

SILICONOVA

AXI-DMA Documentation

**AXI-DMA Version History**

|  |  |  |
| --- | --- | --- |
| **Rev.** | **Date** | **Description** |
| 1.0 | 02/18/2025 | Initial release with basic AXI-DMA functionality |
| 1.1 | 03/10/2025 | Added support for burst transfers |
| 1.2 | 04/05/2025 | Improved performance and reduced latency |
| 1.3 | 05/15/2025 | Added error handling and status reporting |
| 1.4 | 06/20/2025 | Optimized buffer management and cache handling |

**Contact Information**

* **Sequence Developed By**: [amit.sikder@siliconova.com](mailto:amit.sikder@siliconova.com)
* **Monitor Developed By**: [rashid.shabab@siliconova.com](mailto:rashid.shabab@siliconova.com) imran.limon@siliconova.com
* **Driver Developed By:** [amit.sikder@siliconova.com](mailto:amit.sikder@siliconova.com)
* **Sequences developed by** : [mumtahina.orthy@siliconova.com](mailto:mumtahina.orthy@siliconova.com) & [shazarul.islam@siliconova.com](mailto:shazarul.islam@siliconova.com)
* **Testplan and Testcase developed by** : [mumtahina.orthy@siliconova.com](mailto:mumtahina.orthy@siliconova.com) & [shazarul.islam@siliconova.com](mailto:shazarul.islam@siliconova.com)
* Scoreboard Developed by : [imran.limon@siliconova.com](mailto:imran.limon@siliconova.com) (axi4) & [amit.sikder@siliconova.com](mailto:amit.sikder@siliconova.com) (axi4lite)
* **Coverage Developed By** [rashid.shabab@siliconova.com](mailto:rashid.shabab@siliconova.com)

**Content**

[Chapter 1 5](#_Toc120651584)

[Introduction: 5](#_Toc729442405)

[1.1 About axi-dma 5](#_Toc1370067975)

[What is Fly-By DMA used for? 5](#_Toc822610672)

[1.2 Role of AXI-DMA in Data Movement 6](#_Toc1563411565)

[1.3 Define the goal of this project 6](#_Toc799959887)

[1.4 Explain the interfaces involved 6](#_Toc1572048824)

[1.5 AXI4 (Advanced eXtensible Interface 4) 7](#_Toc138860955)

[What is AXI4? 7](#_Toc646560588)

[AXI4 Channels: 8](#_Toc1864320372)

[AXI4-Lite (AXI4-Lite Register Interface) 8](#_Toc883751927)

[1.6 CSR (Control and Status Registers) 8](#_Toc108855386)

[Role of CSR (Control and Status Registers) 10](#_Toc287849880)

[How CSR Works – Functional Flow 11](#_Toc1269897552)

[1. Initialization by CPU 11](#_Toc1456796083)

[2. DMA Reads the CSRs 11](#_Toc1913783074)

[3. Transfer Execution 11](#_Toc383148088)

[4. Status Updates 11](#_Toc1870163855)

[Chapter 2 11](#_Toc1976473794)

[2.1 AXI-DMA Architecture 12](#_Toc1306717559)

[2.2 AXI Channels Operation 12](#_Toc2095566146)

[2.3.3 Write Transaction 12](#_Toc1372814556)

[2.3.4 Read Transaction 13](#_Toc1067119049)

[2.4 Memory Initialization (mem\_write) 14](#_Toc546184257)

[2.5 Captured AXI4-Lite CSR Configuration 14](#_Toc1451742831)

[Chapter 3 14](#_Toc770350138)

[3.1 Testbench Architecture: 14](#_Toc1398853373)

[Chapter 4 16](#_Toc705563073)

[AXI-DMA Test Plan 16](#_Toc646468277)

[Chapter 5 21](#_Toc972622678)

[Axi4lite 22](#_Toc805190277)

[5.1 axi4lite\_m\_agent.sv 22](#_Toc258353562)

[Fields: 22](#_Toc96489290)

[Methods: 22](#_Toc23717486)

[5.2 axi4lite\_m\_driver 23](#_Toc833701813)

[Fields: 23](#_Toc726126492)

[Methods: 23](#_Toc199390022)

[5.3 axi4lite\_m\_seq\_item 24](#_Toc118029071)

[Fields: 24](#_Toc1314686079)

[Methods: 25](#_Toc1002557140)

[axi4lite\_m\_seq\_lib 26](#_Toc1591829997)

[Fields: 26](#_Toc522605856)

[Methods: 26](#_Toc277350052)

[5.4 Axi4lite\_master\_if Interface 26](#_Toc1437626849)

[Description: 26](#_Toc586134134)

[Fields: 27](#_Toc681349145)

[5.5 AXI4-Lite UVM Monitor 28](#_Toc1809321408)

[Responsibilities 28](#_Toc1098300255)

[Methods: 28](#_Toc1927187819)

[Monitored Signals 28](#_Toc769085617)

[Integration in Testbench 31](#_Toc842784598)

[5.6 axi4\_s\_monitor.sv 32](#_Toc571625280)

[Responsibilities: 32](#_Toc1044345657)

[5.7 axi4\_s\_agent.sv : 33](#_Toc190337251)

[Methods: 34](#_Toc1939209783)

[5.8 Axi4\_s\_seq\_item 34](#_Toc632874808)

[Methods: 36](#_Toc719480012)

[5.9 axi4\_s\_driver.sv 37](#_Toc1172549989)

[Responsibilities 37](#_Toc1709499292)

[Field List and Descriptions: 38](#_Toc1020398516)

[Flowchart Explanation: AXI4-S Driver Behavior 39](#_Toc17085064)

[1. Initialization Phase 39](#_Toc1291467290)

[Error Conditions Simulated in axi\_s\_error\_driver 40](#_Toc1926957792)

[5.10 scoreboard.sv 40](#_Toc1902121937)

[Responsibilities 41](#_Toc1742440927)

[Flow chart 41](#_Toc1129239375)

[Matching Logic 42](#_Toc1566055296)

[Abort Handling 43](#_Toc1535450386)

[Logging Format 43](#_Toc940792340)

[Chapter 6 44](#_Toc693837644)

[Test 44](#_Toc245157660)

[6.1 Base Test 44](#_Toc1901447528)

[6.2 Sanity Simple Test 45](#_Toc1321362589)

[Key Components 45](#_Toc510680639)

[Test Flow (run\_phase) 45](#_Toc948712810)

[6.2.1 Test Purpose 46](#_Toc122937534)

[6.3 csr\_wr\_rd\_test 46](#_Toc1683480429)

[6.3.1 Test Purpose 46](#_Toc586203199)

[6.3.2 Sequence used 46](#_Toc98773205)

[6.3.3 Sequence flow 47](#_Toc1887273427)

[6.3.4 Addtional Argument 47](#_Toc1128250032)

[6.4 dma\_error\_test 47](#_Toc1577064857)

[6.4.1 Test purpose 47](#_Toc387937669)

[6.4.2 Sequence used 47](#_Toc359133576)

[6.4.3 Sequence flow 48](#_Toc1283296684)

[6.4.4 Addtional Argument 48](#_Toc574547848)

[6.5 multiple\_burst\_transfer\_test 48](#_Toc1882694094)

[6.5.1 Purpose of the multiple\_burst\_transfer\_test UVM Test 48](#_Toc1353238618)

[6.5.2 Sequence Used 49](#_Toc813508594)

[6.5.3 Sequence Flow 49](#_Toc1186074878)

[6.4.4 Addtional Argument 49](#_Toc2057763436)

[6.6 disable\_descriptor\_test 50](#_Toc1000642456)

[6.6.1 Purpose of the Test 50](#_Toc1391211327)

[6.6.3 Sequence Used 50](#_Toc1594464331)

[6.6.3 Sequence Flow 51](#_Toc1045833177)

[6.4.4 Addtional Argument 51](#_Toc1144768309)

[6.7 max\_burst\_configuration\_test 51](#_Toc955958770)

[6.7.1 Purpose of the test 51](#_Toc1066617177)

[6.7.2 Sequence Used 52](#_Toc596133816)

[6.7.3 Sequence Flow 52](#_Toc1590316178)

[6.4.4 Addtional Argument 52](#_Toc1509271299)

[6.8 Multi-descriptor configuration regression test 53](#_Toc993886601)

[6.8.1 Purpose of the test 53](#_Toc388054572)

[6.8.2 Sequence Used 53](#_Toc221433455)

[6.8.3 Sequence Flow 53](#_Toc2094177304)

[6.8.4 Addtional Argument 54](#_Toc515434269)

[6.9 Single burst transfer test (single descriptor, incremental) 55](#_Toc165669677)

[6.8.1 Purpose of the tes 55](#_Toc303168351)

[6.9.2 Sequence Used 55](#_Toc41428693)

[6.9.3 Sequence flow 55](#_Toc698863962)

[6.9.4 Addtional Argument 56](#_Toc1148613928)

[6.10 Unaligned source transfer test 56](#_Toc549088255)

[6.10.1 Purpose of the test 56](#_Toc683460531)

[6.10.2 Sequence used 56](#_Toc1334276679)

[6.10.3 Sequence Flow 57](#_Toc566733262)

[6.10.4 Addtional Argument 57](#_Toc1659643463)

[6.11 Unaligned destination transfer test 57](#_Toc1246895424)

[6.11.1 Purpose of the test 58](#_Toc562745624)

[6.11.3 Sequence flow 58](#_Toc361451925)

[6.11.4 Addtional Argument 58](#_Toc1482715770)

[6.12 Burst exceeds 4k boundary test (single descriptor, large length 59](#_Toc281949028)

[6.12.1 Purpose of the test 59](#_Toc617258222)

[6.9.4 Addtional Argument 60](#_Toc777926890)

[6.13 4k bytes transfer test (single descriptor) 60](#_Toc641285422)

[Test purpose 60](#_Toc1691302435)

[Sequence used 60](#_Toc372957343)

[Sequence flow 61](#_Toc454871291)

[Addtional Argument 61](#_Toc1022514189)

[6.14 4k bytes transfer test (single descriptor) 62](#_Toc573059082)

[Test purpose 62](#_Toc627915094)

[Sequence used 62](#_Toc1350046240)

[Sequence flow 62](#_Toc187135848)

[Addtional Argument 63](#_Toc568924814)

[6.15 2 Burst: Read Fixed,Write Increment 63](#_Toc1815988799)

[Test purpose 63](#_Toc799519447)

[Sequence used 64](#_Toc1658782234)

[Sequence flow 64](#_Toc177890369)

[Addtional Argument 64](#_Toc333275587)

[6.16 Basic functionality test (single descriptor, aligned) 65](#_Toc1480065317)

[Test purpose 65](#_Toc1602486050)

[Sequence used 65](#_Toc629523526)

[Sequence flow 66](#_Toc1451838522)

[Addtional Argument 66](#_Toc1341841082)

[6.17 Multiple-burst transfer (multi descriptor, incremental burst) 66](#_Toc621306962)

[Test purpose 67](#_Toc1935066278)

[Sequence used 67](#_Toc646881935)

[Sequence flow 67](#_Toc470148781)

[Addtional Argument 67](#_Toc1700155782)

[6.18 Disable descriptor test (single descriptor, control=0 to disable) 68](#_Toc504617109)

[Test purpose 68](#_Toc886950140)

[Sequence used 68](#_Toc747006126)

[Sequence flow 68](#_Toc1398571354)

[Addtional Argument 69](#_Toc1023600811)

[6.19 Max burst configuration test (single descriptor, fixed burst) 69](#_Toc725585580)

[Test purpose 69](#_Toc309901985)

[Sequence used 70](#_Toc361574882)

[Sequence flow 70](#_Toc2113590042)

[Addtional Argument 70](#_Toc1842078940)

[Test purpose 71](#_Toc213781249)

[Sequence used 71](#_Toc784658611)

[Sequence flow 72](#_Toc1053261026)

[Addtional Argument 72](#_Toc948741867)

[Test purpose 73](#_Toc892860008)

[Sequence used 73](#_Toc1352874126)

[Sequence flow 73](#_Toc907211906)

[Addtional Argument 73](#_Toc2098307712)

[Test purpose 74](#_Toc1747224287)

[Sequence used 74](#_Toc1693615809)

[Sequence flow 74](#_Toc1861996629)

[Addtional Argument 75](#_Toc197045244)

[Test purpose 75](#_Toc2142995009)

[Sequence used 76](#_Toc1745252098)

[Sequence flow 76](#_Toc503114738)

[Addtional Argument 76](#_Toc48486831)

[Test purpose 77](#_Toc1217555363)

[Sequence used 77](#_Toc919236689)

[Sequence flow 77](#_Toc1375404113)

[Addtional Argument 78](#_Toc68066286)

[Test purpose 78](#_Toc46944329)

[Sequence used 78](#_Toc1379343732)

[Sequence flow 79](#_Toc91596435)

[Addtional Argument 79](#_Toc611016453)

[Sequence flow 79](#_Toc809987703)

[Addtional Argument 80](#_Toc507313264)

[Chapter 7 80](#_Toc1650808829)

[Coverage 81](#_Toc645445058)

[7.1 Purpose of Coverage 81](#_Toc488579228)

[7.2 Coverage plan 81](#_Toc1028279284)

[7.3 Code coverage – CSR DMA 85](#_Toc786310892)

[7.6 Code coverage – DMA Read Streamer 87](#_Toc2084029521)

[7.8 Code coverage – DMA Master I/F 88](#_Toc1759836250)

[7.9 AXI DMA – Functional 89](#_Toc835395427)

[7.10 Covergroups 89](#_Toc578069299)

[Chapter 1 5](#_Toc1285089715)

[Introduction: 5](#_Toc64532338)

[1.1 About axi-dma 5](#_Toc1911056497)

[What is Fly-By DMA used for? 5](#_Toc2128616252)

[1.2 Role of AXI-DMA in Data Movement 6](#_Toc1996513963)

[1.3 Define the goal of this project 6](#_Toc2032556005)

[1.4 Explain the interfaces involved 6](#_Toc880251)

[1.5 AXI4 (Advanced eXtensible Interface 4) 7](#_Toc2109330217)

[What is AXI4? 7](#_Toc379083980)

[AXI4 Channels: 8](#_Toc1865960709)

[AXI4-Lite (AXI4-Lite Register Interface) 8](#_Toc1590493405)

[1.6 CSR (Control and Status Registers) 8](#_Toc813020228)

[Role of CSR (Control and Status Registers) 10](#_Toc1137329962)

[How CSR Works – Functional Flow 11](#_Toc1188072596)

[1. Initialization by CPU 11](#_Toc1677152549)

[2. DMA Reads the CSRs 11](#_Toc1712930284)

[3. Transfer Execution 11](#_Toc1838967908)

[4. Status Updates 11](#_Toc1727826541)

[Chapter 2 11](#_Toc1167148497)

[2.1 AXI-DMA Architecture 12](#_Toc1971882596)

[2.2 AXI Channels Operation 12](#_Toc1297982711)

[2.3.3 Write Transaction 12](#_Toc883562886)

[2.3.4 Read Transaction 13](#_Toc824150609)

[2.4 Memory Initialization (mem\_write) 14](#_Toc193268589)

[2.5 Captured AXI4-Lite CSR Configuration 14](#_Toc1315559420)

[Chapter 3 14](#_Toc713216192)

[3.1 Testbench Architecture: 14](#_Toc1412784011)

[Chapter 4 16](#_Toc1943268787)

[AXI-DMA Test Plan 16](#_Toc1396701895)

[Chapter 5 21](#_Toc1586124698)

[Axi4lite 22](#_Toc1196071311)

[5.1 axi4lite\_m\_agent.sv 22](#_Toc2128679607)

[Fields: 22](#_Toc284633197)

[Methods: 22](#_Toc1969954362)

[5.2 axi4lite\_m\_driver 23](#_Toc254784476)

[Fields: 23](#_Toc960759259)

[Methods: 23](#_Toc2115208989)

[5.3 axi4lite\_m\_seq\_item 24](#_Toc1249165827)

[Fields: 24](#_Toc208333530)

[Methods: 25](#_Toc733774372)

[axi4lite\_m\_seq\_lib 26](#_Toc546750129)

[Fields: 26](#_Toc990961979)

[Methods: 26](#_Toc1555540734)

[5.4 Axi4lite\_master\_if Interface 26](#_Toc257954474)

[Description: 26](#_Toc1909731323)

[Fields: 27](#_Toc1729435003)

[5.5 AXI4-Lite UVM Monitor 28](#_Toc972274217)

[Responsibilities 28](#_Toc1287735985)

[Methods: 28](#_Toc1367290724)

[Monitored Signals 28](#_Toc459800181)

[Integration in Testbench 31](#_Toc1935995370)

[5.6 axi4\_s\_monitor.sv 32](#_Toc1802164133)

[Responsibilities: 32](#_Toc403510966)

[5.7 axi4\_s\_agent.sv : 33](#_Toc873369051)

[Methods: 34](#_Toc495820413)

[5.8 Axi4\_s\_seq\_item 34](#_Toc131251241)

[Methods: 36](#_Toc2000055496)

[5.9 axi4\_s\_driver.sv 37](#_Toc1516236984)

[Responsibilities 37](#_Toc1175626972)

[Field List and Descriptions: 38](#_Toc519228508)

[Flowchart Explanation: AXI4-S Driver Behavior 39](#_Toc18636180)

[1. Initialization Phase 39](#_Toc1385407655)

[Error Conditions Simulated in axi\_s\_error\_driver 40](#_Toc551766439)

[5.10 scoreboard.sv 40](#_Toc719369063)

[Responsibilities 41](#_Toc1870798295)

[Flow chart 41](#_Toc1256376244)

[Matching Logic 42](#_Toc562053279)

[Abort Handling 43](#_Toc839534691)

[Logging Format 43](#_Toc1057194709)

[Chapter 6 44](#_Toc1902290581)

[Test 44](#_Toc847048068)

[6.1 Base Test 44](#_Toc1140723596)

[6.2 Sanity Simple Test 45](#_Toc273796949)

[Key Components 45](#_Toc351349105)

[Test Flow (run\_phase) 45](#_Toc1005505481)

[6.2.1 Test Purpose 46](#_Toc716006038)

[6.3 csr\_wr\_rd\_test 46](#_Toc455054460)

[6.3.1 Test Purpose 46](#_Toc848909435)

[6.3.2 Sequence used 46](#_Toc2073993927)

[6.3.3 Sequence flow 47](#_Toc110834344)

[6.3.4 Addtional Argument 47](#_Toc1411745898)

[6.4 dma\_error\_test 47](#_Toc2116519422)

[6.4.1 Test purpose 47](#_Toc2037993289)

[6.4.2 Sequence used 47](#_Toc2106222696)

[6.4.3 Sequence flow 48](#_Toc1076040696)

[6.4.4 Addtional Argument 48](#_Toc1741374027)

[6.5 multiple\_burst\_transfer\_test 48](#_Toc1392799787)

[6.5.1 Purpose of the multiple\_burst\_transfer\_test UVM Test 48](#_Toc1478918631)

[6.5.2 Sequence Used 49](#_Toc1082199356)

[6.5.3 Sequence Flow 49](#_Toc1376527315)

[6.4.4 Addtional Argument 49](#_Toc1104234910)

[6.6 disable\_descriptor\_test 50](#_Toc1164250255)

[6.6.1 Purpose of the Test 50](#_Toc402052692)

[6.6.3 Sequence Used 50](#_Toc541027749)

[6.6.3 Sequence Flow 51](#_Toc1682212477)

[6.4.4 Addtional Argument 51](#_Toc1299709439)

[6.7 max\_burst\_configuration\_test 51](#_Toc1606054596)

[6.7.1 Purpose of the test 51](#_Toc2123925183)

[6.7.2 Sequence Used 52](#_Toc1213727495)

[6.7.3 Sequence Flow 52](#_Toc12202676)

[6.4.4 Addtional Argument 52](#_Toc1290035644)

[6.8 Multi-descriptor configuration regression test 53](#_Toc557482157)

[6.8.1 Purpose of the test 53](#_Toc1288477777)

[6.8.2 Sequence Used 53](#_Toc975367321)

[6.8.3 Sequence Flow 53](#_Toc727622301)

[6.8.4 Addtional Argument 54](#_Toc74825556)

[6.9 Single burst transfer test (single descriptor, incremental) 55](#_Toc984258266)

[6.8.1 Purpose of the tes 55](#_Toc765086368)

[6.9.2 Sequence Used 55](#_Toc2066121436)

[6.9.3 Sequence flow 55](#_Toc1226261095)

[6.9.4 Addtional Argument 56](#_Toc1797321106)

[6.10 Unaligned source transfer test 56](#_Toc2142454347)

[6.10.1 Purpose of the test 56](#_Toc609574959)

[6.10.2 Sequence used 56](#_Toc536477818)

[6.10.3 Sequence Flow 57](#_Toc1494233949)

[6.10.4 Addtional Argument 57](#_Toc376578298)

[6.11 Unaligned destination transfer test 57](#_Toc520891278)

[6.11.1 Purpose of the test 58](#_Toc700628454)

[6.11.3 Sequence flow 58](#_Toc532283158)

[6.11.4 Addtional Argument 58](#_Toc1486840588)

[6.12 Burst exceeds 4k boundary test (single descriptor, large length 59](#_Toc2004073785)

[6.12.1 Purpose of the test 59](#_Toc163325086)

[6.9.4 Addtional Argument 60](#_Toc916320020)

[6.13 4k bytes transfer test (single descriptor) 60](#_Toc1924478644)

[Test purpose 60](#_Toc1541451233)

[Sequence used 60](#_Toc917918302)

[Sequence flow 61](#_Toc167055837)

[Addtional Argument 61](#_Toc1821077748)

[6.14 4k bytes transfer test (single descriptor) 62](#_Toc1376932604)

[Test purpose 62](#_Toc1011238888)

[Sequence used 62](#_Toc1337266267)

[Sequence flow 62](#_Toc300012481)

[Addtional Argument 63](#_Toc1053439740)

[6.15 2 Burst: Read Fixed,Write Increment 63](#_Toc1494947700)

[Test purpose 63](#_Toc1044777496)

[Sequence used 64](#_Toc796342936)

[Sequence flow 64](#_Toc820613843)

[Addtional Argument 64](#_Toc1302492741)

[6.16 Basic functionality test (single descriptor, aligned) 65](#_Toc1169194128)

[Test purpose 65](#_Toc591102414)

[Sequence used 65](#_Toc1989875093)

[Sequence flow 66](#_Toc1501786204)

[Addtional Argument 66](#_Toc1809590822)

[6.17 Multiple-burst transfer (multi descriptor, incremental burst) 66](#_Toc1990074724)

[Test purpose 67](#_Toc1176660460)

[Sequence used 67](#_Toc590728041)

[Sequence flow 67](#_Toc384624334)

[Addtional Argument 67](#_Toc726198436)

[6.18 Disable descriptor test (single descriptor, control=0 to disable) 68](#_Toc680032331)

[Test purpose 68](#_Toc594777741)

[Sequence used 68](#_Toc549354732)

[Sequence flow 68](#_Toc1067448851)

[Addtional Argument 69](#_Toc1355902827)

[6.19 Max burst configuration test (single descriptor, fixed burst) 69](#_Toc18373015)

[Test purpose 69](#_Toc1212351185)

[Sequence used 70](#_Toc302970597)

[Sequence flow 70](#_Toc1776562444)

[Addtional Argument 70](#_Toc1544152201)

[Test purpose 71](#_Toc338833635)

[Sequence used 71](#_Toc1834095302)

[Sequence flow 72](#_Toc1238354044)

[Addtional Argument 72](#_Toc1515641314)

[Test purpose 73](#_Toc1930536239)

[Sequence used 73](#_Toc698676259)

[Sequence flow 73](#_Toc552918527)

[Addtional Argument 73](#_Toc1831880198)

[Test purpose 74](#_Toc2062874753)

[Sequence used 74](#_Toc881792719)

[Sequence flow 74](#_Toc1070661065)

[Addtional Argument 75](#_Toc1720720005)

[Test purpose 75](#_Toc132171531)

[Sequence used 76](#_Toc326196482)

[Sequence flow 76](#_Toc771398147)

[Addtional Argument 76](#_Toc2008578584)

[Test purpose 77](#_Toc496425361)

[Sequence used 77](#_Toc1817116790)

[Sequence flow 77](#_Toc650473888)

[Addtional Argument 78](#_Toc506312593)

[Test purpose 78](#_Toc1513507348)

[Sequence used 78](#_Toc1300335259)

[Sequence flow 79](#_Toc727060119)

[Addtional Argument 79](#_Toc1557128745)

[Sequence flow 79](#_Toc18840492)

[Addtional Argument 80](#_Toc942018946)

[Chapter 7 80](#_Toc2090581973)

[Coverage 81](#_Toc1731740143)

[7.1 Purpose of Coverage 81](#_Toc2035624562)

[7.2 Coverage plan 81](#_Toc162379804)

[7.3 Code coverage – CSR DMA 85](#_Toc723619088)

[7.6 Code coverage – DMA Read Streamer 87](#_Toc947547001)

[7.8 Code coverage – DMA Master I/F 88](#_Toc460684195)

[7.9 AXI DMA – Functional 89](#_Toc1503805979)

[7.10 Covergroups 89](#_Toc1551581545)

# Chapter 1

## Introduction:

### 1.1 About axi-dma

AXI-DMA (Advanced eXtensible Interface - Direct Memory Access) is a specialized hardware module designed to efficiently transfer data between memory and peripherals without CPU intervention. It supports high-speed data movement by leveraging the **AXI4 (Advanced eXtensible Interface) protocol**, commonly used in modern System-on-Chip (SoC) architectures.

#### What is Fly-By DMA used for?

In fly-by mode DMA, the function of the RD and WE signals (for DMA only) are reversed. This allows a host to move data between an external device and the ADV202/212 with the use of a single strobe.

In encode mode with fly-by DMA, the host can use the RDFB signal (WE pin) to simultaneously read from the ADV202/212 and write to an external device like memory.

In decode mode with fly-by DMA, the host can use the WEFB signal (RD pin) to simultaneously read from the external device and write to the ADV202/212.

### 1.2 Role of AXI-DMA in Data Movement

AXI-DMA serves as a bridge between memory and peripheral devices, enabling efficient data transfers through the following channels:

* **Memory-to-Peripheral (MM2S - Memory-Mapped to Stream):** Transfers data from system memory to a peripheral device via an AXI stream interface. [github.com](https://github.com/cathalmccabe/PYNQ_tutorials/blob/master/dma/dma_tutorial_part1.md)
* **Peripheral-to-Memory (S2MM - Stream to Memory-Mapped):** Receives data from a peripheral device through an AXI stream and writes it to system memory.

By offloading these data transfer tasks from the CPU, AXI-DMA reduces processor workload, minimizes latency, and enhances overall system performance.

### 1.3 Define the goal of this project

* **Functional Verification:**

Ensure that the AXI-DMA IP core accurately performs data transfers between memory and peripherals, adhering to specified control and status register (CSR) configurations.

* **Protocol Compliance:**

Verify that the AXI-DMA operates in full compliance with the AMBA AXI4 protocol standards, maintaining proper signaling and timing throughout all transactions.

* **Performance Validation:**

Assess the DMA's ability to handle various data transfer scenarios, including different burst lengths and transfer sizes, to confirm that it meets the desired throughput and latency requirements.

* **Error Handling:**

Test the DMA's response to error conditions, such as unaligned transfers or unsupported burst types, to ensure robust error detection and recovery mechanisms are in place.

* **Resource Efficiency:**

Evaluate the design's resource utilization to ensure it meets efficiency targets without compromising functionality or performance.

### 1.4 Explain the interfaces involved

**Interface Used**:

* AXI4 lite for Dma configaration
* AXI4 for memory read/write
* CSR (Control and Status Registers) for DMA control

### 1.5 AXI4 (Advanced eXtensible Interface 4)

### **What is AXI4?**

AXI4 (Advanced eXtensible Interface version 4) is a high-performance, high-frequency bus protocol defined by ARM as part of its AMBA (Advanced Microcontroller Bus Architecture) specification. It is widely used in SoC designs to connect IP blocks and enable efficient memory-mapped transactions.

* **Type:** High-performance memory-mapped interface
* **Role in AXI-DMA:** Used for high-speed data transfers between system memory and peripherals.
* **Key Features:**
  + Supports **burst-based transactions** for efficient data movement.
  + Operates as an **AXI Master** to initiate read and write transactions.
  + Includes **AXI4 Read and Write Channels** (AW, W, B, AR, R).
  + Ensures **high throughput and low-latency data transfers**.
* **AXI-DMA Usage in AXI4:**
  + **Memory-to-Peripheral (MM2S - Memory-Mapped to Stream):** Reads data from system memory and streams it to a peripheral.
  + **Peripheral-to-Memory (S2MM - Stream to Memory-Mapped):** Receives data from a peripheral and writes it to system memory.

|  |  |
| --- | --- |
| **Feature** | **Description** |
| **Burst-based transfers** | Supports burst reads/writes (up to 256 data beats in a single transaction). |
| **Separate address/data** | Separate channels for address, data, and response for both reads and writes. |
| **Out-of-order support** | Can reorder transactions based on ID (AXI3/AXI4), preserving data integrity. |
| **Low latency** | Decoupled handshakes using VALID/READY signals minimize latency. |
| **Protocol flexibility** | Supports multiple interconnect topologies and QoS levels. |

### **AXI4 Channels:**

|  |  |  |
| --- | --- | --- |
| **Channel** | **Direction** | **Purpose** |
| **AW (Write Address)** | Master → Slave | Sends the address and control info for a write burst. |
| **W (Write Data)** | Master → Slave | Carries write data for the burst. |
| **B (Write Response)** | Slave → Master | Acknowledges completion of a write burst. |
| **AR (Read Address)** | Master → Slave | Sends the address and control info for a read burst. |
| **R (Read Data)** | Slave → Master | Carries read data and response for the read burst. |

### AXI4-Lite (AXI4-Lite Register Interface)

* **Type:** Simplified, low-bandwidth AXI interface
* **Role in AXI-DMA:** Used for **control and configuration** of the DMA engine.
* **Key Features:**
  + Operates as an **AXI Slave**, allowing software to configure DMA registers.
  + Supports **single-beat transactions** (no burst support).
  + Used for reading/writing DMA **Control and Status Registers (CSRs)**.
* **AXI-DMA Usage in AXI4-Lite:**
  + Software writes to control registers (e.g., start, stop, configure burst length).
  + Software reads DMA status (e.g., transfer complete, error status).

## 1.6 CSR (Control and Status Registers)

|  |  |
| --- | --- |
| **name** | **offset\_address** |
| [dma\_control](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_control) | 0x00 |
| [dma\_status](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_status) | 0x08 |
| [dma\_error\_addr](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_error_addr) | 0x10 |
| [dma\_error\_stats](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_error_stats) | 0x18 |
| [dma\_desc\_src\_addr[2]](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_desc_src_addr) | 0x20 0x28 |
| [dma\_desc\_dst\_addr[2]](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_desc_dst_addr) | 0x30 0x38 |
| [dma\_desc\_num\_bytes[2]](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_desc_num_bytes) | 0x40 0x48 |
| [dma\_desc\_cfg[2]](https://github.com/aignacio/axi_dma/blob/master/csr_out/csr_dma.md#csr_dma-dma_desc_cfg) | 0x50 0x58 |

### **Role of CSR (Control and Status Registers)**

1. **Configure DMA Transfers**
   1. The CPU writes to the CSRs to set up DMA parameters, such as:
      1. **Source address**
      2. **Destination address**
      3. **Number of bytes to transfer**
      4. **Control flags** (e.g., start, abort)
2. **Control Operation**
   1. The CSRs include control bits to:
      1. **Start** a DMA transfer (GO)
      2. **Abort** a transfer (ABORT)
      3. **Enable/disable descriptors**
3. **Monitor DMA Status**
   1. The software can **read the status register** to:
      1. Detect if a transfer is **complete** (dma\_done\_o)
      2. Check for **errors** (dma\_error\_o)
      3. Retrieve internal status such as error type or problematic address

### **How CSR Works – Functional Flow**

#### **1. Initialization by CPU**

* The CPU writes to the CSRs through an **AXI4-Lite interface**.
* Register fields include:
  + src\_addr (Source)
  + dst\_addr (Destination)
  + num\_bytes (Transfer size)
  + control (Control bits)

#### **2. DMA Reads the CSRs**

* The DMA controller continuously monitors or is triggered to:
  + Load the values from the control registers
  + Begin transfer when GO is set

#### **3. Transfer Execution**

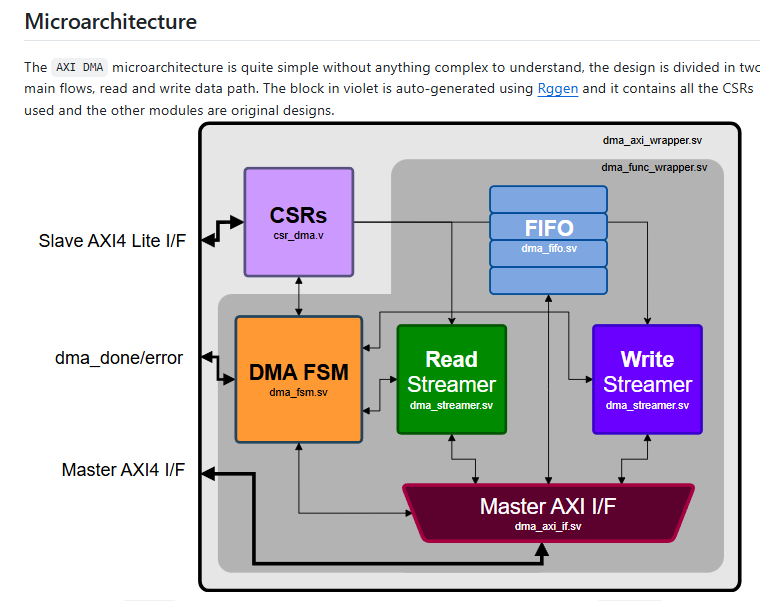
* Based on CSR settings, the DMA generates:
  + AXI Read bursts (from src\_addr)
  + AXI Write bursts (to dst\_addr)
* WSTRB and burst types are calculated accordingly

#### **4. Status Updates**

* When done or if an error occurs:
  + The DMA updates status registers
  + Signals like dma\_done\_o and dma\_error\_o are set
  + Software polls or gets an interrupt

# Chapter 2

## 2.1 AXI-DMA Architecture



## **2.2 AXI Channels Operation**

### **2.3.3 Write Transaction**

**1. AW (Write Address Channel)**

* **Purpose:** Receives and acknowledges write address from master.
* **Operational Steps:**
  + Waits for dma\_m\_awvalid assertion.
  + Checks if received address is within valid memory range.
  + Captures burst length, burst type, and address.
  + Stores address in wr\_queue for matching with write data.
  + Asserts dma\_m\_awready to confirm receipt.

**2. W (Write Data Channel)**

* **Purpose:** Receives actual data from master for memory write.
* **Operational Steps:**
  + Waits for dma\_m\_wvalid.
  + Ensures corresponding address is available (wr\_queue).
  + Captures data and writes to internal memory array (mem1).
  + Increments memory address for burst transfers.
  + Signals completion using internal flags.

**3. B (Write Response Channel)**

* **Purpose:** Sends write response (OKAY or DECERR) back to master.
* **Operational Steps:**
  + Checks for completion of write data reception.
  + Asserts dma\_m\_bvalid and provides appropriate response (dma\_m\_bresp).
  + Waits for master's acknowledgment (dma\_m\_bready).
  + Deasserts response signals after acknowledgment.

### **2.3.4 Read Transaction**

1. AR (Read Address Channel)

* **Purpose:** Receives and acknowledges read address from master.
* **Operational Steps:**
  + Waits for dma\_m\_arvalid assertion.
  + Checks if the address is within the valid memory range.
  + Captures burst length, type, and address.
  + Stores address in rd\_queue for matching with read data.
  + Asserts dma\_m\_arready to confirm receipt.

2. R (Read Data Channel)

* **Purpose:** Sends requested data from memory to master.
* **Operational Steps:**
  + Waits for master readiness (dma\_m\_rready).
  + Ensures corresponding address is available (rd\_queue).
  + Retrieves data from internal memory (mem).
  + Provides data (dma\_m\_rdata), and burst information (dma\_m\_rlast).
  + Sends a response (dma\_m\_rresp) to master (OKAY status).

## **2.4 Memory Initialization (mem\_write)**

* Randomly initializes memory content to simulate realistic memory operations.
* Invoked during the driver's run\_phase.

## 2.5 Captured AXI4-Lite CSR Configuration

|  |  |  |  |
| --- | --- | --- | --- |
| **Address Offset** | **Purpose** | **Width** | **Description** |
| 0x00 | Control | 32 | Bit 1 = abort, Bit 0 = go |
| 0x20 | Source Address | 32 | Source of the DMA read |
| 0x30 | Destination Address | 32 | Destination of the DMA write |
| 0x40 | Num Byte | 32 | Total byte count for this transfer (pushed to queue) |
|  | dma\_done\_o | 1 | Transfer completion indication (pushed separately) |
|  | dma\_error\_o | 1 | Indicates if the DMA failed (pushed separately) |

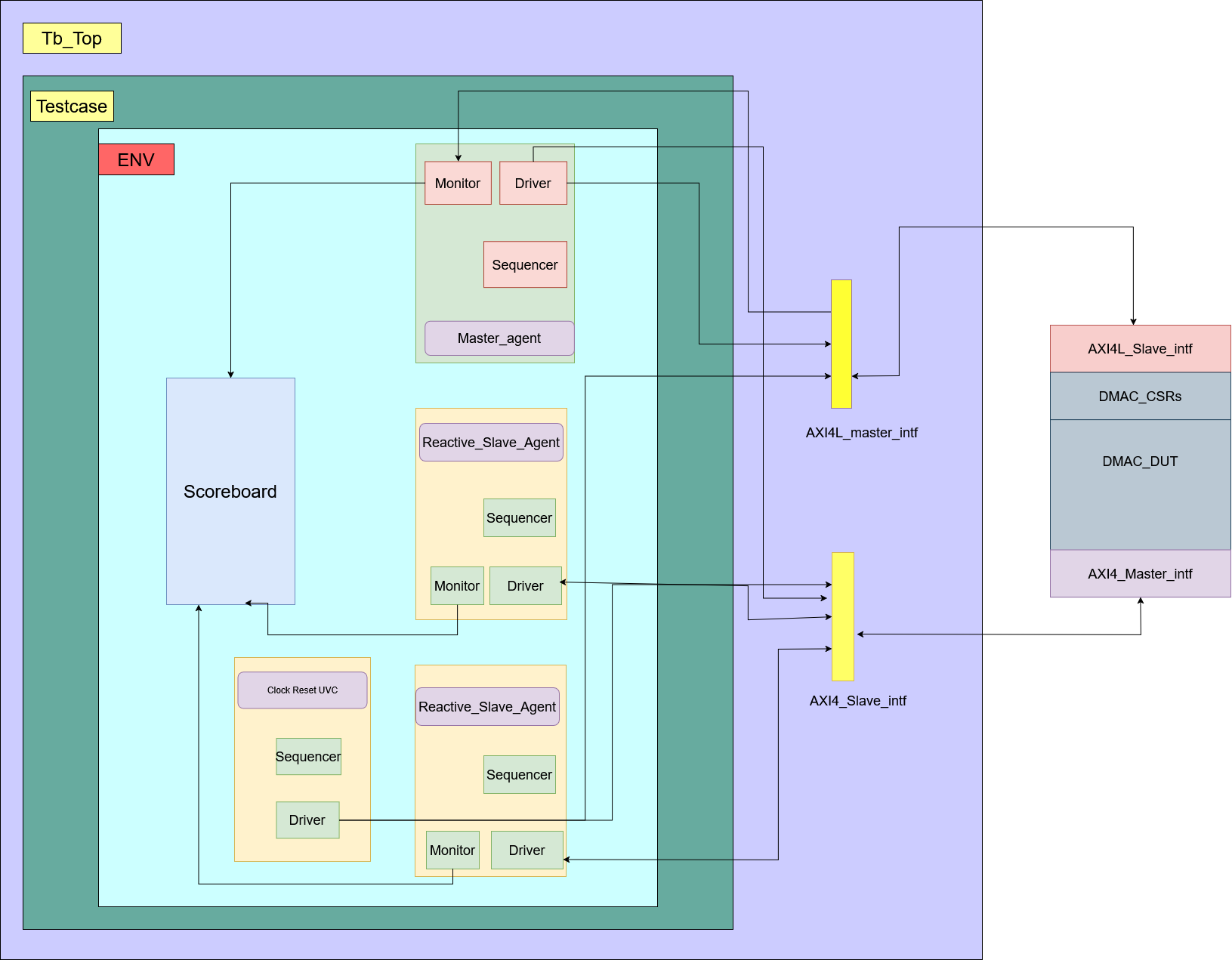
# Chapter 3

**AXI-DMA Verification Environment (UVM)**

## 3.1 Testbench Architecture:

* **Testcase**: Describes the verification scenarios.
* **Environment (ENV)**: Main UVM components.
* **Master Agent**: Generates AXI Master transactions.
* **Slave Agent(s)**: Reactive slaves to verify read/write operations.
* **Scoreboard**: Checks data integrity.
* **Clock and Reset UVC**: Ensures proper timing.

The AXI-DMA project is designed to facilitate high-performance direct memory access (DMA) transfers using the AMBA AXI4-Lite and AXI4 interfaces. The architecture consists of key components including a Master Agent, a Reactive Slave Agent, and the DMAC DUT (Design Under Test). The Master Agent generates and monitors transactions, while the Reactive Slave Agent handles responses. The DMAC DUT comprises AXI4 Slave and Master interfaces along with DMAC control and status registers. A scoreboard is implemented to validate data integrity and transaction correctness.



# Chapter 4

## AXI-DMA Test Plan

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Serial No** | **Test Case** | **Objective** | **Input Configuration** | **Expected Result** |
| 1 | **CSR write read test** | To ensure that all the registers are written and read back properly | write to all configuration registers with their appropiate offset through axi4\_lite. | After reset and before configuration, all the registers should be in their default values. Then we configured the registers.The dma will behave accordingly .Finally we'll read back all the registers and got the same value as we configured and if dma operation is successful on the status register we'll get 1CAFE . If the value we write on the registers are read back with the same values then the test will pass otherwise fail |
| 2 | **Single-Burst Transfer (Aligned, INCR Mode)** | Verify a simple transfer with 64bytes and max\_burst = 16. | - Write to dma\_control: go=1, abort=0, max\_burst=16  - Write descriptor 0 registers: src\_addr,dest\_addr with aligned values, num\_bytes = 0x44, desc\_cfg = 0x4.  - Initiate DMA transfer | - DMA should generates only one read and write bursts based on the configuration.  - ARLEN = 16, AWSIZE = 2, AWBURST = INCR for a 64-byte transfer.  - Write strobe (WSTRB) should be full (0xF) for all 32-bit beats.  - dma\_done\_o is asserted when transfer is complete.   If partial or more than one bursts are created or if dma\_erro\_o signal is asserted during the transaction then the test will fail. |
| 3 | **Unaligned Transfer (Source and destination Unaligned)** | To Verify that an unaligned source and destination address is handled correctly by aligning the address and with proper WSTRB masking. | dma\_control: go = 1, abort = 0, max\_burst = 8  - Descriptor 0: src\_addr =(unaligned), dst\_addr = (unaligned), num\_bytes = 0x24, desc\_cfg = 0x4 | Though we configure unaligned source and destination address ,a nearest aligned address will be issued in the araddr and awaddr. Proper byte masking will be seen on wdata and wstrb.  If dut does not generate proper wdata and wstrb the test will fail.With proper byte masking and wstrb generation along with configured num\_bytes transferring, the test will pass.   - Data transferred matches the descriptor. |
| 4 | **Unaligned Transfer (Destination Unaligned)** | To verify that an unaligned destination address leads to proper alignment and WSTRB computation on the write channel. | - dma\_control: go = 1, abort = 0, max\_burst = 8  - Descriptor 0: src\_addr = 0x10000000 (aligned), dst\_addr = 0x20000002 (unaligned), num\_bytes = 256, desc\_cfg = 0x4 | - AXI Master: AWADDR shows aligned destination; WSTRB on unaligned beats is not 0xF (e.g. if starting at offset 2, WSTRB might be as per internal logic).  - dma\_done\_o is asserted |
| 5 | **Abort Operation Mid-Transfer** | To verify that asserting an abort stops new transactions and leads to a controlled termination. | - dma\_control: Initially, go = 1, abort = 0, max\_burst = 8  - Descriptor 0: any valid settings (e.g., 256 bytes, aligned addresses)   Start DMA transfer.  - In the middle of transfer, need to update dma\_control via CSR write to set abort = 1.  - Observe that new transactions are not initiated and pending FIFO data is ignored | Ongoing transactions may complete but no new AW/AR/wdata transactions are issued.  - The FSM eventually transitions to a DONE state.  - dma\_done\_o is asserted   If dma stops transferring data upon getting abort signal from csr ,the test will pass. If dma still continues to transfer data though it gets the abort signal high,the test will fail. |
| 6 | **Error Injection (Slave Error Response)** | To verify that a slave error (AXI\_SLVERR or AXI\_DECERR) is captured and reported by the DMA. | dma\_control: go = 1, abort = 0, max\_burst = 8  - Descriptor 0: Valid settings (e.g., 256 bytes, aligned) | The error is captured, and the internal error registers (dma\_error.addr, dma\_error.type\_err, dma\_error.src) are updated.  - The transfer stops further processing.  - dma\_error\_o is asserted.  If dma\_error\_o signal is asserted upon reading or writing on the illegal region of the slave the test will pass otherwise fail. |
| 7 | **max\_burst Configuration test** | To verify DMA behavior with maximum max\_burst value that is 255. | - dma\_control: Test with max\_burst =255  - Descriptor 0: src and destination address aligned,num\_bytes=0x400 i.e 1024 bytes,desc\_cfg = 0x4 | - For max\_burst = 255:ARLEN,AWLEN 255 i.e a 256 beat burst will be created  - If dma is able to transfer all the 256 beats properly and dma\_done\_o signal is asserted then the test will pass. |
| 8 | **Disabled Descriptor** | To verify that if a descriptor is not enabled, no transfer occurs. | dma\_control: go = 1, abort = 0  - Descriptor 0: Set enable = 0 (desc\_cfg’s enable bit = 0) | - No AW/AR/wdata transactions should be generated.  - No data movement occurs  If no araddr or awaddr is issued thus no data transfer occurs then the test will pass otherwise fail |
| 9 | **Multiple Descriptor Configuration** | Configure two descriptor with different addresses and control signal | Need to configure src\_addr, dst\_addr, and num\_bytes for Descriptor 0 & 1.  2. Enable descriptors (desc\_enable = 1).  3. Set dma\_control.go = 1 | After finishing transaction of both descriptors,DMA should successfully assert dma\_done\_o. If dma\_done\_o asserts after successfully completing all the transfers of both the descriptor then the test will pass. |
| 10 | **Multiple Burst test** | Set max\_burst and num\_bytes such that multiple burst are generated and Transaction should be successful for each burst | Configure dma\_control and dma\_desc\_num\_byte register with different value for multiple burst generation | For every burst if slave respnses wiith OKAY ,finally dma\_done\_o signal will be asserted then the test will pass |
| 11 | **Burst exceeds 4kb boundary** | To Verify that a transfer crossing a 4KB boundary is properly split into bursts. | Configure dma\_control: go = 1, abort = 0, with a valid max\_burst value.  - Program a descriptor with an address near a 4KB boundary, e.g.,  src\_addr = 0x1FFF0000, dst\_addr = 0x20000000, and a num\_bytes value that would normally span beyond 4KB (e.g., 8192 bytes).  - Drive enable\_descriptor = 1. | The DMA detects the 4KB boundary constraint and splits the transfer into bursts that do not cross the boundary. Each burst completes within a single 4KB region. The transfer continues in multiple bursts until all specified bytes in num\_bytes are transferred. dma\_done\_o is asserted after completing the transfers successfully.   If burst splitting after exceeding 4k boundary does not occur then the test will fail. If burst splitting occurs with proper num\_bytes transfer and dma\_done\_o is asserted then the test will pass. |
| 12 | **4k bytes transfer test** | To configure the registers such that 4k bytes will get transferred at one go | Configure dma\_control: go = 1, abort = 0, with a valid max\_burst value.  - Program a descriptor valid aligned souce and destnation address   and a num\_bytes =4kb  - Drive enable\_descriptor = 1 | If 4k bytes gets transferred from two slave peripheral without any interruption in the middle and dma\_done\_o is asserted then the test will pass.If interruption or data drop occurs in the transaction then the test will fail |
| 13 | **same descriptor multiple transaction test** | to verify if in a single descriptor two different configuration of registers works correctly | Descriptor 0: src\_addr =(unaligned), dst\_addr = (unaligned), num\_bytes = 0x24, desc\_cfg = 0   dma\_control: go = 1, abort = 0, max\_burst = 8   after getting dma\_done\_o we need to set go=0, then after a short delay configure descriptor0 registers again with different values than previous and set the go=1 | Same descriptor will assert done twice with two different register configuration thus the test will pass. If dma does not transfer data with the different configuration and assert dma\_done\_o after completion of each configuration then the test will fail. |

# Chapter 5

**UVM Testbench Components**

* Master Agent:
* Monitor
* Driver
* Sequencer
* Reactive Slave Agent:
* Sequencer
* Monitor
* Driver
* Scoreboard:
* Data checking.
* Functional coverage.

### Axi4lite

## 5.1 axi4lite\_m\_agent.sv

A **UVM Agent** is a reusable verification component that encapsulates all necessary objects required to **drive, monitor, and sequence transactions** for a specific interface in a testbench. It serves as a bridge between the testbench and the DUT (Design Under Test) by **generating, injecting, and capturing** transactions on a given interface.

#### **Fields:**

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| axi4lite\_m\_agnt\_cfg | Configuration object for the AXI4-Lite master agent. |
| axi4lite\_m\_drvr | Driver component responsible for driving AXI4-Lite transactions. |
| axi4lite\_m\_mntr | Monitor component that captures and observes AXI4-Lite transactions. |
| axi4lite\_m\_cov | Coverage component for collecting functional coverage metrics. |
| sqncr | UVM sequencer that controls the flow of transaction sequences. |
| axi4lite\_m\_ag  nt\_port | Analysis port used for forwarding monitored transactions to other components. |

#### **Methods:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| new | Constructor method that initializes the AXI4-Lite agent. |
| build\_phase | - Retrieves the agent configuration from the UVM configuration database.  - Creates instances of the driver, sequencer, monitor, and coverage components (if enabled).  - Initializes the agent’s analysis port. |
| connect\_phase | - Connects the driver’s sequence item port to the sequencer.  - Connects the monitor’s analysis port to the agent's analysis port.  - If coverage is enabled, connects the monitor’s analysis port to the coverage component. |

## 5.2 axi4lite\_m\_driver

The AXI4-Lite Driver (axi4lite\_m\_driver) is a UVM component responsible for driving transactions onto the AXI4-Lite interface. It receives transactions from the sequencer, translates them into signal-level activity, and applies them to the DUT interface.

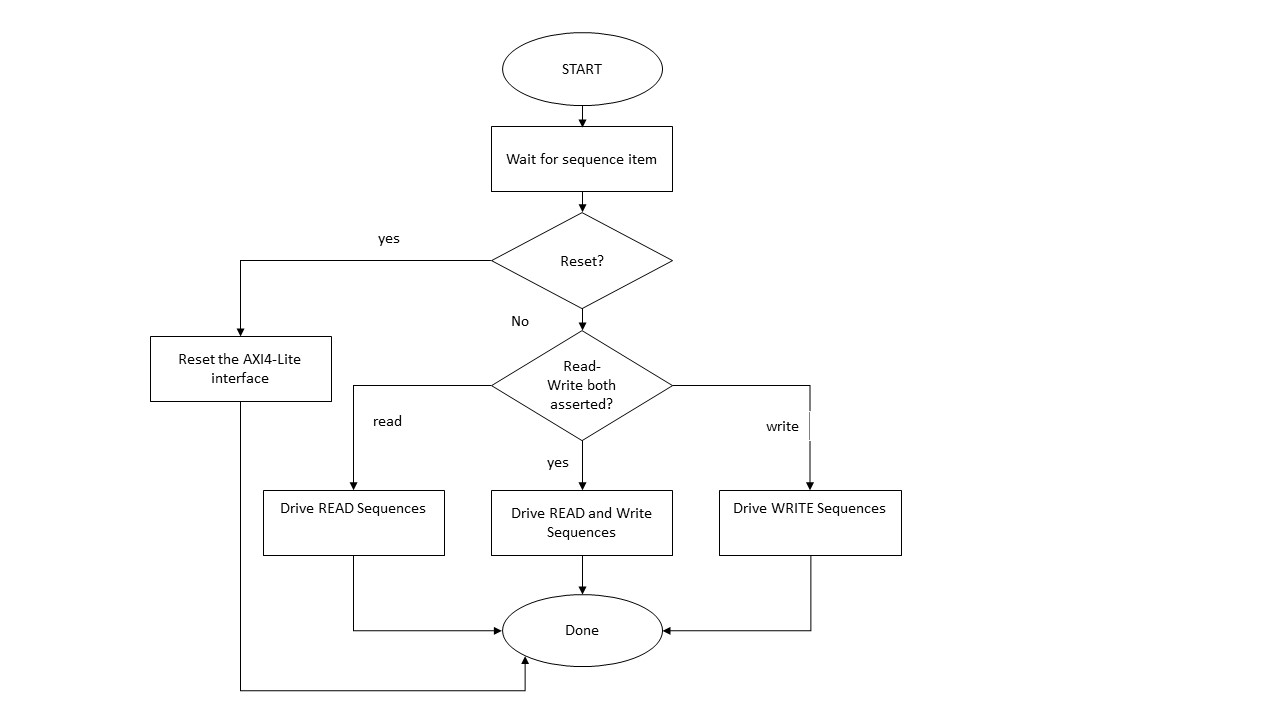
### **Fields:**

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| axi4lite\_m\_intf | Virtual interface that provides access to AXI4-Lite signals. |
| item | Sequence item (axi4lite\_m\_seq\_item) that holds transaction data. |

### **Methods:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| new | Constructor method that initializes the driver. |
| build\_phase | Retrieves the **virtual interface** from the UVM configuration database. |
| reset\_intf | Drives reset signals and initializes all AXI4-Lite interface signals. |

Flow chart



## 5.3 axi4lite\_m\_seq\_item

The axi4lite\_m\_seq\_item class is a UVM sequence item that represents transactions on the AXI4-Lite bus. It includes all the fields required for write and read transactions, making it the data structure used in sequences to generate stimulus for the DUT.

### **Fields:**

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| clk | Clock signal. |
| rst\_n | Active-low reset signal. |
| **Write Address Channel:** |  |
| dma\_s\_awaddr | Write address (32-bit). |
| dma\_s\_awprot | Write protection type (3-bit). |
| dma\_s\_awvalid | Write address valid signal. |
| dma\_s\_awready | Write address ready signal (from DUT). |
| **Write Data Channel:** |  |
| dma\_s\_wdata | Write data (32-bit). |
| dma\_s\_wstrb | Write strobe (4-bit). |
| dma\_s\_wvalid | Write data valid signal. |
| dma\_s\_wready | Write data ready signal (from DUT). |
| dma\_s\_wlast | Indicates the last write data beat. |
| **Write Response Channel:** |  |
| dma\_s\_bresp | Write response (2-bit). |
| dma\_s\_bvalid | Write response valid signal (from DUT). |
| dma\_s\_bready | Write response ready signal. |
| **Read Address Channel:** |  |
| dma\_s\_araddr | Read address (32-bit). |
| dma\_s\_arprot | Read protection type (3-bit). |
| dma\_s\_arvalid | Read address valid signal. |
| dma\_s\_arready | Read address ready signal (from DUT). |
| **Read Data Channel:** |  |
| dma\_s\_rdata | Read data (32-bit). |
| dma\_s\_rresp | Read response (2-bit). |
| dma\_s\_rvalid | Read data valid signal (from DUT). |
| dma\_s\_rready | Read data ready signal. |
| dma\_s\_rlast | Indicates the last read data beat. |
| **Control Signals:** |  |
| READ | Indicates a read transaction. |
| WRITE | Indicates a write transaction. |

### **Methods:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| new | Constructor that initializes the sequence item. |
| uvm\_object\_utils\_begin | Registers the fields for automation (copying, printing, comparing). |

### **axi4lite\_m\_seq\_lib**

The axi4lite\_m\_seq\_lib file defines AXI4-Lite sequences that generate different types of transactions for the AXI4-Lite driver. These sequences extend uvm\_sequence and define stimulus such as reset, read, and write transactions.

### **Fields:**

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| addr | Address for the AXI-Lite transaction (32-bit). |
| data | Data to be written or read (32-bit). |
| strb | Write strobes for write operations (4-bit). |
| prot | Protection bits (AWPROT for write and ARPROT for read, 3-bit). |
| READ | Read transaction control flag. |
| WRITE | Write transaction control flag. |

### **Methods:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| new | Constructor for initializing the sequence. |
| body | Defines the behavior of the sequence. |

## 5.4 Axi4lite\_master\_if Interface

#### **Description:**

The axi4lite\_master\_if interface defines the AXI4-Lite master interface for communication between the testbench and the DUT (Device Under Test). It consists of clock/reset signals, DMA IRQs, and AXI4-Lite protocol signals for read/write transactions. It provides a modport definition for easy connection to the AXI4-Lite master.

### **Fields:**

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| **Clock and Reset Signals** |  |
| clk | Clock signal. |
| rst\_n | Active-low reset signal. |
| **DMA IRQs** |  |
| dma\_done\_o | DMA transfer complete interrupt. |
| dma\_error\_o | DMA error interrupt. |
| **Write Address Channel (AW)** |  |
| dma\_s\_awaddr | Write address (32-bit). |
| dma\_s\_awprot | Protection type (3-bit). |
| dma\_s\_awvalid | Write address valid. |
| dma\_s\_awready | Write address ready (from DUT). |
| **Write Data Channel (W)** |  |
| dma\_s\_wdata | Write data (32-bit). |
| dma\_s\_wstrb | Write strobe (4-bit). |
| dma\_s\_wvalid | Write data valid. |
| dma\_s\_wready | Write data ready (from DUT). |
| dma\_s\_wlast | Indicates the last write data beat. |
| **Write Response Channel (B)** |  |
| dma\_s\_bready | Write response ready. |
| dma\_s\_bresp | Write response (2-bit). |
| dma\_s\_bvalid | Write response valid. |
| **Read Address Channel (AR)** |  |
| dma\_s\_araddr | Read address (32-bit). |
| dma\_s\_arprot | Read protection type (3-bit). |
| dma\_s\_arvalid | Read address valid. |
| dma\_s\_arready | Read address ready (from DUT). |
| **Read Data Channel (R)** |  |
| dma\_s\_rdata | Read data (32-bit). |
| dma\_s\_rresp | Read response (2-bit). |
| dma\_s\_rvalid | Read data valid. |
| dma\_s\_rready | Read data ready. |
| dma\_s\_rlast | Indicates the last read data beat. |

## 5.5 AXI4-Lite UVM Monitor

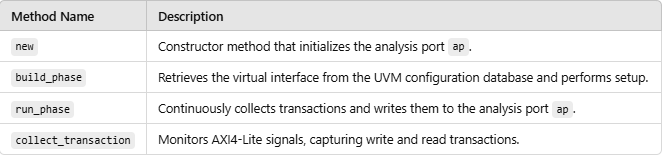
axi4lite\_m\_monitor is a UVM component designed to monitor transactions on an AXI4-Lite master interface. It captures both read and write operations, translates them into axi4lite\_m\_seq\_item transactions, and sends them through an analysis port for further processing (e.g., by a scoreboard or coverage collector).

**Component Type**: Passive  
**Purpose**: Monitors AXI4-Lite transactions (read/write), captures per-phase signal handshakes, logs the data, and broadcasts transactions through a UVM analysis port.

## **Responsibilities**

* Monitor all 5 channels of AXI4-Lite:
  + **Write**: AW (Address), W (Data), B (Response)
  + **Read**: AR (Address), R (Data)
* Capture each phase only during a valid handshake (valid && ready)
* Log all phases into a structured human-readable log file
* Send completed transaction to connected components via uvm\_analysis\_port

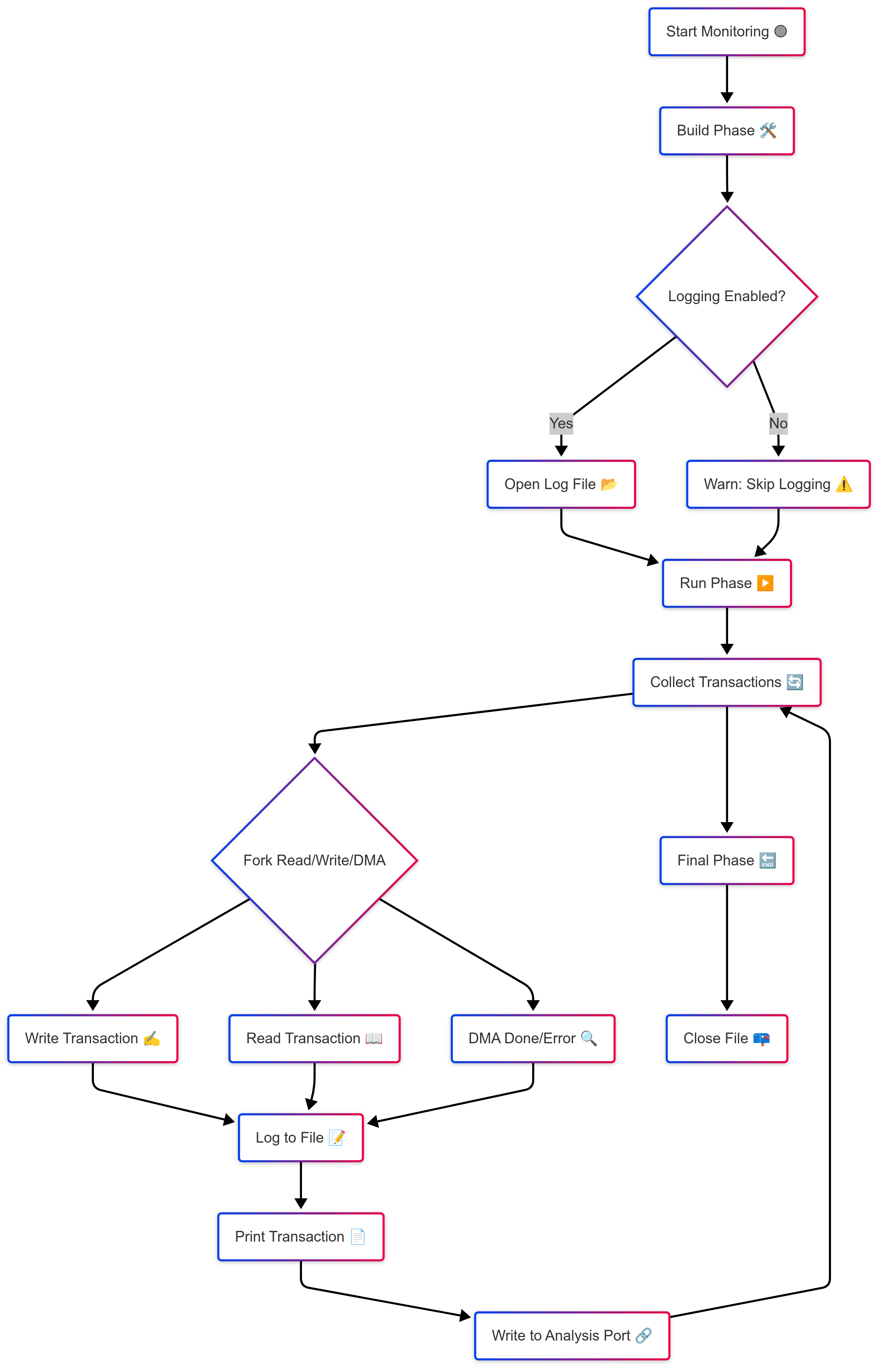
#### **Methods:**



## **Monitored Signals**

|  |  |  |
| --- | --- | --- |
| **Signal** | **Source** | **Description** |
| dma\_s\_awvalid | AXI4-Lite | Write address valid |
| dma\_s\_wvalid | AXI4-Lite | Write data valid |
| dma\_s\_bvalid | AXI4-Lite | Write response valid |
| dma\_s\_arvalid | AXI4-Lite | Read address valid |
| dma\_s\_rvalid | AXI4-Lite | Read data valid |
| dma\_done\_o | DUT | High when DMA operation completes |
| dma\_error\_o | DUT | High when DMA error occurs |

|  |  |  |
| --- | --- | --- |
| **Field Name** | **Type** | **Description** |
| ap | uvm\_analysis\_port #(axi4lite\_m\_seq\_item) | Analysis port used to broadcast captured transaction objects to connected components. |
| vif | virtual axi4lite\_if | Virtual interface handle used to access AXI4-Lite signals for monitoring. |
| item | axi4lite\_m\_seq\_item | Sequence item used to capture and hold transaction data before broadcasting. |
| txn | axi4lite\_m\_seq\_item | Captured transaction item |
|  |  |  |
| **Method Name** | **Type** | **Description** |
| new | Constructor | Initializes the monitor instance with a given name and parent component. |
| build\_phase | UVM Phase Method | Retrieves the AXI4-Lite virtual interface (vif) using uvm\_config\_db. |
| run\_phase | UVM Phase Method | Spawns two parallel threads using fork...join to independently monitor read and write transactions. |
| monitor\_write | Task | Captures write transactions: performs handshakes on AW, W, and B channels, logs, and sends to mon\_ap. |
| monitor\_read | Task | Captures read transactions: performs handshakes on AR and R channels, logs, and sends to mon\_ap. |



## **Integration in Testbench**

systemverilogCopyEdit// In build\_phase of env or agent  
uvm\_config\_db#(virtual axi4lite\_master\_if)::set(this, "monitor", "AXI4LITE\_MASTER\_INTF", vif);

**axi4\_slave\_uvc**

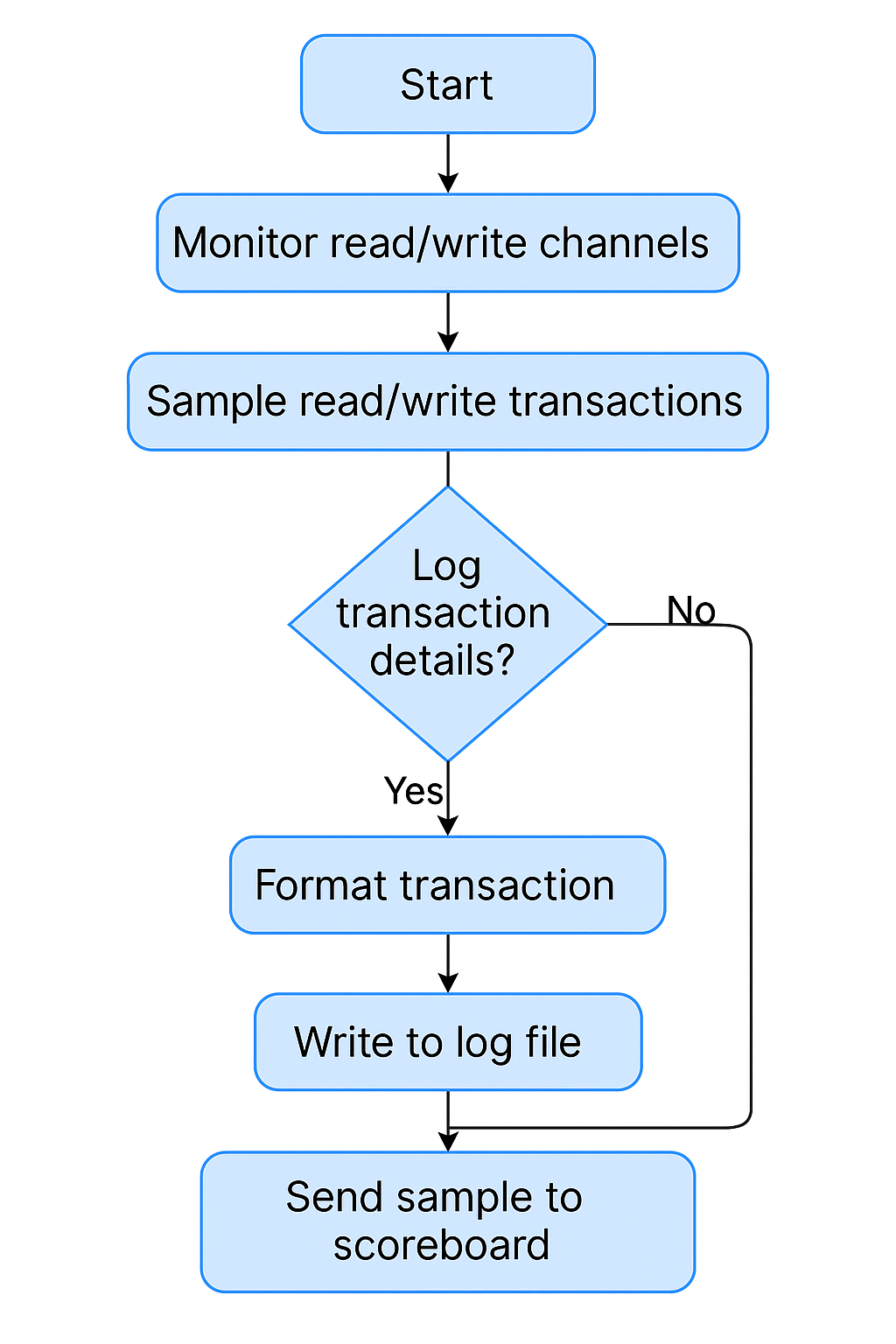
## 5.6 axi4\_s\_monitor.sv

axi4\_s\_monitor is a passive UVM component designed to **observe and log AXI4 transactions** (both read and write) occurring on a slave interface. It captures the full details of each burst transaction and logs them into a file, while also broadcasting the structured transaction object through an analysis port.

### **Responsibilities:**

* Passively monitor **AXI4 slave interface** for read/write transactions.
* Accurately detect and record **burst-based** transfers, including size, length, and address.
* Track timing (start/end) of each transaction for performance metrics.
* Log transaction summaries to a human-readable **log file**.
* Broadcast completed transactions using uvm\_analysis\_port.

**Flow chart**



5.7 axi4\_s\_agent.sv :

The axi4\_s\_agent class extends uvm\_agent and represents a UVM agent designed specifically for AXI4 slave verification environments. It includes components like a driver, monitor, sequencer, and coverage collector based on a configuration. It helps coordinate stimulus generation, monitoring, and coverage collection within the AXI4 slave verification environment.

* **axi4\_s\_agnt\_cfg** (axi4\_s\_agent\_config): Configuration object containing settings for the agent.
* **axi4\_s\_drvr** (axi4\_s\_driver): Driver component to drive AXI4 transactions to the DUT.
* **axi4\_s\_mntr** (axi4\_s\_monitor): Monitor component to observe transactions and interface signals.
* **axi4\_s\_cov** (axi4\_s\_coverage): Coverage collector component for collecting functional coverage data.
* **sqncr** (uvm\_sequencer #(axi4\_s\_seq\_item)): Sequencer component responsible for generating stimulus items.
* **axi4\_s\_agnt\_port** (uvm\_analysis\_port #(axi4\_s\_seq\_item)): Analysis port for broadcasting observed transactions from monitor to subscribers.

#### **Methods:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| New | Constructor method that initializes the AXI4-Lite agent. |
| build\_phase | - Retrieves the agent configuration from the UVM configuration database.  - Creates instances of the driver, sequencer, monitor, and coverage components (if enabled).  - Initializes the agent’s analysis port. |
| connect\_phase | - Connects the driver’s sequence item port to the sequencer.  - Connects the monitor’s analysis port to the agent's analysis port.  - If coverage is enabled, connects the monitor’s analysis port to the coverage component. |

## 5.8 Axi4\_s\_seq\_item

The axi4\_s\_seq\_item class extends uvm\_sequence\_item and represents individual AXI4 transactions (both read and write) in a UVM verification environment. It encapsulates all transaction-specific signals required for AXI4 read/write operations. Each signal field is randomized to generate various test scenarios and can be controlled through constraints in sequences.

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| rst\_n | Active low reset signal |
| dma\_m\_awaddr | Write transaction address |
| dma\_m\_awid | Write transaction ID |
| dma\_m\_awlen | Write burst length |
| dma\_m\_awsize | Write burst size |
| dma\_m\_awburst | Write burst type |
| dma\_m\_awlock | Write lock indicator |
| dma\_m\_awcache | Write cache attributes |
| dma\_m\_awprot | Write protection attributes |
| dma\_m\_awqos | Write quality of service |
| dma\_m\_awregion | Write region |
| dma\_m\_awuser | User-defined signals for write address |
| dma\_m\_awvalid | Indicates validity of the write address |
| dma\_m\_awready | Indicates slave ready to accept write address |
| dma\_m\_wdata | Data to be written |
| dma\_m\_wstrb | Write strobes (byte enable signals) |
| dma\_m\_wlast | Indicates last data of write burst |
| dma\_m\_wvalid | Indicates validity of write data |
| dma\_m\_wready | Indicates slave ready to accept write data |
| dma\_m\_wuser | User-defined signals for write data |
| dma\_m\_bready | Indicates master ready to accept write response |
| dma\_m\_bid | Transaction ID for write response |
| dma\_m\_bresp | Write response (ACK/ERROR) |
| dma\_m\_buser | User-defined signals for write response |
| dma\_m\_bvalid | Indicates validity of write response |
| dma\_m\_araddr | Read transaction address |
| dma\_m\_arid | Read transaction ID |
| dma\_m\_arlen | Read burst length |
| dma\_m\_arsize | Read burst size |
| dma\_m\_arburst | Read burst type |
| dma\_m\_arlock | Read lock indicator |
| dma\_m\_arcache | Read cache attributes |
| dma\_m\_arprot | Read protection attributes |
| dma\_m\_arqos | Read quality of service |
| dma\_m\_arregion | Read region |
| dma\_m\_aruser | User-defined signals for read address |
| dma\_m\_arvalid | Indicates validity of the read address |
| dma\_m\_arready | Indicates slave ready to accept read address |
| dma\_m\_rdata | Data read from slave |
| dma\_m\_rid | Transaction ID for read data |
| dma\_m\_rresp | Read response (ACK/ERROR) |
| dma\_m\_rlast | Indicates last data of read burst |
| dma\_m\_rvalid | Indicates validity of read data |
| dma\_m\_rready | Indicates master ready to accept read data |
| dma\_m\_ruser | User-defined signals for read data |
| READ | User-defined flag for enabling read transactions |
| WRITE | User-defined flag for enabling write transactions |

### **Methods:**

|  |  |
| --- | --- |
| **Method Name** | **Description** |
| new | Constructor that initializes the sequence item. |
| uvm\_object\_utils\_begin | Registers the fields for automation (copying, printing, comparing). |

## 5.9 axi4\_s\_driver.sv

The axi4\_s\_driver class extends uvm\_driver #(axi4\_s\_seq\_item) and serves as a reactive driver for AXI4 slave transactions in a UVM-based verification environment. It interacts with a virtual interface (axi4\_slave\_if) to drive read and write signals, handle AXI protocol handshakes, manage data storage, and reactively respond to AXI transactions generated by the sequencer.

* **Component Type:** Reactive UVM Driver (Slave)
* **Purpose:**
  + Reactively handles AXI4 read and write transactions initiated by an AXI master.
  + Simulates a slave device by responding appropriately to AXI4 master requests.
  + Maintains internal memory arrays to emulate memory read/write operations.

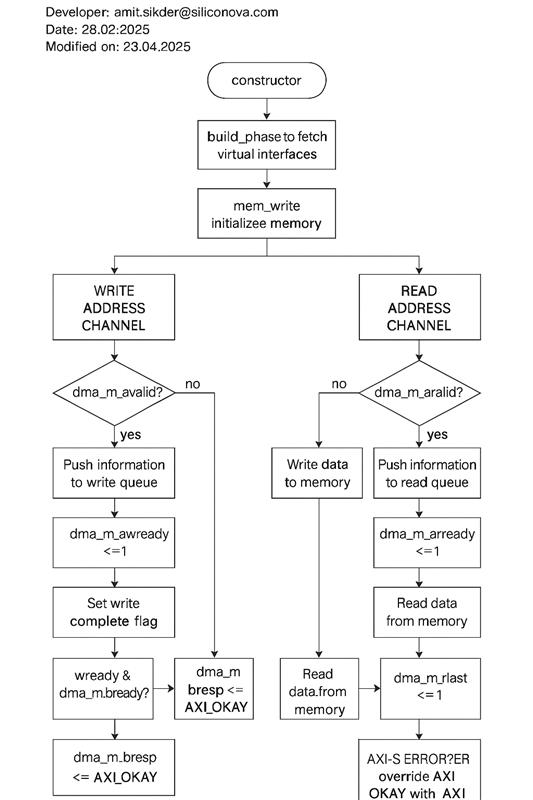
## **Responsibilities**

* **Reactive Behavior:**
  + Responds to AXI master-generated transactions without generating sequences internally.
  + Manages AXI handshake signals to synchronize communication effectively.
* **Memory Management:**
  + Uses internal memory arrays (mem, mem1) to simulate actual slave memory.
  + Supports memory initialization with random data for realistic simulations.
* **AXI Transaction Handling:**
  + Properly handles bursts (multiple data transfers per transaction).
  + Manages outstanding requests using address queues (rd\_queue, wr\_queue).

### **Field List and Descriptions:**

|  |  |
| --- | --- |
| **Field Name** | **Description** |
| axi4\_s\_intf | Virtual interface handle to AXI4 slave signals. |
| clk\_intf | Virtual interface handle for clock and reset signals. |
| item | AXI4 sequence item used for driving stimulus. |
| mem | Internal memory array (addresses 0 to 1023) for data storage. |
| mem1 | Internal memory array (addresses 1024 to 2045) for additional storage. |
| awready | Flag indicating readiness to accept AXI write address (AW channel). |
| wready | Flag indicating readiness to accept AXI write data (W channel). |
| arready | Flag indicating readiness to accept AXI read address (AR channel). |
| waddress | Holds the current write address received from the AXI bus. |
| wdata | Holds the current write data received from the AXI bus. |
| raddress | Holds the current read address received from the AXI bus. |
| rdata | Holds the current read data to be driven on the AXI bus. |
| wr\_burst\_length | Write burst length (number of data transfers in a write burst). |
| wr\_burst\_type | Write burst type (FIXED, INCR, WRAP). |
| wr\_burst\_size | Write burst size (number of bytes transferred per beat). |
| rd\_burst\_length | Read burst length (number of data transfers in a read burst). |
| rd\_burst\_type | Read burst type (FIXED, INCR, WRAP). |
| rd\_burst\_size | Read burst size (number of bytes transferred per beat). |

**Flow Chart**



## **Flowchart Explanation: AXI4-S Driver Behavior**

### **1. Initialization Phase**

* **Constructor**
  + Creates the axi4\_s\_driver or axi\_s\_error\_driver UVM object.
* **build\_phase**
  + Fetches the virtual interfaces (axi4\_slave\_if and clk\_rst\_interface) from the UVM config database.
* **mem\_write**
  + Initializes the internal memory (mem[0:10240]) with random data to simulate real memory behavior.

## **Error Conditions Simulated in axi\_s\_error\_driver**

|  |  |
| --- | --- |
| **Channel** | **Error Behavior** |
| **Write Response** | Always sends AXI\_SLVERR response after write |
| **Read Data** | Always sends AXI\_SLVERR response on all beats |

This class is useful to **simulate protocol violations**, test **resilience of scoreboard**, and **trigger error-handling mechanisms** in the DUT.

**Scoreboard**

## 5.10 scoreboard.sv

**Class**: dma\_scoreboard  
**Extends**: uvm\_scoreboard  
**Component Type**: *Passive*  
**Purpose**:  
Validates end-to-end AXI-DMA transfer correctness by matching read-back data, written data, byte strobe patterns, CSR configurations (e.g., num\_byte), and completion signals (dma\_done\_o, dma\_error\_o). Ensures that DMA sequences complete without error and data is moved accurately from source to destination memory.

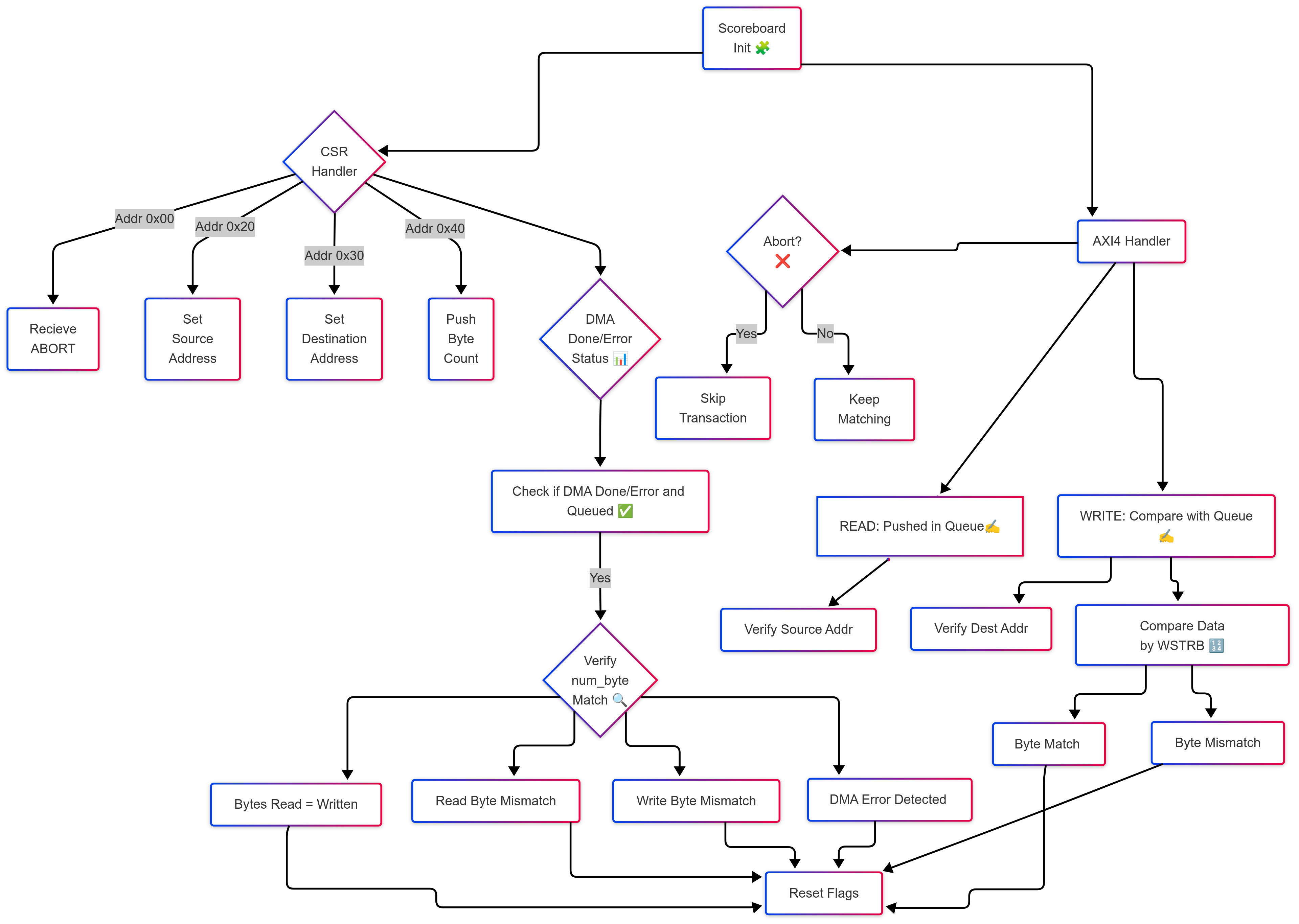
### **Responsibilities**

The dma\_scoreboard performs the following duties per DMA sequence:

1. **Captures CSR settings** including source\_address, destination\_address, and num\_byte from AXI4-Lite writes.
2. **Tracks unaligned AXI4-S data** using WSTRB and performs **byte-by-byte comparisons** between read and written data.
3. **Monitors total number of bytes** read and written and ensures they match with the configured num\_byte.
4. **Detects dma\_done\_o and dma\_error\_o flags** and uses them to validate transfer completion.
5. **Manages multiple DMA sequences** using queues for num\_byte, dma\_done, and dma\_error.
6. **Handles aborts** via CSR control (bit-1 of offset 0x00).
7. **Prints match/mismatch logs** at each byte and transaction level with detailed debug visibility.

* data read from the AXI memory mapped region to be matched later with writes.

### **Flow chart**



### **Matching Logic**

* **Address Matching**
  + Performed once at the start of each sequence using flags source\_addr\_checked and dest\_addr\_checked.
* **Data Comparison**
  + Byte-by-byte comparison using WSTRB.
  + Ignores bytes where WSTRB[i] == 0.
  + Compares data using expected\_byte = expected >> (i \* 8).
* **Sequence Matching**
  + Each time a dma\_done\_o == 1 is received, a full match is triggered:
    - Compares total\_bytes\_read and total\_bytes\_written against num\_byte\_q.pop\_front().
    - Errors flagged if mismatch.
    - Handles success/failure reporting using UVM\_INFO/UVM\_ERROR/UVM\_WARNING.
* **Multiple Sequence Support**
  + Uses separate queues (num\_byte\_q, dma\_done\_q, dma\_error\_q) to track overlapping sequences.
  + Sequences are *matched in FIFO order* as they complete.

### **Abort Handling**

* Controlled by bit[1] at address 0x00.
* If abort is asserted:
  + No more read/write transactions are validated.
  + Scoreboard skips validation silently with a warning log.

### **Logging Format**

|  |  |  |
| --- | --- | --- |
| **Event** | **Logger** | **Message Format Example** |
| CSR Setup | "CSR" | Set Source=0x4, Destination=0xfa4 |
| Read Transaction | "axi4-scb" | READ -> Addr: 0x4, Data: 0x12345678 |
| Write Transaction | "axi4-scb" | WRITE -> Addr: 0xfa4, Data: 0x12345678, WSTRB: 1111 |
| Byte Match | "axi4-scb" | Byte[2] Data Matched! Expected: 0x34, Got: 0x34 |
| Byte Mismatch | "axi4-scb" | Byte[2] Mismatch! Expected: 0x34, Got: 0x12 |
| Transfer Match | "axi4-scb" | DMA DONE - Transfer matched for num\_byte = 36 |
| Read/Write Mismatch | "axi4-scb" | Total bytes WRITTEN (32) != Expected (36) |
| DMA Error Flag | "axi4-scb" | DMA completed with ERROR |
| Abort Triggered | "CSR" | ABORT received. Transfer verification will stop. |

# Chapter 6

## Test

In UVM (Universal Verification Methodology), a **uvm\_test** is the top-level component used to define and control the verification scenario for a DUT (Design Under Test). It inherits from the uvm\_test base class, which itself is a specialization of uvm\_component.

A **uvm\_test** is responsible for:

* Creating and configuring the UVM environment (env).
* Instantiating sequences or test-specific stimulus.
* Controlling simulation phases like build\_phase, connect\_phase, run\_phase, etc.
* Optionally overriding components or configurations to run different scenarios without modifying the environment

### 6.1 Base Test

The base\_test class is extended from uvm\_test class. As it is a component it has been registered to the factory accordingly. The handle of environment and environment\_config has been taken to control the configuration of lower hierarchical components. To print all the components that has been registered with the factory , a handle of in built uvm\_factory and uvm\_coreservice\_t has been taken. In the constructor function we passed the two arguments of component. Then in the build phase env\_config has been created and some default values of agent configuration have been passed such as by default the agent is active ,it has no coverage and scoreboard . Then the environment\_config was set so that this configuration can be accessed from wherever needed. The env has also been created here. In the end\_of\_elaboration phase the in built uvm\_factory variable named factory is assigned and then is printed . And finally in the run phase a reporting macro is added to see whether this sequence is running or not.

|  |  |  |
| --- | --- | --- |
| **Field** | **Type** | **Description** |
| env | environment | The top-level verification environment instance. |
| env\_cfg | environment\_config | Stores configuration settings for all sub-environments and agents. |
| clk\_enb | clock\_enable[2] | Enables clock(s) (array for multi-clock design). |
| clk\_dis | clock\_disable | Disables clock(s). |
| rst\_enb | reset\_enable | Triggers reset signal(s) at appropriate times. |
| factory | uvm\_factory | Access handle to the UVM factory for component control and printing. |

|  |  |  |
| --- | --- | --- |
| **Method** | **Type** | **Description** |
| new() | Constructor | Initializes the base\_test object. |
| build\_phase() | Function | Creates and configures the environment and passes agent configs via config DB. |
| axi4lite\_m\_build\_config() | Function | Custom helper to configure agent activity, coverage, and monitor logging. |
| end\_of\_elaboration\_phase() | Function | Prints test and factory information. |
| run\_phase() | Task | Prints a log that test is running. Typically where sequences are started. |
| report\_phase() | Function | Summarizes the test result (PASS/FAIL) based on UVM errors or fatals. |

### 6.2 Sanity Simple Test

The **Sanity Simple Test** is a minimal, entry-level test designed to verify that the core components of the testbench and DUT (Design Under Test) are correctly initialized and communicating. It acts as a **basic integration test** to ensure all agents, interfaces, clocks, resets, and sequences are functioning before more complex tests are executed.

### **Key Components**

|  |  |  |
| --- | --- | --- |
| **Component** | **Type** | **Description** |
| seq | sanity\_simple\_sequence |  |
| clk\_enb[2] | clock\_enable |  |
| clk\_dis | clock\_disable | (Declared but not used here) For disabling clocks. |
| rst\_enb | reset\_enable | Drives reset to DUT via reset agent. |

### **Test Flow (run\_phase)**

1. **Raise UVM Objection** to hold simulation.
2. **Enable clocks** on two domains using clock\_enable sequences.
3. **Trigger reset** using the reset\_enable sequence.
4. **Start the main sequence** sanity\_simple\_sequence on the AXI4-Lite master sequencer.
5. **Drop UVM Objection** after stimulus completes.

#### 6.2.1 Test Purpose

The purpose of the test is to send a simple reset and directed write and read sequence to verify if the dut is responding correctly according to the protocol.

**Sequence Flow Diagram**

axi4lite\_m\_reset\_seq  
 ↓  
axi4lite\_m\_write\_seq  
 ↓  
axi4lite\_m\_read\_seq

### 6.3 csr\_wr\_rd\_test

#### 6.3.1 Test Purpose

The csr\_wr\_rd\_test is a UVM test class designed to validate the **read and write operations** of DMA control and status registers (CSR) through the AXI4-Lite interface. It leverages the **axi4lite\_m\_csr\_wr\_rd\_seq** sequence to perform the actual read and write transactions.

#### 6.3.2 Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### 6.3.3 Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

#### 6.3.4 Addtional Argument

+UVM\_TESTNAME=csr\_wr\_rd\_test,

+UVM\_TIMEOUT=500000,

+UVM\_VERBOSITY=UVM\_LOW

### 6.4 dma\_error\_test

#### 6.4.1 Test purpose

To verify that a slave error (AXI\_SLVERR or AXI\_DECERR) is captured and reported by the DMA. A specific address range was defined within the AXI4 slave. According to the DUT specification, any access — read or write — to this region must be rejected with a SLVERR (2'b10) response on the appropriate AXI response channel (RRESP or BRESP).

#### 6.4.2 Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

#### 6.4.3 Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### 6.4.4 Addtional Argument

"run\_op : +UVM\_TESTNAME=csr\_custom\_test", "run\_op : +UVM\_TIMEOUT=500000", "run\_op : +UVM\_VERBOSITY=UVM\_LOW", "run\_op : +NUM\_DES=single", "run\_op : +SRC\_ADDR\_TYPE=AL", "run\_op : +DEST\_ADDR\_TYPE=UNAL", "run\_op : +BURST\_MODE=FI", "run\_op : +DATA\_NUM\_BYTES=0x24", "run\_op : +DATA\_DES\_CONFIG=0x4", "run\_op : +DATA\_CTRL=0x21", "run\_op : +uvm\_set\_type\_override=axi4\_s\_driver,axi\_s\_error\_driver" ]

## 6.5 multiple\_burst\_transfer\_test

6.5.1 Purpose of the multiple\_burst\_transfer\_test UVM Test

The multiple\_burst\_transfer\_test is a UVM test class designed to validate **multiple burst transfer operations** via the AXI4-Lite protocol. The primary goal of this test is to verify the **functionality and data integrity** during **burst transfers** between the source and destination addresses in a DMA controller.

6.5.2 Sequence Used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

6.5.3 Sequence Flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### 6.4.4 Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=AL",

+BURST\_MODE=II",

+DATA\_NUM\_BYTES=0x48",

+DATA\_DES\_CONFIG=0x4",

+DATA\_CTRL=0x21"

## 6.6 disable\_descriptor\_test

6.6.1 Purpose of the Test

The dma\_disabled\_descriptor\_test is a UVM test class designed to verify the behavior of the DMA controller when a descriptor is disabled. The primary objective of this test is to ensure that the DMA engine correctly handles disabled descriptors and does not initiate data transfers when the descriptor is marked as inactive.

6.6.3 Sequence Used  
The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

6.6.3 Sequence Flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### 6.4.4 Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=AL",

+DATA\_NUM\_BYTES=0x48",

+DATA\_DES\_CONFIG=0x0",

+DATA\_CTRL=0x21"

### 6.7 max\_burst\_configuration\_test

#### 6.7.1 Purpose of the test

The dma\_vary\_max\_burst\_test is designed to validate the DMA controller's ability to handle varying burst sizes during data transfers. It aims to ensure that the DMA controller correctly processes different maximum burst lengths without data loss or corruption. The test dynamically configures the burst length and initiates data transfers, verifying that the DMA correctly adapts to different burst settings. By varying the burst length, the test assesses the throughput and robustness of the DMA controller under diverse operating conditions. Additionally, it checks for unexpected errors or stalls when the burst size changes dynamically.

#### 6.7.2 Sequence Used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

6.7.3 Sequence Flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

#### 6.4.4 Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+BURST\_MODE=II",

+DATA\_NUM\_BYTES=0x400",

+DATA\_DES\_CONFIG=0x4",

+DATA\_CTRL=0x3FD"

### 6.8 Multi-descriptor configuration regression test

#### 6.8.1 Purpose of the test

In this test the number of descriptors will be more than one, we will check that the dma can handle write/read transaction properly when there are multiple descriptors.

The dma\_multi\_descriptor\_test aims to verify the correct handling and processing of multiple DMA descriptors. It ensures that the DMA controller can manage multiple descriptor sequences efficiently while maintaining proper synchronization between clock and reset signals. The test initiates clock enable sequences on two channels, triggers the reset enable sequence, and executes the multi-descriptor transfer using the environment's sequencer to validate seamless operation.

6.8.2 Sequence Used  
The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

#### 6.8.3 Sequence Flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

#### 6.8.4 Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+NUM\_DES=multi",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=AL",

+BURST\_MODE=II",

# descriptor-0 fields

+DATA\_NUM\_BYTES=0x48",

+DATA\_DES\_CONFIG=0x4",

+DATA\_CTRL=0x21",

+DATA\_SRC=0x0",

+DATA\_DEST=0x1400",

#descriptor-1 fields

+DATA\_NUM\_BYTES2=0x24",

+DATA\_DES\_CONFIG2=0x4",

# control reg is shared, but we still pass it

+DATA\_SRC2=0x04",

+DATA\_DEST2=0x1404"

### 6.9 Single burst transfer test (single descriptor, incremental)

6.8.1 Purpose of the test

In this test the single burst will be created, the CSR is configured in that way.

#### 6.9.2 Sequence Used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

#### 6.9.3 Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

#### 6.9.4 Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+BURST\_MODE=II",

+DATA\_NUM\_BYTES=0x44",

+DATA\_DES\_CONFIG=0x4",

+DATA\_CTRL=0x41"

### 6.10 Unaligned source transfer test

#### 6.10.1 Purpose of the test

In this test the CSR is configured in such a way that the source address is unalign and the destination address is allign.

### 6.10.2 Sequence used

The **axi4lite\_m\_unaligned\_src\_seq** sequence is used to test the DMA controller's ability to handle transfers where the **source address is unaligned**. This scenario is crucial because unaligned addresses can cause data transfer errors or unexpected behavior if not handled properly.

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

#### 6.10.3 Sequence Flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

#### 6.10.4 Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

"run\_op : +UVM\_TIMEOUT=500000",

"run\_op : +UVM\_VERBOSITY=UVM\_LOW",

"run\_op : +NUM\_DES=single",

"run\_op : +SRC\_ADDR\_TYPE=UNAL",

"run\_op : +DEST\_ADDR\_TYPE=AL",

"run\_op : +BURST\_MODE=II",

"run\_op : +DATA\_NUM\_BYTES=0x24",

"run\_op : +DATA\_DES\_CONFIG=0x4",

"run\_op : +DATA\_CTRL=0x21"

### 6.11 Unaligned destination transfer test

#### 6.11.1 Purpose of the test

In this test the CSR is configured in such a way that the source address will be aligned and the destination address will be unalligned.

6.11.2 Sequence used

The **axi4lite\_m\_unaligned\_destination\_seq** sequence is used to verify the DMA controller's ability to handle data transfers where the **destination address is unaligned**. This scenario is crucial for testing the DMA's robustness since unaligned addresses can cause data corruption or transfer errors if not handled properly.

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

#### 6.11.3 Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

#### 6.11.4 Addtional Argument

"run\_op : +UVM\_TESTNAME=csr\_custom\_test",

"run\_op : +UVM\_TIMEOUT=500000",

"run\_op : +UVM\_VERBOSITY=UVM\_LOW",

"run\_op : +NUM\_DES=single",

"run\_op : +SRC\_ADDR\_TYPE=AL",

"run\_op : +DEST\_ADDR\_TYPE=UNAL",

"run\_op : +BURST\_MODE=II",

"run\_op : +DATA\_NUM\_BYTES=0x24",

"run\_op : +DATA\_DES\_CONFIG=0x4",

"run\_op : +DATA\_CTRL=0x21"

### 6.12 Burst exceeds 4k boundary test (single descriptor, large length

#### 6.12.1 Purpose of the test

In this test , we tried to send the maximum number of data for transfer.

Sequence used

The **dma\_burst\_exceeds\_4k\_boundary\_test** sequence is used to verify the DMA controller's behavior when a burst transfer crosses a **4KB address boundary**. Crossing such boundaries can potentially cause errors or unexpected behavior, as some hardware components may not handle such scenarios efficiently.

Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### 6.9.4 Addtional Argument

"run\_op : +UVM\_TESTNAME=csr\_custom\_test",

"run\_op : +UVM\_TIMEOUT=500000",

"run\_op : +UVM\_VERBOSITY=UVM\_LOW",

"run\_op : +NUM\_DES=single",

"run\_op : +BURST\_MODE=II",

"run\_op : +DATA\_SRC=0xFFC"

"run\_op : +DATA\_DEST=0x1FF8"

"run\_op : +DATA\_NUM\_BYTES=0x14", # 4K

"run\_op : +DATA\_DES\_CONFIG=0x4",

"run\_op : +DATA\_CTRL=0x11"

## 6.13 4k bytes transfer test (single descriptor)

Test purpose

To configure the DMA for a single 4KB transfer, set the **Number of Bytes Register** to 0x1000 (4096 bytes). Ensure the **DMA Control Register** is configured to enable the transfer with the appropriate burst size. Additionally, configure the **Descriptor Configuration Register** to enable the descriptor and use **INCR mode** for continuous data flow. This setup allows the DMA to move the entire 4KB block in one go.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

testcase["4kb\_bytes\_transfer\_test\_single\_descriptor\_II"] = [ "run\_op : +UVM\_TESTNAME=csr\_custom\_test",

"run\_op : +UVM\_TIMEOUT=500000",

"run\_op : +UVM\_VERBOSITY=UVM\_LOW",

"run\_op : +NUM\_DES=single",

"run\_op : +BURST\_MODE=II",

+DATA\_SRC=0x0",

+DATA\_DEST=0x1400",

+DATA\_NUM\_BYTES=0x1000",

+DATA\_DES\_CONFIG=0x4",

+DATA\_CTRL=0x3FD" ]

## 6.14 4k bytes transfer test (single descriptor)

Test purpose

To configure the registers such that 4k bytes will get transferred at one go

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=multi",

+BURST\_MODE=II",

+DATA\_SRC=0x0",

+DATA\_DEST=0x1400",

+DATA\_CTRL=0x3FD",

+DATA\_NUM\_BYTES=0x800",

+DATA\_DES\_CONFIG=0x4",

+DATA\_SRC2=0x04",

+DATA\_DEST2=0x1404",

+DATA\_NUM\_BYTES2=0x800" # 4K

## 6.15 2 Burst: Read Fixed,Write Increment

Test purpose

This test checks whether the DMA can successfully read multiple beats from a fixed source address and write them to incrementing destination addresses, while using a single aligned descriptor and transferring 36 bytes. It validates the core data path and AXI burst handling logic for an FI configuration.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=AL",

+BURST\_MODE=FI",

+DATA\_NUM\_BYTES=0x24",

+DATA\_CTRL=0x21"

## 6.16 Basic functionality test (single descriptor, aligned)

Test purpose

This test verifies the basic functionality of the DMA engine using a single, aligned descriptor. It ensures that a fixed burst read (from a constant source address) and an incrementing burst write (to sequential destination addresses) operate correctly. The test transfers 36 bytes (0x24) of data and uses control signals to enable and start the DMA. This setup emulates typical scenarios like reading from a FIFO and writing to memory. It validates correct burst handling, address alignment, and data integrity for FI mode.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=AL",

+BURST\_MODE=FI",

+DATA\_NUM\_BYTES=0x24",

+DATA\_CTRL=0x21"

## 6.17 Multiple-burst transfer (multi descriptor, incremental burst)

Test purpose

Set max\_burst and num\_bytes such that multiple burst are generated and Transaction should be successful for each burst

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=AL",

+BURST\_MODE=FI",

+DATA\_NUM\_BYTES=0x48",

+DATA\_CTRL=0x21"

## 6.18 Disable descriptor test (single descriptor, control=0 to disable)

Test purpose

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

"run\_op : +UVM\_TESTNAME=csr\_custom\_test",

"run\_op : +UVM\_TIMEOUT=500000",

"run\_op : +UVM\_VERBOSITY=UVM\_LOW",

"run\_op : +NUM\_DES=single",

"run\_op : +SRC\_ADDR\_TYPE=AL",

"run\_op : +DEST\_ADDR\_TYPE=AL",

"run\_op : +BURST\_MODE=FI",

"run\_op : +DATA\_NUM\_BYTES=0x48",

"run\_op : +DATA\_DES\_CONFIG=0x2",

"run\_op : +D

## 6.19 Max burst configuration test (single descriptor, fixed burst)

## Test purpose

To verify DMA behavior with maximum max\_burst value that is 255.

To verify the DMA behavior with a maximum max\_burst value of 255, configure the DMA Control Register to set max\_burst to 255. This will instruct the DMA to perform the transfer in the largest burst size possible. Ensure the Descriptor Configuration Register is set for burst transfers. Then, test the system by initiating a transfer and confirming that the DMA operates correctly within the 255 burst limit.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+BURST\_MODE=FI",

+DATA\_NUM\_BYTES=0x14",

: +DATA\_SRC=0x04",

+DATA\_DEST=0x1400",

+DATA\_CTRL=0x11"

6.20 Multi-descriptor configuration regression test

### Test purpose

This test verifies the DMA engine's ability to handle multiple descriptors in a single run, each with aligned source and destination addresses. It uses Fixed burst mode for both read and write (FF) to simulate scenarios like communicating with FIFO-like peripherals. Two descriptors are configured to transfer 36 bytes (0x24) each, with distinct source and destination addresses. The test ensures that the DMA can correctly process back-to-back descriptors, handle separate memory regions, and maintain data integrity and proper sequencing during multi-descriptor transfers.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+NUM\_DES=multi",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=AL",

+BURST\_MODE=FF",

+DATA\_NUM\_BYTES=0x24",

+DATA\_CTRL=0x21",

+DATA\_SRC=0x0",

"r +DATA\_DEST=0x1400",

+DATA\_NUM\_BYTES2=0x24",

+DATA\_SRC2=0x04",

+DATA\_DEST2=0x1404"

6.21 Single burst transfer test (single descriptor, incremental

### Test purpose

Verify a simple transfer with 64bytes and max\_burst = 16.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+BURST\_MODE=FF",

+DATA\_NUM\_BYTES=0x40",

+DATA\_CTRL=0x3D"

6.22 Unaligned source transfer test

### Test purpose

To Verify that an unaligned source address is handled correctly by aligning the address and with proper WSTRB masking.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+SRC\_ADDR\_TYPE=UNAL",

+DEST\_ADDR\_TYPE=AL",

+BURST\_MODE=FF",

+DATA\_NUM\_BYTES=0x24",

+DATA\_CTRL=0x21"

6.23 Unaligned destination transfer test

### Test purpose

To Verify that an unaligned destination address is handled correctly by aligning the address and with proper WSTRB masking.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+SRC\_ADDR\_TYPE=AL",

+DEST\_ADDR\_TYPE=UNAL",

+BURST\_MODE=FF",

+DATA\_NUM\_BYTES=0x24",

+DATA\_CTRL=0x21"

6.25 Burst exceeds 4k boundary test (single descriptor, large length)

### Test purpose

To Verify that a transfer crossing a 4KB boundary is properly split into bursts.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000", "

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single", "

+SRC\_ADDR\_TYPE=AL",

+BURST\_MODE=FF",

+DATA\_SRC=0xFFC",

+DATA\_DEST=0x1FF8",

+DATA\_NUM\_BYTES=0x14", # 4K

+DATA\_CTRL=0x11"

6.26 Single burst transfer test (single descriptor, incremental)

### Test purpose

Verify a simple transfer with 64bytes and max\_burst = 16.

### Sequence used

The main sequence used in this test is axi4lite\_m\_csr\_wr\_rd\_custom\_seq , derived from `axi4lite\_m\_base\_seq. It includes handles for axi4lite\_m\_reset\_seq reset\_seq; (as ` reset\_seq `) and ` axi4lite\_m\_write\_seq write\_seq ` (as ` write\_seq ` for write operations and ` axi4lite\_m\_read\_seq read\_seq; for read operations). This sequence first performs a reset using ` reset\_seq `, followed by write operations with ` write\_seq ` and read operations with ` read\_seq `. The write and read transactions are designed to test the different feature of DMA . The different configarations are generated based on plusargs ( +UVM\_TESTNAME, +UVM\_VERBOSITY, +NUM\_DES, +BURST\_MODE, +DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+BURST\_MODE=II",

+DATA\_NUM\_BYTES=0x1E4",

+DATA\_CTRL=0x1E1"

+DATA\_SRC, +DATA\_DEST, +DATA\_NUM\_BYTES, +DATA\_DES\_CONFIG, +DATA\_CTRL) provided at runtime.

### Sequence flow

1. axi4lite\_m\_csr\_wr\_rd\_custom\_seq

2. axi4lite\_m\_write\_seq write\_seq;

3.axi4lite\_m\_read\_seq read\_seq;

4.axi4lite\_m\_reset\_seq reset\_seq;

### Addtional Argument

+UVM\_TESTNAME=csr\_custom\_test",

+UVM\_TIMEOUT=500000",

+UVM\_VERBOSITY=UVM\_LOW",

+NUM\_DES=single",

+DATA\_SRC=0x4",

+DATA\_DEST=0x1401",

+BURST\_MODE=II",  
+DATA\_NUM\_BYTES=0x24",  
+DATA\_CTRL=0x21"

# Chapter 7

## Coverage

Coverage refers to the measurement of how much of the design and verification space has been exercised during simulation.It's a key metric to assess the completeness and effectiveness of your verification efforts.

## 7.1 Purpose of Coverage

* Coverage measures how well the testbench stimulates the design, not just whether the DUT behaves correctly.
* It helps answer:  
   Did our tests exercise all meaningful scenarios and corner cases?
* Expose dead or unused logic  
   Helps identify unreachable code or incorrect connections in RTL
* Guide test development  
   Uncovered bins or expressions point to missing tests

## 7.2 Coverage plan

* **Design Under Test**: AXI DMA controller
* **Interfaces Verified**:
  + **AXI4-Lite** (control/config)
  + **AXI4-Full** (data movement)

**Coverage Strategy**:

* + Functional coverage via SystemVerilog

covergroups

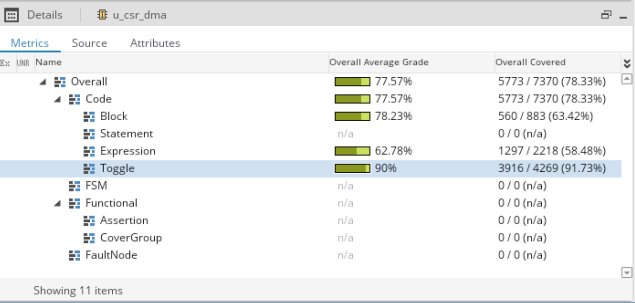
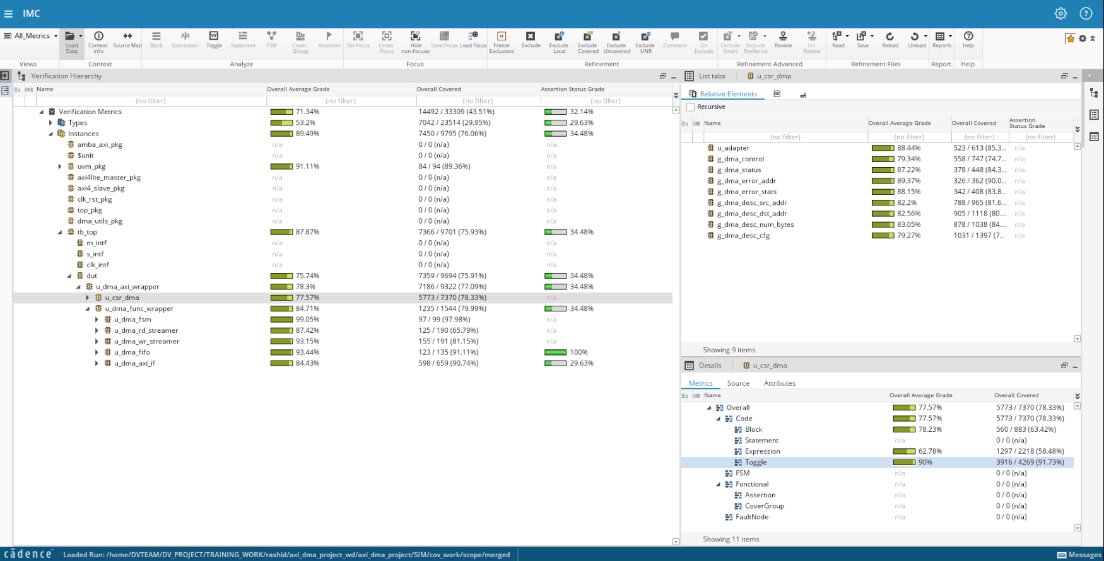
* + Separate covergroups for each interface

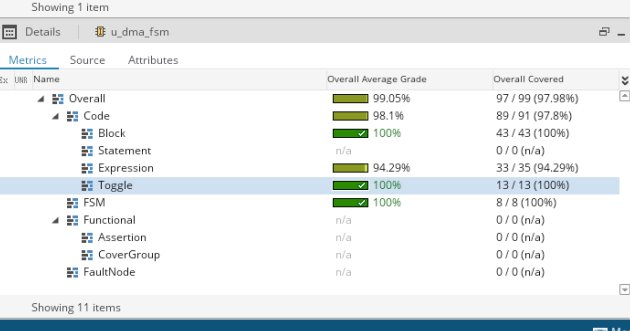
type

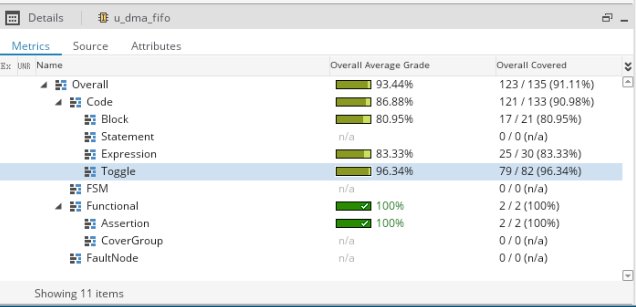
* + Code coverage via IMC tool

|  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- |
| Coverpoint type | Interface | Channel | Coverpoint name | Values to cover | Description |
| Non-cross |  |  | rst\_n\_cp | 1’b0, 1’b1 |  |
| Non-cross |  |  | dma\_done\_o\_cp | 1’b0, 1’b1 |  |
| Non-cross |  |  | dma\_error\_o\_cp | 1’b0, 1’b1 |  |
| Non-cross | AXI4-Lite |  | dma\_s\_awaddr\_cp | Valid addr: individual bins rw registers  For illegals we need only one bin | Covers all the CSR addresses during write operation |
| Non-cross | AXI4-Lite |  | dma\_s\_awprot\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_awvalid\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_awready\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_wdata\_cp |  | Covers all the CSR configuration values during write operation |
| Non-cross | AXI4-Lite |  | dma\_s\_wstrb\_cp | Valid bins 1000    1100    1110    1111    0001    0011    0111 |  |
| Non-cross | AXI4-Lite |  | dma\_s\_wvalid\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_wready\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_bready\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_bresp\_cp | Separate bins for each rsp value |  |
| Non-cross | AXI4-Lite |  | dma\_s\_bvalid\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_araddr\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_arprot\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_arvalid\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_arready\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_rdata\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_rresp\_cp | Separate bins for each rsp value |  |
| Non-cross | AXI4-Lite |  | dma\_s\_rvalid\_cp |  |  |
| Non-cross | AXI4-Lite |  | dma\_s\_rready\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awaddr\_cp |  | Covers all valid addresses of the slave memory within 4KB boundary during slave write operation |
| Non-cross | AXI4 |  | dma\_m\_awid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awlen\_cp | Autobins for all values |  |
| Non-cross | AXI4 |  | dma\_m\_awsize\_cp | Only 3’b010 is valid |  |
| Non-cross | AXI4 |  | dma\_m\_awburst\_cp | WRAP, RESERVED in ignore bins |  |
| Non-cross | AXI4 |  | dma\_m\_awlock\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awcache\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awprot\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awqos\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awregion\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awuser\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awvalid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_awready\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_wdata\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_wstrb\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_wlast\_cp | Checked in scoreboard |  |
| Non-cross | AXI4 |  | dma\_m\_wvalid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_wready\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_wuser\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_bready\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_bid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_bresp\_cp | Separate bins for each response |  |
| Non-cross | AXI4 |  | dma\_m\_buser\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_bvalid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_araddr\_cp | Covering start & end address with separate bins | Covers all valid addresses of the slave memory within 4KB boundary during slave write operation |
| Non-cross | AXI4 |  | dma\_m\_arid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arlen\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arsize\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arburst\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arlock\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arcache\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arprot\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arqos\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arregion\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_aruser\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arvalid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_arready\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_rdata\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_rid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_rresp\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_rlast\_cp | Checked in Scoreboard |  |
| Non-cross | AXI4 |  | dma\_m\_rvalid\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_rready\_cp |  |  |
| Non-cross | AXI4 |  | dma\_m\_ruser\_cp |  |  |
| Cross |  |  | rst\_araddr\_rdata\_cross |  | Covers CSR default values during reset |
| Cross |  |  | aw\_handshake\_cross |  |  |
| Cross |  |  | w\_handshake\_cross |  |  |
| Cross |  |  | b\_handshake\_cross |  |  |
| Cross |  |  | ar\_handshake\_cross |  |  |
| Cross |  |  | r\_handshake\_cross |  |  |
| Cross |  |  | aw\_handshake\_cross |  |  |
| Cross |  |  | w\_handshake\_cross |  |  |
| Cross |  |  | dma\_done\_cross |  | Covers DMA AXI4 master interface write response channel handshake crossed with DMA done status |
| Cross |  |  | dma\_error\_cross |  | Covers DMA AXI4 master interface write response channel handshake crossed with DMA error status |
| Cross |  |  | ar\_handshake\_cross |  |  |
| Cross |  |  | r\_handshake\_cross |  |  |
| Cross |  |  | aw\_burst\_size\_cross |  |  |
| Cross |  |  | ar\_burst\_size\_cross |  |  |
|  |  |  |  |  |  |

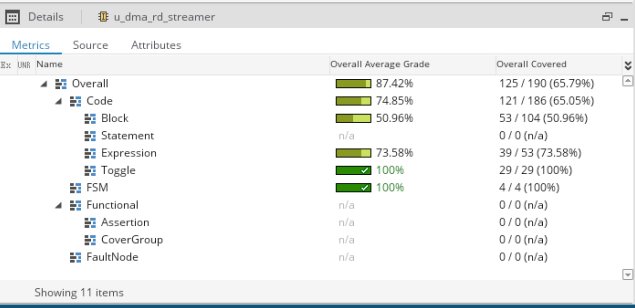
## 7.3 Code coverage – CSR DMA

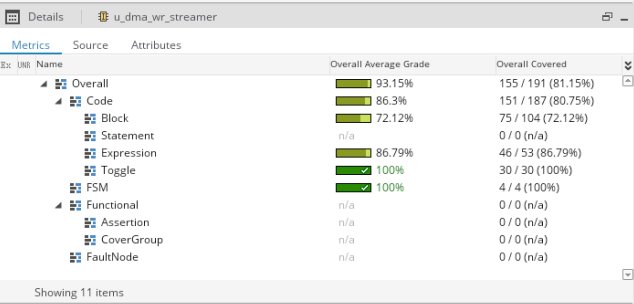
7.4 Code coverage – DMA FSM

7.5 Code coverage – DMA FIFO

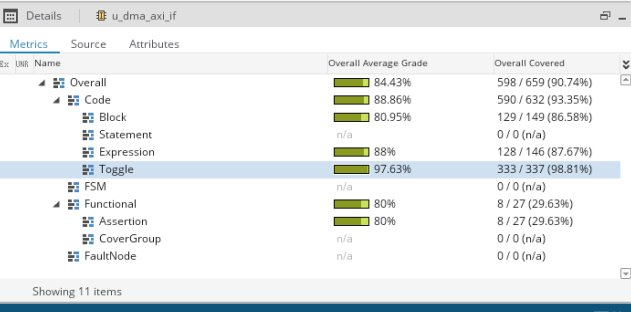


## 7.6 Code coverage – DMA Read Streamer

7.7 Code coverage – DMA Write Streamer



## 7.8 Code coverage – DMA Master I/F



7.9 AXI DMA – Functional

**AXI4-Lite Covergroup**

**Covergroup Name**: dma\_cg\_lite

**Targeted Features**:

* + CSR accesses (read/write to control,

descriptor regs)

* + Start/abort commands and config

parameters

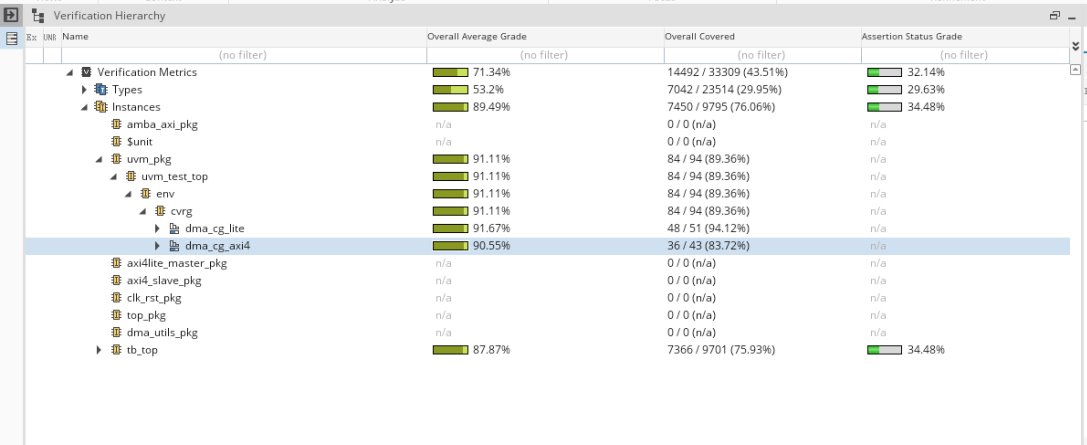
* + Address/data range bins (source, destination,

burst sizes)

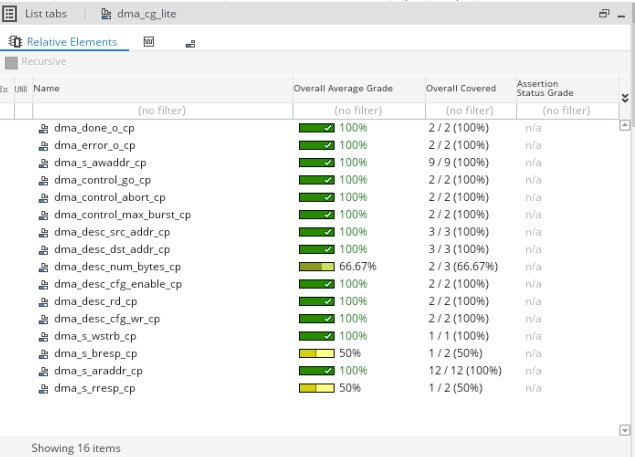
* + Protocol correctness: WSTRB, BRESP,

RRESP values

## 7.10 Covergroups



7.11 dma\_cg\_lite Covergroup coverpoints



**AXI4-Full Covergroup**

**Covergroup Name**: dma\_cg\_axi4

**Targeted Features**:

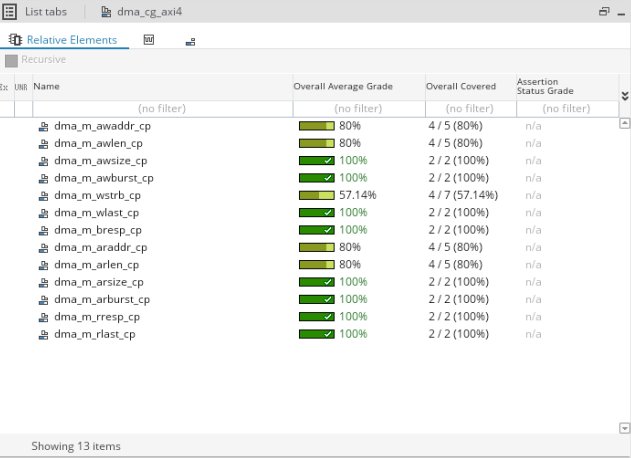
* + AXI4 transaction fields (AWADDR,

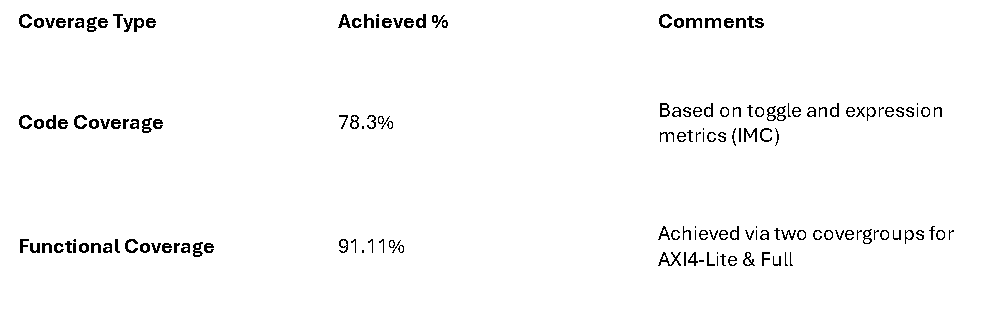
ARADDR, AWLEN, AWSIZE, etc.)

* + Burst types (FIXED, INCR)
  + Transfer sizes and lengths
  + Write/read strobes, last signals, and

Responses

dma\_cg\_axi4 Covergroup coverpoints

**SUMMARY**

Since we did not develop the RTL ourselves and focused solely on achieving toggle coverage, it is important to note that other types of code coverage (such as expression, statement, and block coverage) were deliberately excluded.

Achieving 100% toggle coverage can be challenging when the RTL engineers are not involved, as they possess in-depth knowledge of the design's internal logic and potential corner cases. Consequently, the lack of RTL development from our end limited our ability to fully exercise all possible toggle scenarios, resulting in a toggle coverage percentage that did not reach 100%.

If the RTL engineers had been available, we could have optimized the stimulus and verification environment to target the uncovered toggles, thereby improving the overall code coverage significantly.