AMBA DMA Controller

With 32-bit/64-bit AXI Support

User's Guide
Generic / ASIC
AMBA DMA Controller User’s Guide - Verilog

User’s Guide
Version 1.1 - January 2015

Contents

1 DESIGN KIT INTRODUCTION ........................................................................................................... 5

2 DESIGN FLOW ................................................................................................................................. 6
  2.1 OVERVIEW ................................................................................................................................. 6
  2.2 RECOMMENDED TOOLS ............................................................................................................ 6

2 DMA CONTROLLER CORE DATABASE ......................................................................................... 7

3 SIMULATION ENVIRONMENT ........................................................................................................ 8
  3.1 RUNNING SIMULATION USING MODELSIM ................................................................. 9
    3.1.1 Overview .............................................................................................................................. 9
    3.1.2 Simulation Options ............................................................................................................. 10
    3.1.3 Package Options ................................................................................................................ 10
  3.2 TEST BENCH: THEORY OF OPERATION ............................................................................. 10
    3.2.1 MAC64 ............................................................................................................................... 10
    3.2.2 3-port SWITCH 32-bit FIFO ............................................................................................ 11
    3.2.3 3-port SWITCH 64-bit FIFO ............................................................................................ 11
  3.3 DDR AXI-SLAVE MODEL ......................................................................................................... 11
  3.4 EXTERNAL REG BLOCK .......................................................................................................... 11
  3.5 TOP LEVEL SCRIPT FILES .................................................................................................... 12
    3.5.1 Verilog Test bench Code Walk-Through ........................................................................... 12

4 CORE USAGE ................................................................................................................................. 21
  4.1 CORE CONFIGURATION AND SYNTHESIS OPTIONS .................................................... 21
    4.1.1 Global Synthesis Options .................................................................................................. 21
    4.1.2 Core Configuration Package File ...................................................................................... 21
  4.2 TOP-LEVEL ............................................................................................................................... 24
    4.2.1 Files ..................................................................................................................................... 24
    4.2.2 Instantiation Parameters ................................................................................................... 24
  4.3 SIMULATION ............................................................................................................................. 26
  4.4 MEMORY IMPLEMENTATION ............................................................................................... 27
    4.4.1 Internal memory structures .............................................................................................. 27

5 SYNTHESIS .................................................................................................................................... 28
  5.1 SYNOPSIS SYNTHESIS ......................................................................................................... 28
  5.2 VERILATOR LINT SYNTHESIS ............................................................................................ 28
  5.3 LIBERO SYNTHESIS FROM ACTEL/MICROSEMI .............................................................. 28

6 CONTACT ........................................................................................................................................ 30

7 DOCUMENT HISTORY .................................................................................................................. 31
Document Founder

<table>
<thead>
<tr>
<th>Mr. Paul. P. Bates</th>
<th>Nine Ways Research &amp; Development Ltd (UK)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Written under the governance and agreement of Mr. Daniel Koehler</td>
<td>MorethanIP GmbH</td>
</tr>
</tbody>
</table>

About This Release Document

This document describes specifically coding relevancies, package options and internal architecture from the HDL perspective.

Intended Audience

This document is written for associates and employees of MorethanIP GmbH, for customers/clients using the hardware and integrating into project designs. It is a full open release document that is available with the purchase and deliverable of the AMBA DMA Controller product.
List of Tables

Table 1: Recommended Tools ........................................................................................................... 6
Table 2: Design Kit Directories Description ..................................................................................... 7
Table 3: Testbench Files .................................................................................................................... 8
Table 4: Global Package Definitions (common_header) .................................................................... 21
Table 5: Synthesis Package Definitions (mtip_dma_pack_package.verilog) ....................................... 21
Table 6: Top-Level Files .................................................................................................................... 24
Table 7: Behavioral Models Summary ............................................................................................... 26
Table 8: Memory Descriptions .......................................................................................................... 27

List of Figures

Figure 1 – Test-bench Setup Overview (showing the MAC64 option only) ................................. 9
Figure 2 - Libero from Microsemi with the Nine Ways start-up project open ............................. 29
Figure 3 - Directory in the DMA project for SmartFusion2 start-up, where HDL files were copied 29
1 Design Kit Introduction

The DMA Controller package is agnostic and generic. That is – it can be synthesized and instantiated into different design IDE programs for different device families even though it as a deliverable comes packaged in a defined and simple structure.

This structure is described and listed in detail in the following section of this document.

Despite the many different variations of configuration (all changeable in the package.verilog file) there is only one set of source files and simulation scripts. Usually an IP core package may include different duplicate files some for 32-bit and some 64-bit deviations. Moreover, there can often be a JAVA application to allow GUI configuration that then produces specific package source folders with hardcoded parameters and constants. However, this package does include such a configurator IDE.

This package for the AMBA DMA Controller is monolithic – that being, all configuration options for both synthesis and ModelSIM Testbench simulation are kept in the <package.verilog> file included by all source HDL (located in source/package/verilog). This allows for a single ModelSIM script and also only a single source directory with generic source files capable of being synthesized in a whole variation of different configurations and sizes.

It will be described, later in this document, how the simulation using ModelSIM is performed by setting the parameters of operation in the package.verilog file, then executing a simple script for ModelSIM.

Also, it will show how the instantiation of this package in a synthesis environment such as Libero, is simply the IDE copying the source directory files into its own instance directory in the IDE. At this point, changes to the package.verilog file in the newly copied and separate DMA instance directory in the synthesizer IDE will only affect that instance. It will not affect this initial package directory content.

Furthermore, if multiple instantiations of the same AMBA DMA Controller occur, then the package.verilog file in separate directories is the only duplication needed per instance which then leads to the IDE referring to the different <package.verilog> header file.
2 Design Flow

2.1 Overview
The different steps of the AMBA DMA Controller Design and hardware core’s usage within the provided test environment are shown in the following paragraphs and include:

- Core generation
- RTL Simulation
- Synthesis and Implementation

The full design kit provides scripts for ease of use and fast design, verification and implementation turn-around.

2.2 Recommended Tools

<table>
<thead>
<tr>
<th>Design Flow</th>
<th>Tool</th>
<th>Version</th>
</tr>
</thead>
<tbody>
<tr>
<td>Simulation</td>
<td>ModelSIM SE</td>
<td>6.x or Higher</td>
</tr>
<tr>
<td></td>
<td>ModelSIM PE</td>
<td></td>
</tr>
<tr>
<td></td>
<td>ModelSIM AE</td>
<td></td>
</tr>
<tr>
<td>Synthesis (FPGA)</td>
<td>Synopsis Synplify Pro</td>
<td>ME G-2012.09MSP1 or higher</td>
</tr>
<tr>
<td>Synthesis</td>
<td>Synopsys Design Compiler</td>
<td>10.9.1 or Higher</td>
</tr>
</tbody>
</table>
2 DMA Controller Core Database

After the design is generated, the root directory of the DMA Controller package is called AMBA_DMA_Controller which contains a documentation directory (containing these documents), libero, quartus, simulation, and source, synthesis which houses all of the IP core hardware source, simulation and test-bench facilities, and finally the LinuxDriver directory.

<table>
<thead>
<tr>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
</table>
| libero    | Contains the <MakeLibero> file script that copies the agnostic generic driver in this package into a SmartFusion2 Microsemi project HDL directory.  
*Note: You will need to edit this file to select the target directory.* |
| quartus   | Altera Quartus project example that synthesizes the core (if available – and may be empty as a directory) |
| simulation| Scripts to configure and execute simulation. |
| source    | Design and Testbench Source Files. The Testbench sub-directory contains the DDR-AXI Slave ARM model for verification of the ARM sub-system memory controller. |
| synthesis | Synopsys Synthesis Scripts.  
An example implementation script is provided to show the list of files and their order. If you have the Synopsis/Synplify package installed on your computer, executing the <SynopsisLint.bat> file will perform a full scale synthesis check on the DMA package as stand-alone.  
*Note: You must have Synopsis tools installed on your PC running Windows XP or higher.*  
Linting.  
<LintCheck.bat> file in this directory will perform a standard generic linting analysis using Verilator.  
*Note: You must have Verilator installed on your PC running Windows XP or higher.* |
| documentation | For the customer package, this directory is at the same level as the above and contains all of the PDF documents for this product. |
| LinuxDriver | Contains the entire sub-tree of all the C code, build files, make script and binaries of the device driver software. |
3 Simulation Environment

A system Test bench (see Figure 1 – Test-bench Setup Overview) is provided which implements the AMBA DMA Controller core with simulation control state machines, MAC FIFO and DDR-AXI model drivers / monitors which generate and analyze traffic to and from the DMA Controller core.

The DMA can be tested using the MTIP MAC64 core, or two variants of the 3-Port SWITCH (32-bit or 64-bit FIFO interfaces).

Table 3: Testbench Files

<table>
<thead>
<tr>
<th>File</th>
<th>Directory</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>testbenchmac.v</td>
<td>source/testbench/verilog</td>
<td>Testbench using the DMA top-level with the standard ANYSPEED MTIP MAC looped-back</td>
</tr>
<tr>
<td>testbenchswitch32_64.v</td>
<td>source/testbench/verilog</td>
<td>Test bench using the DMA top-level with the 32/64-bit FIFO interfaced 3-Port MTIP SWITCH looped-back</td>
</tr>
<tr>
<td>testbench_functions.v</td>
<td>source/testbench/verilog</td>
<td>TB lower level functions</td>
</tr>
<tr>
<td>tb_model_host.v</td>
<td>source/testbench/verilog</td>
<td>External &lt;reg&gt; memory block</td>
</tr>
<tr>
<td>tb_ddraxi_slave</td>
<td>source/testbench/verilog</td>
<td>MDDR AXI Slave model (ARM)</td>
</tr>
<tr>
<td>tb_signals.v</td>
<td>source/testbench/verilog</td>
<td>All reg, wire and integer definitions</td>
</tr>
<tr>
<td>tb_clocks</td>
<td>source/testbench/verilog</td>
<td>Clock generation block</td>
</tr>
<tr>
<td>testmac64 &lt;directory&gt;</td>
<td>source/testbench/verilog</td>
<td>Instantiated MAC for loop-back testing</td>
</tr>
<tr>
<td>testswitch32 &lt;directory&gt;</td>
<td>source/testbench/verilog</td>
<td>Instantiated 3-Port SWITCH with 32-bit FIFO interface for loop-back testing</td>
</tr>
<tr>
<td>testswitch64 &lt;directory&gt;</td>
<td>source/testbench/verilog</td>
<td>Instantiated 3-Port SWITCH with 64-bit FIFO interface for loop-back testing</td>
</tr>
</tbody>
</table>

- **External Register Model Host**: This emulates another IP core cascaded onto the external register bus of the DMA Controller. The external register bus allows for other IP cores to cascade off the primary DMA Controller and be mapped into the same AHB-Lite addressing scheme from a processor sub-system. The Testbench performs a series of contiguous read/write transactions to this model to check that the DMA HDL is operating well.

- **MDDR AXI Slave**: This model emulates an MDDR or FDDR AXI Slave interface that the DMA Controller will be expected to connect with. The AXI Master in the DMA Controller in the Testbench is controlled to read and write Ethernet frames to/from the AXI model.

The Testbench has three possible configurations:

1. 32-bit AMBA data width.
2. 64-bit AMBA data width.
3. Variable descriptor table sizes for RX and TX: The four descriptor tables can have their sizes adjusted in the <package.verilog> file along with the 32-bit and 64-bit modes (above).
Figure 1 – Test-bench Setup Overview (showing the MAC64 option only)

Note: For the SWITCH 32-bit and 64-bit test-bench variants, the MAC TX and RX blocks are replaced by the top level SWITCH block and the GMII loop-back still operates the same way.

The test-bench includes several configuration and scenario files to exercise the core.

3.1 Running Simulation Using ModelSIM

3.1.1 Overview

The AMBA DMA Controller core files and dependencies are available in the file `<comp_amba_dma.do>` *(simulation directory)*, the Test bench files and simulation models files are listed in the files `<sim_mac_dma.do, sim_switch32_dma.do, sim_switch64_dma.do>`

To perform simulation, the following steps should be followed:

1. Change to the *simulation* directory.
2. Compile the core database: `<comp_amba_dma.do>`
3. Compile the database and run simulation for MAC64: `<sim_mac_dma.do>` OR
4. Compile the database and run simulation SWITCH32: `<sim_switch32_dma.do>` OR
5. Compile the database and run simulation SWITCH64: `<sim_switch64_dma.do>`

Note: that the test-bench will produce many debugging information during simulation. There are errors and warning messages during the simulation, which are intended (e.g. purposely generated errors). When the Test bench stops the output must show "* Simulation Ends without Error *" at the end indicating a successful pass through all scenarios.
3.1.2 Simulation Options

The simulation is controlled by a set of `define parameters. These parameters are given to the compiler when compiling the Test bench. See the script file `< sim_mac_dma.do, sim_switch32_dma.do, sim_switch64_dma.do >` for a list of available options (see +define+ statements).

3.1.3 Package Options

The simulation is also heavily influenced and controlled by the main package.verilog header file for the main source HDL under test. The testbench.v also observes and uses these parameters and `defines.

For instance, if you comment out or uncomment the `define MTIP_MAC_UUT_64BIT then it will change the simulation scenario between 32-bit and 64-bit data widths for the AMBA system (FIFO and AXI)

Also, you change the parameters for the TX and TX Descriptor sizes;

```verilog
parameter RXCH0_DESCSIZE = 4; // Customer choice: Size of RX channel 0 Descriptors
parameter RXCH1_DESCSIZE = 4; // Customer choice: Size of RX channel 1 Descriptors
parameter TXCH0_DESCSIZE = 4; // Customer choice: Size of TX channel 0 Descriptors
parameter TXCH1_DESCSIZE = 4; // Customer choice: Size of TX channel 1 Descriptors
```

This will change the scenario for the Test bench as well. The DMA controller is monolithic and so changes to the main package.verilog do not change just the package HDL source under test but also the Test bench simulation. This all adds to the simplicity of the whole operation.

**Note:** The remaining parameters in the `<package.verilog>` file can also be changed to help test the widest possible set of scenarios possible.

3.2 Test bench: Theory of Operation

The testing of the DMA and the Anyspeed MAC, 3-Port SWITCH 32-bit FIFO and the 3-Port SWITCH 64-bit FIFO cores are all performed in the directory simulation of the AMBA DMA Controller development. The DMA is the constant and is used for the entire different gateway IP cores under test as this is the block that allows memory to be read/written with Ethernet frames.

The gateway IP cores (MAC, SWITCH etc) are all instantiated optionally from sub-directories. This is the fundamental architectural model of test.

3.2.1 MAC64

With the AMBA DMA Controller `mtip_amba_dma_controller UUT()` as the primary common module, the MorethanIP Anyspeed MAC with 64-bit FIFO interface `top_mac_if64mgmil MAC_TEST()` is instantiated and connected to the DMA. The FIFO interfaces are connected purely as one-to-one and the GMII signals on the network side of the MAC core are looped-back from TX to RX.

The DMA has to be compiled as 64-bit FIFO width before the test can run.

One by one, frames are read from a memory bank (see below DDR AXI-Slave Model) and sent through the DMA TX mechanism, on to the TX of the MAC64, out and back on the GMII signals, into the MAC64 RX block, then into the RX of the DMA. It is important to state that the RX and TX of the DMA and the MAC64 are independent.

Finally, the DMA alerts the test-bench code that a frame is waiting, which gets read into the memory bank for analysis. The length of the frames, the burst size all increase and change. Moreover, timeouts and errors are injected from time to time.
If the received frame is received without errors, and is identical to the transmitted frame, then the test continues without failure. A single failure of any kind, results with the test being terminated.

### 3.2.2 3-port SWITCH 32-bit FIFO

With the AMBA DMA Controller `mtip_amba_dma_controller UUT()` as the primary common module, the MorethanIP 3-port SWITCH with 32-bit FIFO interface `switch_3p_hub_axi SWITCH_TEST()` is instantiated and connected to the DMA. The FIFO interfaces are connected purely as one-to-one and the GMII signals on the network side of the MAC core are looped-back from TX to RX.

The DMA has to be compiled as 32-bit FIFO width before the test can run.

One by one, frames are read from a memory bank (see below DDR AXI-Slave Model) and sent through the DMA TX mechanism, on to the TX of the FIFO block internal side of the SWITCH, through the SWITCH backplane, on to MAC0 and MAC1 TX, out and back on the GMII signals of MAC0 ONLY, into the MAC0 RX block, through backplane, to the FIFO RX, then into the RX of the DMA. It is important to state that the RX and TX of the DMA are independent. However the SWITCH backplane is common to RX and TX.

Finally, the DMA alerts the test-bench code that a frame is waiting, which gets read into the memory bank for analysis. The length of the frames, the burst size all increase and change. Moreover, timeouts and errors are injected from time to time.

If the received frame is received without errors, and is identical to the transmitted frame, then the test continues without failure. A single failure of any kind, results with the test being terminated.

### 3.2.3 3-port SWITCH 64-bit FIFO

With the AMBA DMA Controller `mtip_amba_dma_controller UUT()` as the primary common module, the MorethanIP 3-port SWITCH with 64-bit FIFO interface `switch_3p_hub_axi SWITCH_TEST()` is instantiated and connected to the DMA. The FIFO interfaces are connected purely as one-to-one and the GMII signals on the network side of the MAC core are looped-back from TX to RX.

The DMA has to be compiled as 64-bit FIFO width before the test can run.

**Note:** the rest of the description is the same as the 32-bit FIFO SWITCH (above).

### 3.3 DDR AXI-Slave Model

The memory block used to store the frame ready for transmit, and then receive separately the RX returned frame, is an ARM written AXI-MDDR replica `tb_ddraxi_slave DDR_SDRAM_ETHER_FRAMES()`. The model is stored in the verilog file `tb_ddraxi_slave.v`

Whilst it can never be 100% indicative of all AXI interfaces the DDR memory, this one replicates the characteristics of the SmartFusion2 FDDR-AXI-MDDR-LPDDR1 piped blocks.

This imported ARM block has been modified and specialized slightly to allow for the differing 32-bit and 64-bit FIFO data sizes. This keeps the entire data path all the same width between the memory, DMA and the FIFO in the SWITCH internal side.

### 3.4 External REG BLOCK

As a separate part of the overall DMA and FIFO testing across all variants, a test is carried out initially to an external register block. The DMA Controller IP core not only has internal register for configuration, control and status but also a memory address range for externally connected REG devices.

Such external devices are the registers of connected MAC and/or SWITCH. For this test-bench however, we use the “always” blocks stored in the verilog file `tb_model_host.v`

This functionality is enough to simulate the DMA correctly re-mapping outside registers to the correct addresses to prove the concept.
3.5 Top Level Script Files

All 3 variations of test bench are scripted from a combination of verilog files. Whilst there are 3 separate ModelSIM action commands `sim_mac_dma.do`, `sim_switch32_dma.do`, `sim_switch64_dma.do> these translate down to just 2 top-level test scripts.

`sim_mac_dma.do` compiles all modules necessary to use `testbenchmac.v`

`sim_switch32_dma.do` & `sim_switch64_dma.do` compile all modules to use `testbenchswitch32_64.v`

It is therefore, `testbenchmac.v` and `testbenchswitch32_64.v` which run the important test bench scenario scripts for all current DMA test variations. As these files are similar in operation and functional control of the DMA and the send/receive packet process (just differing in the instantiation of the SWITCH versus MAC), the following sub-section describes the coding operation of both files (`testbenchmac.v`, `testbenchswitch32_64.v`) irrespective of which IP cores are being tested and used.

3.5.1 Verilog Test bench Code Walk-Through

All code above the main entry point to the verilog file is instantiated modules such as the SWITCH/MAC, DDR RAM and the external register block.

The first main action is to put all modules into reset by setting the reset signals and then displaying on ModelSIM debug output the start of the testing:

```verilog
reset = 1'b 1;
macreset = 1'b 1;
ddr_nreset = 1'b 0;
$display("n* * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * n");
$display("----------------------------------------------------------------");
$display("-- MorethanIP AMBA:AXI DMA Master Controller hardware IP core --");
$display("-- Copyright (C) MorethanIP GmbH. All Rights Reserved 2013 --");
reset = 1'b 0;
macreset = 1'b 0;
ddr_nreset = 1'b 1;
NOPX(50);
```

After a reset period, the signals are restored to allow the MAC/SWITCH, DDR, REG block etc to run freely:

```verilog
reset = 1'b 0;
macreset = 1'b 0;
ddr_nreset = 1'b 1;
NOPX(50);
```

The DMA Controller IP core main registers are checked for valid expected values as a sanity test:

```verilog
$display("nTest Host LOCAL DMA CORE REGISTERS default values...");
$display("-----------------------------------------------");
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_DMA_VERSION_REGISTER, {MTIP_CORE_VERSION, MTIP_CUSTOMER_SPECIFIC_REVISION });
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_SCRATCH_REGISTER, {REG_DATA_WIDTH (1'b 0)});
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_GLOBAL_CONTROL, {REG_DATA_WIDTH (1'b 0)});
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_POINTER, {REG_DATA_WIDTH (1'b 0)});
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH1_POINTER, {REG_DATA_WIDTH (1'b 0)});
```

```
The MAC/SWITCH IP core main registers are checked for valid expected values as a sanity test ($pVERIFY$). The MAC/SWITCH registers are then written to with a default standard configuration to enable a normal operation:

```verilog
$display("\nTest <SWITCH32/64> TEST Interface Registers");
$display("-------------------------------");
$display("Revision Register");
pRVERIFY(MTIP_SWITCH_REVISION_REG, 32'h 10302); // SWITCH32/64 Revision Register
$display("Scratch Register [RW] TEST");

pWRITE(MTIP_SWITCH_MAC0_RX_SECTION_EMPTY_REG, 32'b 0);
pWRITE(MTIP_SWITCH_MAC0_RX_SECTION_FULL_REG, 32'b 0);
pWRITE(MTIP_SWITCH_MAC0_TX_SECTION_EMPTY_REG, 32'h 80);
pWRITE(MTIP_SWITCH_MAC0_TX_SECTION_FULL_REG, 32'h 40);
pWRITE(MTIP_SWITCH_MAC0_RX_ALMOST_FULL_REG, 32'h 8);
pWRITE(MTIP_SWITCH_MAC0_RX_ALMOST_EMPTY_REG, 32'h 8);

$display("\n[SETTING UP DMA TX-CORE]");
$display("-----------------------------------------");
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_GLOBAL_FRAMEBUFFER_MAX, 1522);
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_GLOBAL_CONTROL,
   TX_GLOBAL_CONTROL_DMAENABLE +
   TX_GLOBAL_CONTROL_CH0ENABLE +
   TX_GLOBAL_CONTROL_CH1ENABLE +
   TX_GLOBAL_CONTROL_AXIRMW +
   TX_GLOBAL_CONTROL_AXILOCKEDACCESS);
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_GLOBAL_FRAMEBUFFER_MAX, 1522);
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_GLOBAL_CONTROL,
   TX_GLOBAL_CONTROL_DMAENABLE +
   TX_GLOBAL_CONTROL_CH0ENABLE +
   TX_GLOBAL_CONTROL_CH1ENABLE +
   TX_GLOBAL_CONTROL_AXIRMW +
   TX_GLOBAL_CONTROL_AXILOCKEDACCESS);

$display("\n[SETTING UP DMA RX-CORE]");
$display("-----------------------------------------");
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_GLOBAL_CONTROL,
   RX_GLOBAL_CONTROL_DMAENABLE +
   RX_GLOBAL_CONTROL_CH0ENABLE +
   RX_GLOBAL_CONTROL_CH1ENABLE +
   RX_GLOBAL_CONTROL_FULLPOLICY);
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_MEMPOOL, 32'h 00802000);
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH1_MEMPOOL, 32'h 00C02000);
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_COEL_TOUT, 5);
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_GLOBAL_CONTROL,
   RX_GLOBAL_CONTROL_DMAENABLE +
   RX_GLOBAL_CONTROL_CH0ENABLE +
   RX_GLOBAL_CONTROL_CH1ENABLE +
   RX_GLOBAL_CONTROL_FULLPOLICY);
```

Write to the DMA Global Control register to enable the DMA TX and all other general functionality for the test we are about to undertake. Also the maximum frame allowed is set to 1522 octets that are allow to pass through the DMA. Once written the values are read back to check that they were accepted by the DMA Controller ($pVERIFY$):

```
```
Once the DMA Controller and the MAC/SWITCH have been setup and checked for working operation using pVERIFY, the memory block that has been instantiated has to be written with incremental default values and display on output debug the percentage completed as this may take up to 30 seconds:

```verilog
$display("n[WRITING TO <AXI-DDR> SDRAM MEMORY LOCATIONS WITH TEST DATA]");
$display("------------------------------------------------------------");
for (i = 0; i < 131072; i = i + (AMBA_DATA_WIDTH / 8))
begin
  writeSDRAM(32'h 00002000 + i, i + 1024);
  if (!(i % 5000))
    begin
      $display("[%d percent]", i / 1311);
    end
end
```

Next is the start of the big mains loops. There are several all nested inside of each other. The first ensures that the testing flows through channel 0 then after all inside processes are completed it re-tests with channel 1. Before getting to the next nested loop, we write to the interrupt enable register of the DMA as this may have been disabled during the loop processing further down. The next inner loop allows frame sizes from 32 octets up to 1510 octets. Finally for each of the above loops in combination, go through the AXI burst sizes (RX, TX) from 1 to 16. All following text boxes with code are part of the main nested loops until FURTHER NOTICE:

```verilog
for (channel = 0; channel < 1; channel = channel + 1)
begin
  writeindex = 0;
  macerrs = 0;
  burstincr = 1;
  pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_INTERRUPT_ENABLE,
          {REG_DATA_WIDTH {1'b 1}});
  for (framesize = 32; framesize < 1510; framesize = framesize + 1)
    begin
      framesize16b = framesize - 14;
      for (burst = 1; burst < 17; burst = burst + burstincr)
        begin
          pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_AXIBURST, burst);
          pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_AXIBURST, burst);
          pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_AXITIMEOUT, 32'h 01);
          pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_AXITIMEOUT, 32'h 01);
        end
      
      .
      .
      .
    end
end
```

The DMA burst size register is written with the current burst size loop value, and the AXI timeout is set to a single AXI missed transaction (the worst possible test for good reason):
A brief inner loops is then encountered which forces the DMA TX descriptors to fill the DMA with a burst of frames in one go to stress test the whole process. All following text boxes with code are part of this purge loop until FURTHER NOTICE:

```verilog
for (purge = 1; purge < 4; purge = purge + 1)
begin
  .
  .
  .
end
```

Inside this purge loop, we set `INTERNALREGISTER_TX_DESC_CH0_PTR` to the next free TX descriptor available for use (writeindex contains this pointer). This allows the following few register writes to fill in the TX descriptor entries with TX frame information that we want to transmit via the DMA. The second write to the `INTERNALREGISTER_TX_DESC_CH1_PTR` sets all of the bit settings and frame size (commented in the code):

```verilog
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_PTR, writeindex); // Set TX desc ptr to first (zero)
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA1, {{1'b 1}, // generate IRQ when TX completed
  {1'b 0}, // xstat field used?
  {1'b 0}, // DMAerr (set to zero, read when done)
  {1'b 0}, // XMITDONE (set to zero, read when done)
  {1'b 0}, // Bytes discarded (set to zero, read when done)
  {1'b 0}, // Protocol checksum to be forwarded to SWITCH32/64?
  {1'b 1}, // CRC forwarded to SWITCH32/64?
  {9'b 0}, // Blank space (9-bits of ZEROs)
  {framesize[15:0]}}); // 16 bit length in bytes of frame to be sent in SDRAM
```

The rest of the TX_DESC registers are written with all of the other frame information ready to be transmitted via the DMA. `DATA2 = checksum information (not used as MAC/SWITCH does not interpret in hardware yet), DATA3|4|5|6 = 128-bit xstat dummy data (DATA6 we write framesize + burst as a changing value to check against later with an RX descriptor), DATA7 = location in RAM block of the TX frame data (in which writeindex determines which part is used for each descriptor entry):

```verilog
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA2, 32'b 0); // TCP/UDP/IP checksum data for SWITCH32/64
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA3, 32'h ABFE1584); // xstat DATA [31:0]
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA4, 32'h 79D3F2A0); // xstat DATA [63:32]
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA5, 32'h AAAA5555); // xstat DATA [95:64]
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA6, framesize + burst); // xstat DATA [127:96]
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA7, 32'h 00002000 + (writeindex * 2048));
```
The next bit of Verilog code just outputs some debug text showing the start of some of the TX frame about to be transmitted via the DMA, the debug is different between the 32-bit and 64-bit operations as to how it interprets the:

```
ifdef MTIP_MAC_UUT_64BIT
    writeSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h 0, 64'h FFFFFFFFFFFFFFFF);
    writeSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h 8, {16'd 0}, framesize16b[7:0], framesize16b[15:8], (32'h FFFFFFFFFF));
    readSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h 8);
    $display("Overwriting Frame Length field <64-bit> [%x:%x]", memread, memaddr);
else
    writeSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h 0, 32'h FFFFFFFF);
    writeSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h 4, 32'h FFFFFFFF);
    writeSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h 8, 32'h FFFFFFFF);
    writeSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h C, {16'd 0}, framesize16b[7:0], framesize16b[15:8]);
    readSDRAM(32'h 00002000 + (writeindex * 2048) + 32'h C);
    $display("Overwriting Frame Length field <32-bit> [%x:%x]", memread, memaddr);
endif
```

Before the DMA transaction is undertaken, we must clear and zero off all the RX descriptor tables ready for the DMA to overwrite. This is not essential, but if there are errors in the hardware DMA Controller, then zeroing off the tables will ensure that mistakes are shown clearly:

```
$display("n[BLANKING/CLEARING UP THE READ DESCRIPTOR ENTRY FOR A CH[01d] RX DMA TRANSACTION], channel);
$display("n[START SINGLE PACKET FRAME TX/RX PROCESS]");
if (purge > 2)
begin
    pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_POINTER, writeindex); // Set TX desc ptr to first (zero)
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_DATA1, 32'b 0); // Main receive status fields
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_DATA2, 32'b 0); // TCP/UDP/IP checksum data for SWITCH32/64
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_DATA3, 32'b 0); // xstat DATA [31:0]
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_DATA4, 32'b 0); // xstat DATA [63:32]
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_DATA5, 32'b 0); // xstat DATA [95:64]
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_DATA6, 32'b 0); // xstat DATA [127:96]
end
```

Inside the purge loop, only when the purge counter is on the last TX frame being registered with the DMA TX, do we instruct DMA to start the burst transaction process of multiple packets. This is done with the first write (below) pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_WR_INDEX_CH0, (writeindex + 1) % TXCH0_DESCSIZE):

```
$display("n[START SINGLE PACKET FRAME TX/RX PROCESS]");
if (purge > 2)
begin
    pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_WR_INDEX_CH0, (writeindex + 1) % TXCH0_DESCSIZE);
pVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_WR_INDEX_CH0, (writeindex + 1) % TXCH0_DESCSIZE);

    loop = 1000;
    pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_WR_INDEX_CH0);
    while ((reg_data_read != ((writeindex + 1) % RXCH0_DESCSIZE)) & (loop > 0))
begin
    pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_RD_INDEX_CH0);
    NOPX(50);
    loop = loop - 1;
end
end
else
begin
    $display("n[BLANKING/CLEARING UP THE READ DESCRIPTOR ENTRY FOR A CH[%01d] RX DMA TRANSACTION], channel);
    $display("n[START SINGLE PACKET FRAME TX/RX PROCESS]");
    if (purge > 2)
begin
    pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_WR_INDEX_CH0, (writeindex + 1) % TXCH0_DESCSIZE);
pVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_WR_INDEX_CH0, (writeindex + 1) % TXCH0_DESCSIZE);

    loop = 1000;
    pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_WR_INDEX_CH0);
    while ((reg_data_read != ((writeindex + 1) % RXCH0_DESCSIZE)) & (loop > 0))
begin
    pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_RD_INDEX_CH0);
    NOPX(50);
    loop = loop - 1;
end
end
```

```
AT THIS POINT, THE PURGE LOOP FROM A FEW PREVIOUS TEXT BLOCKS ENDS HERE AND FOLLOWING CODE IS PART OF THE DMA TRANSACTION CHECKING

We have to check if the TX descriptor pointers actually incremented and advanced forward and around the buffers to indicate that the DMA TX engine actually transmitted the frames onto the MAC/SWITCH IP core via the FIFO interface. We check if (loop == 0) because this is the timeout from the above text box that shows that the TX descriptor never read back the value expected:

```
if (loop == 0)
begin
  $display("TX Desc READ Counter timeout expired!!!!!");
  $display("n-- This simulation has ERRORS --\n quot;);
  $display("n-- Completed Testbench Simulation DONE --\n quot;);
  $stop();
end
```

If all has gone as expected, then we check INTERNALREGISTER_TX_DESC_CNT_INDEX_CH0 register in the DMA to ensure it has no pending count for TX frames as they should all have been removed. Following this, we repeatedly poll INTERNALREGISTER_RX_DESC_WR_INDEX_CH0 to check for the arriving frames looped-back through the MAC/SWITCH back into the DMA RX. If loop times out to zero, then it indicates that the packs never fully came back and the DMA has not seen them all. If this occurs, you can see the debug text output with values from the MAC/SWITCH statistics trying to show where the failure occurred:

```
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CNT_INDEX_CH0, 32'b 0);
// ...and count also
loop = 1500;
pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_WR_INDEX_CH0);
while ((reg_data_read != ((writeindex + 1) % RXCH0_DESCSIZE)) && (loop > 0))
begin
  pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_WR_INDEX_CH0);
  NOPX(50);
  loop = loop - 1;
end
if (loop == 0)
begin
  $display("RX Desc WRITE Counter timeout expired!!!!!!");
  pREAD(MTIP_SWITCH_MAC0_FramesOK_REG);
  $display("nMAC<0> [RX] Frames=%04d\n quot;,
  reg_data_read);
  pREAD(MTIP_SWITCH_MAC1_FramesOK_REG);
  $display("nMAC<1> [RX] Frames=%04d\n quot;,
  reg_data_read);
  pREAD(MTIP_SWITCH_TOTAL_FRM_REG);
  $display("nSWITCH Good=%04d\n quot;,
  reg_data_read);
  $display("n-- This simulation has ERRORS --\n quot;);
  $display("n-- Completed Testbench Simulation DONE --\n quot;);
  $stop();
end
```
In order to inform the DMA RX engine that we have acknowledged receiving the frames incoming, we advance in software the INTERNALREGISTER_RX_DESC_RD_INDEX_CH0 by the number of frames we have detected. WE then read back for sanity the value in case there was a hardware error simulated (pVERIFY). If all is good, reading INTERNALREGISTER_TX_DESC_CH0_DATA1 allows the test bench to check for DMA indicative errors reported on the TX transmit:

```verilog
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_RD_INDEX_CH0, (writeindex + 1) % RXCH0_DESCSIZE); // Writing to WRITE TX reg update
pRVERIFY(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_RD_INDEX_CH0, (writeindex + 1) % RXCH0_DESCSIZE);
$display("********************************************************************************");
pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_TX_DESC_CH0_DATA1);
$display("Write: %s %s %s", (reg_data_read[29] == 1'b 1) ? "DMA Error" : "",
        (reg_data_read[28] == 1'b 1) ? "Completed" : "",
        (reg_data_read[27] == 1'b 1) ? "Discarded" : "");
$display("********************************************************************************");
if (reg_data_read[29] == 1'b 1)
    begin
        $display("[TX] DMA Error reported!!!!!!!n");
        macerrs = macerrs + 1;
    end
if (reg_data_read[23] == 1'b 1)
    begin
        $display("[RX] DMA Error reported!!!!!!!n");
        macerrs = macerrs + 1;
    end
if (reg_data_read[24] == 1'b 1)
    begin
        $display("[RX] FIFO/PHY Error reported!! !!!!!!!n");
        macerrs = macerrs + 1;
    end
if (((framesize < 60) && (reg_data_read[15:0] != 68)) || ((reg_data_read[15:0] != (framesize + 8)) && (framesize >= 60)))
    begin
        $display("<RX> DMA loop-back received bytes back is NOT equal to <TX> framsize [%05d != (%05d - 8)]!!!!!!!n", framesize, reg_data_read[15:0]);
        macerrs = macerrs + 1;
        pREAD(MTIP_SWITCH_TOTAL_BYT_FRM_REG);
        $display("SWITCH Octets Total=%04d 
        n", reg_data_read);
        pREAD(MTIP_SWITCH_MAC0_TxOctetsOK_REG);
        $display("MAC0 TX Octets Total=%04d
        n", reg_data_read);
        pREAD(MTIP_SWITCH_MAC0_OctetsOK_REG);
        $display("MAC0 RX Octets Total=%04d
        n", reg_data_read);
    end
```

If all is good, reading INTERNALREGISTER_RX_DESC_CH0_DATA1 allows the test bench to check for DMA indicative errors reported on the RX frame received back. The $display command shows the entire breakdown of the bit meanings:

```verilog
pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_RX_DESC_CH0_DATA1); // 16 bit length plus RX flags
$display("Read: %s %s %s %s %s %s %s [%04d bytes received]",
        (reg_data_read[31:27] == 5'b 10101) ? "Preamble" : ",",
        (reg_data_read[26] == 1'b 1) ? "Validation" : ",",
        (reg_data_read[25] == 1'b 1) ? "Discarded" : ",",
        (reg_data_read[24] == 1'b 1) ? "PHY Error" : ",",
        (reg_data_read[23] == 1'b 1) ? "DMA Error" : ",",
        (reg_data_read[22] == 1'b 1) ? "Checksum Protocol" : ",",
        (reg_data_read[21] == 1'b 1) ? "XSTAT" : ",",
        (reg_data_read[20:16] > 5'b 0) ? "PHY Issues" : ",",
        reg_data_read[15:0]);
if (reg_data_read[23] == 1'b 1)
    begin
        $display("[RX] DMA Error reported!!!!!!!n");
        macerrs = macerrs + 1;
    end
if (reg_data_read[24] == 1'b 1)
    begin
        $display("[RX] FIFO/PHY Error reported!!!!!!!n");
        macerrs = macerrs + 1;
    end
if (((framesize < 60) && (reg_data_read[15:0] != 68)) || ((reg_data_read[15:0] != (framesize + 8)) && (framesize >= 60)))
    begin
        $display("n<RX> DMA loop-back received bytes back is NOT equal to <TX> framsize [%05d != (%05d - 8)]!!!!!!!n", framesize, reg_data_read[15:0]);
        macerrs = macerrs + 1;
        pREAD(MTIP SWITCH TOTAL_BYT_FRM_REG);
        $display("nSWITCH Octets Total=%04d
        n", reg_data_read);
        pREAD(MTIP SWITCH MAC0 TxOctetsOK_REG);
        $display("MAC0 TX Octets Total=%04d
        n", reg_data_read);
        pREAD(MTIP SWITCH MAC0 OctetsOK_REG);
        $display("MAC0 RX Octets Total=%04d
        n", reg_data_read);
    end
```

18
Now we must check what information came back for each received incoming RX DMA frame. We have to read the first header of the Ethernet frame for the tag inserted etc, and then move onto general data for checks. `readSDRAM()` is the function used to read and check the RAM block data that the DMA moved data into as a hardware process:

```verilog
framessent = framessent + 1;
$display("n[READING SELECTION FROM THE START <AXI-DDR> SDRAM MEMORY LOCATIONS OF RECEIVED DATA]:");
$display("n-----------------------------------------------");
readSDRAM(32'h 00802000 + (writeindex * 2048) + ((AMBA_DATA_WIDTH == 64) ? 8 : 4));
$display("[Modified frame length field [%X]: %X", memaddr, memread);
readSDRAM(32'h 00802000 + (writeindex * 2048));
$display("[%%X]: %X", memaddr, memread);
if (memread[31:0] != 32'h FFFFFFFF)
begin
  $display("n[RX] Memory mismatch!!!!!!!!!n");
  macerrs = macerrs + 1;
end
readSDRAM(32'h 00802000 + (writeindex * 2048) + ((AMBA_DATA_WIDTH == 64) ? 24 : 24));
$display("[%%X]: %X", memaddr, memread);
if ((memaddr - 32'h 00802000) != (memread[31:0] - 1024 + 8))
begin
  $display("n[RX] Memory mismatch!!!!!!!!!n");
  macerrs = macerrs + 1;
end
writeindex = ((writeindex + 1) % TXCH0_DESCSIZE);
if (cpu_int == 1'b 1)
begin
  pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_INTERRUPT_PENDING);
  $display("nIRQ Generated for FIFO activity [%X]", reg_data_read);
  pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_INTERRUPT_PENDING,
  {REG_DATA_WIDTH {1'b 1}});
end
else
begin
  reg_errors = 1'b 1;
end
if (cpu_int == 1'b 1)
begin
  $display("nFIFO IRQ[s] stuck!!!!!!!!!n");
  macerrs = macerrs + 1;
end
```

Check to ensure that the TX and RX IRQ signals occurred in a crude way by checking the `PENDING` bits in the DMA interrupt register:

```verilog
writeindex = ((writeindex + 1) % TXCH0_DESCSIZE);
if (cpu_int == 1'b 1)
begin
  pREAD(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_INTERRUPT_PENDING);
  $display("nIRQ Generated for FIFO activity [%X]", reg_data_read);
  pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_INTERRUPT_PENDING,
  {REG_DATA_WIDTH {1'b 1}});
end
else
begin
  reg_errors = 1'b 1;
end
```

Make sure that the IRQ signal from the DMA Controller IP core is not asserted continuously which was a common problem in the early design of the DMA:

```verilog
if (cpu_int == 1'b 1)
begin
  $display("nFIFO IRQ[s] stuck!!!!!!!!!n");
  macerrs = macerrs + 1;
end
```
Finally, before the inner nested loops end and the whole process is iterated for burst size, frame size and channel, we check for other general unreported errors that incremented the `macerrs` and the `reg_errors` variables:

```verilog
if ((macerrs > 0) || (reg_errors > 0))
begin
    $display("\n-- This simulation has ERRORS --\n");
    $display("\n-- Completed Testbench Simulation DONE --\n");
    $stop();
end
else
begin
    $display("--------------------------------------------------------------");
    $display("-- CHANNEL <%01d> AXI BURST [%02X] of FRAMESIZE <%04d> Head/Tail (%02d) PASSED <%s> --",
                channel, burst, framesize, writeindex, (AMBA_DATA_WIDTH == 64) ? "64-bit" : "32-bit");
    $display("--------------------------------------------------------------");
    macerrs = 0;
end
```

AT THIS POINT, THE 3-NESTED CHANNEL, SIZE AND BURST SIZE LOOPS FROM NEAR THE START TEXT BLOCKS ENDS HERE AND FOLLOWING CODE IS PART OF THE FINAL COMPLETION SUCCESS DEBUG OUTPUT

```verilog
pWRITE(LOCAL_REGISTER_MEMORY_OFFSET + INTERNALREGISTER_INTERRUPT_ENABLE, {REG_DATA_WIDTH {1'b 0}});
if (reg_errors > 0)
begin
    $display("\n-- This simulation has ERRORS --\n");
end
else
begin
    $display("***************************************************************************");
    $display("--***************** Simulation Ends without Error *****************

    $display("***************************************************************************");
    $display("\n-- Completed Testbench Simulation DONE --\n");
    $stop();
```

If the test manages to get to this final stage, after days worth of iterative exhaustive testing in ModelSIM, then it means that not a single error or glitch of any kind has occurred in the entire system.

This marks a big success.

**Important:** the slightest error, in any of the test-bench code – will result in a failure and the testing is STOPPED. Errors or glitches cannot occur and the test continues. Continuing debug output over the days of testing show a continuing success.

[END]
4 Core Usage

4.1 Core Configuration and Synthesis options

4.1.1 Global Synthesis Options

The following table lists the relevant synthesis options found in the so-called global package file. This file is included by all sources to set global definitions. These are for size optimizations and to include/exclude features that affect pin-outs of modules.

If multiple different Cores are synthesized together, all their "common_header.verilog" files need to be merged into a single file of that name containing all settings of all cores.

The package file is located at:

- source/package/verilog/common_header.verilog.

The following settings are available for change. All others must not be changed.

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>No Relevant Values</td>
<td>N/A</td>
</tr>
</tbody>
</table>

Table 4: Global Package Definitions (common_header)

4.1.2 Core Configuration Package File

The core is configured using the package file:

- source/package/verilog/mtip_dma_pack_package.verilog

The file is included by every relevant source file to configure several options during synthesis. The following settings in this file can be changed. All others must remain unchanged.

<table>
<thead>
<tr>
<th>Setting Name</th>
<th>Type</th>
<th>Description</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>MTIP_CUSTOMER_SPECIFIC_REVISION</td>
<td>parameter</td>
<td>This can be any 16-bit integer value. It is stated in hexadecimal in the package file and is used as a sub-version number by a project integrator using this DMA Controller package in their design.</td>
<td>0001 (HEX)</td>
</tr>
<tr>
<td>MTIP_CORE_VERSION</td>
<td>parameter</td>
<td>DO NOT CHANGE</td>
<td>1301 (HEX)</td>
</tr>
<tr>
<td>AMBA_ADDR_WIDTH</td>
<td>parameter</td>
<td>Global AMBA system data width. This affects the AXI address width only in this case. It sets the number of bit of the AXI Master</td>
<td>32</td>
</tr>
</tbody>
</table>

Table 5: Synthesis Package Definitions (mtip_dma_pack_package.verilog)
<table>
<thead>
<tr>
<th>AMBA DMA Controller User’s Guide - Verilog</th>
<th></th>
</tr>
</thead>
</table>

<table>
<thead>
<tr>
<th>MTIP_MAC_UUT_64BIT</th>
<th>`define</th>
<th>Defined</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>address bus and <strong>MUST</strong> match the width of the system memory bus of the host device system architecture.</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>Must be a power of 2, at least 32.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AXI_ID_WIDTH</th>
<th>parameter</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the number of bits for the write AXI address ID, write AXI data ID, read AXI address ID and read AXI data ID (rid).</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>Usual values are 2, 3 or 4.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>AXI_RXDMA_BACKPRESSURE_SIZE</th>
<th>parameter</th>
<th>32</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Depth of the RX-DMA backpressure FIFO. This FIFO cushions the initial [sop] data to activate the RX mechanism, and also allows for the MAC to react slowly to an RX-DMA ready signal de-assertion. This, if set too low (less than 4 can cause problems with the RX if the ready signal is dropped to the MAC).</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>This can be set as high as 2048 but optimally is operates well at 16 or 32.</td>
<td></td>
</tr>
</tbody>
</table>

### RX & TX Descriptor Table Configuration

<table>
<thead>
<tr>
<th>RXCH0_DESCSIZE</th>
<th>parameter</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the size of the High priority RX-DMA (Channel 0) descriptor table size in terms of entries.</td>
<td></td>
</tr>
<tr>
<td><strong>Note:</strong></td>
<td>This value if <strong>&lt;zero&gt;</strong> effectively disables the RX-DMA CH0 mechanism. Maximum allowed value is 256.</td>
<td></td>
</tr>
<tr>
<td></td>
<td><strong>Note:</strong> Values in the higher range provide for better burst and high throughput performance, but may not synthesize well in smaller device families.</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>RXCH1_DESCSIZE</th>
<th>parameter</th>
<th>4</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Sets the size of the Low priority RX-DMA (Channel 1) descriptor table size in terms of entries.</td>
<td></td>
</tr>
<tr>
<td>Parameter</td>
<td>Description</td>
<td>Value</td>
</tr>
<tr>
<td>--------------------</td>
<td>-----------------------------------------------------------------------------</td>
<td>-------</td>
</tr>
<tr>
<td>TXCH0_DESCSIZE</td>
<td>Sets the size of the High priority TX-DMA (Channel 0) descriptor table size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in terms of entries.</td>
<td></td>
</tr>
<tr>
<td>TXCH1_DESCSIZE</td>
<td>Sets the size of the Low priority TX-DMA (Channel 1) descriptor table size</td>
<td></td>
</tr>
<tr>
<td></td>
<td>in terms of entries.</td>
<td></td>
</tr>
<tr>
<td>Other Settings</td>
<td>All other settings must be left unchanged.</td>
<td>N/A</td>
</tr>
</tbody>
</table>
4.2 Top-Level

4.2.1 Files

<table>
<thead>
<tr>
<th>File</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>mtip_amba_dma_controller.v</td>
<td>source/top/verilog/</td>
<td>The DMA Controller top-level universal for all configurations and variations.</td>
</tr>
</tbody>
</table>

Note that all files rely on packages included from the directory `source/package/verilog`. Hence this directory must be added to the include path of the simulator and synthesizer tools.

4.2.2 Instantiation Parameters

The top-level does NOT have any options that can be configured during instantiation. All parameters are controlled in the `package.verilog` file.

*Instantiation would typically just take the following form (with no parameters but just wire definitions):*

```verilog
module mtip_amba_dma_controller(
    .reset_hclk( ...... ),
    .hclk( ...... ),
    .reset_aclk( ...... ),
    .aclk( ...... ),
    .reg_rd( ...... ),
    .reg_wr( ...... ),
    .reg_addr( ...... ),
    .reg_data_in( ...... ),
    .reg_data_out( ...... ),
    .reg_busy( ...... ),
    .extint_inputs( ...... ),
    .extint_acks( ...... ),
    .hsel( ...... ),
    .haddr( ...... ),
    .hwrite( ...... ),
    .htrans( ...... ),
    .hsize( ...... ),
    .hburst( ...... ),
    .hwdata( ...... ),
    .hprot( ...... ),
    .hready( ...... ),
    .hmastlock( ...... ),
    .hreadyout( ...... ),
    .hresp( ...... ),
    .hrdata( ...... ),
    .awlen( ...... ),
    .awsize( ...... ),
    .awburst( ...... ),
    .awlock( ...... ),
    .awvalid( ...... ),
```
.awready( …… ),
.wvalid( …… ),
.wlast( …… ),
.wready( …… ),
.bresp( …… ),
.bvalid( …… ),
.bready( …… ),
.arvalid( …… ),
.arlen( …… ),
.arsize( …… ),
.arburst( …… ),
arlock( …… ),
arready( …… ),
.rlast( …… ),
rresp( …… ),
rvalid( …… ),
rready( …… ),
aradr( …… ),
.awaddr( …… ),
.wdata( …… ),
wstrb( …… ),
.awid( …… ),
.wid( …… ),
bid( …… ),
arid( …… ),
rid( …… ),
rdata( …… ),
.rmw( …… ),
.ff_tx_dsav( …… ),
.ff_tx_data( …… ),
.ff_tx_sop( …… ),
.ff_tx_eop( …… ),
.ff_tx_err( …… ),
.ff_tx_dval( …… ),
.ff_tx_mod( …… ),
.ff_tx_xstat( …… ),
.ff_tx_crc_fwd( …… ),
.ff_tx_protocol_checksum_enable( …… ),
.ff_tx_protocol_checksum( …… ),
.ff_tx_rdy( …… ),
.ff_tx_ch( …… ),
.ff_tx_septy( …… ),
.ff_rx_protocol_checksum_valid( …… ),
.ff_rx_err_stat( …… ),
.ff_rx_data( …… ),
.ff_rx_sop( …… ),
.ff_rx_eop( …… ),
.ff_rx_err( …… ),
.ff_rx_mod( …… ),
.ff_rx_xstat( …… ),
.ff_rx_dval( …… ),
.ff_rx_rdy( …… ),
.ff_rx_dsav( …… ),
.ff_rx_ch( …… ),
.ff_rx_protocol_checksum( …… ),
cpu_int( …… ),
ext_resetn( …… )
);
4.3 Simulation

To simulate the AMBA DMA Controller when implemented in a custom project, compile the HDL source core in the design work directory as shown in the script `<simulation/comp_amba_dma.do followed by sim_mac_dma.do, sim_switch32_dma.do, sim_switch64_dma.do>`

Behavioral models (external register device, MDDR AXI Slave, Anyspeed MAC64, 3-Port SWITCH32, 3-Port SWITCH64) are provided in verilog source in the `<source/testbench/verilog>` directory. The models can be re-used in a custom project Testbench and provide a robust and flexible verification tool.

Table 7: Behavioral Models Summary

<table>
<thead>
<tr>
<th>Verilog Model</th>
<th>Location</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>tb_model_host.v</td>
<td>source/testbench/verilog</td>
<td>Simulates another MorethanIP hardware core cascaded off the DMA Controller using the external register bus for the IO Register of that 3rd party core.</td>
</tr>
<tr>
<td>tb_ddraxi_slave.v</td>
<td>source/testbench/verilog</td>
<td>Behaves according to the standard AMBA ARM memory bus sub-system MDDR that exposes an AXI Slave interface that connects to the DMS Controller AXI Master.</td>
</tr>
<tr>
<td>&lt;testmac64&gt;</td>
<td>source/testbench/verilog/testmac64/</td>
<td>MorethanIP independent well-established 32/64 MAC. This allows for the frame on the TX of the DMA Controller to be looped-back through a MAC core into the RX of the DMA Controller core. This is the primary function of the Testbench simulation.</td>
</tr>
<tr>
<td>&lt;testswitch32&gt;</td>
<td>source/testbench/verilog/testswitch32/</td>
<td>MorethanIP independent well-established 32-bit FIFO interfaced 3-Port SWITCH. This allows for the frame on the TX of the DMA Controller to be looped-back through a SWITCH core into the RX of the DMA Controller core. This is the primary function of the Testbench simulation.</td>
</tr>
<tr>
<td>&lt;testswitch64&gt;</td>
<td>source/testbench/verilog/testswitch64/</td>
<td>MorethanIP independent well-established 64-bit FIFO interfaced 3-Port SWITCH. This allows for the frame on the TX of the DMA Controller to be looped-back through a SWITCH core into the RX of the DMA Controller core. This is the primary function of the Testbench simulation.</td>
</tr>
</tbody>
</table>
4.4 Memory Implementation

4.4.1 Internal memory structures

The AMBA DMA Controller has only one instance of an implementation of memory structures. This is the back pressure FIFO in the RX-DMA (AXI write) inner-block. This is used to allow the RX of the MAC FIFO to burst some transaction words whilst the AXI write bus is setting up. Moreover, if there are many scenarios whereby the AXI memory system experiences in a large complex system slowdowns and delays due to bus contention – then the FIFO allows the MAC to continue transactions for a short while by filling the FIFO whilst the AXI bus waits.

It is widely seen that the MAC cores do take a finite number of clock cycles to respond to `ff_rx_rdy` being de-asserted by the DMA Controller – which gives rise for the need for the RX-DMA FIFO.

**Note:** The RX backpressure FIFO is ideally around 32 words in depth.

This is enough to handle:

1. The MAC slow response to `ff_rx_rdy`.
2. The time between an SOP seen and the AXI bus activated for write transactions
3. The start of a next Ethernet frame from a MAC whilst the previous AXI bus transaction completes
4. Not to use too much hardware look-up tables in the Fabric/ASIC.

The files describing memories used in the design are:

Table 8: Memory Descriptions

<table>
<thead>
<tr>
<th>Directory/File</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>source/aximaster/verilog/mtip_aximaster_fifo.v</td>
<td>Single-Clock Dual-Port Memory. Used for the RX-DMA backpressure FIFO in <code>mtip_aximaster_wr.v</code> To change the depth, use the parameter: <code>AXI_RXDMA_BACKPRESSURE_SIZE</code> in <code>&lt;package.verilog&gt;</code></td>
</tr>
</tbody>
</table>
5 Synthesis

5.1 Synopsis Synthesis

The `<synthesis>` directory contains a synthesis script `<SynopsisLint.prj>` example used to synthesize the design. Synthesis scripts are provided for Synopsys Synplify Pro as a starting point. The script needs to be updated to the proper library and other environmental options as necessary.

For synthesis it is important to have the directory `source/package/verilog` added to the include search path of the compiler as most of the files include the package files found in that directory during synthesis.

**Note:** You must have the Synopsis Synplify IDE and tools installed on your PC. This is not included with the deliverable DMA Controller package as this is far too large file size.

5.2 Verilator Lint Synthesis

The `<synthesis>` directory also contains a Linting script `<Lintcheck.bat>` for Windows. Use this to check that the generic syntax and verilog coding is good. You can edit the `Lintcheck.bat` file of you want to check other modules or you create different top modules for this project.

**Important Note:** You must have Verilator copied into the root directory of your computer as `/Verilator` with the following files:

cygwin1.dll
verilator
verilator.bin
verilator_compiled_win

You can obtain the Verilator package from Nine Ways Research & Development Ltd. See website [www.nineways.co.uk](http://www.nineways.co.uk) to contact us and obtain the file.

**Note:** This is not included in the deliverable package of the AMBA DMA Controller as the file sizes are too big and bloats the content.

5.3 Libero Synthesis from Actel/Microsemi

As a start point, the directory within the package called `<libero>` contains a Windows BATCH file script `<MakeLibero.bat>` to directly install the source HDL content of the DMA Controller into a Microsemi project. This then is ready for instantiation and use within an FPGA/ASIC IDE environment.

This is very specific to the Microsemi environment and only acts as a start point. Over time, more scripts can be added that will help end users install the HDL source into specific IDE environments.

The IDE package for Microsemi is typically called Libero Pro or similar derivatives. However, this script is setup specifically for if you are using Nine Ways Research & Developments start-up project that provides a SmartFusion2 environment with the DMA Controller and MorethanIP’s MAC as fabric cores.

This is available from Nine Ways as downloadable ZIP package from the Nine Ways website. Contact details can be obtained from [www.nineways.co.uk](http://www.nineways.co.uk)

If you wish to have your own Libero project then where ever this is located in your file system on your PC, you will have to edit the file `<MakeLibero.bat>` in the `<libero>` directory.
Once you have installed the files into the Libero system, they are separate entities along with the other core verilog files in that target Microsemi directory. They can then have their own parameter settings and `define configuration as discussed previously in this document.

**Figure 2 - Libero from Microsemi with the Nine Ways start-up project open**

![Figure 2 - Libero from Microsemi with the Nine Ways start-up project open](image)

**Figure 3 - Directory in the DMA project for SmartFusion2 start-up, where HDL files were copied**

![Figure 3 - Directory in the DMA project for SmartFusion2 start-up, where HDL files were copied](image)
# Contact

<table>
<thead>
<tr>
<th>MorethanIP GmbH</th>
<th>E-Mail</th>
<th><a href="mailto:info@morethanip.com">info@morethanip.com</a></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Internet</td>
<td><a href="http://www.morethanip.com">www.morethanip.com</a></td>
</tr>
<tr>
<td><strong>Europe</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Muenchner Str. 199</td>
<td>Tel</td>
<td>+49 (0) 8131 333939 0</td>
</tr>
<tr>
<td>D-85757 Karlsfeld</td>
<td>FAX</td>
<td>+49 (0) 8131 333939 1</td>
</tr>
<tr>
<td>Germany</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Nine Ways Research &amp; Development Ltd</th>
<th>E-Mail</th>
<th><a href="mailto:pbates@nineways.co.uk">pbates@nineways.co.uk</a></th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Internet</td>
<td><a href="http://www.nineways.co.uk">www.nineways.co.uk</a></td>
</tr>
<tr>
<td><strong>UK</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Unit A.3, iDCentre, Lathkill House, rtc Business Park</td>
<td>Tel</td>
<td>+44 (0) 1332 258847</td>
</tr>
<tr>
<td>London Road, Derby, DE24 8UP</td>
<td>FAX</td>
<td>+44 (0) 1332 258823</td>
</tr>
<tr>
<td>United Kingdom</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
## 7 Document History

Document Change Notices (DCO)

<table>
<thead>
<tr>
<th>Version</th>
<th>Description</th>
<th>Created/Changed By</th>
<th>Date</th>
</tr>
</thead>
<tbody>
<tr>
<td>Version 1.0</td>
<td>Initial release as according to Version 1.0 of the Verilog coding package</td>
<td>Paul Bates: Nine Ways R&amp;D (UK) Ltd</td>
<td>31st October 2013</td>
</tr>
<tr>
<td>Version 1.1</td>
<td>Updated file structures, file scripts and parameter setting</td>
<td>Paul Bates: Nine Ways R&amp;D (UK) Ltd</td>
<td>20th January 2015</td>
</tr>
</tbody>
</table>

Copyright © MorethanIP GmbH 2013. All Rights Reserved.