

**Hochschule Karlsruhe
Technik und Wirtschaft**
UNIVERSITY OF APPLIED SCIENCES

Fast Memory Access of Large Amount of Data in Xilinx Zynq System on Chip

Master Thesis

Muzamil Farid (64389)

M.Sc. Sensor System Technology

Supervisors:

Prof Dr. Gerhard Schäfer

Prof Dr. Manfred Litzenburger

Date: 20/10/2020

Acknowledgments

I would like to thank both of my Professors Dr Gerhard Schäfer and Manfred Litzenburger for their supervision and evaluation of this master thesis. Moreover I would like to thank CoSynth GmbH for providing academic and financial support in this thesis and all the colleagues in the company for their support and most importantly my supervisor at the company Christian Stehno for providing his deep insights in to the topic. Christian's ability to guide and giving appropriate directions was incredible, which helped me to learn and grow into the topic quicker. Christian ability to understand my arguments and then steering me into right direction resulted in better and cohesive work. Additionally I would like to express my gratitude to all my previous mentors who supported me in my previous jobs and internships at Bosch Engineering , who helped me to instil deep passion and motivation for engineering and specifically in FPGA area. I would also like to thank my family members who supported me as best as they could.

Abstract

White light interferometry (WLI) has been widely used as contact free measurement method for surface topographies. While the commonly used vertical scanning white light interferometry (VSWLI) provides height of resolution under 1nm, it is not suitable to measure objects in motion. An alternative technique lateral scanning white light interferometry (LSWLI) is used to measure surface topographies of moving surfaces. White light interferometry uses the principle of interference, where a white light is split in two beams using a beam splitter. One light source is directed to reference mirror and other one is directed to the measuring object. Both light sources reflect from reference surface and measuring surface and recombine at the detector. The detected signal intensity indicates whether constructive or destructive interference occurred. Due to shorter coherence length of white light maximum interference only occurs when the optical path difference is zero. In vertical scanning white light interferometry (VSWLI), a measuring object is moved in the direction of the optical axis and camera captures the images at discrete intervals. A correlogram is obtained for every pixel of the image, a correlogram represents the maximum intensity of a particular pixel. A pixel has a maximum intensity when maximum interference occurs. As the object is moved vertically with discrete interval several correlograms are measured for all the pixels at every discrete distance. These correlograms are stitched together to evaluate surface topography using an algorithm running on powerful and fast hardware FPGA. In lateral scanning white light interferometry (LSWLI) measuring object is moved laterally with a specific tilt angle thus eliminating the need to stitch different correlograms together as opposed to in vertical scanning white light interferometry (VSWLI). As the camera in white light interferometry system captures large number of images, these images are required to be buffered before any algorithm can be applied to evaluate surface topography. As the algorithm is running on powerful hardware FPGA, a special memory access mechanism is required to store the images in the memory and subsequently stream them to the algorithm in a specific pattern. The goal of this thesis was to develop an IP core which would store the images in memory and stream them to another IP core which is an algorithm in this case. A special data structure is implemented on Block RAM which resides in the programmable logic (PL) side of Zynq and DRAM which resides outside the chip connected to the processing system (PS) side of Zynq. Write and Read times of ring buffer were calculated thus giving critical information regarding the speed of memory access.

Table of Contents

Acknowledgments	2
Abstract	3
List of Figures	6
Chapter 1	8
Introduction	8
1.1 White Light Interferometry	9
1.2 Lateral Scanning White Light Interferometry	9
1.3 Structure of the Thesis	11
Chapter 2	12
Xilinx Zynq Soc	12
2.1 IP cores	13
2.2 AXI Protocol	14
2.2.1 Key features of AXI	14
2.2.2 Transaction Channels	14
2.2.3 Write Transactions	15
2.3 AXI Signals	17
2.3.1 Write Address Channel	17
2.3.2 Write Data Channel	18
2.3.3 Read Address Channel	19
2.3.4 Read Data Channel	19
2.4 AXI-LITE	19
2.5 AXI STREAM	20
2.5.1 Timing Diagram of AXI STREAM	20
2.6 Signal Dependencies	21
2.6.1 Write Transaction Dependencies	21
2.6.2 Read Transaction Dependencies	21
2.7 AXI DMA	22
2.7.1 Key features of AXI DMA	22
2.7.2 Channels of AXI DMA	23
2.7.3 Interfaces of AXI DMA	23
2.8 AXI DMA Registers	24
2.8.1 S2MM Registers	24
2.8.2 MM2S Registers	25
2.8.3 Internal Settings of AXI DMA	26
2.9 Structure of the IP core	27

2.9.1 Sub IP core	27
2.9.2 Block Design Setup	28
2.9.3 Design Changes with BRAM and DRAM	30
Chapter 3	32
3.1 Memory Access Pattern	32
3.2 Design Methodology	35
3.3 State Machine Design	37
3.4 Description of State Machine	38
3.4.1 External Interface to the State Machine	39
3.5 Analysis of Different Access Patterns	39
3.6 Measurements for Speed Calculation.....	40
Chapter 4	42
Conclusion.....	42
4.1 Future Recommendations.....	42
References	43
Appendix	44

List of Figures

Figure 1.1 Correlogram Signal at Detector in WLI.....	7
Figure 1.2 White Light Interferometry Setup.....	8
Figure 1.3 Correlograms for Lateral Scanning White Light Interferometry.....	9
Figure 2.1 Xilinx Zynq Soc Architecture.....	12
Figure 2.2 IP core in the current setup.....	13
Figure 2.3 AXI Protocol Topology.....	14
Figure 2.4 AXI Write Transaction.....	15
Figure 2.5 AXI Write Address Channel.....	16
Figure 2.6 AXI Write Data Channel.....	16
Figure 2.7 AXI Write Transaction.....	16
Figure 2.8 AXI Burst Type.....	17
Figure 2.9 AXI Burst Type.....	18
Figure 2.10 AXI LITE Protocol.....	19
Figure 2.11 AXI Stream Protocol.....	20
Figure 2.12 AXI Stream Timing	20
Figure 2.13 Write Transaction Dependencies.....	21
Figure 2.14 Read Transaction Dependencies.....	21
Figure 2.15 AXI DMA internal Structure.....	22
Figure 2.16 AXI DMA Interface.....	23
Figure 2.17 S2MM Control Register.....	24
Figure 2.18 S2MM Status Register.....	24
Figure 2.19 S2MM Length Register.....	25
Figure 2.20 MM2S Control Register.....	25
Figure 2.21 MM2S Status Register.....	26
Figure 2.22 MM2S Length Register.....	26

Figure 2.23 AXI DMA Internal Settings.....	26
Figure 2.24 Structure of the IP Core.....	27
Figure 2.25 Modules of the IP Core.....	27
Figure 2.26 Block Design with BRAM.....	28
Figure 2.27 Block Design with BRAM.....	29
Figure 2.28 Block Design with DRAM.....	29
Figure 2.29 Block Design with DRAM.....	30
Figure 2.30 Address Editor with BRAM.....:	30
Figure 2.31 Address Editor with DRAM.....	31
Figure 3.1 Memory Access Pattern.....	32
Figure 3.2 Overwriting 0 th Slot.....	34
Figure 3.3 Overwriting 1024 th Slot.....	34
Figure 3.4 Design Methodology.....	35
Figure 3.5 State Machine Design.....	37

Chapter 1

Introduction

The measurement of surface profiles has been a very critical part of industrial applications. The quality of technical surfaces is an essential factor for the quality of overall result. In the semiconductor industry, excessive roughness and surface imperfections have a detrimental effect on the quality and functionality of electronic components, whereas in the solar industry a certain degree of roughness improves the functionality of the cells. In medical technology, excessive roughness prevents the small and minute volumes of liquid. In the printing industry, the volume and web widths of printing cells on the printing rollers must be checked. In metal processing industry, product quality often directly depends on the surface quality. The quantitative evaluation of surface topographies is based on standardized roughness parameters. Conventional measuring methods such as profilometry, confocal microscopy, vertical scanning white light interferometry (VSWLI) allows roughness measurements in accordance with standards. These methods are designed to use on stationary objects under laboratory conditions. Lateral scanning white light interferometry (LSWLI) is a measurement method designed for moving objects both translatory and rotatory motions. LSWLI system continuously captures images during the movement of object. Based on this image stack, a relative height value of respective surface point can be determined for each pixel position. Standard compliant filters such as ISO 11562, 13565 can be applied on to the 3D topography data in order to calculate both area related roughness parameters (e.g. S_a and S_q) and profilometric roughness parameters (e.g. R_a and R_q). In addition, a complete large area 3D topography data set is the basis for evaluating surface imperfections (ISO 8785). This possibility of being able to calculate values of almost all standardized surface parameters based on data set of an in-process measurement offers a considerable advantage for the exchange of information and measurement data. The white light enables contactless high resolution and standard compliant topography measurements. In a white light interferometer setup, a beam splitter splits the white light in two beams, a reference beam which illuminates the reference sample and an object beam which illuminates the measuring object. Both beams reflect and recombine and form an interference pattern. The interference only occurs if the optical path difference is zero which means the distance travelled by both the beams is the same. Due to the shorter coherence length of the white light, interference can only be observed in a very short range, as the white light is composed of seven different wavelengths. The signal detected has intensity as shown in figure 1.1.

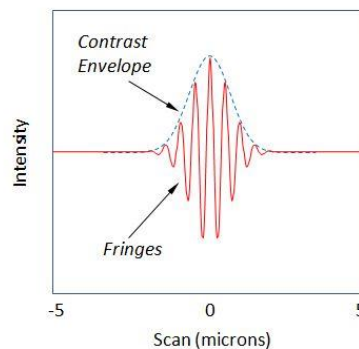


Figure 1.1

1.1 White Light Interferometry

The working principle of white light interferometry is shown in figure 1.2. LED emits a beam of white light and passes through a beam splitter. A beam splitter splits the beam in to reference beam and measurement beam. Measurement beam is directed to the measuring object whereas reference beam is directed to the reference plane which acts as a reference sample and whose surface topography is known.

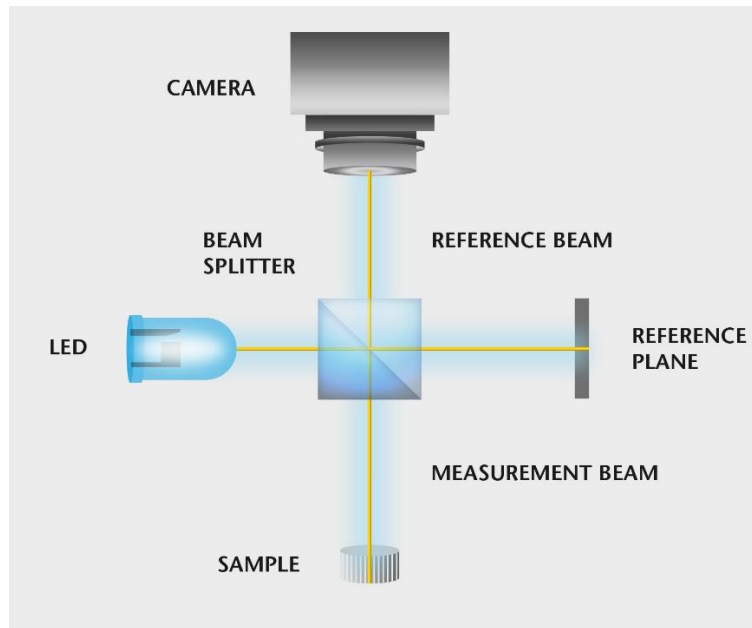


Figure 1.2

1.2 Lateral Scanning White Light Interferometry

Lateral scanning white light interferometry is the measuring method used to measure surface roughness of moving surfaces. In case of vertical scanning white light interferometry, the object is moved vertically and depending on the field of vision of camera only a certain area can be captured, to measure the whole object multiple parts need to be stitched together which might require a stitching algorithm which is not very efficient. To solve this problem a modified version known as lateral scanning white light interferometry is developed which allow measurements of objects without the need of stitching them together as the object is moving laterally in two directions X axis and Z axis. Depending on the point where optical path difference is zero a correlogram for a surface point can be detected. Lateral scanning white light interferometry allows the measurements on fast moving rotating surfaces which are critical in several areas of electronics industry

The below figure illustrates the basic principle of lateral scanning white light interferometry. The surface height between point A and B can be calculated by a standard trigonometry. X_1 refers to the amount of distance object moved horizontally and Z_1 refers to the amount of distance object moved in the vertical direction where surface point A has the maximum intensity as the optical path difference at this point is zero. Surface point B on the other hand has a significant height difference to point A and achieves maximum intensity at position X_2 and Z_2 respectively where its optical path difference is zero. The angle that the object makes with a surface normal is tilt angle which must be known in advance to calculate the height difference between surface point A and B.

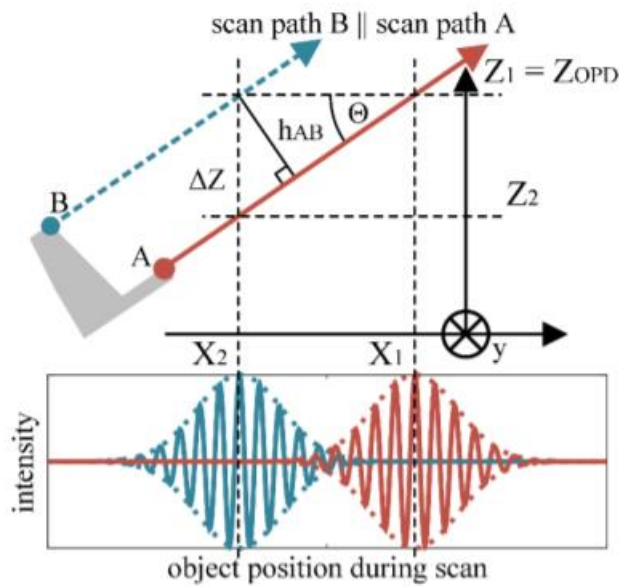


Figure 1.3

The intensity of the signal detected at the detector is as follows

$$I_z = I_0 \cdot \left[1 + \exp \left[-\ln(2) \cdot \left(\frac{2(l - l_{ref})}{l_c} \right)^2 \right] \cdot \cos \left(\frac{4\pi(l - l_{ref})}{l_c} \right) \right] \quad (1)$$

Where I_0 is the mean intensity, l is the distance between the sample surface and the beam splitter ($l = z(\text{surface}) - z(\text{beam splitter})$). l_{ref} represents the distance between reference mirror and the beam splitter. The coherence length of l_c determines the decay of the intensity signal and reads for a gaussian light source

$$l_c = \frac{4 \cdot \ln(2)}{\pi} \cdot \frac{\lambda_0^2}{n \cdot \Delta\lambda} \quad (2)$$

Where λ_0 is the central wavelength of the light spectrum, n as the refractive index of the surrounding medium and $\Delta\lambda$ represents the full width half maximum of the spectrum.

According to equation 1, the intensity fluctuates between local maxima and minima if the length of the measurement path is changed continuously. Such a signal curve is henceforth referred to as a correlogram. The correlogram itself and the interference contrast between a maximum and its adjacent minima in the correlogram are maximal for the surface point where $z=z_{\text{opd}}$ where $I - I_{\text{ref}} = 0$.

In the figure 1.3 the vertical distance between $\Delta z = (x_1 - x_2) \cdot \tan(\theta)$ between the scan paths can be converted to actual height difference h_{AB} by evaluating the cosine of the tilt angle.

$$h_{AB} = \Delta z \cdot \cos(\theta) = (x_1 - x_2) \cdot \sin(\theta). \quad (3)$$

The position x_1 and x_2 are known from the controlled surface movements and the evaluation of the corresponding correlograms. For ease of calculation the reference position x_1 is assigned to a pixel of the camera sensor in the scan direction. As a digital camera in the x-y plane is used to record the signals a continuous stream of surface points is observed moving through the FOV in horizontal x-direction (line measurement). Depending on the number of camera pixels in y-direction, multiple rows of points are recorded in parallel (areal measurement).

According to the equation 3, the tilt angle of the scan path needs to be known, which is either obtained from a calibration with a reference object or a self-calibration method developed by Florin Muntuneua. Florin method uses the frequency of the intensity fringes in the correlogram to calculate the tilt angle of the scanning path

$$|\theta| = \arctan\left(\frac{f_{\text{corr}} \lambda_0}{2 s_{\text{pixel}} N_{\text{pixel},x}}\right), \quad (4)$$

Where λ_0 is the central wavelength of the illumination, s_{pixel} is the width of the surface observed by a pixel and $N_{\text{pixel},x}$ is the number of pixels in scan direction over which the frequency is evaluated

1.3 Structure of the Thesis

This Master thesis is divided into four chapters including this chapter which is an introduction to this thesis.

- Introduction: This chapter explains the project application and necessary background to understand the project. The basic principles of white light interferometry and its different versions are described with graphical and mathematical descriptions.
- Project Planning and Setup: This chapter covers the necessary components required to develop a working project, it also explains the research carried out to understand the task and to develop an implementable project setup
- Implementation: This chapter deals with the technique and analytical approach required to solve a task and to evaluate observations and measurements carried out during the implementation to form a cohesive outcome.
- Conclusion: This is the final chapter of this thesis where it concludes this thesis with a sound conclusion and presents some future recommendations regarding possible changes and different implementation ideas.

Chapter 2

Xilinx Zynq Soc

This master thesis uses all programmable system on chip (Soc) device known as Xilinx Zynq. Zynq offers a powerful solution to electronics and embedded engineers to develop powerful and efficient designs. Its hybrid hardware-software functionality allows designers to create complex hardware-software codesigns making them extremely attractive to software developers as well and thus enabling software developers to develop systems which otherwise would not be possible. Zynq features an integrated FPGA with ARM cores on a single chip. The chip is divided in two parts, the programmable logic (PL) and processing system (PS). Programmable logic part of the Zynq is a powerful 7 series FPGA with all the necessary components such as Block memories, DSP slices and Logic cells. Processing system (PS) part of the Zynq comprises of ARM Cortex CPU and set of peripherals including OCM (On chip memory), L2 Caches, DDR controller and I/O peripherals such as USB, CAN, I2C. The interface between PS and PL is the AXI which is a high-speed communication interface developed by ARM corporation. A simplified version of Zynq looks like as shown in below figure.

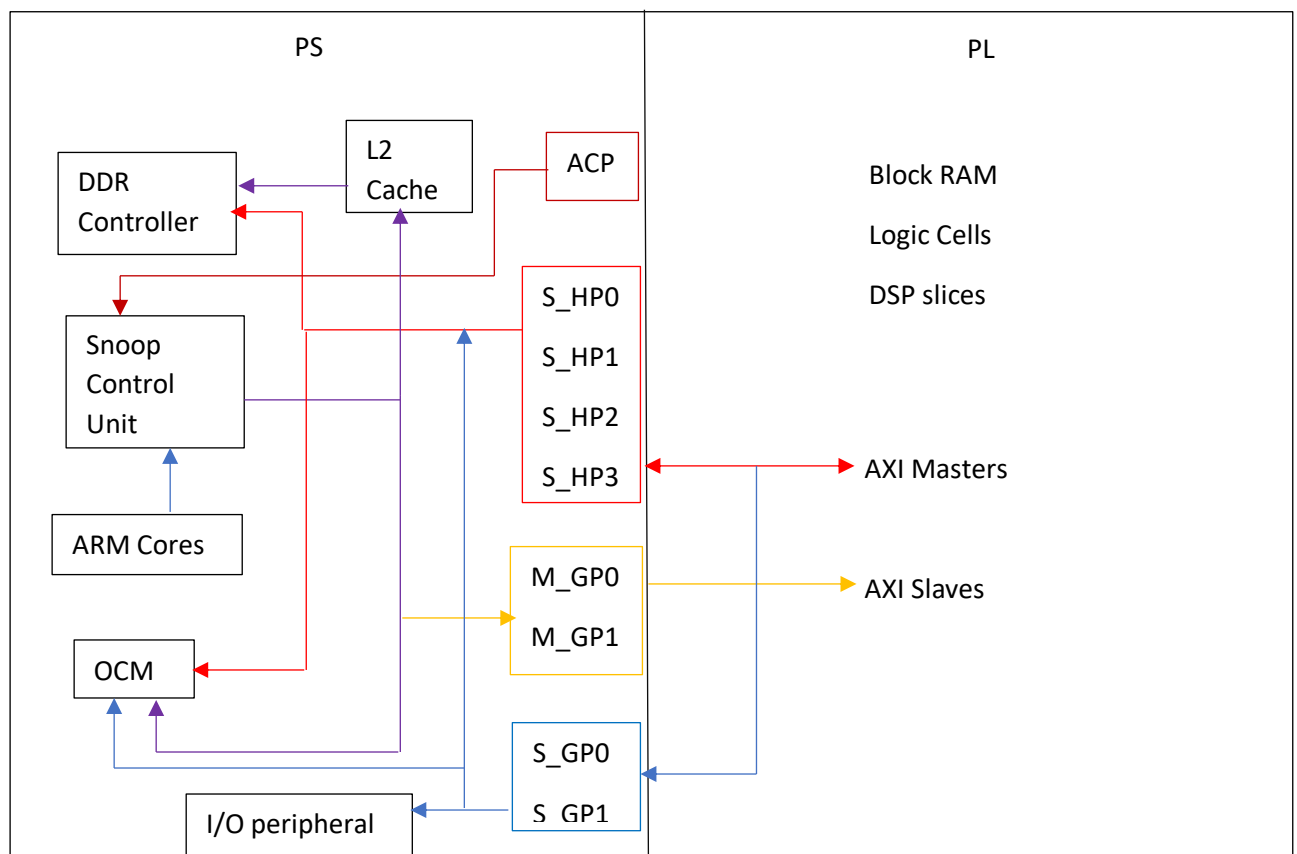


Figure 2.1

As shown in figure 2.1, PL part of the Zynq can access DRAM through DDR controller using one of the four high performance AXI slave ports (S_HP). General purpose ports (S_GP) can also be used to access the DRAM on board through a DDR controller although using S_GP ports are slower than

using S_HP ports. Zynq also features a high-speed accelerator coherency port (ACP) which is directly connected to the snoop control unit (SCU) inside the PS. Snoop control unit maintains cache coherency between two caches inside the ARM cores and provides access to L2 cache and on chip memory (OCM). There is also a dedicated port from L2 cache toward DDR controller thus allowing ARM cores to access DRAM directly.

2.1 IP cores

The main goal of this thesis was to develop an IP core which is simulatable as well as runs appropriately on actual hardware as intended. The IP core stands for Intellectual property core which consists of several modules of hardware description languages (HDL) designed for a specific logical functionality. The figure below explains how the IP core is connected to other components in the FPGA and its intended functionality.

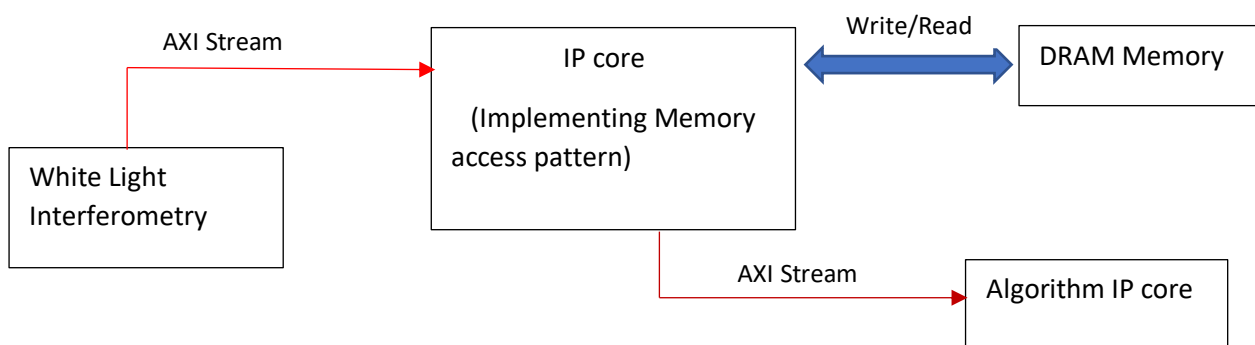


Figure 2.2

White light interferometry system as explained in previous chapter produces large number of images as it scans the objects whose surface roughness is needed to be measured, these images are converted to suitable protocol before entering the FPGA fabric. AXI Stream is the protocol which is used for images and video applications. The IP core developed for this thesis was acquiring these images which are converted to AXI stream and writes them in to the DRAM memory and then reads from the memory from specific addresses in a very specific pattern thus the IP core is implementing a very complex memory access pattern. The pixels that are read from the memory are streamed out towards another IP core which is an algorithm and calculates surface roughness. IP core which is implementing memory access pattern was simulated on Block ram and practically tested on the DRAM memory by programming the FPGA. Several measurements were also taken regarding the write/read times of both DRAM and Block ram to do a comparative analysis regarding the speed. Different settings and different access patterns were tested to evaluate the speed of memory access to form a suitable conclusion.

2.2 AXI Protocol

The AXI protocol is a high-speed communication interface designed for high performance and high frequency designs. The AXI protocol is suitable for high bandwidth and low latency designs. It provides high frequency operation without using complex bridges, meets the interface requirements of a wide range of components. AXI protocol is ideal for memory controllers such as DDR controller in Zynq PS with high initial access latency. AXI protocol is also backward compatible with AHB and APB interfaces

2.2.1 Key features of AXI

The key features of AXI protocol are as follows.

- Separate address/control and data phases.
- Support for unaligned data transfers using byte strobes.
- Uses burst based transactions using only start address issued.
- Separate read and write data channels
- AXI protocol includes the AXI-Lite specifications which is low speed suitable for accessing control and status registers in a component.
- Supports for issuing multiple outstanding addresses.

AXI protocol is based on master-slave topology, which are connected through one interconnect. Master can access multiple slaves through an interconnect and slaves can only respond to a write or a read request.

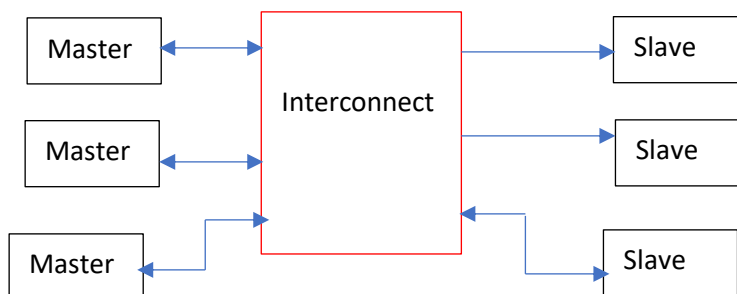


Figure 2.3

2.2.2 Transaction Channels

The AXI protocol is burst based and defines five independent transaction channels

- Write address channel
- Write data channel
- Read address channel
- Read data channel
- Write response channel

The address channel carries control information that describes the nature of data to be transferred between master and slave using a write data channel to transfer a data from master to slave. In a write transaction a slave uses the write response channel to indicate that a write transaction is

successful. The address channel also consists of read address channel where slave transfers the data back to master. Write address channel consists of the address where data needs to be written, write data channel consists of the data to an address location specified in the write address channel. Write address channel specifies the burst type and length where total number of transfers in the burst can be specified. Read address channel specifies the address from where the data needs to be read. Read data channel consists of the data requested by the master. Read data channel can only be active after the read address channel according to AXI specifications. A set of signal dependencies needs to be observed while writing an AXI protocol otherwise protocol can stall and will not work properly.

2.2.3 Write Transactions

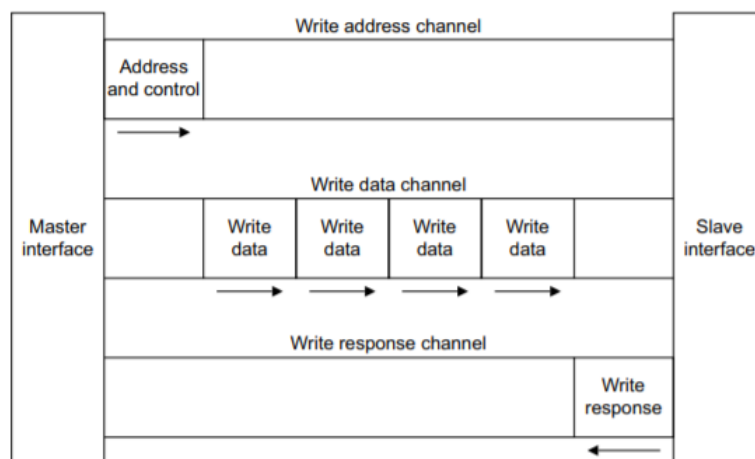


Figure 2.4

A typical AXI write transaction looks like as in figure 2.4. It consists of three write channels.

- Write Address Channel
- Write Data Channel
- Write Response Channel

All three channels use handshake mechanism controlled by two control signals "Valid and Ready" signal. Master asserts the valid signal to indicate that it is driving a valid address on the bus, Slave asserts the ready signal and when both signals are asserted then specified address value is delivered to the slave. Similarly, with write data channel, master asserts the valid signal indicating a valid data on the bus, slave asserts the ready signal and when both signals are asserted then data is delivered to the address location which was written using a write address channel. Both address and data can be written at the same time or data can be written after the address has been written. After both the address and data are written, slave asserts a "Bvalid" signal indicating both address and data has been written to the slave. Master then asserts the "Bready" signal indicating that its ready to accept that address and data has been written and when both signals are asserted then write response transaction is successfully completed. A timing diagram of write address transaction is shown below.

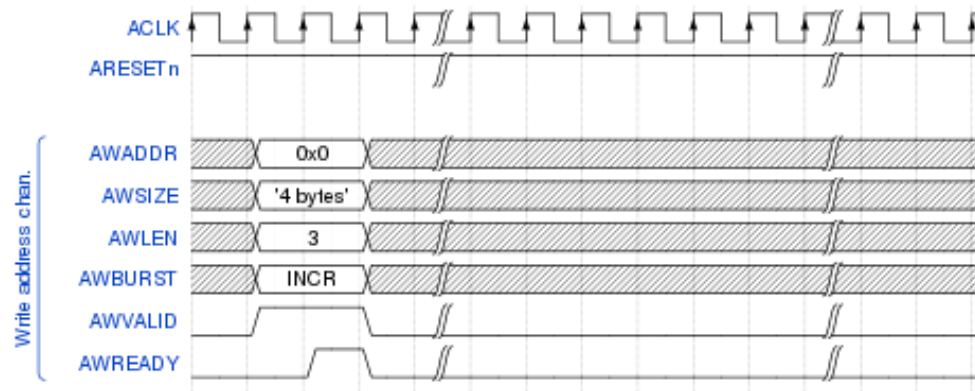


Figure 2.5

A timing diagram of write data channel looks like as below.

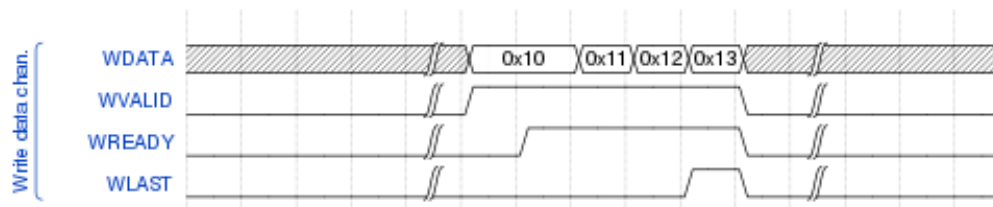


Figure 2.6

Write response channel activates once address and data are both written to the slave, Timing diagram of write response channel along with write address and write data channel looks like as below in figure 2.7

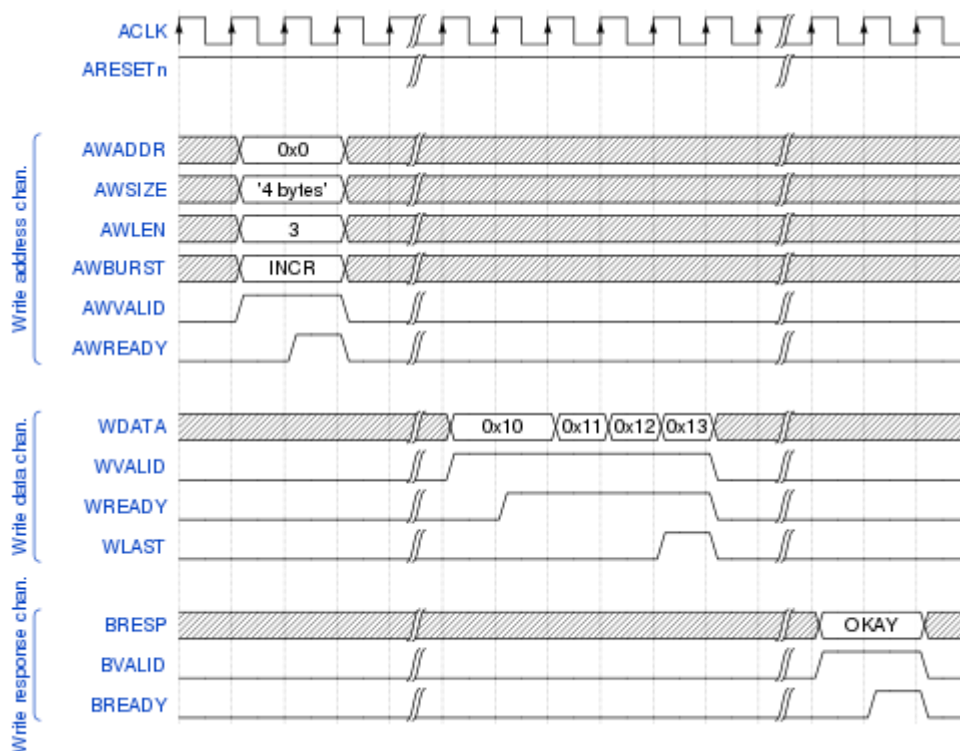


Figure 2.7

2.3 AXI Signals

2.3.1 Write Address Channel

AXI Write address channel defines a set of signals.

- AWVALID
- AWREADY
- AWSIZE
- AWLEN
- AWBURST
- AWADDR

AWVALID and AWREADY are single bit control signals used for handshaking between master and slave. AWSIZE specifies the amount of data needs to be transferred in a single burst and AWLEN specifies the total number of transfers in one single burst.

Burst Size: The maximum number of bytes of data to be transferred in each data beat in a burst.

- AWSIZE [2:0] for write transfers
- ARSIZE [2:0] for read transfers

AxSIZE[2:0]	Bytes in transfer
0b000	1
0b001	2
0b010	4
0b011	8
0b100	16
0b101	32
0b110	64
0b111	128

Figure 2.8

As shown in the table maximum number of bytes that can be transferred in one transfer in a burst are 128 bytes.

Burst Length:

The burst length is specified by

- AWLEN [7:0] for write transfers
- ARLEN [7:0] for read transfers

AXI protocol supports burst transfers up to 256 transfers in single burst. The early termination of burst is not supported.

Burst Type:

The AXI protocol defines three types of burst.

- Fixed
- Incremental
- Wrapping

In a fixed burst, every address is the same for every transfer in the burst, this burst type is used for repeated accesses to the same memory location such as when loading or emptying a FIFO. In the Incrementing burst the address for each transfer in the burst is an increment of the address of the previous transfer. The increment value depends on the size of each transfer. For example, the address for each transfer in a burst with a size of four bytes is the previous address plus four. This burst type is used for accessing the sequential memories. The wrapping burst is similar to incrementing burst except that the address wraps around to a lower address if an upper address limit is reached. The start address must be aligned to the size of each transfer and the length of the burst must be 2,4,8,16 transfers

The burst type is specified by

- AWBURST [1:0] for write transfers
- ARBURST [1:0] for read transfers

AxBURST[1:0]	Burst type
0b00	FIXED
0b01	INCR
0b10	WRAP
0b11	Reserved

Figure 2.9

2.3.2 Write Data Channel

AXI Write data channel defines a set of signals

- WDATA
- WVALID
- WREADY
- WLAST

WVALID and WREADY signal comprises of handshake signals, when both signals are asserted the data is delivered to the slave which is contained in the WDATA signal. WDATA bus can 8,16,32,64,128,256,512,1024 bits wide. WLAST signal indicates the last transfer of a burst.

2.3.3 Read Address Channel

AXI protocol defines the read address channel with set of signals

- ARADDR
- ARVALID
- ARREADY
- ARBURST
- ARLEN
- ARSIZE

The read address channel works exactly as the write address channel, it consists of handshake signals ARVALID and ARREADY. For burst support, ARLEN, ARLEN, ARSIZE provides the parameters for burst transactions.

2.3.4 Read Data Channel

AXI protocol specifies the read data channel as follows

- RDATA
- RVALID
- RREADY
- RLAST

The RDATA consists of the data requested by the master from the slave and slave returns this data from a requested address location, RVALID and RREADY signals perform basic handshake control and RLAST signal indicates the last transfer in the read burst.

2.4 AXI-LITE

The AXI specifications include a low speed memory mapped interface known as AXI-Lite interface. AXI Lite protocol does not provide the burst support so AXLEN, AXSIZE, AXBURST and WLAST are not included in AXI Lite. AXI Lite provides a single beat of transaction in all five channels and ideally suited to configure control and status registers in peripherals that contain AXI interfaces. For example AXI DMA (Direct memory access) consists of set of registers which are required to be programmed in order to run DMA controller, and the only interface that can be used to program DMA controller is S_AXI_LITE (Slave AXI Lite interface) as shown in figure 2.10.

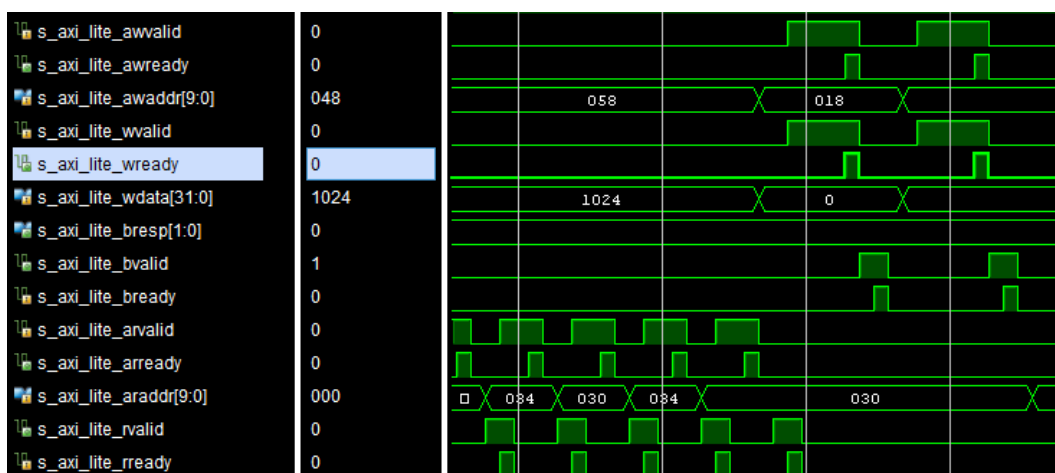


Figure 2.10

2.5 AXI STREAM

AXI specifications define AXI Stream as point to point protocol with unlimited burst support. AXI stream is ideally suited for video and image applications in FPGA. AXI stream does not contain any addressing information and only contains data bus and control signals to control the flow of data.

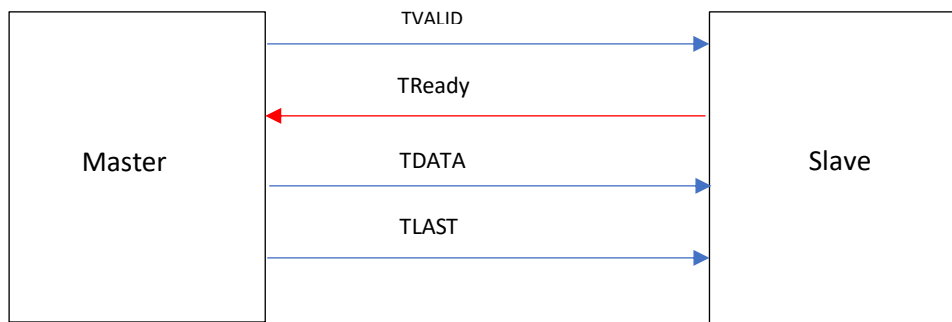


Figure 2.11

AXI stream specifies only one channel with a specific set of signals such as

- TVALID
- TREADY
- TDATA
- TLAST

TVALID and TREADY comprise of basic handshake mechanism, TDATA consists of actual data stream. Note that AXI stream is burst based and capable of unlimited burst, TLAST signal indicate the last transfer in the burst

2.5.1 Timing Diagram of AXI STREAM

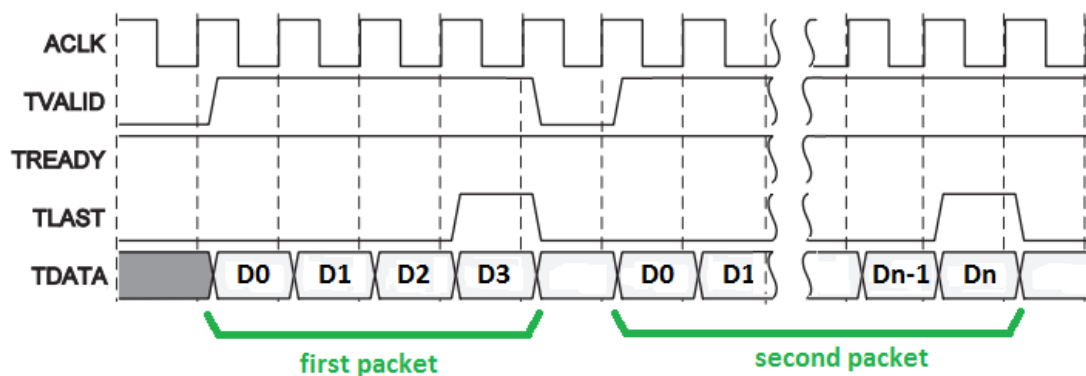


Figure 2.12

Its important to note that once TVALID signal is asserted indicating that master is driving valid data on the bus, it must not deassert the TVALID signal until the transaction occurred. Master must wait for TREADY signal for a transaction and then TVALID signal can be deasserted otherwise Protocol will not function properly and AXI bus can hang. There are some optional signals in AXI Stream such as TSTROBE and TKEEP which indicate which bytes in the data are valid or not valid.

2.6 Signal Dependencies

The AXI specification specifies some very critical dependencies in the protocol which must be followed otherwise it will result in a deadlock situation and AXI bus will hang and not function properly.

2.6.1 Write Transaction Dependencies

AXI protocol follows write transaction dependencies as follows

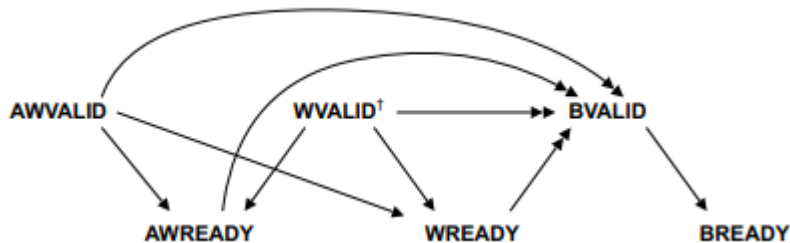


Figure 2.13

As can be seen in figure 2.13, BVALID signal from the write response can only be asserted once WREADY, WDATA, AWREADY, AWVALID signals are asserted indicating both write address and write data transactions have been successful. On the other hand, AWREADY, WREADY, BREADY signals can be asserted before or after AWVALID, WVALID, BVALID are asserted.

2.6.2 Read Transaction Dependencies

AXI protocol follows read transaction dependencies as follows

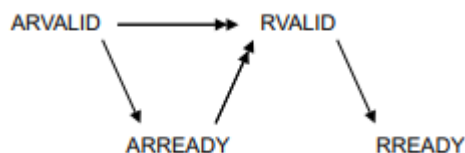


Figure 2.14

As shown in figure 2.14, ARREADY signal can be asserted before or after ARVALID signal. RVALID signal can only be asserted after ARVALID and ARREADY signals are asserted indicating slave can only return the data after master has requested to access the data from the slave from a specific address location. Master can assert RREADY signal before or after RVALID signal from the slave is asserted. RVALID signal must be independent of RREADY signal and must not wait for RREADY to assert RVALID signal.

2.7 AXI DMA

AXI DMA refers to Direct memory access. AXI DMA is a Xilinx soft IP core which provides high bandwidth direct memory access between memory and AXI Stream target peripherals. Its optional scatter/gather mode also offloads data processing tasks from central processing unit (CPU). Initialization and status registers are accessed through a slave AXI-Lite interface (S_AXI_LITE).

2.7.1 Key features of AXI DMA

- AXI compliant
- Optional scatter/gather support
- AXI data width support 32, 64, 128, 256, 512, 1024 bits
- Converts Streaming interface to memory mapped interface
- Provides Burst support

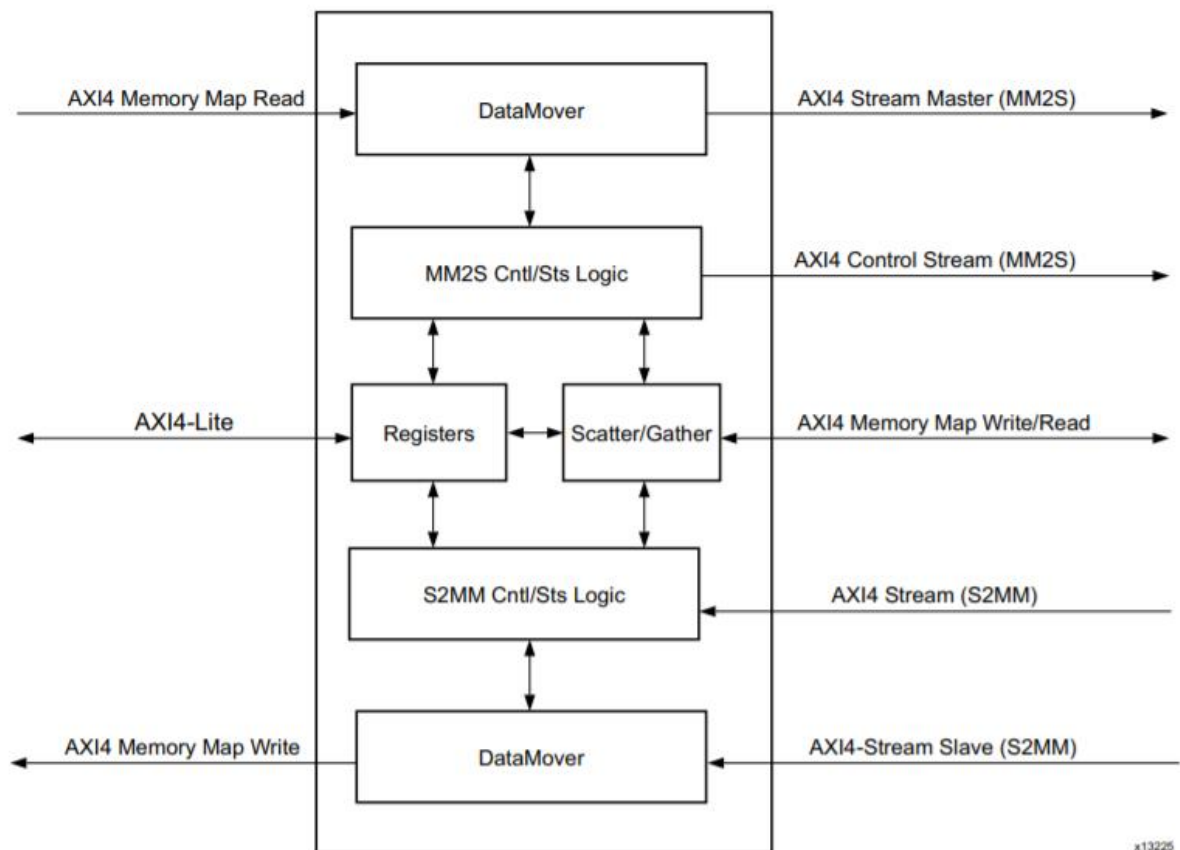


Figure 2.15

AXI DMA is critical component of this project as large number of images are needed to be stored in the memory and read back subsequently and forwarded to another component inside FPGA. Programmable logic side of the Zynq receives these images in the form of AXI Stream through a protocol converter prior to entering the FPGA fabric.

2.7.2 Channels of AXI DMA

AXI DMA contains two channels for writing and reading purposes.

- S2MM (Stream to Memory mapped) for write channel
- MM2S (Memory mapped to Stream) for reading channel

S2MM channel as the name suggests takes in the input data from AXI Stream interface and writes them in to the memory using burst support. Burst length is changeable from 2 beats to 256 beats in a burst. MM2S channel is responsible for reading the data from the memory and outputs the AXI Stream data to be used elsewhere in the design. MM2S channel also uses burst support similarly to S2MM channel. AXI DMA contain specific set of registers which can be programmed to run both S2MM and MM2S channels.

2.7.3 Interfaces of AXI DMA

AXI DMA consists of following interfaces as shown in figure 2.16

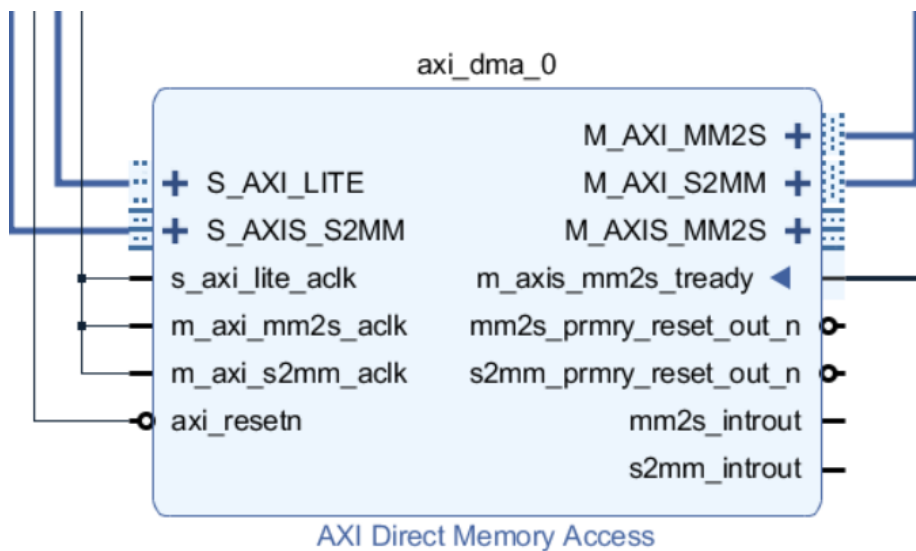


Figure 2.16

- S_AXI_LITE interface is responsible for accessing the control and status registers of the S2MM and MM2S channels
- S_AXIS_S2MM interface is responsible for accepting the input data stream which can come from a camera, ethernet, or any other source which generates AXI Stream.
- M_AXI_S2MM interface is memory mapped interface and uses burst support to access memory and to write input data stream coming through S_AXIS_S2MM into the memory
- M_AXI_MM2S interface performs memory read with burst support and outputs the data as AXI Stream.
- M_AXIS_MM2S interface is responsible to output the streaming data (AXI Stream) to be used elsewhere in FPGA fabric. This data was read from the memory through M_AXIS_MM2S channel

2.8 AXI DMA Registers

AXI DMA consists of two channels S2MM and MM2S which can be accessed through a set of registers inside the DMA.

2.8.1 S2MM Registers

The S2MM channel can be programmed by a set of registers as described below.

- S2MM_DMA_CR (Address offset 30h)
- S2MM_DMA_SR (Address offset 34h)
- S2MM_DA (Address offset 48h)
- S2MM_LENGTH (Address offset 58h)

S2MM_DMA_CR: This is a control register in S2MM channel of AXI DMA and can be used to start the writing channel. It is a 32 bit register as shown in figure below.

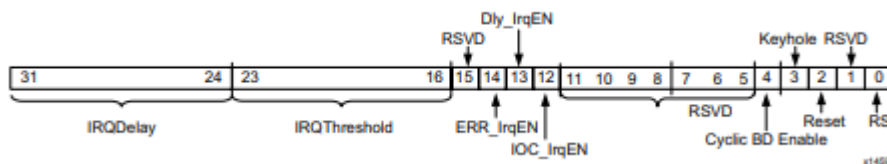


Figure 2.17

The first bit of the register RS is responsible for starting or stopping a channel, programming this bit to 1 will start the S2MM channel and 0 will stop the channel. Third bit of the register is responsible for resetting the DMA controller, asserting this bit to 1 will reset the DMA. Fourth bit is known as keyhole which when asserted to 1 result in S2MM writes in non-incrementing address mode similar to burst type fixed, this bit is non-functional when DMA is used in multi-channel mode. S2MM_DMA_CR contains several other bits intended for different functionalities, only the relevant ones are described here.

S2MM_DMA_SR: This is a status register in S2MM channel of AXI DMA and can be used to check weather a write operation has been successful or not, it is also a 32 bit register similar to control register.

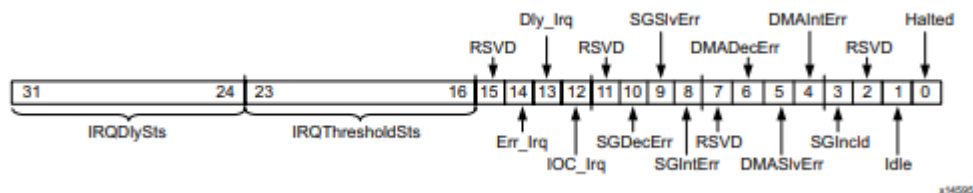


Figure 2.18

The first bit of this register is called a halted bit which when asserted to 1 indicates that S2MM channel is halted and not running whereas 0 indicates vice versa. DMA CR.RS is connected to the DMA SR.Halted. Halted as when DMA CR.RS is 0, DMA SR.Halted is automatically set to 1. The idle bit is the most important which indicates weather the current transfer is complete or not. The fifth bit in

DMA_SR register can be useful in assessing whether DMA produces any internal error or not, DMASR.DMAIntErr bit asserts to 1 when amount of streaming data entering the core does not match with the amount of data programmed in the S2MM_LENGTH register. Bit number 12 is also very critical in checking whether the write transfer is successful or not, its known as IOC_Irq also known as interrupt on complete. After the write transfer is complete this bit asserts to 1. DMASR.IOC_Irq and DMASR.Idle are both used in actual VHDL design to keep track of all the successful transfers

S2MM_DA: This register specifies the destination address in the system memory where data needs to be transferred. It is a 32-bit register, and all 32 bits are allocated for an address of memory.

S2MM_LENGTH: This register specifies the amount of data in bytes to be written in the memory location specified in the register S2MM_DA.

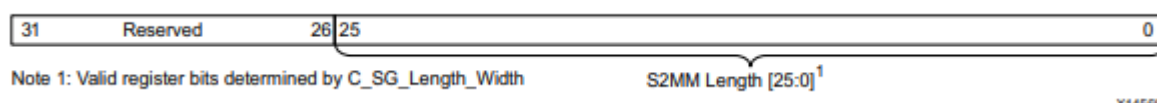


Figure 2.19

S2MM_LENGTH is a 32 bit register in which only first 26 bits are allocated for the total amount of data that can be transferred to the memory. The S2MM_LENGTH must only be programmed in the end, other registers can be programmed in any order. Remaining 6 bits are reserved and writing to them does not result in anything.

2.8.2 MM2S Registers

The MM2S channel can be programmed by a set of registers as follows

- MM2S_DMA_CR (Address offset 00h)
- MM2_DMA_SR (Address offset 04h)
- MM2S_SA (Address offset 18h)
- MM2S_LENGTH (Address offset 28h)

MM2S_DMA_CR: This register specifies the start of reading channel and streams out the data on to M_AXIS_STREAM interface.

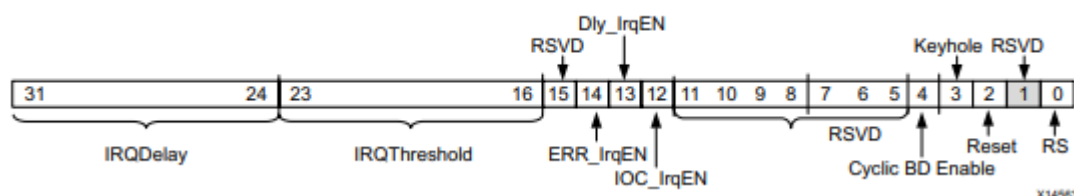


Figure 2.20

The first bit of the register is responsible for starting or stopping a channel by asserting and deasserting a bit respectively similar to S2MM channel. Bit number 1 is reserved meaning writing to it has no effect. Bit number 2 is responsible for resetting a DMA controller if an error is detected.

MM2S_DMA_SR: This register is responsible for checking whether the read transfer is successful or not by checking relevant bits in the register.

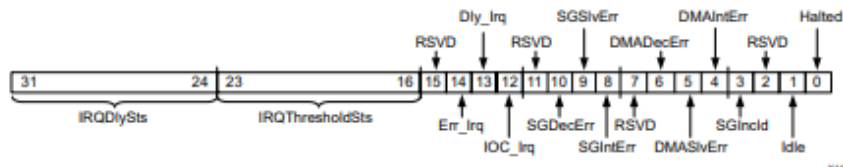


Figure 2.21

The first bit is known as Halted bit similar to S2MM channel and indicates whether the MM2S channel is running or halted. Bit number 1 also known as Idle bit indicates whether the transfer is successful. Idle bit asserts to 1 when a read transfer is successful and stays to 0 in default mode. Bit number 4 also known as DMAIntrErr indicates if MM2S channel produced any error.

MM2S_SA: This register is known as source address register and specifies the address in memory from where data needs to be read. It's a 32-bit register, and all 32 bits are allocated for address.

MM2S_LENGTH: This register is responsible for the amount of data in bytes needed to be read from the memory.

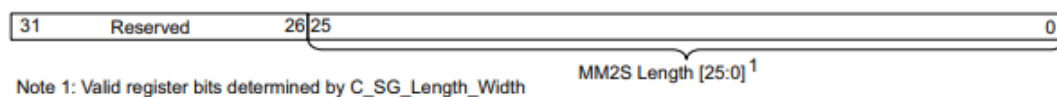


Figure 2.22

MM2S_LENGTH is also a 32 bit register in which only the first 26 bits are used, remaining 6 bits are reserved meaning writing to them has no effect.

2.8.3 Internal Settings of AXI DMA

AXI DMA provides multiple options in its settings. Width of the buffer length register must not be less than the amount of data needs to be written or read. The data bus is also changeable to 1024 bits. The Burst sizes for both channels are configurable

Width of Buffer Length Register (8-23) bits

Address Width (32-64) bits

☒ Enable Read Channel

Number of Channels

Memory Map Data Width

Stream Data Width

Max Burst Size

☐ Allow Unaligned Transfers

☒ Enable Write Channel

Number of Channels

☐ Auto ☐ Memory Map Data Width

Stream Data Width (Auto)

Max Burst Size

☐ Allow Unaligned Transfers

☐ Use Rlength In Status Stream

Figure 2.23

2.9 Structure of the IP core

The structure of the IP core looks like as in figure. The IP core implements a memory access pattern and its written in VHDL. The IP core consists of two main components AXI DMA and a sub IP core which contains several modules of VHDL.

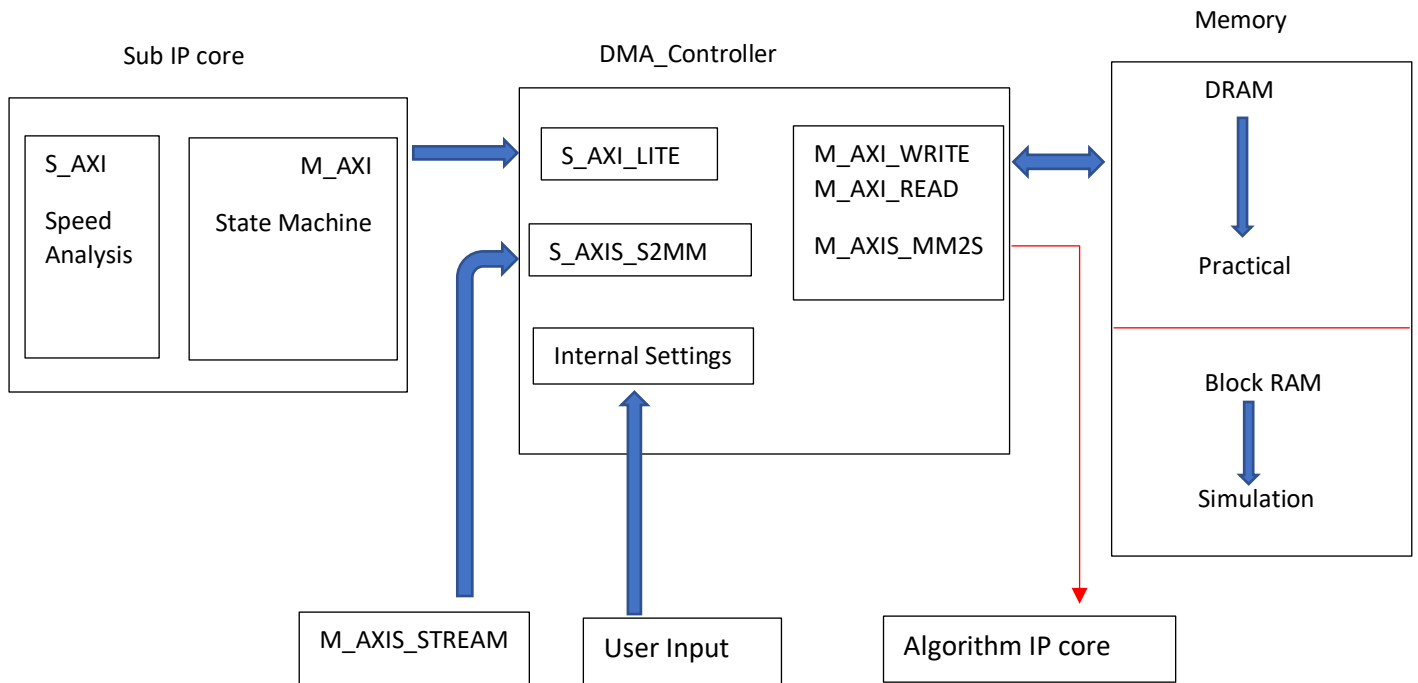


Figure 2.24

2.9.1 Sub IP core

The Sub IP core consists several VHDL modules along with an AXI interface. This sub IP core is named as DMA_DRAM and modules inside are as follows.

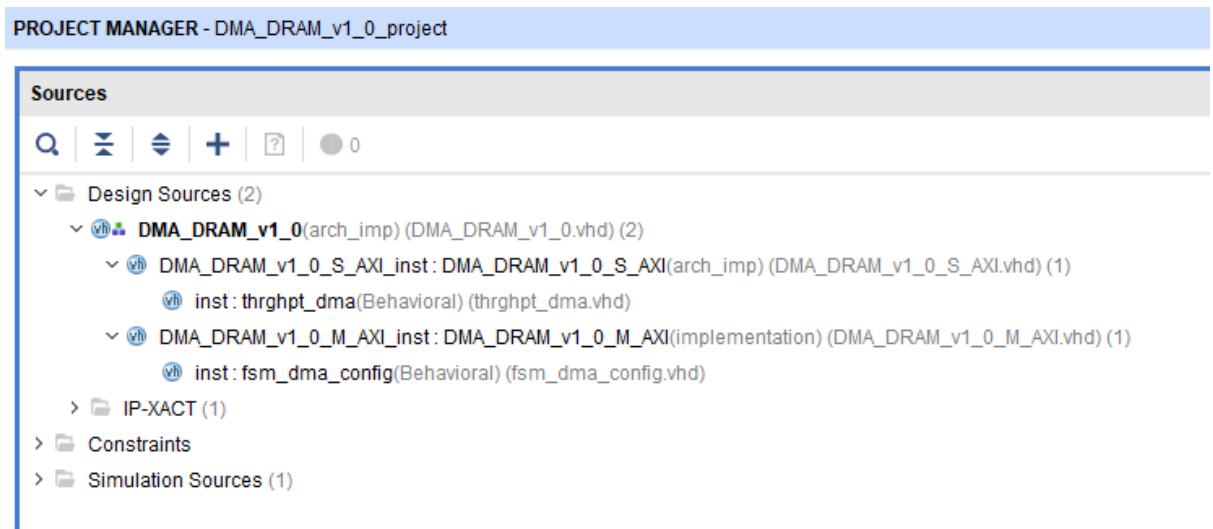


Figure 2.25

The AXI interface needed to program a DMA controller is M_AXI_LITE (Master AXI Lite interface). A state machine is designed to configure the S2MM and MM2S channels. Sub IP core also contains a S_AXI (Slave AXI Lite interface) which is responsible for calculating writing and reading times and to calculate how much time a memory access pattern would take. S_AXI contains memory mapped registers which can be read from Xilinx software command line tool (XSCT console). The design is implemented on both the Block Ram and DRAM. The design was simulated on Block Ram and practically tested on DRAM by programming FPGA.

DMA_DRAM_v1_0: This module is known as the top-level file of the design and contains the instantiations of both master and slave interfaces which are used to program DMA controller and to perform speed measurements, respectively.

DMA_DRAM_v1_0_S_AXI_inst: DMA_DRAM_v1_0_S_AXI: This module is instantiated in the top level file and provides access to memory mapped registers inside which can be read from Xilinx software command line tool to read how many clock cycles it took for a specific memory access pattern to function. This module also contains the instantiation of a module named as thrghpt_dma.

thrghpt_dma: This module contains logic to calculate clock cycles to check how much time it takes for a specific memory access pattern to function. Its also responsible for measuring writing and reading times with different data lengths and different burst sizes by changing different burst sizes in DMA controller.

DMA_DRAM_v1_0_M_AXI_inst: DMA_DRAM_v1_0_M_AXI: This module implements AXI Lite master interface by appropriately following AXI specifications. This module also contains instantiation of fsm_dma_config which is a state machine instantiated with AXI Lite master interface.

fsm_dma_config: This module implements a state machine implementing a memory access pattern.

2.9.2 Block Design Setup

The design was simulated with Block Ram and practically tested on DRAM. The block design with Block Ram looks as follows. One half of the block design is shown in figure 2.26 where AXI DMA is connected to Block Ram using AXI BRAM controller.

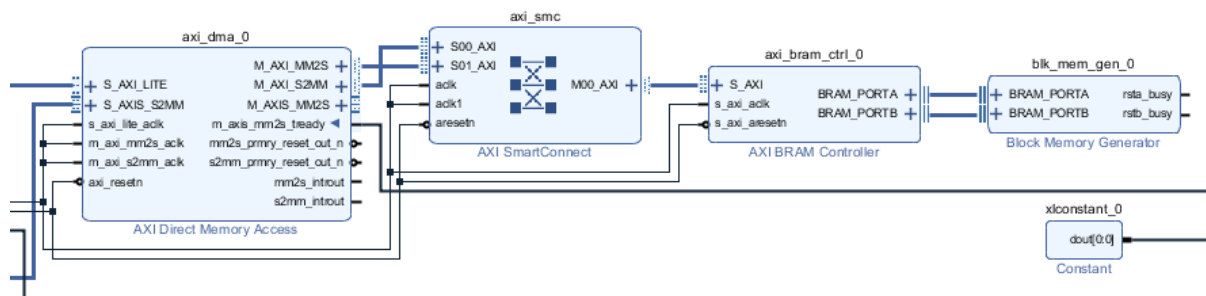


Figure 2.26

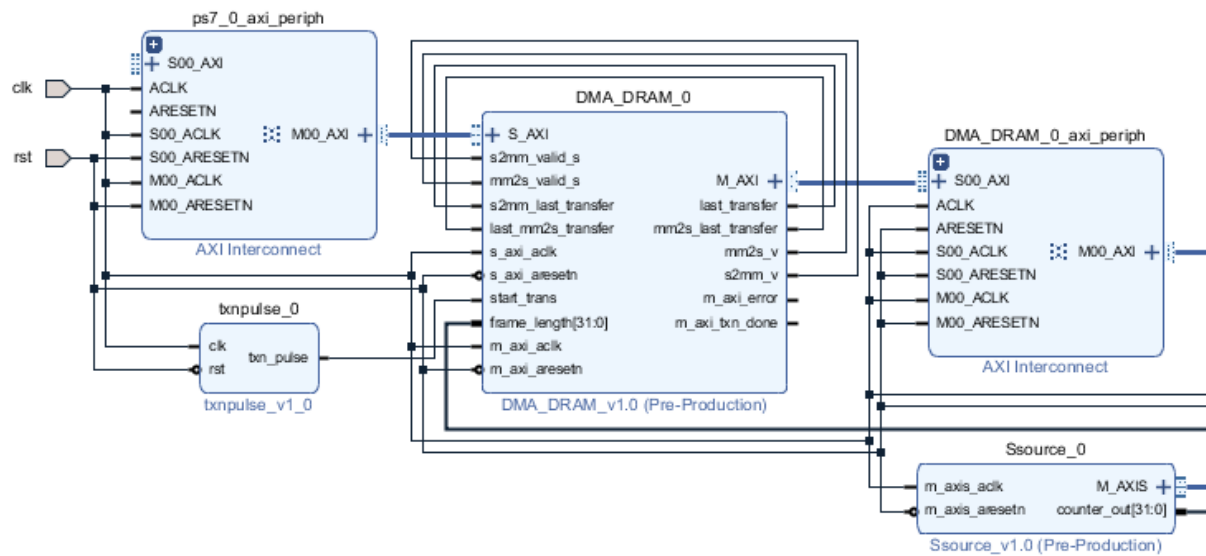


Figure 2.27

The figure 2.27 depicts the other half of block design with Block Ram where DMA_DRAM IP core is connected to AXI DMA using an AXI Interconnect. Ssource_v1.0 is the AXI Stream master which generates streaming data for simulation and is AXI compliant. The one half of block design setup with DRAM looks as follows.

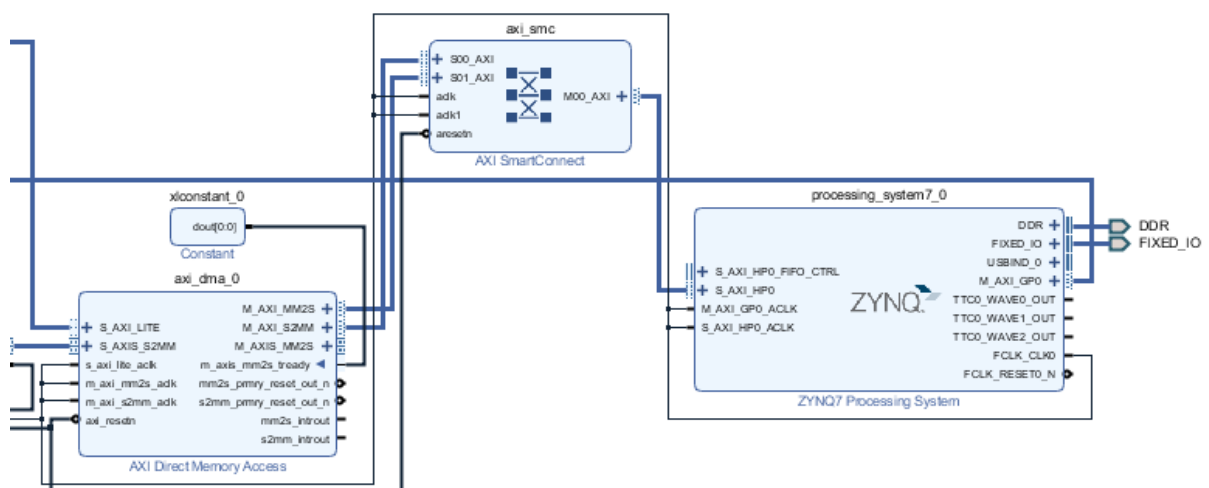


Figure 2.28

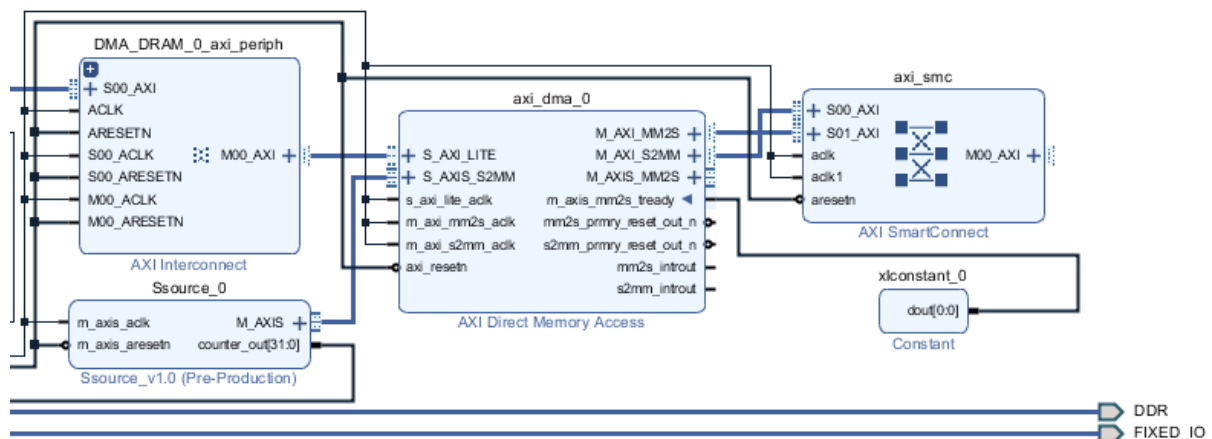


Figure 2.29

The figure 2.29 depicts the other half of block design where AXI DMA is connected to DRAM through Zynq PS

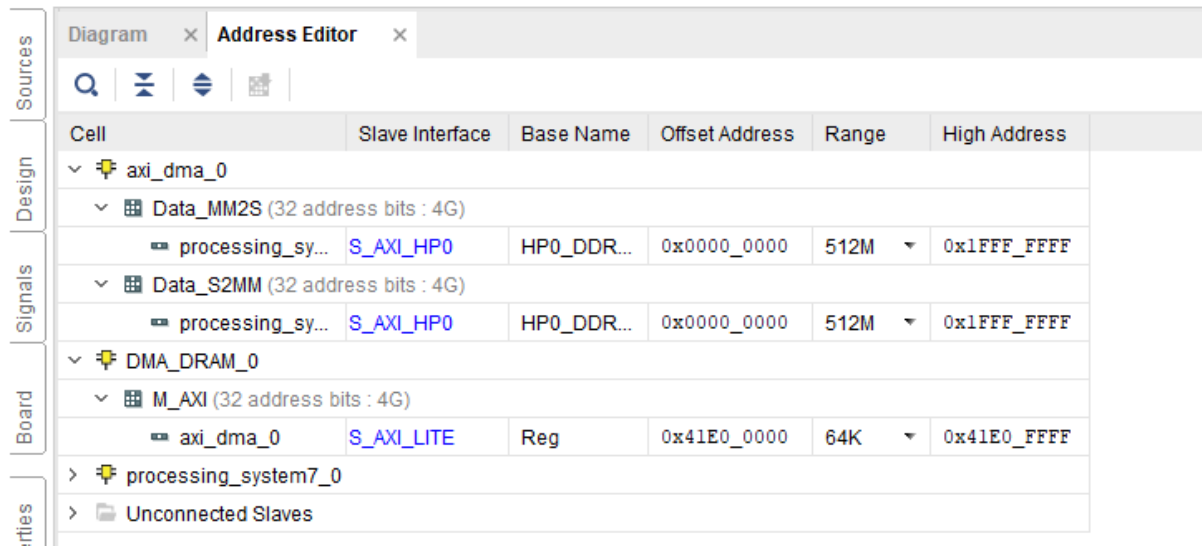
2.9.3 Design Changes with BRAM and DRAM

While developing a block design with BRAM and DRAM, several changes were made for each of the memory. As BRAM resides inside the fabric so it is perfectly simulatable whereas DRAM lies outside the chip and cannot be simulated. While developing a block design with BRAM, clock and reset signals are provided by test bench and the design can be simulated. With DRAM clock and reset signals come from the Zynq PS. Zynq PS generates a clock for the PL side which is typically in the range of 0-250 MHz. The Address editor in VIVADO also requires changes with both memories as the address of BRAM needs to be the same as the address defined in state machine in IP core. For DRAM VIVADO automatically detects the address range of DRAM and only base address inside the state machine of IP core should be the same as of DRAM which is 0x00100000.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
Data_S2MM (32 address bits : 4G)					
axi_bram_ctrl_0	S_AXI	Mem0	0xC000_0000	8K	0xC000_1FFF
DMA_DRAM_0					
M_AXI (32 address bits : 4G)					
axi_dma_0	S_AXI_LITE	Reg	0x41E0_0000	64K	0x41E0_FFFF

Figure 2.30

As can be seen in the figure 2.30, offset address indicates the base address of memory in case of BRAM it should be the same as in state machine design in IP core. For a DRAM, the offset address is automatically adjusted to 0x00000000 and range is 512 MB in case of ZED Board. Different evaluation boards have different sizes of DDR memories.



The screenshot shows the 'Address Editor' window with a tree view on the left and a table of memory addresses. The tree view includes 'axi_dma_0', 'Data_MM2S (32 address bits : 4G)', 'Data_S2MM (32 address bits : 4G)', 'DMA_DRAM_0', 'M_AXI (32 address bits : 4G)', 'axi_dma_0', 'processing_system7_0', and 'Unconnected Slaves'. The table has columns for 'Cell', 'Slave Interface', 'Base Name', 'Offset Address', 'Range', and 'High Address'.

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
axi_dma_0					
Data_MM2S (32 address bits : 4G)					
processing_sy...	S_AXI_HP0	HP0_DDR...	0x0000_0000	512M	0x1FFF_FFFF
Data_S2MM (32 address bits : 4G)					
processing_sy...	S_AXI_HP0	HP0_DDR...	0x0000_0000	512M	0x1FFF_FFFF
DMA_DRAM_0					
M_AXI (32 address bits : 4G)					
axi_dma_0	S_AXI_LITE	Reg	0x41E0_0000	64K	0x41E0_FFFF
processing_system7_0					
Unconnected Slaves					

Figure 2.31

As can be seen in the address editor the address for DRAM is shown as from 0x00000000 to 0x1FFFFFFF. The reason is that it overlaps the addresses with OCM (On chip memory) whose base address is 0x00000000. The base address of DRAM is 0x00100000 and it overlaps with memory region of OCM.

Chapter 3

This chapter focuses on the technique required to implement memory access pattern and approach to solve the problem. This chapter also discusses the measurements taken in context of speed analysis where memory access times with different settings and different access patterns are discussed.

3.1 Memory Access Pattern

The desired memory access pattern is described in figure below.

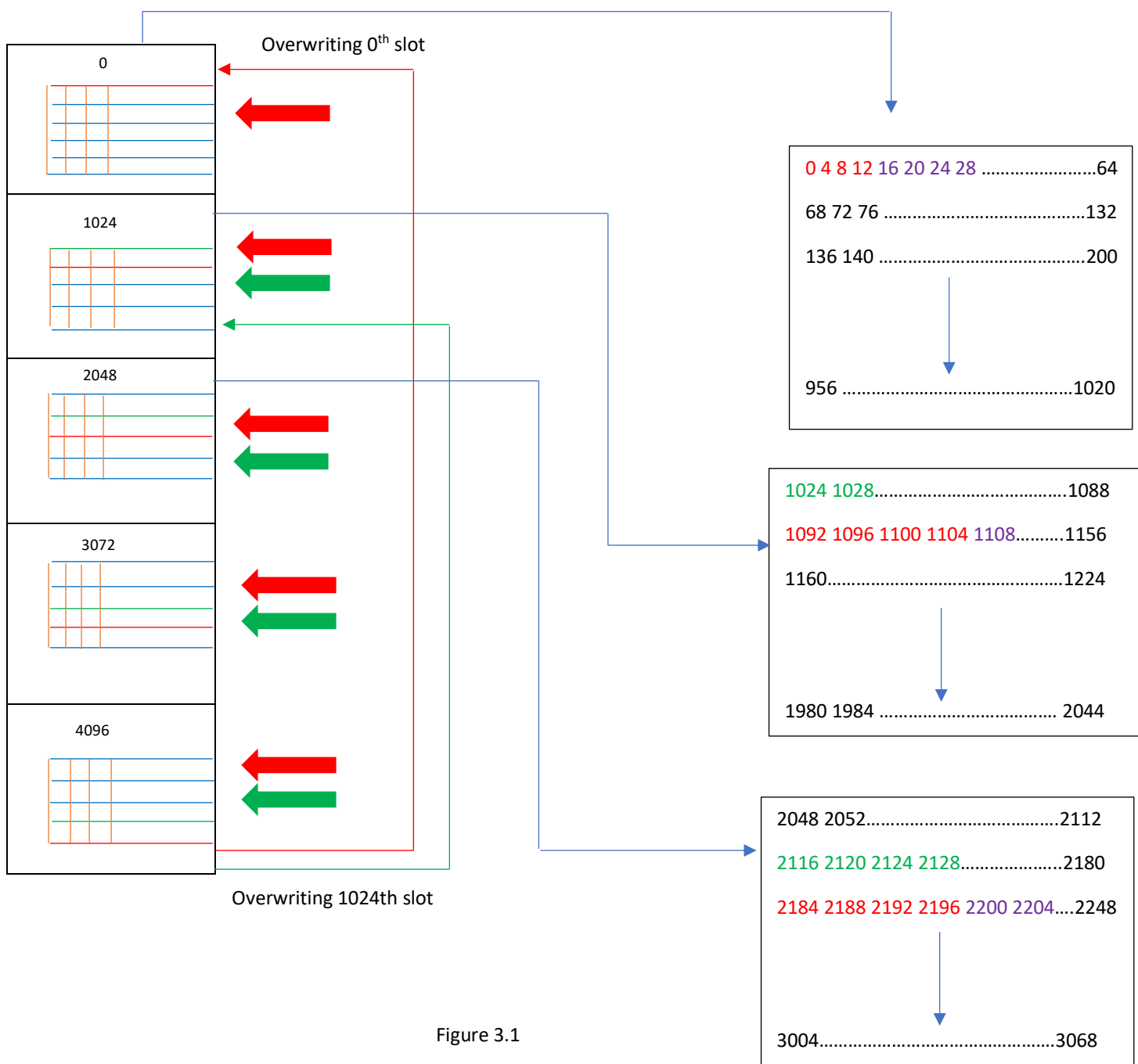


Figure 3.1

For simplicity only five slots of the buffer are shown in figure 3.1. The buffer is a data structure implemented on both DRAM and BRAM. The buffer works like ring buffer but with a very specific memory access pattern. For simulation, the images that AXI Stream master generates are 1Kilobyte in size which means there are total 256 pixels in every image each pixel being 4 bytes. If 1 Kilobyte of image is written at the 0th slot in buffer/RAM, then next image must be directed to 1024th location in buffer/RAM. The right side of figure 3.1 depicts the zoomed version of images inside the slots of buffer which shows how the images inside the buffer look like with their addresses. As each pixel is 4 bytes and AXI DMA performs writes/read in burst, the burst type is always incrementing which means that the next address for a next pixel is previous address plus the size of transfer in a burst. For example, the first pixel of the first image is stored in 0th location (Base location) in memory, the next pixel is stored in 4th location in memory, the next in 8th location and so on. With 1Kilobyte of image there are 256 pixels which means that 256th pixel of first image is stored in 1020th location. The second image must be directed to 1024th (Base location) in the memory. The first pixel of second image is stored in 1024th location in memory, the second in 1028th location and 256th pixel of second image in 2044th location and 2048th is now the base location of third image and so on. The figure 3.1 gives a concise idea as how the pixels are arranged inside the memory which makes it easier to read and write in order to implement a specific memory access pattern. According to a desired memory access pattern, only the chunk of four pixels are needed to be read from every row of every image. For example in the first iteration, the first four pixels from the first row of the first image are read then first four pixels from the second row of second image are read then first four pixels of third row of the third image are read and so on. After all the elements of buffer are read then next four pixels of first row of first image are read, then next four pixels of second row of second image are read and subsequently next four pixels of third row of the third image are and so on for the remaining elements of the buffer. With 1Kilobyte of image there are 16 pixels in every row and column of the image. If the 4 pixels are read in every iteration then total 4 iterations are required in order to read full first row of first image, second row of second image, third row of the third image and so on. After all the rows are read in this manner then the image at 0th location needs to be overwritten with a new image and the slot next to overwritten becomes the first slot. After overwriting the 0th location, the first row of image stored at 1024th location is required to be read, second row of image stored at 2048th location, third row of image stored at 3072nd location and so on until the last image in this case would be at 0th location which means fifth row of the image stored at 0th location is required to be read and all the reads must be in chunk of 4 pixels as described earlier. The pixels are read in chunks of 4 because it is a requirement of an algorithm IP core running on FPGA which will receive these pixels from this IP core and calculate surface roughness. The algorithm IP core can only accept pixels from arbitrary locations in order to be able to perform as intended. The memory map of pixels and their address locations is best described in figure 3.1. As each slot is overwritten after all the pixels of all the respective rows are read this creates a ring buffer-based data structure as the pointer rotates in a circular fashion. The figures below depict how this data structure looks like in simulation on BRAM.

- Overwriting 0th slot

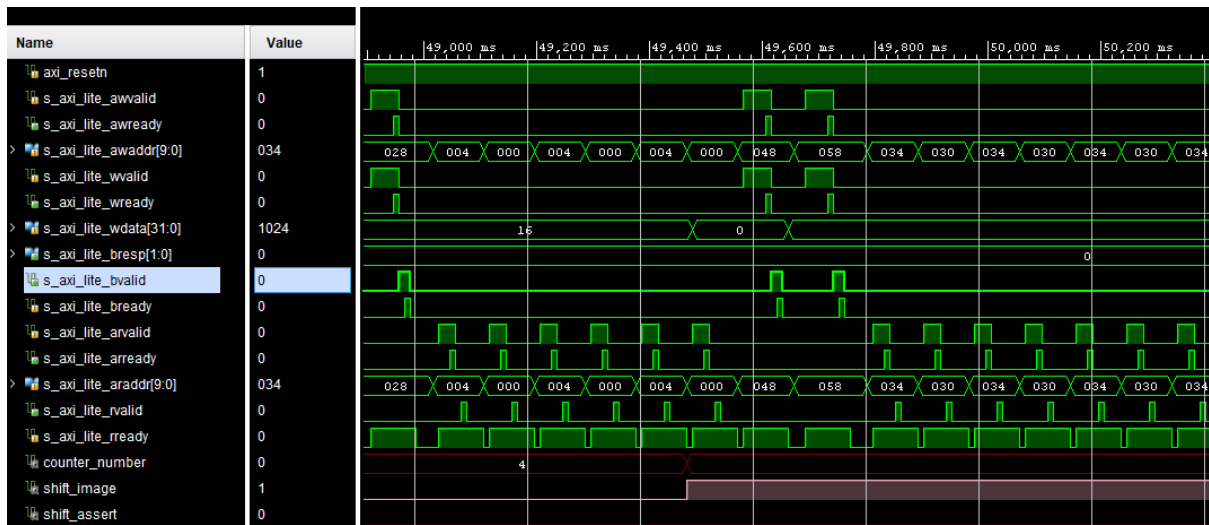


Figure 3.2

The simulation trace shown in figure 3.2 depicts the overwriting the 0th slot after the first iteration is complete which means all the respective rows of all the images in slot are read in a specific pattern as described earlier. The “shift_image” signal in pink colour indicates when the slot needs to be overwritten. The S2MM registers 48h and 58h correspond to destination address and length register of S2MM channel.

- Overwriting 1024th location

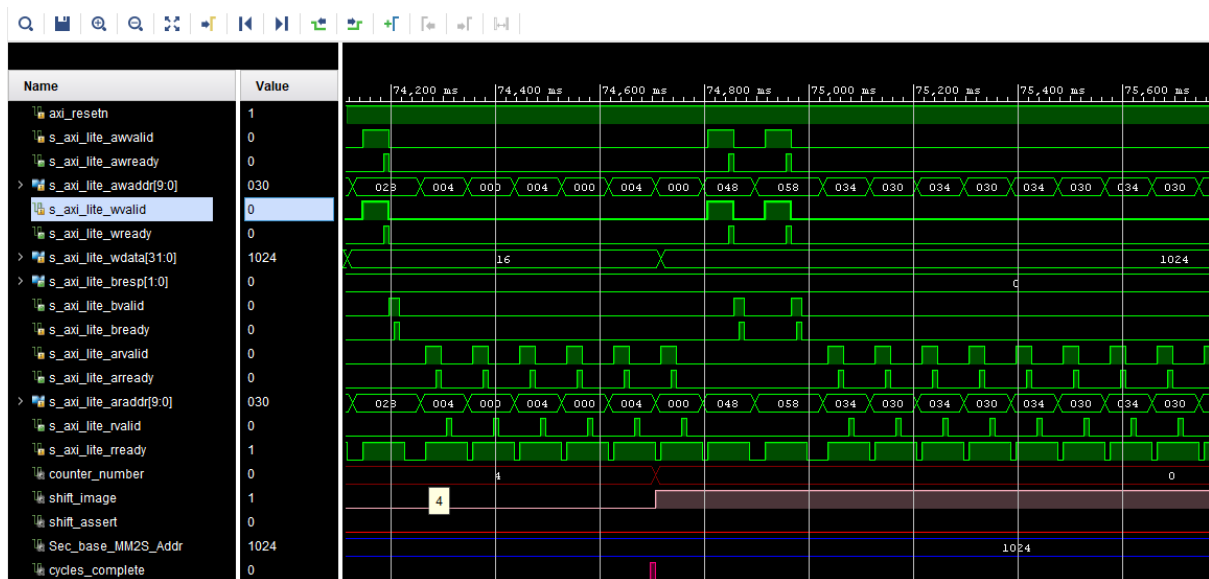


Figure 3.3

The simulation trace in figure 3.3 depicts the second over writing at 1024th slot after all the respective rows of all the images are read according to a desired access pattern. In this case the first image was located at 1024th position and the last image was located at 0th slot. The cycles_complete signal indicates the completion of iteration when 4 pixels from every respective row of every image are read. To read the full rows of all the images 4 iterations are required. The same principle applies to other remaining slots as well. In order to change the number of pixels needed to be read, the

number of iterations are required to change as well, for example if chunks of 2 pixels are read from every row of every image then total 8 iterations are required to read 16 pixels of every row.

3.2 Design Methodology

The approach to solve this task and to implement this memory access pattern a state machine is designed. A state machine is embedded in the module “fsm_dma_config”. After the first iteration when all the rows are read, 0th slot is overwritten and design flow is shown in figure below, For the full iteration of all the images in buffer state machine design is described in section 3.3.

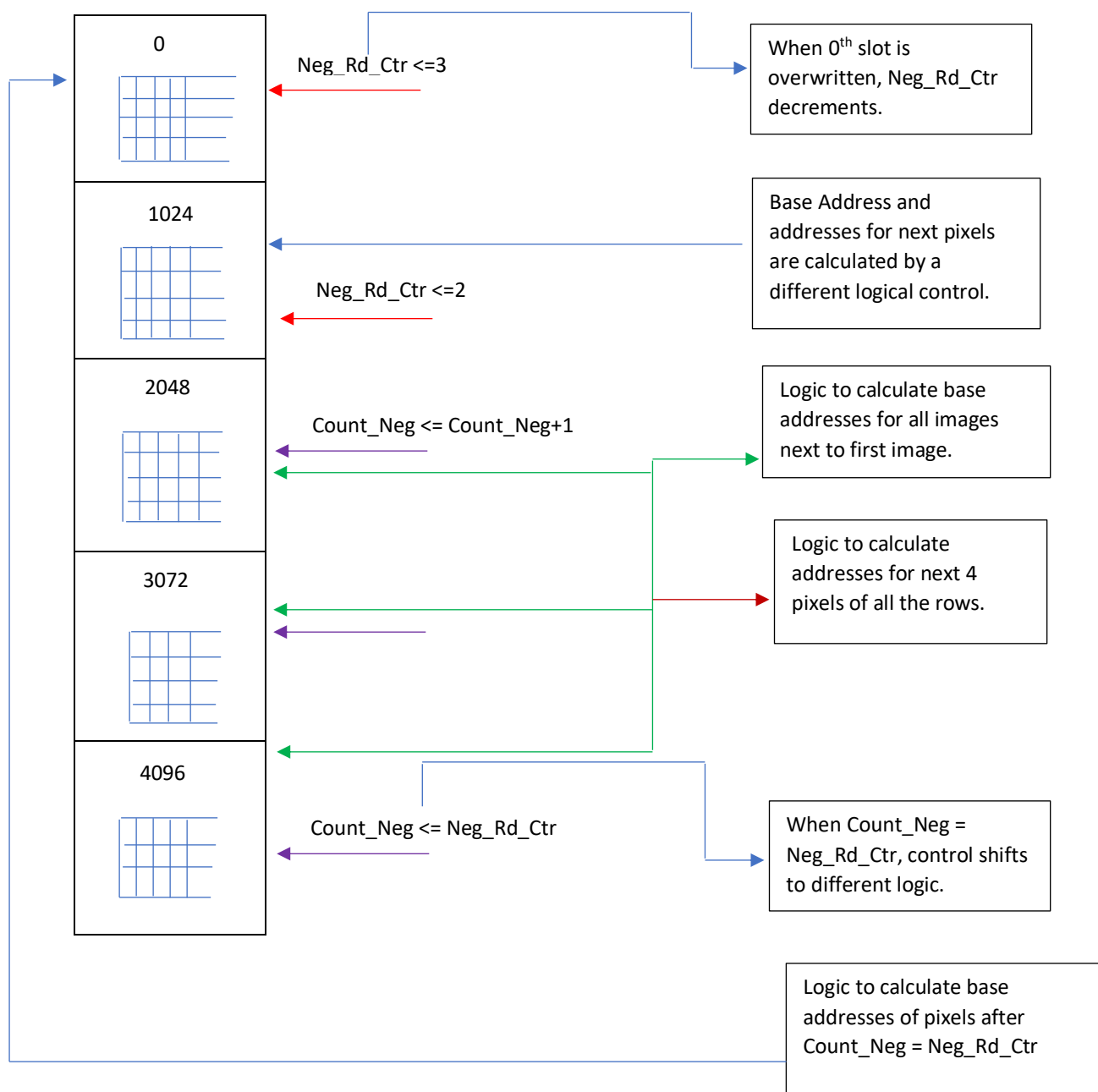


Figure 3.4

The design flow shown in the figure 3.4 depicts the simplified version of the actual design and only explains when the 0th slot is over written, When the remaining slots of the buffer are overwritten buffer wraps around and logical flow will be presented in section 3.3. As shown in figure 3.4, the design is split into several control segments, each control segment is responsible for a logical functionality in the actual design. The design is fundamentally divided into two parts, Base address calculation and next pixels calculation. Base addresses of pixels which are first address of first pixel of the row as the DMA controller reads in burst it automatically reads the next 3 pixels. The next 4 pixels addresses are calculated through a counter logic where several iterations of rows are done to read required number of pixels. There are two counters extensively used in this design, Neg_Rd_Ctr and Count_Neg. Neg_Rd_Ctr functions in reverse order thus implementing a reverse logic. When the 0th slot is overwritten the Neg_Rd_Ctr decrements from an initialized value of buffer_depth -1. The buffer depth is the total capacity of the buffer and can not exceed the size of DRAM. The slot next to overwritten is handled by a different logic control. The base address for slot next to overwritten which is now the first image is calculated separately as well as next 4 pixels for this slot. After the pixels are read from slot next to overwritten, different logical control takes over and calculates the base addresses for the images till Count_Neg=Neg_Rd_Ctr. Count_Neg increments every time when the pixels from remaining images are read. These remaining images in this case where 0th slot was overwritten will be located at 2048th, 3072nd 4096th location in the buffer. When Count_Neg reaches at 4096th location its equal to Neg_Rd_Ctr and control shifts to a different logic where base address for last image which is located at 0th slot will be calculated. When the 1024th location is overwritten, 2048th location is now a first image and the base address and addresses for next 4 pixels are calculated by a different logical control as it was done when 0th slot was overwritten. When 1024th location was overwritten Neg_Rd_Ctr decrements to 2 from 3. When pixels from 2048th location are read, control shifts to a different logic where base addresses for pixels at 3072nd and 4096th location are calculated. Count_Neg increments and becomes 2 after reading pixels from 4096th location. At this stage its again equal to Neg_Rd_Ctr and control shifts to another logic where base address for image at 0th slot is calculated. This time pixels from 4th row of image at 0th slot is needed to be read and the image at 1024th location is the last image, and its 5th row needs to be read. The read pointer used in the design is critical as its used in calculating addresses from necessary rows of the images as required. The read pointer increments every time when a reading of pixels is successful. When the read pointer reaches to the last element of the buffer, a signal known as cycles_complete asserts indicating one full iteration is completed and then the remaining pixels of respective rows are read. When the 3072nd location is overwritten, Neg_Rd_Ctr becomes zero. As already explained the addresses of pixels for slot next to overwritten are calculated by a different logic control so base address and next addresses of 4096th location is calculated without any problem. At this stage addresses for pixels from 0th location till 3072nd location is controlled by a different logic and when all the rows are read, 4096th location is overwritten and now buffer once again starts from 0th location being the first image and 4096th location being the last image. Entirely different control structure calculates addresses of pixels in this case when all the respective rows according to access pattern are read then the image at the 0th slot is once again overwritten and the buffer repeats exactly the same way as it did in the first place.

3.3 State Machine Design

The state machine design is described in figure below.

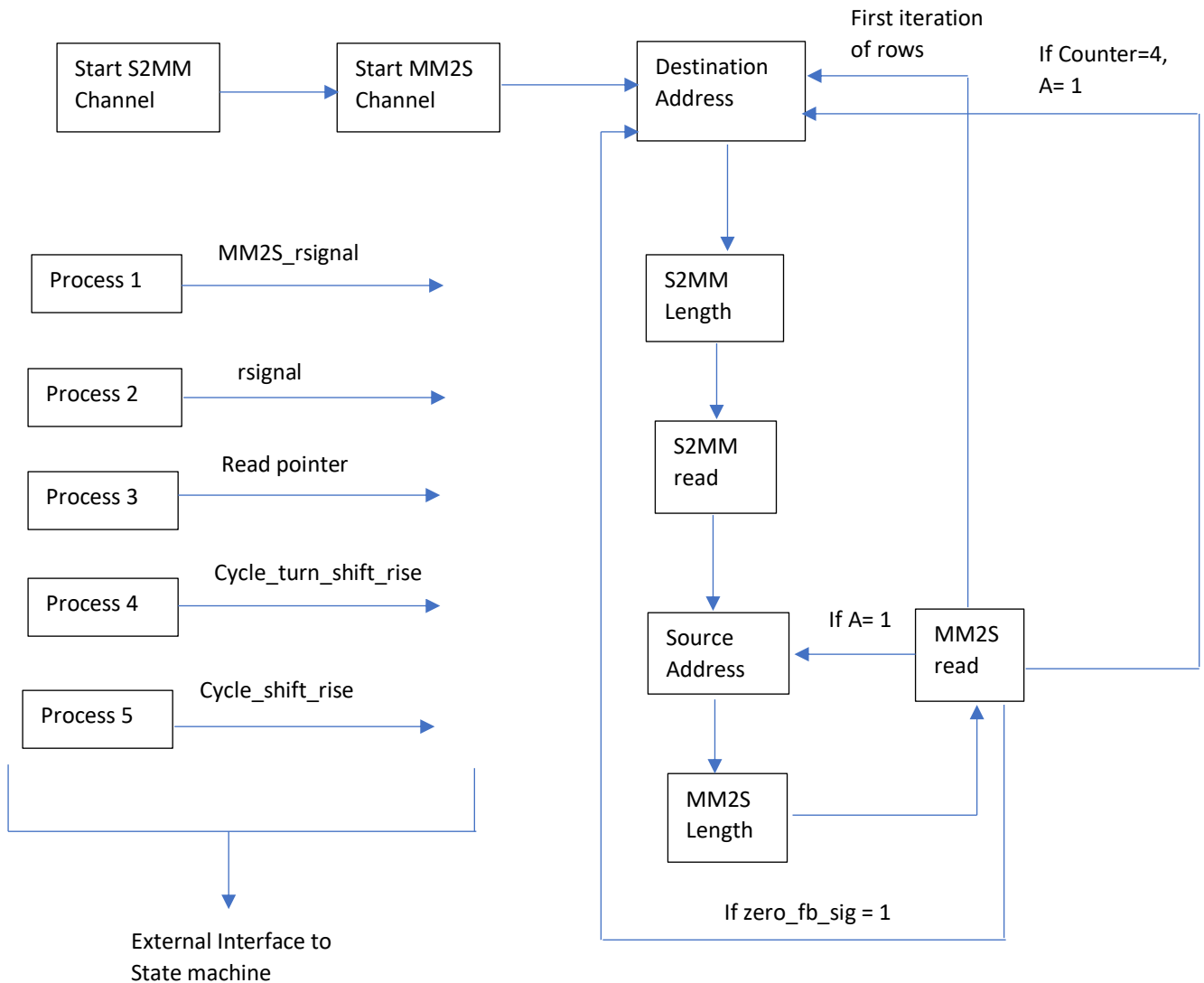


Figure 3.5

The above state machine describes a flow between states and external interfaces with the state machine. Several signals are input to the state machine and vice versa. Each state consists of several nested statements implementing a desired logical functionality intended for a specific state. Below is a precise description of each individual state. The VHDL source code is included in the appendix.

3.4 Description of State Machine

Each state in the state machine is implementing several logical functionalities, description of the states is precisely explained below.

- **Start S2MM:** This state starts the S2MM channel by programming the control register inside the DMA controller.
- **Start MM2S:** This state starts MM2S channel by programming its control register in DMA controller.
- **Destination Address:** This state configures the address in memory where the image is going to be stored after every time image is overwritten. This state also checks whether all the elements in the buffer are overwritten or not and then calculating the address for 0th slot all over again
- **S2MM Length:** This state is responsible for configuring the amount of data needed to be written, for example in this case, 1Kilobyte of data would consist of 256 pixels, 16*16 image.
- **S2MM Read:** This state is responsible for checking whether the image has been successfully written to the memory slot or not. This state also configures the addresses for slot next to overwritten. Neg_Rd_Ctr also decrements in this state by 1 after every time overwrite occurs.
- **Source Address:** This state is responsible for calculating addresses for all the pixels reading. There are 5 different control segments for calculation of addresses for pixels. For a first iteration a control is handled by “mm2s_asignal” statement and to calculate addresses for remaining pixels of the rows control is shifted toward (mm2s_asignal and mm2s_other_rows) as referred in the appendix. When the image is overwritten the base addresses for pixels are controlled by (mm2s_asignal and shift_assert) statement and remaining pixels addresses are calculated by (mm2s_asignal and mm2s_other_rows) statements. When the last element of the buffer has been reached, control is shifted toward (mm2s_asignal and shift_reverse) statement for base addresses and to calculate addresses for the remaining pixels control is shifted toward the (mm2s_asignal and mm2s_shift_other_rows).
- **MM2S Length:** This state is responsible for configuring the amount of data in bytes needed to be read, in this case 16 bytes (4 pixels) are read in every read transfer.
- **MM2S Read:** This state checks whether a read is successful or not. This state also checks whether a cycle is completed which means that one iteration of reading chunks of pixels from respective rows is successful. It also controls the address calculation for next 4 pixels for the slot next to overwritten and increments the counter every time an iteration is completed. When the cycle is completed an image is overwritten by triggering a set of control signals controlling segments in Source Address state. It also controls the value of addresses when all the slots of buffer are overwritten, and buffer starts all over again by overwriting again at the 0th slot. This state also manages address offset values needed to be used in Source Address state to calculate the appropriate addresses as well as it manages the reverse logic controlled by Neg_Rd_Ctr and Count_Neg counter as referred in the appendix.

3.4.1 External Interface to the State Machine.

This section describes the external interface controlling a state machine and vice versa. There are a set of processes in VHDL which are controlling a set of control signals including read pointer and other important signals used within the state machine.

Process 1: This process is incrementing a read pointer and wrapping it to zero when pointer reaches to the last element of the buffer.

Process 2: This process is responsible for checking whether a successful write has occurred or not by checking a status register and then asserting a control signal known as rsignal.

Process 3: This process is responsible for checking whether a successful read has occurred or not by checking a status register, This process also asserts a cycles_complete signal by checking whether a read pointer has reached to the last element of the buffer or not.

Process 4: This process triggers when a shift_image signal changes. Shift_image signal asserts when an image is overwritten in the slot. This process is sensitive to the rising edge of the shift_image signal. This process is critical in calculating base addresses for pixels in conjunction with reverse logic implemented by Neg_Rd_Ctr and Count_Neg.

Process 5: This process is sensitive to falling edge of shift_image signal. This process is critical in calculating addresses for the remaining pixels in conjunction with reverse logic implemented by Neg_Rd_Ctr and Count_Neg.

Additionally, there are set of other processes controlling AXI write and read transactions and to calculate clock cycles required to check the speed of memory access with this access pattern and with different settings.

3.5 Analysis of Different Access Patterns

As described in the previous sections the access pattern required for this task was to read 4 pixels from 1st row of first image, 2nd row of second image and so on and then the overwriting occurs and buffers revolves in circular fashion. Another access pattern implemented was to jump between the rows of the image, for example first row of the first image, 4th row of 2nd, 8th row of third image and so on. By alternating between the rows several access patterns were implemented in VHDL design. The goal of implementing these different access patterns was to calculate the speed and amount of time it takes to read the whole buffer. Alternatively, the goal was also to change the chunks of pixels read in every iteration from 4 pixels to 2 or 8 pixels and calculating the speed with these number of pixels. Several measurements were made with different pixel sizes, different burst settings of DMA controller and with different access patterns. Interesting results were found regarding the different access patterns. The time it took for the desired access pattern was exactly the same as the time it took for access patterns where rows were skipped. For example, the total time the buffer took when 1st row of 1st image, 2nd row of 2nd image pattern was compared with a 1st row of 1st image, 4th row of 2nd image, 8th row of 3rd image, the results were surprisingly exactly the same. These observations were verified by implementing an access pattern where 1st row of 1st image, 6th row of 2nd image, 9th row of 3rd image was read, and all these different patterns resulted in the same speed. Although measurements with different pixel reading from every row were different as discussed in next section.

3.6 Measurements for Speed Calculation

Several measurements are recorded both on Block RAM and DRAM where write and read times are calculated with different data sizes and with different burst settings of DMA controller. Additionally, the time the whole ring buffer took was also calculated with different number of pixels read in every iteration.

Buffer Sizes	Number of Pixels 2	Number of Pixels 4	Number of Pixels 8
50	3073 us	103.6 us	125.7 us
100	13235.6 us	768.4 us	4764.4 us
150	29636.4 us	16879.4 us	10525.27 us

Table 1. Time for different pixels reading

The above table indicates the total amount of time it takes for the whole buffer with different pixels read in every iteration. When 4 pixels are read in every iteration buffer runs the fastest. This has been verified with different buffer sizes except when the buffer size is 150. With the buffer size 150, when 8 pixels are read from every row in every iteration buffer produces the fastest speed. Several measurements were also taken to calculate the writing and reading times with both Block RAM and DRAM and a through comparison was evaluated in order to check the speed with different burst sizes of DMA controller. These measurements were for 5 consecutive writes and 5 consecutive reads from the same address locations. For example, images were written to 0th, 1024th, 2048th, 3072nd, 4096th locations and the images were read from the same locations in the ascending order. Speed was evaluated with BRAM and DRAM both.

#50 Kilobytes of data

Burst Sizes	BRAM	DRAM
256	288.4 us	234.14 us
128	289.8 us	232.97 us
64	294.64 us	232.4 us
8	371.1 us	239.01 us
4	459 us	444.4 us
2	705.34 us	784.97 us

The above table describes the total time consecutive writes and reads into both BRAM and DRAM with 50 kilobytes of data written in every next address location. In all the burst sizes of DMA controller DRAM appears to be faster than BRAM whereas with burst size 2 BRAM appears to be faster than DRAM. This observation can be verified with a different data size of 15 kilobytes as well.

#15 Kilobytes of data

Burst Sizes	BRAM	DRAM
256	89.39 us	72.87 us
128	88.8 us	71.7 us
64	89.8 us	70.02 us
8	112.44 us	72.71 us
4	163.9 us	134.07 us
2	212.55 us	236.37 us

The above table describes the writing and reading times from 5 consecutive writes and reads from consecutive address locations, As can be seen with 15 Kilobytes of data the same pattern can be seen as with 50 Kilobytes. The DRAM appears to be slightly faster than BRAM with all burst sizes of DMA controller except burst size 2 where BRAM appears to be slightly faster than DRAM.

Chapter 4

Conclusion

This master thesis was aimed at developing an IP core which was implementing a specific memory access pattern. This IP core was also calculating speed of access pattern with both BRAM and DRAM and evaluating speed with different settings for example burst sizes and different pixels reading along with different access patterns. Interesting results were found regarding different access pattern where different rows were jumped to read pixels. Time taken for the buffer to function was the same as with the first memory access pattern. Measurements with different pixels reading produced also quite some interesting results as the reading 4 pixels from every row in the age seems to be the fastest option until buffer size 150 where reading 8 pixels from every row produced fastest results. There were also interesting results with different burst sizes in benchmarking comparison with BRAM and DRAM where write and read times of both memories are calculated with different data sizes and different burst sizes. Burst size 2 yielded a very interesting result which showed that BRAM was faster than DRAM whereas DRAM appeared to be faster than BRAM with other burst sizes.

4.1 Future Recommendations

- The memory access pattern was implemented in VHDL language. Zynq PS offers a very powerful set of ARM cores, future recommendations could be a software implementation running on ARM cores and comparative analysis with VHDL design running on PL side of the Zynq.
- As the current memory access pattern works on the reading of rows and chunks of pixels are read from all respective rows and buffer revolves in circular fashion. Reading from columns in this way could produce different results although reading from columns in current setup will be extremely slow as the DMA controllers reads in burst, the first 4 pixels in the column are not possible to read as the addresses are incremented according to AXI protocol.
- Development of a system that can read from columns in the same way as reading from the rows can result in interesting findings and can prove to be a good comparative analysis.

References

1. <https://www.polytec.com/eu/surface-metrology/technology/>
2. http://www.gstitt.ece.ufl.edu/courses/fall15/eel4720_5721/labs/refs/AXI4_specification.pdf
3. https://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf
4. <https://www.osapublishing.org/ao/abstract.cfm?uri=ao-39-22-3906>
5. https://www.researchgate.net/publication/342848253_Lateral_scanning_white-light_interferometry_on_rotating_objects
6. https://www.xilinx.com/support/documentation/user_guides/ug585-Zynq-7000-TRM.pdf
7. https://www.xilinx.com/support/documentation/ip_documentation/processing_system7_vip/v1_0/ds940-zynq-vip.pdf
8. <https://www.osapublishing.org/ao/abstract.cfm?uri=ao-49-12-2371>
9. <https://www.spiedigitallibrary.org/conference-proceedings-of-spie/4737/0000/White-light-interferometry/10.1117/12.474947.short?SSO=1>
10. <https://www.osapublishing.org/josab/abstract.cfm?uri=josab-13-6-1120>
11. https://www.xilinx.com/support/documentation/ip_documentation/axi_interconnect/v2_1/pg059-axi-interconnect.pdf
12. https://www.xilinx.com/support/documentation/ip_documentation/axi_cdma/v4_1/pg034-axi-cdma.pdf
13. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_2/sdsoc_doc/topics/tutorials/concept_working_system_optimization.html
14. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1119-vivado-creating-packaging-ip-tutorial.pdf
15. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug1165-zynq-embedded-design-tutorial.pdf
16. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2016_3/ug1198-vivado-revision-control-tutorial.pdf
17. https://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug873-zynq-ctt.pdf
18. https://www.xilinx.com/support/documentation/sw_manuals/xilinx2020_1/ug936-vivado-tutorial-programming-debugging.pdf

Appendix

This appendix contains just a partial and relevant segment of the design.

1.VHDL Program to configure AXI write address channel

```

process(M_AXI_ACLK)
begin
    if (rising_edge (M_AXI_ACLK)) then
        --Only VALID signals must be deasserted during reset per AXI spec
        --Consider inverting then registering active-low reset for higher fmax
        if (M_AXI_ARESETN = '0' or start_trans = '1') then
            axi_awvalid <= '0';
        else
            --Signal a new address/data command is available by user logic
            if (start_writing = '1') then
                axi_awvalid <= '1';
            elsif (M_AXI_AWREADY = '1' and axi_awvalid = '1') then
                --Address accepted by interconnect/slave (issue of M_AXI_AWREADY by slave)
                axi_awvalid <= '0';
            end if;
        end if;
    end if;
end process;

```

2. VHDL Program to configure AXI Write response channel

```

process(M_AXI_ACLK)
begin
    if (rising_edge (M_AXI_ACLK)) then
        if (M_AXI_ARESETN = '0' or start_trans = '1') then
            axi_bready <= '0';
        else
            if (M_AXI_BVALID = '1' and axi_bready = '0') then
                -- accept/acknowledge bresp with axi_bready by the master
                -- when M_AXI_BVALID is asserted by slave
                axi_bready <= '1';
            elsif (axi_bready = '1') then
                -- deassert after one clock cycle
                axi_bready <= '0';
            end if;
        end if;
    end if;
end process;

```

```
        end if;  
    end if;  
end process;
```

3. VHDL Program to configure Read Address Channel

```
process(M_AXI_ACLK)  
begin  
    if (rising_edge (M_AXI_ACLK)) then  
        if (M_AXI_ARESETN = '0' or start_trans = '1') then  
            axi_arvalid <= '0';  
        else  
            if (start_reading = '1') then  
                --Signal a new read address command is available by user logic  
                axi_arvalid <= '1';  
            elsif (M_AXI_ARREADY = '1' and axi_arvalid = '1') then  
                --Address accepted by interconnect/slave (issue of M_AXI_ARREADY by slave)  
                axi_arvalid <= '0';  
            end if;  
        end if;  
    end if;  
end if;  
end process;
```

4. Entity of the State Machine

```
library IEEE;  
use IEEE.STD_LOGIC_1164.ALL;  
  
-- Uncomment the following library declaration if using  
-- arithmetic functions with Signed or Unsigned values  
use IEEE.NUMERIC_STD.ALL;  
  
-- Uncomment the following library declaration if instantiating  
-- any Xilinx leaf cells in this code.
```

```
--library UNISIM;
--use UNISIM.VComponents.all;
entity fsm_dma_config is
generic (
    Base_DMA_addr : std_logic_vector := x"41e00000";
    FIFO_depth : integer := 6
);
Port (
    clk : in std_logic;
    rst : in std_logic;
    -- address write valid signal of write address channel of AXI
    valid_w : in std_logic;
    -- address write ready signal of write address channel of AXI
    ready_w : in std_logic;
    -- Address Read valid signal of Read Address Channel of AXI
    valid_r : in std_logic;
    -- Address Read ready signal of Read Address channel of AXI
    ready_r : in std_logic;
    -- Write valid and Write ready signals of Write data channel of AXI
    wvalid : in std_logic;
    wready : in std_logic;
    -----
    -- signals of write response channel of AXI, Bvalid is asserted by slave when address and data has
    -- been successfully written,
    -- These two signals are used in state machine to switch between states in the design.
    -----
    Bvalid : in std_logic;
    Bready : in std_logic;
    -----
    -- Start write and Start read triggers the AXI write and Read transactions, start_write controls the
    -- address write valid and write valid signal.
    -- start_read controls the address read valid signal of Axi
```

```

start_write : out std_logic;
start_read  : out std_logic;
-- start trans signal triggers the state machine
start_trans : in std_logic;
addrw       : out std_logic_vector(31 downto 0);
dataw       : out std_logic_vector(31 downto 0);
addrR       : out std_logic_vector(31 downto 0);
dataR       : in std_logic_vector(31 downto 0);
Rvalid      : in std_logic;
last_transfer : out std_logic;
mm2s_last_transfer : out std_logic;
mm2s_v      : out std_logic;
s2mm_v      : out std_logic;
-- Frame length of frame generated by AXI stream source used in write address calculation.
frame_length : in std_logic_vector(31 downto 0)
);
end fsm_dma_config;

```

5. State Machine declaration inside the architecture

architecture Behavioral of fsm_dma_config is

-- State machine composed of several different states, programming a DMA controller for memory access pattern.

```

type state_m is ( idle, dma_s, dram_addr, dma_length,dma_read,dma_status, mm2s_start, dram_sa,
mm2s_length, stop_dma, mm2s_read,mm2s_status,mm2s_stop);

```

```

signal dma_state : state_m;

```

6. State Machine Design

```
process(clk)
```

```
variable internal_shift_counter : integer := 0;
```

```
variable internal_turnoff_counter : integer := 0;
```

```
begin
```

```
if rising_edge(clk) then
```

```
if(rst = '0') then
```

```
    dma_state <= idle;
```

```
    s2mm_valid <= '0';
```

```
else
```

```
case dma_state is
```

```
when idle =>
```

```
    -- start_trans is the trigger pulse triggers the whole state machine
```

```
    if(start_trans = '1') then
```

```
        dma_state <= dma_s;
```

```
    else
```

```
        dma_state <= idle;
```

```
        addrw <= (others => '0');
```

```
        dataw <= (others=> '0');
```

```
        addrR <= (others => '0');
```

```
    end if;
```

```
    -- This state starts the DMA controller write channel
```

```
when dma_s =>
```

```
    cntrlstart <= '1';
```

```
    cntrlread <= '0';
```

```
    drcntr <= '1';
```

```
    Incntr <= '1';
```

```
addrw <= std_logic_vector(to_unsigned(dma_init_reg_addr, 32));
```



```
dataw <= "00000000000000000000000000000001";

if(Bvalid = '1' and Bready = '1') then
    dma_state <= mm2s_start;
    s2mm_valid <= '1';
else
    dma_state <= dma_s;
end if;

-- This state starts the DMA controller read channel
when mm2s_start =>
    cntrlstart <= '1';
    cntrlread <= '0';
    addrw <= std_logic_vector(to_unsigned(dma_mm2s_cntrl_reg_addr, 32));
    dataw <= "00000000000000000000000000000001";
    mm2s_base_addr_cntrl <= '1';
    if(Bvalid = '1' and Bready = '1') then
        dma_state <= dram_addr;
        mm2s_valid <= '1';
        cntrlstart <= '0';
        pixel_count <= '1';
    else
        dma_state <= mm2s_start;
    end if;

-- This state configures the address register of DMA controller where the data has to be written.
when dram_addr =>
    cntrlstart <= '1';
    cntrlread <= '0';
    addrw <= std_logic_vector(to_unsigned(dma_dram_reg_addr, 32));

    -- Feedback logic configuring the address register after the buffer has reached its final element
    and starts all over again.

    if(fb_shift_wr = '1') then
```

```
if(zero_sig_fb = '1') then
    Write_DMEM_Addr <= 0;
    dataw <= std_logic_vector(unsigned(DMEM_Addr) + Write_DMEM_Addr);
    fb_shift_wr <= '0';
    buffer_throughput_complete <= '1';
else
    Write_DMEM_Addr <= Write_DMEM_Addr + 1024;
    dataw <= std_logic_vector(unsigned(DMEM_Addr) + Write_DMEM_Addr);
    fb_shift_wr <= '0';
end if;
else
    buffer_throughput_complete <= '0';
end if;
-- Base DRAM address, the other addresses are calculated separatly.
if(drcntr = '1') then
    dataw <= DMEM_Addr;
    drcntr <= '0';
end if;
if(dram_alert = '1') then
    if(scntr = 1) then
        scntr <= 0;
        dram_alert <= '0';
        dataw <= DMEM_Addr;
    elsif(dsignal = 1) then
        dsignal <= 0;
        asignal <= '1';
        length_signal <= '1';
    else
        Next_Dram_Addr <= to_integer(unsigned(frame_length)*4);
        dsignal <= dsignal + 1;
```


-- dma_read and dma_status checks the status of transfer weather the data has been completely transferred or not.

```
when dma_read =>
```

```
  cntrlstart <= '0';
```

```
  cntrlread <= '1';
```

```
  addrR <= std_logic_vector(to_unsigned(dma_init_reg_addr, 32));
```

```
    if(valid_r = '1' and ready_r = '1') then
```

```
      dma_state <= dma_status;
```

```
    else
```

```
      dma_state <= dma_read;
```

```
      -- Logic for checking the read/status register weather transfer is successfull or not.
```

```
      if(rsignal= '1') then
```

```
        dma_state <= dram_sa;
```

```
        recon_proc_ctrl <= '0';
```

```
        --dram_alert <= '1';
```

```
        cntrlread <= '0';
```

```
      -- Image is shifted, When the one full iteration of reading has occured, image is shifted (overwritten) and the next image is the first image.
```

```
      if(shift_image = '1') then
```

```
      -- Base address provided to dram_sa ( state which configures the address register for reading, After every time image shifts, address is provided by dma_read state)
```

```
        Sec_base_MM2S_Addr <= Sec_base_MM2S_Addr + 1024;
```

```
        neg_rd_ctr <= neg_rd_ctr -1;
```

```
        dram_fb_ctrl <= '1';
```

```
        -- Address wraps to zero after the end of buffer.
```

```
        if(Sec_cntrl = '1') then
```

```
          Sec_base_MM2S_Addr <= 0;
```

```
          end_buf_iter <= '1';
```

```
          end_buf_s <= '1';
```

```
          neg_rd_ctr <= FIFO_depth-1;
```

```
        end if;
```

```

        end if;

    else

        dma_state <= dma_read;

        end if;

    end if;

when dma_status =>

    cntrlstart <= '0';

    cntrlread <= '1';

addrR <= std_logic_vector(to_unsigned(dma_status_reg_addr, 32));

if(valid_r = '1' and ready_r = '1') then

    dma_state <= dma_read;

else

    dma_state <= dma_status;

end if;

-- dram_sa state is responsible for configuring the address register for reading channel.

when dram_sa =>

    cntrlstart <= '1';

    cntrlread <= '0';

    --counter_assert <= '0';

    addrw <= std_logic_vector(to_unsigned(dma_mm2s_dram_reg_addr, 32));

    if(Bvalid = '1' and Bready = '1') then

        dma_state <= mm2s_length;

    else

        dma_state <= dram_sa;

    end if;

```

-- Below logic is controlled by 3 different control signals at 3 different times.

-- mm2s_base_addr_cntrl controls the first time the address which needs to be read from the first row of image which is 0.

-- mm2s_cycle_ctrl controls the addition of 4 more pixels every time read is successful, As the first row contains 16 pixels, 4 pixels are read in every cycle, so 4 cycles are required

-- to read 16 pixels from every row.

-- dram_fb_ctrl controls the Sec_base_MM2S_Addr as its added every time the image shifts. For example if 0th location is overwritten, 1024th location contains the first image.

```

if(mm2s_base_addr_cntrl = '1' or mm2s_cycle_ctrl = '1' or dram_fb_ctrl = '1') then
    dataw <= std_logic_vector(to_unsigned(Sec_base_MM2S_Addr,32) +
mm2s_base_offset);

```

```

    mm2s_base_addr_cntrl <= '0';

```

```

    mm2s_cycle_ctrl <= '0';

```

```

    dram_fb_ctrl <= '0';

```

```

    mm2s_base_ctrl <= '0';

```

```

end if;

```

```

    if(mm2s_dram_alert = '1') then

```

```

        if(mm2s_scntr = 1) then

```

```

            mm2s_scntr <= 0;

```

```

            mm2s_dram_alert <= '0';

```

```

            dataw <= MM2S_DMEM;

```

```

        else

```

--Logic to control the delay of one clock cycle before the value mm2s_offset can be updated.

```

        if( off_mm2s_addr = '1') then

```

```

            mm2s_asignal <= '1';

```

```

            if(shift_image = '1') then

```

```

                mm2s_offset <= sec_mm2s_offset;

```

```

            elsif(cycle_turn_off = '1') then

```

```

                -- add_cycle_turnoff_offset <= add_cycle_turnoff_offset +1024;

```

```

                mm2s_offset <= sec_mm2s_offset;

```

```

            else

```

```

                mm2s_offset <= 1024 + mm2s_offset;

```

```

            end if;

```

```

            off_mm2s_addr <= '0';

```

```

        end if;

```

-- Below statement control the logic for addition of 4 more pixels from every row, counter number indicates the iteration, for next 4 pixels, counter number increments by one.

```

if(mm2s_asetal = '1' and mm2s_other_rows = '1') then

    MM2S_DMEN <= std_logic_vector(to_unsigned(mm2s_offset + (rd_ptr*68)
+16*counter_number,32));

    mm2s_asetal <= '0';

    mm2s_dsignal <= 0;

    mm2s_scntr <= mm2s_scntr +1;

    mm2s_other_rows <= '0';

```

--This statement controls the addition of pixels when the last element of buffer has been reached.

```

elseif (mm2s_asetal = '1' and mm2s_shift_other_rows = '1') then

    MM2S_DMEN <= std_logic_vector(to_unsigned(shift_offset + (rd_ptr*68)
+16*counter_number,32));

    mm2s_asetal <= '0';

    mm2s_dsignal <= 0;

    mm2s_scntr <= mm2s_scntr +1;

    mm2s_shift_other_rows <= '0';

```

-- This statement controls the addition of pixels when the last element of buffer has been reached.

```

elseif(mm2s_asetal = '1' and shift_reverse = '1') then

    MM2S_DMEN <= std_logic_vector(to_unsigned(shift_offset + (rd_ptr*68),32));

    shift_reverse <= '0';

    mm2s_asetal <= '0';

    mm2s_dsignal <= 0;

    mm2s_scntr <= mm2s_scntr + 1 ;

```

-- This statement controls the reading first iteration of base address values.

```

elseif(mm2s_asetal = '1' and shift_assert = '1') then

    MM2S_DMEN <= std_logic_vector(to_unsigned(mm2s_offset + (rd_ptr*68),32));

    mm2s_asetal <= '0';

    mm2s_dsignal <= 0;

    mm2s_scntr <= mm2s_scntr + 1 ;

    shift_assert <= '0';

    elseif(mm2s_asetal = '1') then

```

```
MM2S_DMEM <= std_logic_vector(to_unsigned(mm2s_offset + (rd_ptr*68),32));

ctrl_shift <= '0';

mm2s_asiqnal <= '0';

mm2s_dsignal <= 0;

mm2s_scntr <= mm2s_scntr + 1 ;

end if;

end if;

end if;
```

--This state configures the amount of data needed to be read, which is 16 bytes (4 pixels) in this case.

```
when mm2s_length =>
    cntrlstart <= '1';

    cntrlread <= '0';

    addrw <= std_logic_vector(to_unsigned(dma_mm2s_lngth_reg_addr, 32));
        --Configured to 16 bytes as 4 pixels are read
        dataw <= "0000000000000000000000000000000010000";
        mm2s_output_data <= "00000000000000000000000000000000100";
    mm2s_recon_proc_ctrl <= '1';
if(Bvalid = '1' and Bready = '1') then
    dma_state <= mm2s_status;
    cntrlstart <= '0';
else
    dma_state <= mm2s_length;
end if;
```

-- mm2s_read state controls the entire feedback logic to read pixels in a very specific way as desired.

```
when mm2s_read =>
    cntlstart <= '0';
    cntlread <= '1';
    off_mm2s_addr <= '1';
    addrR <= std_logic_vector(to_unsigned(dma_mm2s_cntl_reg_addr, 32));
    if(valid_r = '1' and ready_r = '1') then
```



```
    dma_state <= mm2s_status;  
else  
    dma_state <= mm2s_read;
```

--This sequential logic controls the iteration of every read cycle, cycles_complete signal checks whether one full iteration of ring buffer has occurred or not.

```
    if(cycles_complete = '1' and mm2s_rsignal = '1') then  
        dma_state <= dram_sa;  
        cntrlread <= '0';  
        loop_signal <= '1';  
        mm2s_base_offset <= mm2s_base_offset + 16;  
        mm2s_cycle_ctrl <= '1';  
        cycle_turn_off <= '1';  
        mm2s_base_ctrl <= '1';  
        mm2s_offset <= 0;  
        shift_image <= '0';  
        rnd_counter <= '0';  
        rnd_turnoff_counter <= '0';  
        counter_number <= counter_number + 1;  
        mm2s_recon_proc_ctrl <= '0';  
        sec_mm2s_offset <= 0;  
        shift_offset <= 0;  
        end_buf_iter <= '0';  
        end_buf_other_rows <= '1';  
        if(cntrl_count_turnoff = '1') then  
            count_turnoff_neg <= FIFO_depth-1;  
            if(end_buf_sig = FIFO_depth+1) then  
                cntrl_count_turnoff <= '0';  
                count_turnoff_neg <= 0;  
            end if;  
        end if;
```

-- This sequential logic controls whether all 4 pixels from every image and every respective rows are successfully read and then image is shifted.

-- For example, reading starts from 0th element of buffer to 4th element. According to desired memory access pattern, 1st full row of first image,

-- 2nd row of 2nd image, 3rd row of 3rd image and so on has to be read. After all the rows are read in this manner, new image can be written at 0th location which is controlled by "shift_image" signal.

-- Counter_number performs iterations for every next 4 pixels reading.

```
if(counter_number = 4 and rd_ptr = FIFO_depth-1) then
    dma_state <= dram_addr;
    counter_number <= 0;
    shift_image <= '1';
    mm2s_offset <= 0;
    mm2s_other_rows <= '0';
    cycle_turn_off <= '0';
    mm2s_cycle_ctrl <= '1';
    sec_mm2s_offset <= 0;
    mm2s_base_offset <= 0;
    fb_shift_wr <= '1';
    end_buf_s <= '0';
    fall_edge_wrap_ctr <= fall_edge_wrap_ctr +1;
    count_turnoff_neg <= 0;
    count_neg <= 0;
    end_buf_other_rows <= '0';
    DMEM_Addr <= (others => '0');
    if(zero_sig_fb = '1') then
        Write_DMEM_Addr <= 0;
    end if;
end if;
elsif(end_buf_iter = '1' and mm2s_rsignal = '1') then
    dma_state <= dram_sa;
```

```

    shift_offset <= shift_offset + 1024;
    mm2s_recon_proc_ctrl <= '0';
    mm2s_dram_alert <= '1';
    shift_reverse <= '1';
elseif(end_buf_s = '1' and mm2s_rsignal = '1') then
    if( end_buf_other_rows = '1' and mm2s_rsignal = '1') then
        dma_state <= dram_sa;
        shift_offset <= shift_offset + 1024;
        mm2s_recon_proc_ctrl <= '0';
        mm2s_dram_alert <= '1';
        mm2s_shift_other_rows <= '1';
        end if;
elseif(shift_image = '1' and count_neg = neg_rd_ctr and mm2s_rsignal = '1') then
    dma_state <= dram_sa;
    shift_offset <= 0;
    shift_reverse <= '1';
    mm2s_dram_alert <= '1';
    count_neg <= 0;
    rnd_counter <= '1';
    shift_counter <= shift_counter +1;
    mm2s_recon_proc_ctrl <= '0';
    add_shift_rise_offset <= 0;
    if(neg_rd_ctr = 0) then
        neg_rd_ctr <= FIFO_depth-1;
        count_turnoff_neg <= FIFO_depth-1;
        cntrl_count_turnoff <= '1';
        end if;

```

-- count_neg and neg_rd_ctr controls the flow of reading from 0th element to final element of buffer,

-- neg_rd_ctr decrements everytime the image is over written (shifted)

-- count_neg increments and checks until its equal to neg_rd_ctr and controls shifts to above statement where count_neg = neg_rd_ctr

```
-----

elsif(shift_image = '1' and rnd_counter='1' and mm2s_rsignal = '1') then

    dma_state <= dram_sa;

    shift_offset <= shift_offset + 1024;

    mm2s_recon_proc_ctrl <= '0';

    mm2s_dram_alert <= '1';

    shift_reverse <= '1';

elsif(shift_image = '1' and mm2s_rsignal = '1') then

    dma_state <= dram_sa;

    --sec_mm2s_offset <= sec_mm2s_offset + 1024;

    mm2s_recon_proc_ctrl <= '0';

    --count_neg <= count_neg+1;

    mm2s_dram_alert <= '1';

    shift_assert <= '1';

    if(cylce_shift_rise = '1') then

        add_shift_rise_offset <= add_shift_rise_offset + 1024;

        count_neg <= count_neg + 1;

        if(count_neg = neg_rd_ctr) then

            add_shift_rise_offset <= 0;

            sec_mm2s_offset <= 0;

        else

            sec_mm2s_offset <= shift_incrementer_shift + 1024 + add_shift_rise_offset;

        end if;

    end if;

elsif(cycle_turn_off = '1' and count_turnoff_neg = neg_rd_ctr and mm2s_rsignal = '1') then

    dma_state <= dram_sa;

    shift_offset <= 0;

    mm2s_shift_other_rows <= '1';
```

```

mm2s_dram_alert <= '1';
count_turnoff_neg <= 0;
rnd_turnoff_counter <= '1';
shift_counter <= shift_counter + 1;
mm2s_recon_proc_ctrl <= '0';
add_cycle_turnoff_offset <= 0;
sec_mm2s_offset <= 0;

elsif(cycle_turn_off = '1' and rnd_turnoff_counter='1' and mm2s_rsignal = '1') then
    dma_state <= dram_sa;
    shift_offset <= shift_offset + 1024;
    mm2s_recon_proc_ctrl <= '0';
    mm2s_dram_alert <= '1';
    mm2s_shift_other_rows <= '1';
    --shift_reverse <= '1';

```

-- cycle_turn_off controls the addition of pixels, 4 pixels in this case from every row . Same logic is applied here

-- count_turnoff_neg is incremented and checked against neg_rd_ctr and
 -- control shifts to above statement where count_turnoff_neg = neg_rd_ctr

```

-----
elsif(cycle_turn_off = '1' and mm2s_rsignal = '1') then
    dma_state <= dram_sa;
    mm2s_dram_alert <= '1';
    mm2s_other_rows <= '1';
    mm2s_recon_proc_ctrl <= '0';
    sec_mm2s_offset <= sec_mm2s_offset + 1024;
    if(cycle_turn_shift_rise = '1') then
        add_cycle_turnoff_offset <= add_cycle_turnoff_offset + 1024;
        count_turnoff_neg <= count_turnoff_neg + 1;
        if(count_turnoff_neg = neg_rd_ctr) then
            add_cycle_turnoff_offset <= 0;

```

```

        sec_mm2s_offset <= 0;
    else
        sec_mm2s_offset <= shift_incrementer + 1024 + add_cycle_turnoff_offset;
    end if;
end if;

elsif(mm2s_rsignal= '1') then
    dma_state <= dram_addr;
    mm2s_dram_alert <= '1';
    cntrlread <= '0';
    mm2s_recon_proc_ctrl <= '0';
    dram_alert <= '1';
else
    dma_state <= mm2s_read;
end if;
end if;

when mm2s_status =>
    cntrlstart <= '0';
    cntrlread <= '1';

    addrR <= std_logic_vector(to_unsigned(dma_mm2s_status_reg_addr, 32));
    if(valid_r = '1' and ready_r = '1') then
        dma_state <= mm2s_read;
    else
        dma_state <= mm2s_status;
    end if;

-- mm2s_stop state is precautionary, in case if there is a need to halt the reading process.
when mm2s_stop =>
    cntrlstart <= '1';
    cntrlread <= '0';
    mm2s_valid <= '0';
    addrw <= std_logic_vector(to_unsigned(dma_mm2s_cntrl_reg_addr, 32));

```

```
dataw <= (others => '0');  
    if(Bvalid = '1' and Bready = '1') then  
        dma_state <= mm2s_stop;  
        --last_transfer <= '0';  
    end if;  
when others =>  
    dma_state <= idle;  
    end case;  
    end if;  
    end if;  
end process;
```