PCI Express

Contents

Chapter 1. Introduction

PCI Express (Peripheral Component Interconnect Express), commonly known as PCIe, is a high-speed serial computer expansion bus standard used for connecting various hardware devices to a computer's motherboard.

Features:

- Backward compatible: PCIe accommodates devices from earlier versions for use in newer slots, operating at the speed of the specific slot.
- Direct device-to-CPU connections via dedicated lanes, unlike PCI's shared bus.
- Facilitates simultaneous two-way data transmission, enhancing transfer efficiency with bidirectional communication.
- PCIe supports hot-plugging, allowing devices to be connected or disconnected without shutting down the system.
- PCIe utilizes separate data lanes for transfer, offering scalability with configurations like x1, x4, x8, and x16 to achieve varied bandwidths.
- Scalable: Lanes can be combined for higher bandwidth.

Chapter 2. Operation of PCIe - How does PCIe function?

Here's a brief overview of how PCIe functions, highlighting its significance in modern computing and the efficiency it brings to data transfer.

1. PCIe initializes devices during boot, allocates resources, and loads drivers.
2. Link training negotiates data lanes and establishes a reliable communication link.
3. Data transfer occurs through packets, managed by Transaction and Data Link layers.
4. Flow control regulates data pace, and clock synchronization uses Reference Clock (RefCLK).
5. Devices independently initiate data transfers using Transaction Layer Packets (TLPs).
6. Bus arbitration mechanisms ensure orderly access to the shared communication pathway.
7. PCIe includes error handling with detection and correction mechanisms.
8. Buffers may temporarily store data during transmission for smooth communication.

PCIe operates by establishing a point-to-point communication architecture, where devices connect directly to the CPU or chipset through dedicated data lanes. The system utilizes a combination of transaction layers, data link layers, clock signals, and error-handling mechanisms to enable efficient and reliable data transfer between devices. The architecture is designed for scalability, allowing for the connection of various components in modern computer systems.

Chapter 3. Interrupt Request (IRQ)

In a PCIe (Peripheral Component Interconnect Express) system, interrupts play a crucial role in facilitating communication between the CPU and PCIe devices. PCIe uses interrupts to notify the CPU about specific events or conditions that require attention, such as the completion of a data transfer or the need for data processing.

Typyes of Interrupts:

1. Supported interrupts (Endpoint) EP Mode:

   In EP mode, the core supports Legacy PCI INTx, MSI, and MSI-X interrupts.

   a. Legacy PCI INTx: It is type of interrupt signaling mechanism used by legacy PCI devices to request attention from the CPU. Legacy PCI devices use INTx interrupts as a way to inform the system (specifically the CPU) that they require attention. These interrupts are associated with specific PCI pins and are typically referred to as INTA#, INTB#, INTC#, and INTD#, where the "#" denotes a specific interrupt line associated with a device.
   b. MSI: MSI-capable devices deliver interrupts by performing memory write transactions. Your application logic makes MSI requests through the MSI interface; the core then generates the corresponding memory write. MSIs are always subject to translation by the internal address translation unit (iATU).
   c. MSI-X extends the capabilities of MSI by allowing more flexible interrupt routing and servicing. It provides a dedicated mechanism for handling interrupts efficiently, and the configuration involves setting up tables and registers in system memory. The application communicates with the core using specific input ports, distinguishing between shared MSI and dedicated MSI-X operations.

Chapter 4. Operation of Interrupts

Interrupts in computing play a crucial role in handling events that require immediate attention. Here's a simplified overview of how interrupts operate:

1. Triggering Event: An external event or condition occurs, such as a hardware device completing an operation or a specific time interval elapsing
2. Interrupt Request (IRQ): The triggering event generates an interrupt request, signaling the need for the CPU's attention.
3. Interrupt Controller: The CPU receives the IRQ and communicates with the interrupt controller, a hardware component responsible for managing multiple interrupt sources.
4. Interrupt Handling: The interrupt controller prioritizes and directs the CPU to the corresponding interrupt service routine (ISR), a predefined piece of code associated with the specific IRQ.
5. Context Switch: The CPU temporarily suspends the current task, saving its state, and shifts to the ISR to address the interrupt.
6. Interrupt Service Routine: The ISR executes the necessary actions to respond to the triggering event. This could involve processing data, updating registers, or initiating further actions.
7. Return from Interrupt: After completing the ISR, the CPU restores the saved state of the interrupted task through a context switch.
8. Resumption of Task: The CPU resumes the interrupted task, continuing its normal operation.

Receiving MSI Interrupts with AHB/ AXI Bridge Integration: User-Guide

Contents

Chapter 1. Introduction

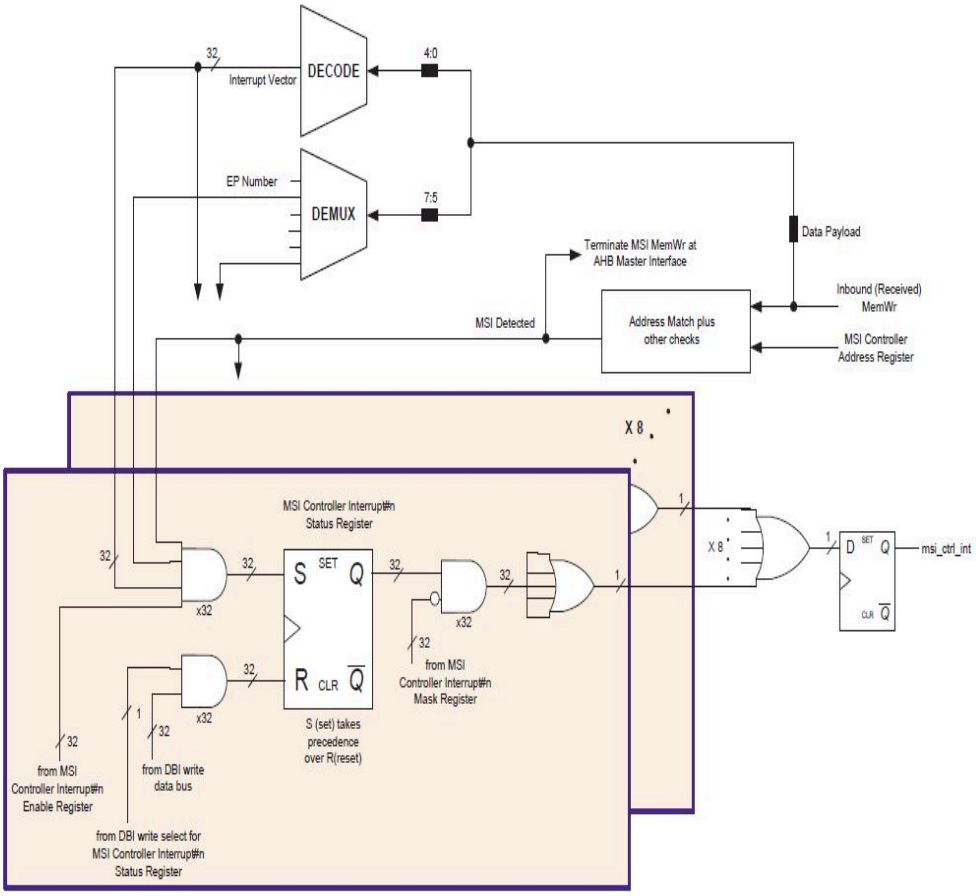
The AHB (Advanced High-Performance Bus) / AXI (Advanced eXtensible Interface) bridge facilitates data transfers in digital systems. This guide focuses on Memory Write (MWr) transactions, specifically addressing the reception of MSI (Message Signaled Interrupt) requests. The standard AHB/AXI bridge handles MSI requests similarly to MWr transactions. However, the bridge doesn't inherently distinguish between MSI and MWr requests, and the termination of an MSI request must be managed by the application or by utilizing an optional MSI controller.

Chapter 2. MSI Request Detection in AXI Bridge

An MSI interrupt request is detected when the core receives a MWr TLP that satisfies the following conditions:

- Header attributes bits are zero. No snoop (NS) and relaxed ordering (RO) must be zero.
- Length field is 0x01 to indicate a payload of one DWORD.
- First byte enable (FBE) is 4'bxx11.
- Last byte enable (LBE) is 4'b0000.
- TLP address corresponds to system's chosen MSI address as programmed in the "MSI Controller Address Register" `(MSI_CTRL_ADDR_REG and MSI_CTRL_UPPER_ADDR_REG)` This register is not the "MSI Lower 32 Bits Address Register" which is part of the PCI Express MSI capability register structure.

Figure 1. MSI Controller Interrupt Detection Process



Chapter 3. Receiving MSI Interrupts with AXI Bridge Integration

This comprehensive guide outlines the detailed steps for configuring and receiving MSI (Message Signaled Interrupt) interrupts using the AXI (Advanced eXtensible Interface) bridge in Root Complex (RC) mode.

Before proceeding with the MSI interrupt configuration, ensure the following:

- Knowledge of PCI Express architecture and the role of MSI in signaling interrupts.
- Familiarity with the AHB/AXI bridge and its modes (RC and EP).
- Access to the relevant documentation for your AHB/AXI bridge.

Configuring MSI Interrupt Reception through the AXI Bridge:

1. Program the "MSI Data Register" `MSI_DATA_` of each Endpoint (EP) to allow the MSI interrupt controller to decode the interrupt source.
2. Program the "MSI Lower 32 Bits Address Register" `MSI_LOWER_32` of every EP with one common MSI address.
3. Program the common MSI address used for the EPs into the interrupt controller's "MSI Controller Address Register" `MSI_CTRL_ADDR_REG`.
4. Read the MSI capability of each EP to determine the number of vectors enabled in each EP. Use this information to program the interrupt enable registers in the MSI interrupt controller.
   The interrupt enable register allows up to 32 MSI interrupt vectors to be enabled within the MSI interrupt controller for a given EP.
5. Program the interrupt enable register based on the "Multiple Message Enable" field in an EP's MSI capability structure.
6. Set the corresponding bit in the `MSI_CTRL_INT_0_STATUS_REG` to indicate that the vector is enabled and not masked.
7. Assert the top-level core output `msi_ctrl_int.`This signal remains asserted until cleared by the host CPU.
8. Verify the successful reception of MSI interrupts by monitoring the relevant status registers and confirming proper functionality.
9. Integrate the configured MSI interrupt reception into your application logic running on the system-on-chip

Integrating Multiple Message Signaled Interrupts (MSI) with an AXI bridge involves configuring registers for MSI data, address, and control. Program the interrupt enable registers based on endpoint capabilities,

assert the top-level core output signal upon receiving an interrupt, and ensure proper functionality through verification. Consult hardware documentation for accurate details and implement error handling for a robust system.

Sending an MSI/MSI-X Interrupt Using the MSI/MSI-X Interface

Contents

Chapter 1. MSI/MSI-X Interface

The Message Signaled Interrupt (MSI) interface is a method used in computer systems to deliver interrupt messages from peripheral devices to the CPU. Message Signaled Interrupt eXtended (MSI-X) is an extension of the original Message Signaled Interrupt (MSI) technology. Both MSI and MSI-X are methods used for devices to generate interrupts and communicate them to the CPU in a more efficient and scalable manner compared to traditional interrupt mechanisms. MSI-X further extends the capabilities of MSI, providing additional features and flexibility.

Chapter 2. Steps to Send an MSI Interrupt Using the MSI Interface

The MSI Interface enables an application to request the core to send an MSI. The MSI interface protocol involves a simple synchronous request/acknowledge handshake.

- Ensure that the MSI interface is exclusively used for upstream ports.

  MSI Interrupt Transmission Procedure:

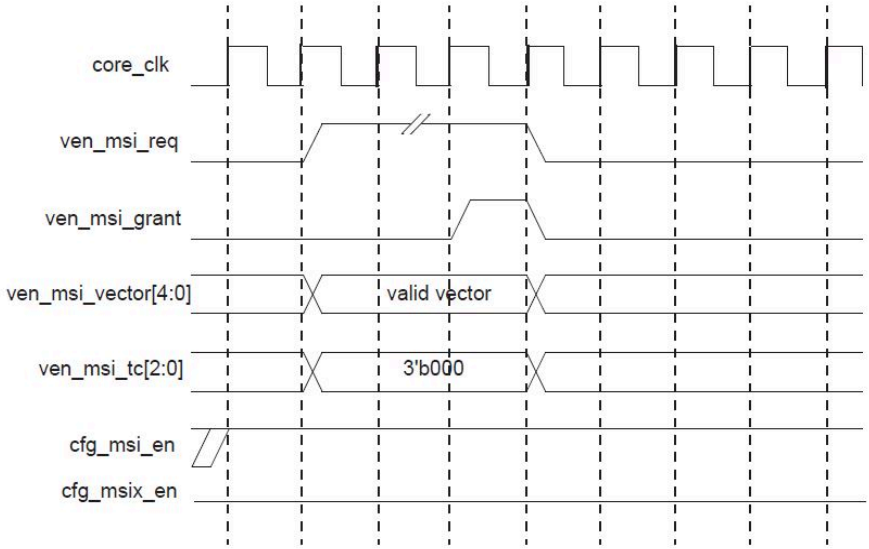
1. Assert the `ven_msi_req`input in the application logic when you want to request the core to send an MSI.
2. Ensure that `ven_msi_req` remains asserted until the core acknowledges the request by asserting `ven_msi_grant`.
3. Once `ven_msi_grant` is asserted, the core generates the corresponding memory write to initiate the MSI.
4. Understand and appropriately use the MSI interface signals.
5. Be aware that the core supports Per Vector Masking (PVM), which allows masking interrupts on a per-vector basis.
6. Refer to the documentation for any specific configurations, registers, or additional parameters related to the MSI interface.
7. Implement Per Vector Masking (PVM) based on the information available in the provided documentation.

Chapter 3. Example of a MSI Transaction

Figure 1. shows an example of your application requesting the core to send an MSI message upstream when `cfg_msix_en` is asserted. `ven_msi_req`stays asserted until the core asserts `ven_msi_grant.`

The internal MSI arbiter performs arbitration for MSI requests from different functions. `ven_msi_grant`is one-cycle pulse acknowledging `ven_msi_req.ven_msi_req` is not required to be de-asserted before reasserting again. When `ven_msi_req` remains asserted, the core generates another MSI.

Figure 1. Fig 1. MSI Message Request



Chapter 4. Steps to Send an MSI-X Interrupt Using the MSI-X Interface

The MSI-X Interface enables an application to request the core to send an MSI-X. The MSI-X interface protocol involves a simple synchronous request/acknowledge handshake.

Ensure that the MSI-X interface is exclusively used for upstream ports.

MSI-X Interrupt Transmission Procedure:

1. Assert the `ven_msi_req`input in the application logic when you want to request the core to send an MSI-X.
2. Ensure that `ven_msi_req` remains asserted until the core acknowledges the request by asserting `ven_msi_grant`.
3. Once `ven_msi_grant` is asserted, the core generates the corresponding memory write to initiate the MSI-X.
4. Understand and appropriately use the MSI-X interface signals.
5. In addition to the MSI interface signals, your application must supply the MSI-X address and data on `msix_addr` and `msix_data`, respectively.
6. In addition to the MSI interface signals, your application must supply the MSI-X address and data on `msix_addr`and `msix_data`, respectively.
7. Be aware that the core supports Per Vector Masking (PVM), which allows masking interrupts on a per-vector basis.
8. Refer to the documentation for any specific configurations, registers, or additional parameters related to the MSI interface.
9. Implement Per Vector Masking (PVM) based on the information available in the provided documentation.

Chapter 5. Example of a MSI-X Transaction

Figure 2. shows an example of an application requesting the core to send an MSI-X Message upstream when `cfg_msix_en` is asserted.`ven_msi_req` stays asserted until the core asserts `ven_msi_grant`.

As with MSI, the internal arbiter performs arbitration for MSI-X requests from different functions. `ven_msi_grant` is a one-cycle pulse, after which the core does not wait for `ven_msi_req` to be de-asserted. When `ven_msi_req` remains asserted, the core generates another MSI-X.

Figure 2. Fig. 2 MSI-X Message Request