AXI DMA Debug Guide

- Guidelines to debug issues With DMA based designs

Introduction

A direct memory access (DMA) is an operation in which data is copied (transported) from one resource to another resource without the involvement of the processor.

It allows additional bus masters to read or write system memory independently of the CPU(s).

DMA channels can transfer blocks of data to or from devices with no CPU overhead.

The CPU manages DMA operations via a DMA controller unit. While the DMA transfer is in progress, the CPU can continue executing code. When the DMA transfer is completed, the DMA controller will signal the CPU with an interrupt.

Typical scenarios of block memory copy where DMA can be useful are network packet routing and video streaming. DMA is a particular advantage in situations where the blocks to be transferred are large or the transfer is a repetitive operation that would consume a large portion of potentially useful CPU processing time.

Debug categories

While debugging AXI-DMA issues, the root cause could lie on the hardware (IP) or software side.

In the following sections, we will illustrate different debugging techniques to resolve system level issues.

Hardware

- Clocks and Reset
- Interfacing issues (tkeep/tlast, etc.)
- Configuration of the GUI (Alignment, 'Width of Length Register', etc.)
- Excessive/unexpected throttling
- Debug Techniques
    - Try to trap the moment it locks/hangs up in ChipScope or Vivado Probe.
    - Check on the slave interface
    - Other Techniques

Software

- Status register errors
- Cache control
- Examples of using various modes (simple, SG, multi-channel, etc)
- Linux drivers

These categories will be described in detail in the next sections.

Hardware

Clocks and Reset

There are 4 clock inputs. With respect to the clocking mode of operation (Synchronous/Asynchronous), domain clocks have to be checked.

In synchronous mode, all logic runs in a single clock domain. s_axi_lite_aclk, m_axi_sg_aclk, m_axi_mm2s_aclk and m_axi_s2mm_aclk must be tied to the same source.

In asynchronous mode clocks can be run asynchronously, however s_axi_lite_aclk must be less than or equal to m_axi_sg_aclk and m_axi_sg_aclk must be less than or equal to the slower of m_axi_mm2s_aclk or m_axi_s2mm_aclk.

The axi_resetn signal needs to be asserted a minimum of 16 of the slowest clock cycles and needs to be synchronized to s_axi_lite_aclk.

Interfacing issues

This is one of the most common root causes.

Unless the interface is correctly set and the signals are appropriately connected and driven, DMA Engine can hang or show incorrect behavior.

- Ensure that you are driving tlast appropriately on the last beat of the transfer.
  If you do not do this, the DMA engine will sit and wait because it believes it has not yet received a complete packet.
  Tlast cannot be simply tied to ground.

- Ensure that you are driving tkeep correctly.
  If it is tied to ground, all bytes in every beat will be treated as null bytes and effectively ignored.

Also, tlast must be asserted on or before the number of bytes configured in the length register.

GUI Configuration

While configuring the GUI for AXI DMA, check for the width of buffer length register. Ensure that you are setting the "Buffer Length Register Width" parameter (C_SG_LENGTH_WIDTH in ISE versions of the core) appropriately with respect to your requested DMA transfer length.

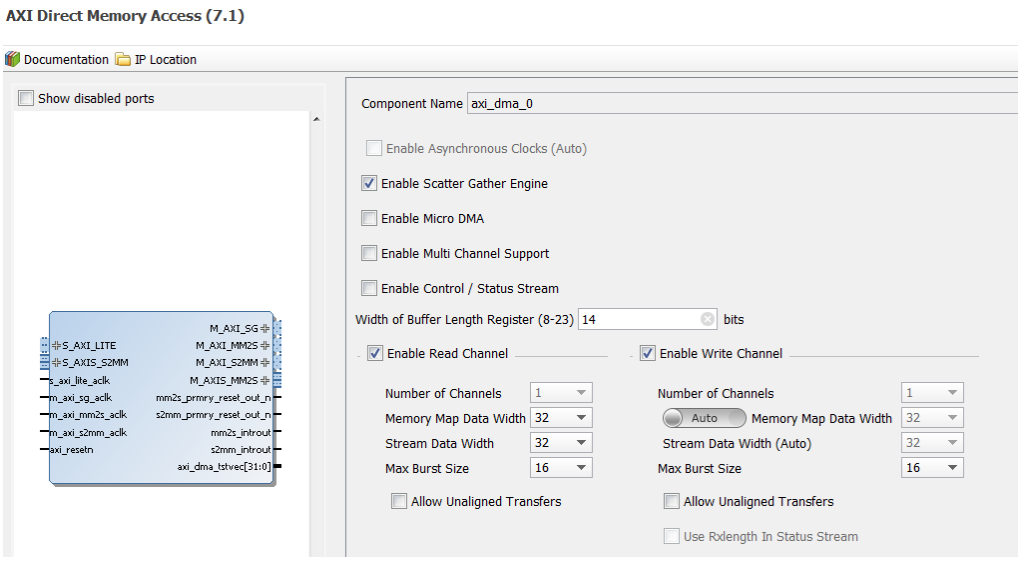
In SG mode (on S2MM), the next descriptor will be fetched when tlast is received.
If tlast occurrs before the number of bytes set in the length register, the SG engine updates the length register with the actual number of bytes transferred and then starts working on the next descriptor as the next packet is coming in.
You can also refer to the comments in the software driver for a more verbose description of BD management.

For example:

```
* <b> Actual Transfer Length </b>
*
* The actual transfer length for receive could be smaller than the requested
* transfer length. The hardware sets the actual transfer length in the
* completed BD. The API to retrieve the actual transfer length is
* XAxiDma_GetActualLength().
```

```
* For RX channels, the value passed in should be the size of the RX buffer
* associated with the given BD in bytes. This is to notify the RX channel
* the capability of the RX buffer to avoid buffer overflow.
*
* The actual receive length can be equal or smaller than the specified length.
* The actual transfer length will be updated by the hardware in the
* XAXIDMA_BD_STS_OFFSET word in the BD.
```

Also, if you are doing unaligned transfers, make sure to check the box for “Allow Unaligned Transfers” for both the read and write channel.



Excessive or Unexpected Throttling

The AXI control stream is provided from the Scatter Gather Descriptor to a target device for User Application data. The control data is associated with the MM2S primary data stream and can be sent out of AXI DMA prior to, during, or after the primary data packet. Throttling by the target device is allowed, and throttling by AXI DMA can occur.

If you are seeing excessive or unexpected throttling on the S2MM side, it should first be noted that these behaviors may be indicative of other issues.
One possible solution might be to enable the Data Realignment Engine (DRE) by checking the 'Allow Unaligned Transfers' option when configuring the core.

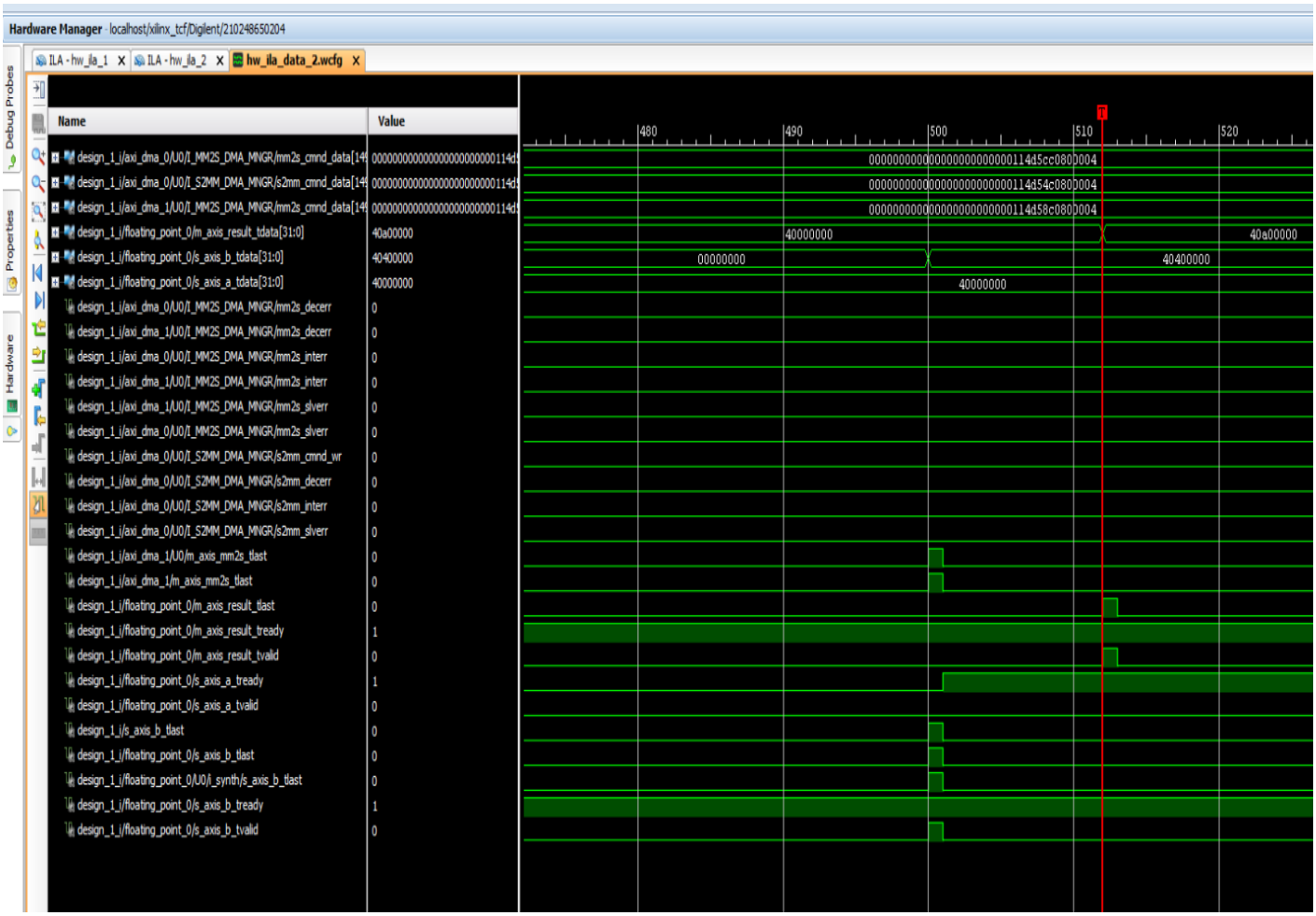
Also, in the absence of any S2MM command, AXI DataMover will pull the s_axis_s2mm_tready signal to Low after taking in four beats of streaming data.

This will throttle the input data stream. To have a minimum amount of throttling, ensure that a valid command is issued to the S2MM interface much before the actual data arrives.

Debug Techniques

- ***Try to trap the moment it locks/hangs up in ChipScope or Vivado Probe.***

  If you have checked the earlier points, then it's best to insert Vivado Logic Analyzer (or ChipScope) on both the stream (S_AXIS_S2MM_*) interface and the memory-mapped (M_AXI_S2MM) interface to determine how far into the transfer you are getting. With this, you can look at tlast and also make sure the SG interface is fetching descriptors.



- ***Check on the slave interface***

    - ✓ Is the endpoint slave device (not just interconnect) asserting awready?
    - ✓ Is anything else trying to access the slave device simultaneously?

- ***Other Techniques***
    - ✓ Are you using interrupts or polling to check for transfers to complete?
    - ✓ How are your BDs setup? What are the src/dest addresses, length, etc.
    - ✓ Are there any resp errors anywhere?
    - ✓ Have you tried applying reset(s) before doing anything?

Software

Status register errors and BD chains

- ✓ Please check the status register to see if any errors were reported. You can find all of the information on address map and bit field details in the product guide for both MM2S and S2MM.
  http://www.xilinx.com/support/documentation/ip_documentation/axi_dma/v7_1/pg021_axi_dma.pdf

- ✓ DMA configuration and status registers before and after starting DMA.
- ✓ DMA BD chain before and after starting DMA.

Issues with Operating Systems

It is important to maintain DMA buffer coherence across context switches. Either you can reserve sections of DDR which are not to be used by the OS or let the kernel/driver worry about it.

Examples of using various modes (simple, SG, multi-channel, etc)

You can check the examples listed in

C:\Xilinx\SDK\2014.3.1\data\embeddedsw\XilinxProcessorIPLib\drivers\axidma_v7_02_a\examples

Also, you can refer the following AR.

http://xkb/Pages/57/57550.aspx

Linux drivers

You can find the kernel driver here. However, you will need a test example to use this.

https://github.com/Xilinx/linux-xlnx/blob/master/drivers/dma/xilinx/xilinx_axidma.c

Conclusion:

If this document does not help to resolve your issues with AXI DMA, please create a WebCase with Xilinx Technical Support. Attach all of the captured ChipScope Pro tool waveforms, and the details of your investigation and analysis.