Black text on a white background

Description automatically generated

**Technical University of Regensburg**

**[Faculty of Electrical Engineering and Information Technology](https://elektro-informationstechnik.oth-regensburg.de/en/)**

**VPL Project**

**DMA and Example Projects**

Authors: Philipp Aurbach (3399206)

Maximilian Rötzer (3399446)

Course of Study: Master Elektro- und Informationstechnik

Academic Supervisor: Prof. Dr. Aschauer

Submission Date: 20.01.2024

**Abstract**

This project presents an implementation of Direct Memory Access (DMA) technology, with a specific focus on its application in programmable logic and AXI Bus systems, including the scatter-gather method. The document first introduces various types of DMA, such as CMDA, DMA, MCDMA, and VDMA, providing an overview of each type of DMA. Explanations are given for each DMA category. Additionally, the documentation explores the basic functionality of the DataMover, which acts as the base for most DMAs.

The primary goal is to get to know the mechanisms and functionalities of different DMA types in conjunction with the AXI Bus. To help practical understanding, the document includes a series of example projects corresponding to each DMA type. These projects serve as examples, demonstrating the applications and basic implementations of DMA technology. The project offers theoretical knowledge paired with practical knowledge, and examples, making it a possible reference guide for developing applications with DMAs in future projects.

A repository containing all projects discussed here is available at the corresponding [github repository](https://github.com/aurbachphilipp/VPL_DMA/tree/main).

**Table of Contents**

[1 Introduction to DMA with AXI Bus 4](#_Toc156217775)

[1.1 AXI4-Memory Mapped 4](#_Toc156217776)

[1.2 AXI Stream 4](#_Toc156217777)

[1.2.1 S2MM (Stream to Memory-Mapped) 5](#_Toc156217778)

[1.2.2 MM2S (Memory-Mapped to Stream) 6](#_Toc156217779)

[1.3 AXI DataMover 7](#_Toc156217780)

[1.4 Scatter – Gather 7](#_Toc156217781)

[2 Example Projects of AXI DMAs 8](#_Toc156217782)

[2.1 AXI CMDA 8](#_Toc156217783)

[2.1.1 CDMA Simple Polling BRAM 9](#_Toc156217784)

[2.1.2 CDMA Simple Polling DRAM 10](#_Toc156217785)

[2.2 AXI Direct Memory Access (DMA) 11](#_Toc156217786)

[2.2.1 DMA Simple Polling 11](#_Toc156217787)

[2.2.2 DMA Scatter-Gather Polling 12](#_Toc156217788)

[2.2.3 DMA Simple Interrupt 13](#_Toc156217789)

[2.2.4 DMA Scatter-Gather Interrupt 14](#_Toc156217790)

[2.2.5 AXI MCDMA 15](#_Toc156217791)

[2.3 AXI VDMA 17](#_Toc156217792)

[2.3.1 Frames to DDR 17](#_Toc156217793)

[3 Outlook 18](#_Toc156217794)

[4 Literature & Attachments 19](#_Toc156217795)

[4.1 Literature 19](#_Toc156217796)

[4.2 GitHub Repository 19](#_Toc156217797)

[5 List of Figures 20](#_Toc156217798)

# Introduction to DMA with AXI Bus

Direct Memory Access (DMA) is a technology that allows hardware within a computer to access the system memory independently of the Central Processing Unit (CPU). The primary purpose of DMA is to enhance overall data throughput and improve system efficiency. By offloading memory access tasks from the CPU, DMA reduces CPU workload, enabling it to focus on other critical processing tasks. The Xilinx Intellectual Property (IP) Cores use the AXI Bus to transfer data, which comes in two forms for the DMA Cores: AXI4 Stream, and AXI4 (MM to MM).

## AXI4-Memory Mapped

Utilized for high-bandwidth memory operations, this interface supports complex memory-mapped transactions, making it suitable for DMA operations that require access to system memory.

**MM to MM (Memory-Mapped to Memory-Mapped):** This mode refers to data transfers where both the source and destination are memory-mapped addresses. The DMA controller in this scenario transfers data from one memory location to another, typically used in operations where large blocks of data need to be moved within the memory.

## AXI Stream

The Advanced eXtensible Interface (AXI) bus, part of the ARM AMBA specifications, is a high-performance, high-bandwidth bus system. There are different DMA IP cores with different types of AXI buses to manage data transfers efficiently.

AXI Stream is designed for unidirectional data transfers from source to destination with minimal handshaking. It is used in applications where data streaming is essential, like video- or signal processing. Usually, the interfaces to the IP cores are named S2MM (Stream to Memory-Mapped) and MM2S (Memory-Mapped to Stream)

### S2MM (Stream to Memory-Mapped)

Here, the DMA controller moves data from a streaming source, like an input/output device, to a memory-mapped destination. This mode is particularly useful for capturing streaming data, such as video frames or network packets, into system memory.

A diagram of a computer system

Description automatically generated

Figure 1: S2MM block diagram [2]

The block diagram presents the S2MM (Stream to Memory-Mapped) data transfer path utilized in DMA operations. The elements within the diagram are as follows:

**AXI4-Stream Slave (Command):** This interface is responsible for receiving the initial setup commands that outline the details of the data transfer, such as destination address and transfer size.

**AXI4-Stream Master (Status)**: This interface communicates the current status of the write operations back to the control entity, often providing vital information like transfer completion and error status.

**AXI4 Master (Write):** This is the interface through which the DMA controller issues write requests to the system memory, pushing data into the memory-mapped destination as instructed by the command logic.

**Write Engine**: The write engine orchestrates the data flow from the AXI4-Stream Slave to the memory-mapped destination. It manages the sequencing of write operations, ensuring that data is accurately and efficiently transferred to the target memory region.

**AXI4-Stream Slave:** This interface receives the incoming data stream to be written to memory. The streaming data might originate from a variety of sources, including peripheral devices or other system components.

### MM2S (Memory-Mapped to Stream)

In contrast, MM2S involves transferring data from a memory-mapped source to a streaming destination. This mode is often employed in applications like playing audio or video, where data stored in memory needs to be sent to an output device.

A diagram of a computer system

Description automatically generated

Figure 2: MM2S block diagram [2]

The block diagram illustrates the MM2S (Memory-Mapped to Stream) data transfer path within a DMA system. The components of the diagram represent the following:

**AXI4-Stream Slave (Command):** This interface receives commands from a controlling entity, such as a processor or another IP block, which directs the data transfer process.

**AXI4-Stream Master (Status):** This interface sends status information back to the controlling entity, indicating the progress and completion status of the data transfer.

**AXI4 Master (Read):** This is the interface that initiates read requests from the memory. It operates under the control of the DMA's read engine and is responsible for pulling data from the memory-mapped source.

**Read Engine:** This is the core component that manages the reading of data from the memory-mapped source. It orchestrates the timing and sequence of read requests and the subsequent forwarding of data to the AXI4-Stream Master interface.

**AXI4-Stream Master**: This interface outputs the data stream resulting from the DMA read operation. It is typically connected to a device that operates on streaming data, such as a FIFO, a network interface, or a digital signal processor.

## AXI DataMover

The Xilinx IP core AXI DataMover uses an AXI4 Read to AXI4-Stream and an AXI4-Stream to AXI4 Write interface to transport its given data in full duplex mode. It is a basic soft core that is used as the base core of several other, more complex memory access cores like the AXI DMA or AXI CDMA. It features the same Memory Mapped data width of 32 to 1024 bits and Stream data width of 8 to 1024 bits as the AXI DMA, but also supports a parameterized Memory Map Burst transfer. Using that feature, for example a data block on RAM of up to 256 bytes can be accessed more easily by just using one address initialization.

The AXI DataMover is mainly used in custom applications where more complex DMAs do either not provide the functionalities like burst transfer or are not necessary and can be ignored to improve the resource utilization. To save resources the function set of the AXI DataMover can be even more reduced via the Vivado IDE. [2]

## Scatter – Gather

Scatter-Gather (SG) is an advanced form of DMA. Normally, DMA works by transferring a contiguous block of data from one memory location to another. However, a Scatter-Gather DMA allows the transfer of data to and from multiple, non-contiguous blocks of memory in a single DMA transaction. This is particularly useful when the data to be processed is not stored contiguously in memory.

Central DMA (MM2MM), and the “standard” DMA (S2MM MM2S) both have a Scatter-Gather engine that makes use of so-called **Scatter-Gather Lists.** We have provided an example for both blocks being used in Scatter-Gather mode.

**Scatter-Gather Lists:** These are data structures used to manage scatter-gather DMA operations. A Scatter-Gather List is essentially a list of pointers or descriptors, each pointing to a different memory block. The DMA engine uses this list to know where to fetch data from (scatter) and where to store data to (gather).

# Example Projects of AXI DMAs

## AXI CMDA

Often referred to as MM to MM (Memory-Mapped to Memory-Mapped) DMA, this is a DMA type where data transfer occurs between two memory-mapped areas. CDMA is used in scenarios where large blocks of data need to be moved or copied within the system's memory without CPU intervention. This method is efficient for bulk data transfers, as it can handle large data blocks autonomously, freeing up the CPU for other tasks. CDMA is ideal for operations like memory initialization, buffer copying, or moving data between different levels of memory hierarchy. The CDMA IP block is mainly used in embedded processing systems. It can be used with the Xilinx MicroBlaze processor. [3]

### CDMA Simple Polling BRAM

In our example **CDMA\_simple\_polling\_BRAM**, we use the Central DMA to write from a Memory Mapped to another Memory Mapped address. Specifically, we write from the ZYNQ´s DDR to the BRAM of the FPGA. The following block design was synthesized to realize it:

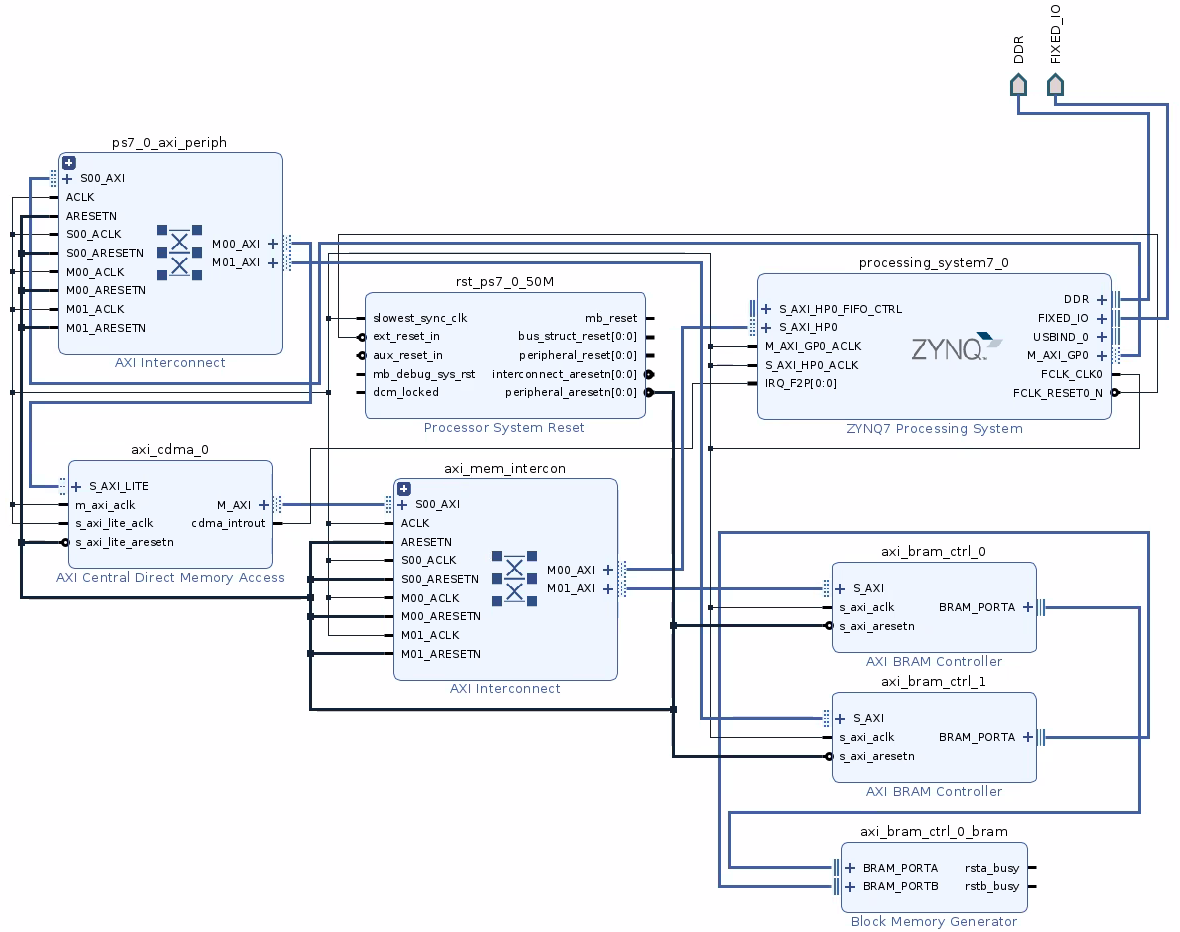


Figure 3: CDMA example block design for BRAM transfer

The source code *cdma\_simple\_polling.c* first initializes the system and then initializes a Transfer Request to the CDMA with *DoSimplePollTransfer()* in line 197.

This function first creates a test pattern inside the DDR, which is the *SrcBuffer*. It then calls *XAxiCdma\_SimpleTransfer(),* specifying the source and destination address, which is our *SrcBuffer* and *DestBuffer*, respectively. The *DestBuffer’s* address was set to XPAR\_BRAM\_0\_BASEADDR, which is the address of our BRAM cells on the FPGA. The *SrcBuffer* variable can just be passed as a pointer (see line 254 in source code).

The project finishes with the *CheckData()* function, which simply compares the bytes of the DDR with those of the BRAM, and returns success if everything went well.

### CDMA Simple Polling DRAM

In our example **CDMA\_simple\_polling\_DDR**, we use the Central DMA to write from a Memory Mapped to another Memory Mapped address. Specifically, we write from the ZYNQ´s DDR address to another DDR address of the ZYNQ processor, which means we do a simple “memcopy”, just with the CDMA. The following block design was synthesized to realize it:

A diagram of a computer network

Description automatically generated

Figure 4: CDMA example block design for DRAM transfer

The source code *cdma\_simple\_polling\_DDR.c* first initializes the system and then initializes a Transfer Request to the CDMA with *DoSimplePollTransfer()* in line 197.

This function first creates a test pattern inside the DDR, which is the *SrcBuffer*. It then calls *XAxiCdma\_SimpleTransfer(),* specifying the source and destination address, which is our *SrcBuffer* and *DestBuffer*, respectively. The *DestBuffer* and *SrcBuffer* addresses do not have to be set manually, as the C compiler already allocates space for our variables. They can simply be passed to the *XAxiCdma\_SimpleTransfer()* function as pointers (see line 254 in source code).

The project finishes with the *CheckData()* function, which simply compares the bytes of the DDR with those of the BRAM, and returns success if everything went well.

## AXI Direct Memory Access (DMA)

Utilizing S2MM (Stream to Memory-Mapped) and MM2S (Memory-Mapped to Stream) interfaces, the DMA facilitates efficient data handling for various applications. In S2MM mode, DMA is commonly used for reading data from a peripheral device (like a network card) into system memory. Conversely, in MM2S mode, it is typically employed to send data from memory to a peripheral device. [4]

### DMA Simple Polling

In our example **DMA\_simple\_polling**, we use the DMA to read and write from and to a FIFO, which represents our streaming device. Instead of using a device that generates a stream, for example an Analog to Digital Converter (ADC), we create our own stream by writing from DDR to FIFO, with the DMA. We then read the stream by capturing it with the DMA and storing it to DDR. The following block design was synthesized to realize it:

A diagram of a computer

Description automatically generated

Figure 5: DMA example block design in polling mode

The source code *dma\_simple\_polling.c* first initializes the system and then starts the DMA Transfers with the *XAxiDma\_SimpleTransfer()* function.

As parameters, the Rx or Tx Buffer, and the direction are given (line 258 in source code). Two directions are possible:

The first direction called XAXIDMA\_DEVICE\_TO\_DMA reads streaming data from the FIFO and stores it in the Rx Buffer.

The second, called XAXIDMA\_DMA\_TO\_DEVICE, writes DDR data from the Tx Buffer to the FIFO to create our stream.

In line 244 of the source code, the Tx Buffer was initialized with a test pattern. In line 272, the DMA is being polled until the transfers are finished. The function *CheckData()* (line 277) verifies that the Rx Buffer has the same data as the Tx Buffer after the transfers, and returns success if all went well.

### DMA Scatter-Gather Polling

Looking at our project **DMA\_SG\_polling,** the code in dma\_SG\_polling.c is designed to demonstrate how to use the DMA in Scatter Gather Mode for data transfer in polling mode. This mode is useful when data needs to be transferred in and out of non-contiguous memory areas.

The project basically uses the same block design as for DMA data transfer in simple polling mode, as shown in Figure 5, with only the Scatter-Gather option enabled in the AXI Direct Memory Access IP block.

The code defines memory buffers for transmit (TX) and receive (RX) operations (line 106, 107). These buffers are areas where data is stored temporarily before being sent out and after being received.

The code also sets up Buffer Descriptors (BDs) for both TX and RX operations with the function *TxSetup()* (line 201) and *RxSetup()* (line 206). BDs are data structures that contain information about data packets, such as where they are in memory and how big they are. The DMA engine uses this information to 'scatter' (read) data from various memory locations for transmission and 'gather' (write) incoming data into different memory locations.

With *SendPacket()*, a transfer is initialized. The function *CheckDmaResult()* verifies the behavior by polling (lines 621 and 635) the Tx and Rx channels. It returns success when finished and all went well.

### DMA Simple Interrupt

Our example project **DMA\_simple\_interrupt** uses the DMA to read and write from and to a FIFO, which again represents our streaming device. The project demonstrates how to use the DMA for data transfer in interrupt mode. This mode is useful when data needs to be transferred in and out of non-contiguous memory areas and polling is not an option. The following block design was synthesized:

A diagram of a computer

Description automatically generated

Figure 6: DMA example block design in interrupt mode

The source code *dma\_simple\_interrupt.c* first initializes the system, sets up the interrupt system as well as the interrupt flags (line 287), and then starts the DMA Transfers with the *XAxiDma\_SimpleTransfer()* function.

As parameters, the Rx or Tx Buffer, and the direction are given (lines 308f in source code). Two directions are possible.

The first, XAXIDMA\_DEVICE\_TO\_DMA, reads streaming data from the FIFO and stores it in the Rx Buffer.

The second, XAXIDMA\_DMA\_TO\_DEVICE, writes DDR data from the Tx Buffer to the FIFO to create our stream.

In line 293 of the source code, the Tx Buffer was initialized with a test pattern. As soon as the transfers finish, interrupts are generated, and the flags are set by the corresponding Interrupt Handlers (lines 447 and 522). The function *CheckData()* (line 342) verifies that the Rx Buffer has the same data as the Tx Buffer after the transfers and returns success if all went well.

### DMA Scatter-Gather Interrupt

Looking at our project **DMA\_SG\_interrupt,** the code in dma\_SG\_interrupt.c is designed to demonstrate how to use the DMA in Scatter Gather Mode for data transfer in interrupt mode.

The project basically uses the same block design as for DMA data transfer in simple interrupt mode, as shown in Figure 6, with only the Scatter-Gather option enabled in the AXI Direct Memory Access IP block.

The code defines memory buffers for transmit (TX) and receive (RX) operations (lines 126 and 127). These buffers are areas where data is stored temporarily before being sent out and after being received.

The code also sets up Buffer Descriptors (BDs) for both TX and RX operations with the function *TxSetup()* (line 281) and *RxSetup()* (line 289). BDs are data structures that contain information about data packets, such as where they are in memory and how big they are. The DMA engine uses this information to 'scatter' (read) data from various memory locations for transmission and 'gather' (write) incoming data into different memory locations. Interrupt flags are created in line 305, which our Interrupt Callbacks (lines 444 and 582) add += 1. When the flag has the number of BD descriptors, all transfers have finished.

With *SendPacket()* a transfer is initialized. The function *CheckDmaResult()* verifies the behavior by looking at the *TxDone* and *RxDone* flags. It returns success when finished and all went well.

### AXI MCDMA

Multi-Channel DMA (MCDMA) extends the capabilities of standard DMA by supporting multiple, independent data transfer channels simultaneously. Like DMA, it operates with both S2MM and MM2S modes. MCDMA is particularly useful in complex systems requiring concurrent data processing tasks. In S2MM mode, MCDMA can simultaneously receive multiple data streams, such as sensor inputs in an industrial control system, or managing multiple networks. In MM2S mode, it can handle tasks like outputting different audio channels in real-time.

Ein Bild, das Text, Diagramm, Plan, technische Zeichnung enthält.

Automatisch generierte Beschreibung

Figure 7: MCDMA block diagram [5]

Figure 7 shows the basic block diagram of a MCDMA and its main components:

**Multiple Channels:** The AXI MCDMA supports up to 16 independent channels, allowing it to handle multiple data streams simultaneously. A single DMA typically handles one data stream at a time.

**Scatter Gather (SG) Interface:** This feature offloads the MCDMA management work from the CPU, fetching and updating Buffer Descriptors independently from the primary data bus. This is more advanced than the typical linear data transfer method used in single DMAs.

The **Packet Filter** is a user implemented logic which lies outside the MCDMA and handles streaming data. It is necessary and should do the following jobs:

*Data Sorting Based on TDEST*: In systems using AXI MCDMA, data packets on AXI4-Stream interfaces are associated with a TDEST (destination identifier). The packet filter sorts these packets into separate descriptor queues based on their TDEST values.

*Routing and Processing*: On the MM2S (Memory-Mapped to Stream) side, the packet filter routes AXI4-Stream data based on the TDEST value. On the S2MM (Stream to Memory-Mapped) side, it assigns the appropriate TDEST value to incoming data packets. [5]

Because of the increased complexity, we did not create this custom hardware.

## AXI VDMA

Video DMA (VDMA) specializes in handling video data streams, integrating both S2MM and MM2S modes for flexible video processing. In S2MM mode, VDMA is essential for applications like video capture, where it transfers incoming video frames from a camera or other video sources into system memory. In MM2S mode, VDMA facilitates tasks like video playback, e.g. sending video data from memory to a display device. VDMA also supports a circular buffer to store up to 32 video frames with 4k resolution, provided you have the memory necessary for that. [6]

### Frames to DDR

Looking at the project **VDMA\_framesToDDR,** the code of the vdma\_continuous\_to\_DDR.c source provides the basis for the usage of the VDMA for frame buffer transfer. A Video Test Pattern Generator creates a continuous video stream, and the VDMA stores 3 of those frames at a time into the DDR, and operates continuously in the background. The following block design was synthesized:

A diagram of a computer

Description automatically generated

Figure 8: VDMA example block design

A buffer of 3 frames is used, each 480x480 pixels. After configuring the DMA and the buffer addresses, the continuous transfer from the Test Pattern Generator to DDR is initiated in line 56, and then runs in the background.

From DDR, the video data could be streamed via http, or being sent to a video output interface like VGA or HDMI to a display device.

# Outlook

Because of the increased complexity with the Packet Filter when using the Multi-Channel DMA (MCDMA), we have not implemented this. Perhaps in a future project one could create such a user-implanted Packet Filter and then input and output multiple streams of data.

A newer development seems to be the Time Aware DMA, where there is not yet a proper documentation available from AMD. In a future project, an example may be created.

Further, using the DataMover (which is the base of the various DMAs), one could go about creating a custom DMA for a specific purpose, perhaps a neural network.

# Literature & Attachments

## Literature

1. AXI Stream Documentation,

<https://docs.xilinx.com/r/en-US/ug1399-vitis-hls/AXI-Adapter-Interface-Protocols> (accessed: 11.01.2024)

1. AXI DataMover Documentation

<https://docs.xilinx.com/r/en-US/pg022_axi_datamover>

(accessed: 10.01.2024)

1. AXI CDMA Documentation

<https://docs.xilinx.com/r/en-US/pg034-axi-cdma>

(accessed: 10.01.2024)

1. AXI DMA Documentation

<https://docs.xilinx.com/r/en-US/pg021_axi_dma>

(accessed: 10.01.2024)

1. AXI Multichannel DMA Documentation

<https://docs.xilinx.com/v/u/1.0-English/pg288-axi-mcdma>

(accessed: 10.01.2024)

1. AXI VDMA Documentation

<https://docs.xilinx.com/r/en-US/pg020_axi_vdma>

(accessed: 10.01.2024)

## GitHub Repository

[aurbachphilipp/VPL\_DMA: Example Project using various DMAs in vivado (github.com)](https://github.com/aurbachphilipp/VPL_DMA)

# List of Figures

[Figure 1: S2MM block diagram [2] 5](#_Toc156119859)

[Figure 2: MM2S block diagram [2] 6](#_Toc156119860)

[Figure 3: CDMA example block design for BRAM transfer 9](#_Toc156119861)

[Figure 4: CDMA example block design for DRAM transfer 10](#_Toc156119862)

[Figure 5: DMA example block design in polling mode 11](#_Toc156119863)

[Figure 6: DMA example block design in interrupt mode 13](#_Toc156119864)

[Figure 7: MCDMA block diagram [5] 15](#_Toc156119865)

[Figure 8: VDMA example block design 17](#_Toc156119866)