu_cortexr5_0/standalone_psu_cortexr5_0/bsp/
psu_cortexr5_0/libsrc/zupl_xmpu_v1_0(Vitis)
```

The source file `pl_xmpu_example.c` includes the declarations shown in the following figure.

- `SetupInterruptSystem` installs the general interrupt controller (GIC) and enables exception handling for interrupts and synchronous data aborts.
- `SAbort_DataAbortHandler` clears the ArmR5 aborts exception, returns the program pointer to the next instruction, and allows the application to continue operation.
- The `readReg` and `writeReg` memory tests use the exception detection to determine PASS/FAIL and prints the result.
- The `XMpuPl_IntrHandler` responds to interrupts triggered by the `zupl_xmpu` core's irq signal. It stores the violation data and clears the interrupt status register.
- The `exceptionDetected` flag is set by `SAbort_DataAbortHandler` and indicates that exception has occurred.
- `XMpuPl_IntrHandler` stores the number of interrupt occurrences in `xmpu_intr` and the status of the most previous interrupt in `xmpu_isr`.

Figure 65: `pl_xmpu_example` declarations

```

/***** Function Prototypes *****/
static int      SetupInterruptSystem(XScuGic *XicInstPtr);
void           SAbort_DataAbortHandler(int);

static void readReg(char registerName[30], u32 registerAddress);
static void writeReg(char registerName[30], u32 registerAddress, u32 regVal);

void XMpuPl_IntrHandler(void * data);
Responds to interrupts triggered by the zupl_xmpu core's irq signal. Stores the
violation data and clears the interrupt status register.

/***** Variables *****/
/* Flag for register test functions */
bool           exceptionDetected = false;

/* Storage for interrupt data */
static u32 xmpu_intr = {0U};
static u32 xmpu_isr = {0U};
    
```

The main (A) begins with instance declarations for the general interrupt controller and XMPU_PL, followed by the `SetupInterruptSystem` function call to set up the interrupt controller and exception handling.

Figure 66: pl_xmpu_example Main (A)

```

/***** MAIN *****/
int main(void)
{
    /* Generic Interrupt Controller Instance */
    XScuGic XicInst;

    /*
     * XmpuPl Instance Array. Supports 1 or more zupl_xmpu cores.
     */
    XmpuPl XMPU_PL_Inst[XMPU_PL_NUM_INST];

    /*
     * Install the generic interrupt system. This configures the GIC and
     * Exception Handlers
     */
    SetupInterruptSystem(&XicInst);
}
    
```

Although this example PL design only implements a single XMPU_PL, the demonstration code declares the XmpuPl instance as an array to support any number of instances, defined by XMPU_PL_NUM_INST in pl_xmpu_example.h:

```
#define XMPU_PL_NUM_INST      XPAR_ZUPL_XMPU_NUM_INSTANCES
```

Note: The zupl_xmpu SW driver supports a maximum of 16 zupl_xmpu instances. Each instance can support a maximum of 16 regions.

The ZUPL_XMPU parameters are defined in xparameters.h:

Figure 67: xparameters.h ZUPL_XMPU Parameters

```

/* Definitions for driver ZUPL_XMPU */
#define XPAR_ZUPL_XMPU_NUM_INSTANCES 1

/* Definitions for peripheral ZUPL_XMPU_0 */
#define XPAR_ZUPL_XMPU_0_DEVICE_ID 0
#define XPAR_ZUPL_XMPU_0_S_AXI_XMPU_BASEADDR 0xA0002000
#define XPAR_ZUPL_XMPU_0_S_AXI_XMPU_HIGHADDR 0xA0002FFF
#define XPAR_ZUPL_XMPU_0_M_AXI_BASEADDR 0xFFFFFFFF
#define XPAR_ZUPL_XMPU_0_M_AXI_HIGHADDR 0x00000000
#define XPAR_ZUPL_XMPU_0_REGIONS_MAX 16
    
```

Initialization of the XMPU_PL instance(s), shown in the following figure, is carried out in a FOR loop. Each instance number represents the Device ID.

Figure 68: pl_xmpu_example Main (B)

```

/*
 * Initialize all XMPU(s) in the PL. This design only contains one, but
 * this example supports multiple.
 */
u32 Status;
XmpuPl *InstancePtr;
u8 XpmuPl_Id = {0U};
for (XpmuPl_Id = 0U; XpmuPl_Id < XMPU_PL_NUM_INST; XpmuPl_Id++) {

    /* Retrieve Base Address of XMPU Device */
    XmpuPl_Config *ConfigPtr = XmpuPl_LookupConfig(XpmuPl_Id);

    /* Assign XMPU Instance Pointer */
    InstancePtr = &XMPU_PL_Inst[XpmuPl_Id];

    /* Initialize XMPU_PL Instance */
    Status = XmpuPl_CfgInitialize(InstancePtr, ConfigPtr,
        ConfigPtr->BaseAddress);
    if (Status!=0U) {
        xil_printf("\n\rERROR: XMPU_PL %d "
            "Config Initialization Failed!\n\r", XpmuPl_Id);
    }

    /* Interrupt ID */
    u16 IntrId = XMPU_PL_INTR_ID + XpmuPl_Id;

    /* Assign Interrupt Handler for XMPU */
    (void)XScuGic_Connect(
        &XicInst,
        IntrId,
        (Xil_ExceptionHandler)XmpuPl_IntrHandler,
        (void*)XMPU_PL_Inst);

    /* Enable the interrupt for the device */
    XScuGic_Enable(&XicInst, IntrId);
}

```

The interrupt ID for instance 0 is defined `pl_xmpu_example.h`

```
#define XMPU_PL_INTR_ID      XPAR_FABRIC_ZUPL_XMPU_0_IRQ_INTR
```

For each instance, the interrupt ID is registered to the `XMpuPl_IntrHandler` function which is passed the starting address of the instance array as its parameter. Since the design only contains a single instance, only instance 0 is configured.

Figure 69: pl_xmpu_example Main (C)

```

/*
 * Configure XmpuPL Inst 0
 */

/* Assign XMPU Instance Pointer */
XpmuPl_Id = 0U;
InstancePtr = &XMPU_PL_Inst[XpmuPl_Id];

/* Configure XMPU_PL CTRL Register */
InstWriteReg(InstancePtr, XMPU_PL_CTRL_OFFSET, XMPU_CTRL_VAL);

/* Select Masters to Bypass LOCK */
InstWriteReg(InstancePtr, XMPU_PL_BYPASS_OFFSET, XMPU_LOCK_MASTERS);

/* Lock XMPU Config Registers */
InstWriteReg(InstancePtr, XMPU_PL_LOCK_OFFSET, 1U);

/* Enable XMPU Interrupts */
XmpuPl_EnableInterrupts(InstancePtr, XMPU_INT_EN);

/* Add REGION 0 */
Status = XmpuPl_AddRegion(InstancePtr,
                          REGION_0_ADDR, 1U, REGION_0_MASTERS, REGION_0_CFG);
if (Status != 0U) {
    xil_printf("\n\rXMPU Add Region 0 Failed!\n\r");
}

/* Add REGION 1 */
Status = XmpuPl_AddRegion(InstancePtr,
                          REGION_1_ADDR, 1U, REGION_1_MASTERS, REGION_1_CFG);
if (Status != 0U) {
    xil_printf("\n\rXMPU Add Region 1 Failed!\n\r");
}

/* Update XmpuPl Instance */
Status = XmpuPl_GetConfig(InstancePtr);
if (Status != 0U) {
    xil_printf("\n\rXMPU Get Config Failed!\n\r");
}
    
```

The CTRL register is configured with default read allowed, default write allowed, poison attribute and poison address enabled, and poisoned AXI response DECERR, by XMPU_CTRL_VAL defined in the following pl_xmpu_example.h::

```

#define XMPU_CTRL_VAL ( XMPU_PL_CTRL_DEFRD \
                       | XMPU_PL_CTRL_DEFWR \
                       | XMPU_PL_CTRL_PSNATTREN \
                       | XMPU_PL_CTRL_PSNADDREN \
                       | XMPU_PL_CTRL_ARSP_DEC )
    
```

The defined register offsets and configuration options are found in the zupl_xmpu SW driver file zupl_xmpu_hw.h. The LOCK BYPASS register configuration allows the PMU and RPU0 to have write access after the LOCK is enabled.

```

#define XMPU_LOCK_MASTERS ( XMPU_PL_MID_PMU | XMPU_PL_MID_RPU0 )
    
```

Read and write violations are enabled interrupts by XMPU_INT_EN.

```
#define XMPU_INT_EN          ( XMPU_PL_IXR_WRVIO_MSK \
| XMPU_PL_IXR_RDVIO_MSK )
```

Region 0 is set to a 1 KB size starting at the base of the secure (S) BRAM area, and configured with the following parameters:

```
#define REGION_0_ADDR        PL_BRAM_S_BASE
#define REGION_0_MASTERS    ( XMPU_PL_MID_RPU0 )
#define REGION_0_CFG        ( XMPU_PL_REGION_WR_ALLOW \
| XMPU_PL_REGION_RD_ALLOW \
| XMPU_PL_REGION_ENABLE )
```

Only RPU0 has read and write privileges.

Region 1 is also set to a 1 KB size starting at the base of the non-secure (NS) BRAM area, and configured with the following parameters:

```
#define REGION_1_ADDR        PL_BRAM_NS_BASE
#define REGION_1_MASTERS    ( XMPU_PL_MID_APU )
#define REGION_1_CFG        ( XMPU_PL_REGION_WR_ALLOW \
| XMPU_PL_REGION_RD_ALLOW \
| XMPU_PL_REGION_ENABLE )
```

Only the APU has read and write privileges. PL_BRAM_NS_SHARED is set to an address between region 0 end and region 1 start. A region miss falls to the default settings specified in the CTRL registers that gives read and write access to all masters making the memory space shared.

The rest of main () runs the read/write tests and finally prints the number of interrupts recorded by the interrupt handler, XMpuPI_IntrHandler, shown in the following figure. In this example, one interrupt handler is shared by all instances. The interrupt status register of each instance is checked until an active violation is found. The interrupt status is stored, the number of interrupts is incremented, and then the interrupt status is cleared. If there is more than one instance issuing an interrupt, the handler gets recalled until all interrupts are cleared.

Figure 70: pl_xmpu_example XMpuPI_IntrHandler

```

/***** Interrupt Handler *****/
void XMpuPI_IntrHandler(void * data)
{
    /* Variables */
    u8 exit_loop = {0U};
    u32 reg_isr = {0U};
    XmpuPI *XMPU_PL_Ptr = (XmpuPI *)data;

    /* Search XMPU Instances for Interrupt Status */
    for (int i=0; i<XMPU_PL_NUM_INST; i++) {
        /* NULL Check */
        if (XMPU_PL_Ptr != NULL) {
            /* Get ISR Status */
            reg_isr = XMpuPI_GetInterruptStatus(XMPU_PL_Ptr);
            if (reg_isr!=0U) {

                /* Store event in static variable */
                xmpu_isr = reg_isr;
                xmpu_intr++;

                /* Clear ISR */
                XMpuPI_ClearInterruptStatus(XMPU_PL_Ptr, reg_isr);
                reg_isr = XMpuPI_GetInterruptStatus(XMPU_PL_Ptr);
                exit_loop = 1U;
            }
        } else {
            exit_loop = 1U;
            xil_printf("\n\rXMPU_PL Handler: NULL Pointer! ");
        }
        /* Exit or Continue */
        if (exit_loop) {
            break;
        } else {
            XMPU_PL_Ptr++;
        }
    }

    if (reg_isr!=0U) {
        xil_printf("\n\rXMPU_PL Handler: ISR Clear Failure! ");
        xil_printf("\n\rISR 0x%08X \n\r", reg_isr);
    }
}
    
```

This is an example of one way a designer chooses to configure and handle the `zupl_xmpu_v1_0` core. Additionally, you can add multiple instances into the PL design and add their configurations to this application. This is left for you as an exercise.

Conclusion

The `zupl_xmpu_v1_0` bridges PL and PS security and isolation for AXI based embedded designs in Zynq UltraScale+ devices. The following appendix provides the Master ID list and SW driver details.

Appendix A: Master ID List

Table 25: PS Master IDs

Master	ID	Mask	Master	ID	Mask
MID_RPU0	x"0000"	x"03F0"	MID_GPU	x"00C4"	x"03FF"
MID_RPU1	x"0010"	x"03F0"	MID_DAP_AXI	x"00C5"	x"03FF"
MID_PMU	x"0040"	x"03FF"	MID_PCIE	x"00D0"	x"03FF"
MID_USB0	x"0060"	x"03FF"	MID_DP_DMA0	x"00E0"	x"03FE"
MID_USB1	x"0061"	x"03FF"	MID_DP_DMA1	x"00E1"	x"03FE"
MID_DAP_APB	x"0062"	x"03FF"	MID_DP_DMA2	x"00E2"	x"03FE"
MID_LPD_DMA0	x"0068"	x"03FE"	MID_DP_DMA3	x"00E3"	x"03FE"
MID_LPD_DMA1	x"0069"	x"03FE"	MID_DP_DMA4	x"00E4"	x"03FE"
MID_LPD_DMA2	x"006A"	x"03FE"	MID_DP_DMA5	x"00E5"	x"03FE"
MID_LPD_DMA2	x"03FB"	x"03FE"	MID_FPD_DMA0	x"00E8"	x"03FE"
MID_LPD_DMA4	x"006C"	x"03FE"	MID_FPD_DMA1	x"00E9"	x"03FE"
MID_LPD_DMA5	x"006D"	x"03FE"	MID_FPD_DMA2	x"00EA"	x"03FE"
MID_LPD_DMA6	x"006E"	x"03FE"	MID_FPD_DMA3	x"00EB"	x"03FE"
MID_LPD_DMA7	x"006F"	x"03FE"	MID_FPD_DMA4	x"00EC"	x"03FE"
MID_SD0	x"0070"	x"03FF"	MID_FPD_DMA5	x"00ED"	x"03FE"
MID_SD1	x"0071"	x"03FF"	MID_FPD_DMA6	x"00EE"	x"03FE"
MID_NAND	x"0072"	x"03FF"	MID_FPD_DMA7	x"00EF"	x"03FE"
MID_QSPI	x"0073"	x"03FF"	MID_HPC0_FPD	x"0200"	x"03C0"
MID_GEM0	x"0074"	x"03FF"	MID_HPC1_FPD	x"0240"	x"03C0"
MID_GEM1	x"0075"	x"03FF"	MID_HP0_FPD	x"0280"	x"03C0"
MID_GEM2	x"0076"	x"03FF"	MID_HP1_FPD	x"02C0"	x"03C0"
MID_GEM3	x"0077"	x"03FF"	MID_HP2_FPD	x"0300"	x"03C0"
MID_APU	x"0080"	x"03FF"	MID_HP3_LPD	x"0340"	x"03C0"
MID_APU	x"00C0"	x"03C0"	MID_PL_LPD	x"0380"	x"03C0"
MID_SATA1	x"00C1"	x"03FF"	MID_ACE_FPD	x"03C0"	x"03C0"

Appendix B: SW Driver Library

Overview

The `zupl_xmpu` driver provides standard C functions and macros for Zynq UltraScale+ MPSoC PS and PL processor applications that initializes, configures, and manages the `XMPU_PL` memory and peripheral protection unit implemented by the `zupl_xmpu_v1_0` reference core.

The `zupl_xmpu_v1_0` source and include directories contain the files shown in the following figure:

Figure 71: ZUPL_XMPU SW Driver Files

Source	Header	Description
	xparameters.h	Exported Device Parameters (example): <pre> /* Definitions for Fabric interrupts connected to psu_acpu_gic */ #define XPAR_FABRIC_ZUPL_XMPU_0_IRQ_INTR 121U /* Definitions for driver ZUPL_XMPU */ #define XPAR_ZUPL_XMPU_NUM_INSTANCES 1 /* Definitions for peripheral ZUPL_XMPU_0 */ #define XPAR_ZUPL_XMPU_0_DEVICE_ID 0 #define XPAR_ZUPL_XMPU_0_S_AXI_XMPU_BASEADDR 0xA0002000 #define XPAR_ZUPL_XMPU_0_S_AXI_XMPU_HIGHADDR 0xA0002FFF #define XPAR_ZUPL_XMPU_0_M_AXI_IN_BASEADDR 0xFFFFFFFF #define XPAR_ZUPL_XMPU_0_M_AXI_IN_HIGHADDR 0x00000000 #define XPAR_ZUPL_XMPU_0_REGIONS_MAX 16 </pre>
zupl_xmpu.c	zupl_xmpu.h	Device instance data structs; Device user utilities (operation)
zupl_xmpu_selftest.c		SelfTest Function
	zupl_xmpu_hw.h	Register address offsets, control and interrupt data masks.
zupl_xmpu_sinit.c	zupl_xmpu_sinit.h	Config initialization table data; Device user utilities (config)
zupl_xmpu_g.c		Boot initialization of config table

Structs

XmpuPl_Config Struct

The XmpuPl_Config struct passes exported device parameters.

```

typedef struct {
    u16 DeviceId;          /**< Unique ID for device */
    u32 BaseAddress;      /**< Base address for device */
    u32 M_Axi_BaseAddress; /**< Base Address for Protected Master */
    u32 M_Axi_HighAddress; /**< Base Address for Protected Master */
    u32 MaxRegions;      /**< Maximum allowed Regions for device */
} XmpuPl_Config;
    
```

XmpuPl_Regions Struct

The XmpuPl_Regions struct stores a copy of region configuration register values.

```

typedef struct {
    u64 Start;
    u64 End;
    u32 Masters;
    u32 Config;
} XmpuPl_Regions;
    
```

XmpuPl_Regs Struct

The XmpuPl_Regs struct stores a copy of device instance register values. This includes XmpuPl_Regions.

```
typedef struct {
    u32 CTRL;
    u32 POISON;
    u32 IMR;
    u32 LOCK;
    u32 BYPASS;
    u32 REGIONS;
    XmpuPl_Regions Region_Regs[16U];
} XmpuPl_Regs;
```

XmpuPl Struct

The XmpuPl struct stores and passes all device instance register, configuration, and exported values. This includes XmpuPl_Config and XmpuPl_Regs.

```
typedef struct {
    XmpuPl_Config Config;          /**< Configuration structure */
    XmpuPl_Regs Regs;
    u32 IsReady;                  /**< Device is initialized and ready */
} XmpuPl;
```

Functions

XMpuPl_LookupConfig

```
XmpuPl_Config *XMpuPl_LookupConfig(u16 DeviceId);
```

This searches the XMpuPlInst_ConfigTable for the device configuration based on the unique device ID, and returns a pointer to the element at the associated table index.

Parameters

- **DeviceId:** DeviceId contains the unique ID of the device

Return

- **XmpuPl_Config *:** Pointer to XMpuPlInst_ConfigTable element

XMpuPl_CfgInitialize

```
u32 XMpuPl_CfgInitialize(XmpuPl *InstancePtr, XmpuPl_Config *ConfigPtr, u32 EffectiveAddr);
```

This initializes the XMpuPl Instance Configuration

Parameters

- **InstancePtr *:** Pointer to XmpuPl instance

- **XmpuPl_Config ***: Pointer to XMpuPlInst_ConfigTable element
- **EffectiveAddr**: Base address of the device. This is typically set to `XmpuPl_Config -> BaseAddress`, but is also used for system address mapping.

Return

- **Status**: Function execution status: 0U Success; 1U Error.

XMpuPI_IsActive

```
u32 XMpuPl_IsActive(XmpuPl *InstancePtr);
```

Note: This checks if the device has been configured.

Parameters

- **InstancePtr ***: Pointer to XmpuPI instance

Return

- **Status**: Instance configuration status: 0U Active; 1U Unconfigured.

XMpuPI_AddRegion

```
u32 XMpuPl_AddRegion(XmpuPl *InstancePtr, u64 start, u32 size, u32 masters, u32 config);
```

This configures a protected address region in to the next available region.

Parameters

- **InstancePtr ***: Pointer to XmpuPI instance
- **Start**: Upper 32 bits of a 40-bit starting address for the region.
- **Size**: Size of the region in KB(s)
- **Masters**: Value written to R[n]_MASTERS register. Each bit authorizes a PS Master.
- **Config**: Value written to R[n]_CONFIG register.

Return

- **Status**: Function execution status: 0U Success; 1U Error.

XMpuPL_GetConfig

```
u32 XMpuPl_GetConfig(XmpuPl *InstancePtr);
```

This loads all device and region configuration data into instance.

Parameters

- **InstancePtr ***: Pointer to XmpuPl instance

Return

- **Status:** Function execution status: 0U Success; 1U Error.

XMpuPL_SelfTest

```
u32 XMpuPL_SelfTest(XmpuPl *InstancePtr);
```

This runs a read and write self-test on the device.

Parameters

- **InstancePtr ***: Pointer to XmpuPl instance

Return

- **Status:** Function execution status: 0U Success; 1U Error.

Macros

InstReadReg

```
#define InstReadReg(InstancePtr, RegOffset) \
    (Xil_In32(((InstancePtr)->Config.BaseAddress) + (u32) \
    (RegOffset)))
```

This returns the value of the selected device register.

Parameters

- **InstancePtr ***: Pointer to XmpuPl instance
- **RegOffset:** Use register offset values provided in `zupl_xmpu_hw.h`

Return

- Returns register value.

InstWriteReg

```
#define InstWriteReg(InstancePtr, RegOffset, Data) \
    (Xil_Out32(((InstancePtr)->Config.BaseAddress) + (u32) (RegOffset), \
    (u32) (Data)))
```

This writes the value to the selected device register.

Parameters

- **InstancePtr***: Pointer to XmpuPl instance

- **RegOffset:** Use register offset values provided in `zupl_xmpu_hw.h`
- **Data:** Value to be written to register

Return

- **None:** none

XMpuPI_EnableInterrupts

```
#define XMpuPl_EnableInterrupts(InstancePtr, InterruptMask) \
    InstWriteReg((InstancePtr), XMPU_PL_IER_OFFSET, \
    (InstReadReg((InstancePtr), XMPU_PL_IER_OFFSET) | \
    (InterruptMask)))
```

This enables the selected interrupts. The unselected interrupts maintain their current settings.

Parameters

- **InstancePtr *:** Pointer to XmpuPI instance
- **InterruptMask :** Use interrupt mask values provided in `zupl_xmpu_hw.h`

Return

- **None:** none

XMpuPI_DisableInterrupts

```
#define XMpuPl_DisableInterrupts(InstancePtr, InterruptMask) \
    InstWriteReg((InstancePtr), XMPU_PL_IDS_OFFSET, \
    (~InstReadReg((InstancePtr), XMPU_PL_IMR_OFFSET) & \
    (InterruptMask)))
```

This disables the selected interrupts. The unselected interrupts maintains their current settings.

Parameters

- **InstancePtr *:** Pointer to XmpuPI instance
- **InterruptMask:** Use interrupt mask values provided in `zupl_xmpu_hw.h`

Return

- **None:** none

XMpuPI_GetInterruptStatus

```
#define XMpuPl_GetInterruptStatus(InstancePtr) \
    InstReadReg((InstancePtr), XMPU_PL_ISR_OFFSET)
```

This returns the value of the interrupt status register.

Parameters

- **InstancePtr ***: Pointer to XmpuPl instance

Return

- **Return Status:** Returns ISR register value

XMpuPl_ClearInterruptStatus

```
#define XmpuPl_ClearInterruptStatus(InstancePtr, InterruptMask) \
    InstWriteReg((InstancePtr), XMPU_PL_ISR_OFFSET, (InterruptMask))
```

This clears the selected interrupts. The unselected interrupts maintain their current settings.

Parameters

- **InstancePtr ***: Pointer to XmpuPl instance
- **InterruptMask:** Use interrupt mask values provided in `zupl_xmpu_hw.h`

Return

- **Return Status:** None

Constants

```
/*REGISTER OFFSETS*/
#define XMPU_PL_CTRL_OFFSET          0x0U
#define XMPU_PL_ERRS1_OFFSET        0x4U
#define XMPU_PL_ERRS2_OFFSET        0x8U
#define XMPU_PL_POISON_OFFSET       0xCU
#define XMPU_PL_ISR_OFFSET          0x10U
#define XMPU_PL_IMR_OFFSET          0x14U
#define XMPU_PL_IER_OFFSET          0x18U
#define XMPU_PL_IDS_OFFSET          0x1CU
#define XMPU_PL_LOCK_OFFSET         0x20U
#define XMPU_PL_BYPASS_OFFSET       0x24U
#define XMPU_PL_REGIONS_OFFSET      0x28U
#define XMPU_PL_R00_START_OFFSET    0x100U
#define XMPU_PL_R00_END_OFFSET      0x104U
#define XMPU_PL_R00_MASTERS_OFFSET  0x108U
#define XMPU_PL_R00_CONFIG_OFFSET   0x10CU
#define XMPU_PL_R01_START_OFFSET    0x110U
#define XMPU_PL_R01_END_OFFSET      0x114U
#define XMPU_PL_R01_MASTERS_OFFSET  0x118U
#define XMPU_PL_R01_CONFIG_OFFSET   0x11CU
#define XMPU_PL_R02_START_OFFSET    0x120U
#define XMPU_PL_R02_END_OFFSET      0x124U
#define XMPU_PL_R02_MASTERS_OFFSET  0x128U
#define XMPU_PL_R02_CONFIG_OFFSET   0x12CU
#define XMPU_PL_R03_START_OFFSET    0x130U
#define XMPU_PL_R03_END_OFFSET      0x134U
#define XMPU_PL_R03_MASTERS_OFFSET  0x138U
#define XMPU_PL_R03_CONFIG_OFFSET   0x13CU
#define XMPU_PL_R04_START_OFFSET    0x140U
#define XMPU_PL_R04_END_OFFSET      0x144U
#define XMPU_PL_R04_MASTERS_OFFSET  0x148U
#define XMPU_PL_R04_CONFIG_OFFSET   0x14CU
#define XMPU_PL_R05_START_OFFSET    0x150U
#define XMPU_PL_R05_END_OFFSET      0x154U
#define XMPU_PL_R05_MASTERS_OFFSET  0x158U
```

```

#define XMPU_PL_R05_CONFIG_OFFSET    0x15CU
#define XMPU_PL_R06_START_OFFSET    0x160U
#define XMPU_PL_R06_END_OFFSET      0x164U
#define XMPU_PL_R06_MASTERS_OFFSET  0x168U
#define XMPU_PL_R06_CONFIG_OFFSET   0x16CU
#define XMPU_PL_R07_START_OFFSET    0x170U
#define XMPU_PL_R07_END_OFFSET      0x174U
#define XMPU_PL_R07_MASTERS_OFFSET  0x178U
#define XMPU_PL_R07_CONFIG_OFFSET   0x17CU
#define XMPU_PL_R08_START_OFFSET    0x180U
#define XMPU_PL_R08_END_OFFSET      0x184U
#define XMPU_PL_R08_MASTERS_OFFSET  0x188U
#define XMPU_PL_R08_CONFIG_OFFSET   0x18CU
#define XMPU_PL_R09_START_OFFSET    0x190U
#define XMPU_PL_R09_END_OFFSET      0x194U
#define XMPU_PL_R09_MASTERS_OFFSET  0x198U
#define XMPU_PL_R09_CONFIG_OFFSET   0x19CU
#define XMPU_PL_R10_START_OFFSET    0x1A0U
#define XMPU_PL_R10_END_OFFSET      0x1A4U
#define XMPU_PL_R10_MASTERS_OFFSET  0x1A8U
#define XMPU_PL_R10_CONFIG_OFFSET   0x1ACU
#define XMPU_PL_R11_START_OFFSET    0x1B0U
#define XMPU_PL_R11_END_OFFSET      0x1B4U
#define XMPU_PL_R11_MASTERS_OFFSET  0x1B8U
#define XMPU_PL_R11_CONFIG_OFFSET   0x1BCU
#define XMPU_PL_R12_START_OFFSET    0x1C0U
#define XMPU_PL_R12_END_OFFSET      0x1C4U
#define XMPU_PL_R12_MASTERS_OFFSET  0x1C8U
#define XMPU_PL_R12_CONFIG_OFFSET   0x1CCU
#define XMPU_PL_R13_START_OFFSET    0x1D0U
#define XMPU_PL_R13_END_OFFSET      0x1D4U
#define XMPU_PL_R13_MASTERS_OFFSET  0x1D8U
#define XMPU_PL_R13_CONFIG_OFFSET   0x1DCU
#define XMPU_PL_R14_START_OFFSET    0x1E0U
#define XMPU_PL_R14_END_OFFSET      0x1E4U
#define XMPU_PL_R14_MASTERS_OFFSET  0x1E8U
#define XMPU_PL_R14_CONFIG_OFFSET   0x1ECU
#define XMPU_PL_R15_START_OFFSET    0x1F0U
#define XMPU_PL_R15_END_OFFSET      0x1F4U
#define XMPU_PL_R15_MASTERS_OFFSET  0x1F8U
#define XMPU_PL_R15_CONFIG_OFFSET   0x1FCU
    
```

/*CONTROL REGISTER*/

```

#define XMPU_PL_CTRL_DEFRD          0x00000001U
#define XMPU_PL_CTRL_DEFWR          0x00000002U
#define XMPU_PL_CTRL_PSNADDREN      0x00000004U
#define XMPU_PL_CTRL_PSNATTREN      0x00000008U
#define XMPU_PL_CTRL_EXTSINKEN      0x00000010U
#define XMPU_PL_CTRL_ARSP_OKA       0x00000000U
#define XMPU_PL_CTRL_ARSP_EXO       0x00000020U
#define XMPU_PL_CTRL_ARSP_SLV       0x00000040U
#define XMPU_PL_CTRL_ARSP_DEC       0x00000060U
#define XMPU_PL_CTRL_DEFRD_MSK      0x00000001U
#define XMPU_PL_CTRL_DEFWR_MSK      0x00000002U
#define XMPU_PL_CTRL_PSNADDREN_MSK  0x00000004U
#define XMPU_PL_CTRL_PSNATTREN_MSK  0x00000008U
#define XMPU_PL_CTRL_EXTSINKEN_MSK  0x00000010U
#define XMPU_PL_CTRL_ARSP_MSK       0x00000060U
#define XMPU_PL_CTRL_ADDRHIGH_MSK   0x00FF0000U
    
```

/*MASTERS*/

```

#define XMPU_PL_MID_FPD_DMA_6_7     (1U << 30U)
#define XMPU_PL_MID_FPD_DMA_4_5     (1U << 29U)
#define XMPU_PL_MID_FPD_DMA_2_3     (1U << 28U)
#define XMPU_PL_MID_FPD_DMA_0_1     (1U << 27U)
#define XMPU_PL_MID_DP_DMA_4_5      (1U << 26U)
    
```

```

#define XMPU_PL_MID_DP_DMA_2_3      (1U << 25U)
#define XMPU_PL_MID_DP_DMA_0_1      (1U << 24U)
#define XMPU_PL_MID_PCIE            (1U << 23U)
#define XMPU_PL_MID_DAP_AXI        (1U << 22U)
#define XMPU_PL_MID_GPU            (1U << 21U)
#define XMPU_PL_MID_SATA1          (1U << 20U)
#define XMPU_PL_MID_SATA0          (1U << 19U)
#define XMPU_PL_MID_APU            (1U << 18U)
#define XMPU_PL_MID_GEM3           (1U << 17U)
#define XMPU_PL_MID_GEM2           (1U << 16U)
#define XMPU_PL_MID_GEM1           (1U << 15U)
#define XMPU_PL_MID_GEM0           (1U << 14U)
#define XMPU_PL_MID_QSPI           (1U << 13U)
#define XMPU_PL_MID_NAND           (1U << 12U)
#define XMPU_PL_MID_SD1            (1U << 11U)
#define XMPU_PL_MID_SD0            (1U << 10U)
#define XMPU_PL_MID_LPD_DMA_6_7    (1U << 9U)
#define XMPU_PL_MID_LPD_DMA_4_5    (1U << 8U)
#define XMPU_PL_MID_LPD_DMA_2_3    (1U << 7U)
#define XMPU_PL_MID_LPD_DMA_0_1    (1U << 6U)
#define XMPU_PL_MID_DAP_APB        (1U << 5U)
#define XMPU_PL_MID_USB1           (1U << 4U)
#define XMPU_PL_MID_USB0           (1U << 3U)
#define XMPU_PL_MID_PMU            (1U << 2U)
#define XMPU_PL_MID_RPU1           (1U << 1U)
#define XMPU_PL_MID_RPU0           (1U << 0U)
    
```

```

/*REGION CONFIGURATION*/
#define XMPU_PL_REGION_ENABLE      0x00000001U
#define XMPU_PL_REGION_RD_ALLOW    0x00000002U
#define XMPU_PL_REGION_WR_ALLOW    0x00000004U
#define XMPU_PL_REGION_REGIONNS    0x00000008U
#define XMPU_PL_REGION_NSCHCK      0x00000010U
#define XMPU_PL_REGION_MIDDISABLE  0x00000020U
    
```

```

/*INTERRUPTS*/
#define XMPU_PL_I XR_RDVIO_MSK      0x00000002U /* RdPermVIO Interrupt */
#define XMPU_PL_I XR_WRVIO_MSK      0x00000004U /* WrPermVIO Interrupt */
#define XMPU_PL_I XR_SECVIO_MSK     0x00000008U /* SecurityVIO Interrupt */
*/
    
```

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
01/14/2021 Version 1.0	
Initial release.	N/A

References

These documents provide supplemental material useful with this guide:

1. *Zynq UltraScale+ Device Technical Reference Manual* ([UG1085](#))
2. *Isolation Methods in Zynq UltraScale+ MPSoCs* ([XAPP1320](#))
3. *ZCU102 Evaluation Board User Guide* ([UG1182](#))

4. *Zynq UltraScale+ Device Register Reference* ([UG1087](#))
5. *Zynq UltraScale+ MPSoC: Software Developers Guide* ([UG1137](#))

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see [Xilinx Support](#).

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open 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 DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the [Design Hubs](#) page.

Note: For more information on DocNav, see the [Documentation Navigator](#) page on the Xilinx website.

Please Read: Important Legal Notices

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

© Copyright 2021 Xilinx, Inc. Xilinx, the Xilinx logo, Alveo, Artix, Kintex, Spartan, Versal, Virtex, Vivado, Zynq, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. AMBA, AMBA Designer, Arm, ARM1176JZ-S, CoreSight, Cortex, PrimeCell, Mali, and MPCore are trademarks of Arm Limited in the EU and other countries. PCI, PCIe, and PCI Express are trademarks of PCI-SIG and used under license. All other trademarks are the property of their respective owners.