



# **DesignWare ARC EM Databook**

---

## Copyright Notice and Proprietary Information Notice

© 2014 Synopsys, Inc. All rights reserved. This software and documentation contain confidential and proprietary information that is the property of Synopsys, Inc. The software and documentation are furnished under a license agreement and may be used or copied only in accordance with the terms of the license agreement. No part of the software and documentation may be reproduced, transmitted, or translated, in any form or by any means, electronic, mechanical, manual, optical, or otherwise, without prior written permission of Synopsys, Inc., or as expressly provided by the license agreement.

### Destination Control Statement

All technical data contained in this publication is subject to the export control laws of the United States of America. Disclosure to nationals of other countries contrary to United States law is prohibited. It is the reader's responsibility to determine the applicable regulations and to comply with them.

### Disclaimer

SYNOPSYS, INC., AND ITS LICENSORS MAKE NO WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, WITH REGARD TO THIS MATERIAL, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE.

### Trademarks

Synopsys and certain Synopsys product names are trademarks of Synopsys, as set forth at

<http://www.synopsys.com/Company/Pages/Trademarks.aspx>.

All other product or company names may be trademarks of their respective owners.

Synopsys, Inc.  
700 E. Middlefield Road  
Mountain View, CA 94043

[www.synopsys.com](http://www.synopsys.com)

# Contents

|                                                    |    |
|----------------------------------------------------|----|
| Chapter 1                                          |    |
| Product Overview                                   | 37 |
| 1.1 Introduction to DesignWare ARC Processor Cores | 37 |
| 1.1.1 Processors in the ARC EM Series              | 37 |
| 1.2 Design Goals                                   | 40 |
| 1.3 Architecture Highlights                        | 40 |
| 1.4 ARCV2 Instruction-Set Architecture             | 42 |
| 1.5 Gate Count                                     | 42 |
| 1.6 Customization                                  | 42 |
| 1.7 Fast Configurability                           | 42 |
| 1.8 Floating Point Extensions                      | 42 |
| Chapter 2                                          |    |
| Release Notes                                      | 45 |
| 2.1 About This Release                             | 45 |
| 2.1.1 Deliverables                                 | 45 |
| 2.1.2 Documents                                    | 45 |
| 2.2 Supported Platforms                            | 46 |
| 2.3 Supported Standards                            | 47 |
| 2.4 Release Change Information                     | 48 |
| 2.4.1 New in Release 2.0                           | 48 |
| 2.5 Release History                                | 48 |
| 2.5.1 Changes in Release 1.1a                      | 48 |
| 2.5.2 Changes in Release 1.1                       | 49 |
| 2.5.3 Changes in Release 1.0a                      | 49 |
| 2.5.4 Changes in Release 1.0                       | 49 |
| 2.6 Known Issues                                   | 49 |
| Chapter 3                                          |    |
| Getting Started                                    | 51 |
| 3.1 Introduction                                   | 51 |
| 3.2 ARChitect Configuration Tool                   | 51 |
| 3.2.1 Creating a Project                           | 52 |
| 3.2.2 Selecting a Configuration Template           | 53 |
| 3.2.3 Adding and Configuring Components            | 53 |
| 3.2.4 Choosing Design Flows                        | 53 |
| 3.2.5 ARChitect Command Line                       | 54 |
| 3.2.6 Building and Debugging with a TCF File       | 54 |
| 3.2.7 Simulation Tool Support                      | 56 |

|                                                                   |           |
|-------------------------------------------------------------------|-----------|
| 3.3 Compiler and Debugger Support .....                           | 56        |
| 3.3.1 Compiling C/C++ Code for the DesignWare ARC EM Series ..... | 56        |
| 3.3.2 Assembling for the DesignWare ARC EM Series .....           | 57        |
| 3.3.3 MetaWare-Debugger Support .....                             | 57        |
| 3.4 Licensing Your Software .....                                 | 57        |
| 3.5 Where to Go from Here .....                                   | 58        |
| <b>Chapter 4</b>                                                  |           |
| <b>Configurability .....</b>                                      | <b>59</b> |
| 4.1 Overview .....                                                | 59        |
| 4.2 Configuring an ARC EM Processor .....                         | 60        |
| 4.2.1 Adding an Optional Component .....                          | 60        |
| 4.2.2 Specifying Optional Features of Components .....            | 60        |
| 4.3 Options for Build Components .....                            | 60        |
| 4.3.1 Instance Features .....                                     | 60        |
| 4.3.2 Implementation Options .....                                | 60        |
| 4.3.3 Reset Options .....                                         | 61        |
| 4.3.4 Interrupt Options .....                                     | 61        |
| 4.3.5 Bus and External Memory .....                               | 61        |
| 4.3.6 Register File .....                                         | 62        |
| 4.3.7 Instruction Set .....                                       | 63        |
| 4.3.8 Floating-Point Extensions .....                             | 64        |
| 4.3.9 Closely Coupled Memories .....                              | 65        |
| 4.3.10 Instruction Cache .....                                    | 65        |
| 4.3.11 Instruction Fetch Queue .....                              | 66        |
| 4.3.12 Data Cache .....                                           | 66        |
| 4.3.13 Data Memory Initiator .....                                | 66        |
| 4.3.14 JTAG Interface .....                                       | 66        |
| 4.3.15 Debug Interface .....                                      | 68        |
| 4.3.16 Timers .....                                               | 68        |
| 4.3.17 Code Protection .....                                      | 68        |
| 4.3.18 Stack Checking .....                                       | 69        |
| 4.3.19 Memory Protection Unit .....                               | 69        |
| 4.4 Optional Components for Debug-Assist .....                    | 69        |
| 4.4.1 Actionpoints (Advanced Breakpoint Assist) .....             | 69        |
| 4.4.2 SmaRT (Small Real-Time Trace Buffer) .....                  | 70        |
| 4.4.3 Advanced Real-Time Trace System .....                       | 70        |
| 4.5 Adding Instructions and More with APEX .....                  | 71        |
| 4.6 Multiprocessor Support .....                                  | 71        |
| 4.7 Other Configuration Features .....                            | 72        |
| <b>Chapter 5</b>                                                  |           |
| <b>Memory Components .....</b>                                    | <b>73</b> |
| 5.1 Memory Map .....                                              | 73        |
| 5.1.1 Instruction and Data Closely Coupled Memories .....         | 75        |
| 5.1.2 Instruction-Fetch Queue .....                               | 81        |
| 5.1.3 Data Memory Initiator .....                                 | 82        |
| 5.1.4 Peripheral Data Bus .....                                   | 82        |
| 5.1.5 Non-Cacheable Memory Locations .....                        | 83        |

|                                                         |     |
|---------------------------------------------------------|-----|
| 5.1.6 Memory-Map Constraints .....                      | 83  |
| 5.2 Caches .....                                        | 84  |
| 5.2.1 Data Cache .....                                  | 84  |
| 5.2.2 Instruction Cache .....                           | 103 |
| 5.2.3 Invalidate Instruction Cache, IC_IVIC .....       | 104 |
| 5.2.4 Instruction Cache Control Register, IC_CTRL ..... | 104 |
| 5.3 Debugger View of Memory .....                       | 115 |
| Chapter 6                                               |     |
| Memory-Protection Unit .....                            | 117 |
| 6.1 Stack Checking .....                                | 117 |
| 6.1.1 Configuring Stack Checking .....                  | 117 |
| 6.1.2 Build Configuration Register .....                | 117 |
| 6.1.3 Enabling Stack Checking .....                     | 117 |
| 6.1.4 Specifying Stack Regions .....                    | 118 |
| 6.1.5 Stack-Checking Operation .....                    | 118 |
| 6.2 Code Protection .....                               | 121 |
| 6.2.1 Configuring Code Protection .....                 | 121 |
| 6.2.2 Code Protection Scheme .....                      | 121 |
| 6.2.3 Code Protection Behavior .....                    | 121 |
| 6.3 Memory-Protection Unit .....                        | 123 |
| 6.3.1 Configuring an MPU .....                          | 123 |
| 6.3.2 Enabling the MPU .....                            | 123 |
| 6.3.3 Overview of MPU Operation .....                   | 123 |
| Chapter 7                                               |     |
| Interrupts .....                                        | 125 |
| 7.1 Interrupt Architecture .....                        | 126 |
| 7.1.1 Interrupt-Unit Features .....                     | 126 |
| 7.1.2 Interrupt Unit Configuration .....                | 126 |
| 7.1.3 Enabling Interrupts .....                         | 127 |
| 7.1.4 Interrupt Priorities .....                        | 127 |
| 7.1.5 Pulse-Triggered Interrupts .....                  | 127 |
| 7.1.6 Bank Configuration and Second Register Set .....  | 128 |
| 7.1.7 Reducing Context-Switching Overhead .....         | 128 |
| 7.1.8 Determining Configured Interrupts .....           | 129 |
| Chapter 8                                               |     |
| Cycle-Timing Characteristics .....                      | 131 |
| 8.1 ARC EM Pipeline Overview .....                      | 131 |
| 8.1.1 Instruction Sequencing .....                      | 132 |
| 8.1.2 Pipeline Hazards .....                            | 132 |
| 8.2 Instruction Timing .....                            | 133 |
| 8.2.1 Instruction-Fetch Timing .....                    | 134 |
| 8.2.2 ALU Instruction Timing .....                      | 135 |
| 8.2.3 Load and Store instruction Timing .....           | 136 |
| 8.2.4 Branch and Jump Instruction Timing .....          | 137 |
| 8.2.5 Two-Cycle Branches .....                          | 138 |
| 8.2.6 Indexed Execution Timing .....                    | 138 |
| 8.2.7 Multiplier Instruction Timing .....               | 139 |

|                                                               |     |
|---------------------------------------------------------------|-----|
| 8.2.8 Divide/Remainder Instruction Timing .....               | 141 |
| 8.2.9 Multi-Operation Instruction Timing .....                | 142 |
| 8.2.10 Extension Instruction Timing .....                     | 142 |
| 8.2.11 Serializing Auxiliary-Register Writes .....            | 143 |
| 8.2.12 Serializing Control Instructions .....                 | 143 |
| 8.2.13 Miscellaneous Instruction Delays .....                 | 144 |
| 8.3 Exception Timing .....                                    | 144 |
| 8.4 Interrupt Response Times (Interrupt Latency) .....        | 144 |
| 8.5 Cache Miss Penalties .....                                | 145 |
| 8.5.1 Instruction Cache Miss Penalties .....                  | 146 |
| 8.6 Multi-cycle Floating Point Instructions .....             | 146 |
| Chapter 9                                                     |     |
| External Interfaces .....                                     | 147 |
| 9.1 Bus Interfaces .....                                      | 147 |
| 9.1.1 Memory Interface Types .....                            | 148 |
| 9.1.2 Instruction-Bus Interface Unit .....                    | 149 |
| 9.1.3 Data-Bus Interface Unit .....                           | 151 |
| 9.1.4 Combined Instruction- and Data-Bus Interface Unit ..... | 153 |
| 9.1.5 Peripheral-Bus Interface Unit .....                     | 153 |
| 9.1.6 Bus Command Attributes .....                            | 155 |
| 9.1.7 Direct Memory Interface Unit .....                      | 155 |
| 9.2 Interrupts Interface .....                                | 158 |
| 9.3 Test and Debug Interface .....                            | 158 |
| 9.4 Halt and Run Controls .....                               | 159 |
| 9.5 Actionpoint Signals .....                                 | 161 |
| 9.6 Real Time Trace Interface .....                           | 161 |
| 9.6.1 Real-Time Trace Interface Signals .....                 | 162 |
| 9.7 Clock and Reset .....                                     | 165 |
| 9.8 Code Protect .....                                        | 165 |
| Chapter 10                                                    |     |
| Clock, Reset, and Run Controls .....                          | 167 |
| 10.1 Signal Interface .....                                   | 167 |
| 10.2 Clock-Signal Requirements .....                          | 167 |
| 10.3 ICCM1 Clock Details .....                                | 168 |
| 10.4 Reset Signal Requirements .....                          | 168 |
| 10.5 External Halt and Run Control Signals .....              | 168 |
| 10.6 Debug Reset and Watchdog Reset .....                     | 169 |
| 10.7 Reset Configuration .....                                | 169 |
| 10.8 Run Status Signals .....                                 | 170 |
| Chapter 11                                                    |     |
| ARC EM DSP Features .....                                     | 171 |
| 11.1 Fractional Data Types .....                              | 171 |
| 11.2 Rounding .....                                           | 171 |
| 11.2.1 Rounding Up .....                                      | 172 |
| 11.2.2 Convergent Rounding .....                              | 172 |
| 11.3 Saturation .....                                         | 172 |
| 11.4 Accumulator Operations .....                             | 172 |

|                                                             |     |
|-------------------------------------------------------------|-----|
| 11.5 Vector Operations .....                                | 173 |
| 11.6 Complex Operations .....                               | 173 |
| 11.7 Butterfly Acceleration .....                           | 173 |
| 11.8 ARC EM DSP Pipeline Behavior .....                     | 173 |
| 11.9 Instruction Timing .....                               | 175 |
| 11.10 ARC EM DSP Configurations .....                       | 175 |
| Chapter 12                                                  |     |
| Extending the Processor .....                               | 177 |
| Chapter 13                                                  |     |
| Multi-Core Option .....                                     | 179 |
| 13.1 Multi-Core Integrated with ARConnect IP .....          | 180 |
| 13.2 Introduction of ARConnect Components .....             | 182 |
| 13.2.1 Inter-Core Interrupt Unit .....                      | 182 |
| 13.2.2 Inter-Core Semaphore Unit .....                      | 183 |
| 13.2.3 Inter-Core Message Unit .....                        | 184 |
| 13.2.4 Multi-Core Debug Unit .....                          | 185 |
| 13.2.5 Global Real-Time Counter (GRTC) .....                | 187 |
| 13.2.6 Power Management Unit .....                          | 187 |
| 13.3 Programming Model of ARConnect .....                   | 196 |
| 13.3.1 Auxiliary Registers Supporting ARConnect .....       | 196 |
| 13.3.2 ARConnect Command List .....                         | 203 |
| 13.3.3 ARConnect Command Usage Examples .....               | 207 |
| 13.3.4 Multi-Core Debug Examples .....                      | 212 |
| 13.3.5 Power Management Related Examples .....              | 213 |
| 13.3.6 ARConnect Configurability .....                      | 216 |
| Chapter 14                                                  |     |
| Debug Features .....                                        | 219 |
| 14.1 Host Access via JTAG .....                             | 219 |
| 14.2 Four-Wire JTAG Interface (IEEE 1149.1) .....           | 220 |
| 14.2.1 JTAG 1149.1 Port .....                               | 220 |
| 14.2.2 JTAG 1149.1 Programmer's Model .....                 | 220 |
| 14.2.3 The JTAG 1149.1 TAP Controller .....                 | 224 |
| 14.2.4 The JTAG 1149.1 TAP Controller State Machine .....   | 225 |
| 14.3 Two-Wire JTAG Interface (IEEE 1149.7) .....            | 227 |
| 14.3.1 Compact JTAG 1149.7 Core Features .....              | 227 |
| 14.3.2 Adding the Compact JTAG 1149.7 Core .....            | 228 |
| 14.3.3 Compact JTAG 1149.7 Synthesis Constraints .....      | 228 |
| 14.3.4 Compact JTAG 1149.7 Core System-Level Overview ..... | 229 |
| 14.3.5 JTAG 1149.7 System-Level Block Diagrams .....        | 229 |
| 14.3.6 JTAG 1149.7 Chip-Level Block Diagrams .....          | 230 |
| 14.3.7 JTAG 1149.7 Signals .....                            | 232 |
| 14.3.8 JTAG 1149.7 Functional Description: Overview .....   | 234 |
| 14.3.9 JTAG 1149.7 Operating Modes .....                    | 235 |
| 14.3.10 JTAG 1149.7 Initialization .....                    | 235 |
| 14.3.11 JTAG 1149.7 Startup Options .....                   | 236 |
| 14.3.12 JTAG 1149.7 Reset Types .....                       | 236 |
| 14.3.13 JTAG 1149.7 Bypass Mode .....                       | 237 |

|                      |                                                              |     |
|----------------------|--------------------------------------------------------------|-----|
| 14.3.14              | JTAG 1149.7 Selection/Deselection                            | 237 |
| 14.3.15              | JTAG 1149.7 Control Levels                                   | 238 |
| 14.3.16              | JTAG 1149.7 Command Processing                               | 239 |
| 14.3.17              | TAP.7 Registers and Programming Model                        | 240 |
| 14.3.18              | JTAG 1149.7 Controller ID Allocation and Deallocation        | 242 |
| 14.3.19              | JTAG 1149.7 Scan Formats                                     | 242 |
| 14.4                 | The Debug Port                                               | 243 |
| 14.5                 | The Host Interface to BVCI Target                            | 243 |
| 14.6                 | Setting Up Read/Write Transactions                           | 244 |
| 14.6.1               | Setting up a Write Access to the ARC EM Processor or Memory  | 244 |
| 14.6.2               | Accessing the Status Register                                | 245 |
| 14.6.3               | Setting up a Read Access from the ARC EM Processor or Memory | 245 |
| 14.7                 | Actionpoints                                                 | 246 |
| 14.7.1               | Actionpoints Mechanism                                       | 247 |
| 14.7.2               | Overview of Actionpoint Auxiliary Registers                  | 249 |
| 14.7.3               | Programming Actionpoints                                     | 252 |
| 14.7.4               | Paired Actionpoints                                          | 256 |
| 14.7.5               | Independent-Mode Actionpoints                                | 258 |
| 14.7.6               | Quad Actionpoints                                            | 259 |
| 14.7.7               | Processor-Halt Actionpoints                                  | 260 |
| 14.7.8               | Software-Exception Actionpoints                              | 260 |
| 14.7.9               | Actionpoints on External Parameters                          | 261 |
| 14.8                 | SmaRT                                                        | 262 |
| 14.8.1               | Features                                                     | 262 |
| 14.8.2               | Overview of SmaRT Operation                                  | 263 |
| 14.8.3               | Overview of SmaRT Auxiliary Registers                        | 263 |
| 14.8.4               | Trace-Stack Operation                                        | 267 |
| 14.9                 | Real Time Trace Module Interface                             | 281 |
| 14.9.1               | Overview                                                     | 281 |
| 14.9.2               | RTT Interface Registers                                      | 281 |
| 14.9.3               | Programming and Timing                                       | 282 |
| Chapter 15           |                                                              |     |
| Performance Counters |                                                              | 285 |
| 15.1                 | Introduction                                                 | 285 |
| 15.2                 | Components                                                   | 286 |
| 15.2.1               | Hardware Components                                          | 286 |
| 15.2.2               | Software Components                                          | 287 |
| 15.3                 | Usage Example                                                | 288 |
| 15.4                 | Countable Conditions                                         | 288 |
| 15.4.1               | Condition Filter                                             | 289 |
| 15.5                 | Condition Registers                                          | 289 |
| 15.6                 | Available Conditions                                         | 289 |
| 15.6.1               | Logical Conditions                                           | 290 |
| 15.6.2               | ISA Conditions                                               | 290 |
| 15.6.3               | Microarchitecture-Specific Conditions                        | 292 |
| 15.7                 | Performance Counters                                         | 296 |
| 15.7.1               | Adding Counters                                              | 296 |
| 15.7.2               | Characteristics of Counters                                  | 296 |



|                                                                  |     |
|------------------------------------------------------------------|-----|
| 15.7.3 Performance Counter Registers .....                       | 297 |
| Chapter 16                                                       |     |
| UAUX Interface .....                                             | 299 |
| 16.1 Overview .....                                              | 299 |
| 16.2 Block Diagram .....                                         | 300 |
| 16.3 Interface Signals .....                                     | 301 |
| 16.3.1 Violations .....                                          | 302 |
| 16.3.2 Serialization .....                                       | 302 |
| 16.4 Transactions .....                                          | 302 |
| 16.4.1 Basic Transaction .....                                   | 303 |
| 16.4.2 Wait Transaction .....                                    | 304 |
| 16.4.3 Transaction Aborted .....                                 | 304 |
| Chapter 17                                                       |     |
| Power-Saving Features .....                                      | 305 |
| 17.1 Overview .....                                              | 305 |
| 17.2 Power-Saving Modes .....                                    | 306 |
| 17.2.1 Sleep State .....                                         | 306 |
| 17.2.2 Halt State .....                                          | 307 |
| 17.2.3 Modules Always on in Sleep or Halted States .....         | 308 |
| 17.2.4 External Interfaces Alive in Sleep or Halted States ..... | 308 |
| 17.3 Architectural Clock Gating .....                            | 308 |
| 17.3.1 Configuring Architectural Clock Gating .....              | 308 |
| 17.3.2 Actionpoints .....                                        | 309 |
| 17.3.3 Performance Counters and Hardware Profiler .....          | 309 |
| 17.4 Reset Control .....                                         | 309 |
| 17.5 Unit Level Clock Gating .....                               | 309 |
| Chapter 18                                                       |     |
| Integration and Verification .....                               | 311 |
| 18.1 IP Directory Structure .....                                | 311 |
| 18.1.1 Verilog Files .....                                       | 311 |
| 18.1.2 Documents .....                                           | 312 |
| 18.1.3 Build Logs .....                                          | 312 |
| 18.1.4 Argument Files .....                                      | 312 |
| 18.2 Support Libraries .....                                     | 312 |
| 18.2.1 ARC Memory Subsystem (MSSv2) .....                        | 312 |
| 18.2.2 Confidence Tests: CCT Library .....                       | 332 |
| 18.2.3 Other Libraries .....                                     | 332 |
| 18.3 Build-Sequence Options .....                                | 333 |
| 18.4 Building the Design .....                                   | 334 |
| 18.5 Verification Assistance .....                               | 335 |
| 18.5.1 ARCTest .....                                             | 335 |
| 18.5.2 Waveform Generation .....                                 | 336 |
| 18.5.3 Pipeline Monitor .....                                    | 337 |
| 18.5.4 MetaWare Debugger .....                                   | 337 |
| Appendix A                                                       |     |
| Converting from ARC 600 .....                                    | 339 |

|                                                                       |     |
|-----------------------------------------------------------------------|-----|
| A.1 Hardware Considerations .....                                     | 339 |
| A.2 Software Considerations .....                                     | 340 |
| Appendix B                                                            |     |
| Key IP Library Command-Line Options .....                             | 343 |
| B.1 Key Options .....                                                 | 343 |
| B.1.1 HTML Documentation and Software .....                           | 343 |
| B.1.2 Scripts and HDL .....                                           | 343 |
| B.1.3 Test Code .....                                                 | 344 |
| B.1.4 IP-XACT .....                                                   | 344 |
| B.1.5 Verilog and Simulator .....                                     | 344 |
| B.1.6 RASCAL and Trace .....                                          | 345 |
| B.1.7 ARCHitect Tools .....                                           | 345 |
| B.1.8 Processor .....                                                 | 346 |
| B.1.9 Synthesis .....                                                 | 346 |
| B.1.10 Endian Orientation and Counter Sizes .....                     | 346 |
| B.1.11 Interrupts and Reset .....                                     | 346 |
| B.1.12 Register File .....                                            | 347 |
| B.1.13 Memory .....                                                   | 348 |
| B.1.14 Closely Coupled Memory .....                                   | 348 |
| B.1.15 Instruction Optimizations .....                                | 349 |
| B.1.16 Build Optimization .....                                       | 350 |
| B.1.17 Busses .....                                                   | 350 |
| B.1.18 Clock Frequency .....                                          | 351 |
| B.1.19 JTAG Clock, Latency Unit, and HDL Library .....                | 351 |
| B.1.20 Actionpoints .....                                             | 351 |
| B.1.21 SmaRT Stack Size .....                                         | 351 |
| B.1.22 Interrupt-Timer Levels .....                                   | 352 |
| B.1.23 UART .....                                                     | 352 |
| B.1.24 Instruction Fetch Queue .....                                  | 352 |
| B.1.25 Instruction Cache and Instruction Closely Coupled Memory ..... | 353 |
| B.1.26 Data Cache .....                                               | 353 |
| B.1.27 Memory Protection Unit .....                                   | 354 |
| B.1.28 com.arc.hardware.EM.Floating_point_unit .....                  | 354 |
| B.1.29 com.arc.hardware.EM.Real_time_trace_producer .....             | 354 |
| B.1.30 com.arc.hardware.ARCv2MSS.BusFabric .....                      | 354 |
| B.1.31 com.arc.hardware.ARCv2MSS.ClkCtrl .....                        | 356 |
| B.1.32 com.arc.hardware.ARCv2MSS.Profiler .....                       | 356 |
| B.1.33 com.arc.hardware.ARCv2MSS.SRAMCtrl .....                       | 356 |
| B.1.34 com.arc.hardware.UAUX_Interface.1_0 .....                      | 357 |
| Chapter C                                                             |     |
| Synthesis and Physical Implementation .....                           | 359 |
| C.1 Register-File Implementation .....                                | 359 |
| C.2 Integration: Memory Insertion .....                               | 359 |
| C.3 System Integration .....                                          | 360 |
| C.4 DFT .....                                                         | 360 |
| C.5 Interfacing ARC EM Memory .....                                   | 360 |
| C.5.1 Memory Functions .....                                          | 360 |

|                                                                                         |     |
|-----------------------------------------------------------------------------------------|-----|
| C.5.2 Memory Implementation Specification Using ram_sizes.txt .....                     | 361 |
| C.5.3 Memory-Port Specifications .....                                                  | 362 |
| C.5.4 Memory-Wrapper Reference Implementation .....                                     | 366 |
| Appendix D                                                                              |     |
| ARC UART .....                                                                          | 375 |
| D.1 UART Features .....                                                                 | 375 |
| D.1.1 Block Diagram .....                                                               | 375 |
| D.1.2 System Block Diagram .....                                                        | 376 |
| D.1.3 Register Base Address .....                                                       | 377 |
| D.1.4 Control Registers .....                                                           | 377 |
| D.2 UART Functions .....                                                                | 382 |
| D.2.1 Interrupts .....                                                                  | 382 |
| D.2.2 Transmit Operation .....                                                          | 383 |
| D.2.3 Receive Operation .....                                                           | 383 |
| D.2.4 Baud Rate Generation .....                                                        | 383 |
| Appendix E                                                                              |     |
| ARConnect Commands .....                                                                | 385 |
| E.1 CMD_CHECK_CORE_ID .....                                                             | 385 |
| E.1.1 Parameter Format .....                                                            | 385 |
| E.1.2 MCIP_WDATA Register Format .....                                                  | 385 |
| E.1.3 MCIP_READBACK Register Format .....                                               | 385 |
| E.1.4 Command Usage: Check the CORE_ID of the core issuing CMD_CHECK_CORE_ID Command .. | 386 |
| E.2 CMD_INTRPT_GENERATE_IRQ .....                                                       | 386 |
| E.2.1 Parameter Format .....                                                            | 386 |
| E.2.2 MCIP_WDATA Register Format .....                                                  | 387 |
| E.2.3 MCIP_READBACK Register Format .....                                               | 387 |
| E.2.4 Command Usage: Generate Inter-core Interrupts .....                               | 387 |
| E.3 CMD_INTRPT_GENERATE_ACK .....                                                       | 387 |
| E.3.1 Parameter Format .....                                                            | 387 |
| E.3.2 MCIP_WDATA Register Format .....                                                  | 388 |
| E.3.3 MCIP_READBACK Register Format .....                                               | 388 |
| E.3.4 Command Usage: Generate Inter-core Interrupts Acknowledgment .....                | 388 |
| E.4 CMD_INTRPT_READ_STATUS .....                                                        | 388 |
| E.4.1 Parameter Format .....                                                            | 388 |
| E.4.2 MCIP_WDATA Register Format .....                                                  | 388 |
| E.4.3 MCIP_READBACK Register Format .....                                               | 389 |
| E.4.4 Command Usage: Check the Status of Inter-core Interrupts .....                    | 389 |
| E.5 CMD_INTRPT_CHECK_SOURCE .....                                                       | 390 |
| E.5.1 Parameter Format .....                                                            | 390 |
| E.5.2 MCIP_WDATA Register Format .....                                                  | 390 |
| E.5.3 MCIP_READBACK Register Format .....                                               | 390 |
| E.5.4 Command Usage: Check the Status of Inter-core Interrupts .....                    | 391 |
| E.6 CMD_SEMA_CLAIM_AND_READ .....                                                       | 391 |
| E.6.1 Parameter Format .....                                                            | 391 |
| E.6.2 MCIP_WDATA Register Format .....                                                  | 391 |
| E.6.3 MCIP_READBACK Register Format .....                                               | 392 |

|                                                                          |     |
|--------------------------------------------------------------------------|-----|
| E.6.4 Command Usage: Claim Semaphore and Check Semaphore ownership ..... | 392 |
| E.7 CMD_SEMA_RELEASE .....                                               | 392 |
| E.7.1 Parameter Format .....                                             | 393 |
| E.7.2 MCIP_WDATA Register Format .....                                   | 393 |
| E.7.3 MCIP_READBACK Register Format .....                                | 393 |
| E.7.4 Command Usage: Release Semaphore Owned by Core .....               | 393 |
| E.8 CMD_SEMA_FORCE_RELEASE .....                                         | 393 |
| E.8.1 Parameter Format .....                                             | 393 |
| E.8.2 MCIP_WDATA Register Format .....                                   | 393 |
| E.8.3 MCIP_READBACK Register Format .....                                | 393 |
| E.8.4 Command Usage: Release Semaphore Owned by Core .....               | 393 |
| E.9 CMD_MSG_SRAM_SET_ADDR .....                                          | 394 |
| E.9.1 Parameter Format .....                                             | 394 |
| E.9.2 MCIP_WDATA Register Format .....                                   | 394 |
| E.9.3 Command Usage: .....                                               | 394 |
| E.10 CMD_MSG_SRAM_READ_ADDR .....                                        | 394 |
| E.10.1 Parameter Format .....                                            | 394 |
| E.10.2 MCIP_WDATA Register Format .....                                  | 395 |
| E.10.3 MCIP_READBACK Register Format .....                               | 395 |
| E.10.4 Command Usage: Read internal SRAM Address .....                   | 395 |
| E.11 CMD_MSG_SRAM_SET_ADDR_OFFSET .....                                  | 395 |
| E.11.1 Parameter Format .....                                            | 395 |
| E.11.2 MCIP_WDATA Register Format .....                                  | 395 |
| E.11.3 MCIP_READBACK Register Format .....                               | 396 |
| E.11.4 Command Usage: Read internal SRAM Address .....                   | 396 |
| E.12 CMD_MSG_SRAM_READ_ADDR_OFFSET .....                                 | 396 |
| E.12.1 Parameter Format .....                                            | 396 |
| E.12.2 MCIP_WDATA Register Format .....                                  | 396 |
| E.12.3 MCIP_READBACK Register Format .....                               | 396 |
| E.12.4 Command Usage: Read SRAM Address Offset Value .....               | 397 |
| E.13 CMD_MSG_SRAM_WRITE .....                                            | 397 |
| E.13.1 Parameter Format .....                                            | 397 |
| E.13.2 MCIP_WDATA Register Format .....                                  | 397 |
| E.13.3 MCIP_READBACK Register Format .....                               | 397 |
| E.13.4 Command Usage: Write data into message passing SRAM .....         | 398 |
| E.14 CMD_MSG_SRAM_WRITE_INC .....                                        | 398 |
| E.14.1 Parameter Format .....                                            | 398 |
| E.14.2 MCIP_WDATA Register Format .....                                  | 398 |
| E.14.3 MCIP_READBACK Register Format .....                               | 398 |
| E.14.4 Command Usage: Write data in incremental mode .....               | 399 |
| E.15 CMD_MSG_SRAM_WRITE_IMM .....                                        | 399 |
| E.15.1 Parameter Format .....                                            | 399 |
| E.15.2 MCIP_WDATA Register Format .....                                  | 400 |
| E.15.3 MCIP_READBACK Register Format .....                               | 400 |
| E.15.4 Command Usage: Write data with immediate address .....            | 400 |
| E.16 CMD_MSG_SRAM_READ .....                                             | 400 |
| E.16.1 Parameter Format .....                                            | 400 |
| E.16.2 MCIP_WDATA Register Format .....                                  | 400 |
| E.16.3 MCIP_READBACK Register Format .....                               | 401 |

|                                                                 |     |
|-----------------------------------------------------------------|-----|
| E.16.4 Command Usage: Write data with immediate address .....   | 401 |
| E.17 CMD_MSG_SRAM_READ_INC .....                                | 401 |
| E.17.1 Parameter Format .....                                   | 402 |
| E.17.2 MCIP_WDATA Register Format .....                         | 402 |
| E.17.3 MCIP_READBACK Register Format .....                      | 402 |
| E.17.4 Command Usage: Read data in incremental mode .....       | 402 |
| E.18 CMD_MSG_SRAM_READ_IMM .....                                | 403 |
| E.18.1 Parameter Format .....                                   | 403 |
| E.18.2 MCIP_WDATA Register Format .....                         | 403 |
| E.18.3 MCIP_READBACK Register Format .....                      | 403 |
| E.18.4 Command Usage: Read data in immediate mode .....         | 403 |
| E.19 CMD_DEBUG_RESET .....                                      | 404 |
| E.19.1 Parameter Format .....                                   | 404 |
| E.19.2 MCIP_WDATA Register Format .....                         | 404 |
| E.19.3 MCIP_READBACK Register Format .....                      | 405 |
| E.19.4 Command Usage: Selective Reset Cores .....               | 405 |
| E.20 CMD_DEBUG_RESET .....                                      | 405 |
| E.20.1 Parameter Format .....                                   | 405 |
| E.20.2 MCIP_WDATA Register Format .....                         | 405 |
| E.20.3 MCIP_READBACK Register Format .....                      | 405 |
| E.20.4 Command Usage: Selective Halt Cores .....                | 405 |
| E.21 CMD_DEBUG_RUN .....                                        | 406 |
| E.21.1 Parameter Format .....                                   | 406 |
| E.21.2 MCIP_WDATA Register Format .....                         | 406 |
| E.21.3 MCIP_READBACK Register Format .....                      | 406 |
| E.21.4 Command Usage: Selective Run Cores .....                 | 406 |
| E.22 CMD_DEBUG_SET_MASK .....                                   | 406 |
| E.22.1 Parameter Format .....                                   | 406 |
| E.22.2 MCIP_WDATA Register Format .....                         | 407 |
| E.22.3 MCIP_READBACK Register Format .....                      | 407 |
| E.22.4 Command Usage: .....                                     | 407 |
| E.23 CMD_DEBUG_READ_MASK .....                                  | 407 |
| E.23.1 Parameter Format .....                                   | 407 |
| E.23.2 MCIP_WDATA Register Format .....                         | 407 |
| E.23.3 MCIP_READBACK Register Format .....                      | 408 |
| E.23.4 Command Usage: Read Mask Register of Specific Core ..... | 408 |
| E.24 CMD_DEBUG_SET_SELECT .....                                 | 408 |
| E.24.1 Parameter Format .....                                   | 409 |
| E.24.2 MCIP_WDATA Register Format .....                         | 409 |
| E.24.3 MCIP_READBACK Register Format .....                      | 409 |
| E.24.4 Command Usage: Read Mask Register of Specific Core ..... | 409 |
| E.25 CMD_DEBUG_READ_SELECT .....                                | 409 |
| E.25.1 Parameter Format .....                                   | 409 |
| E.25.2 MCIP_WDATA Register Format .....                         | 410 |
| E.25.3 MCIP_READBACK Register Format .....                      | 410 |
| E.25.4 Command Usage: Read Select Register Value .....          | 410 |
| E.26 CMD_DEBUG_READ_EN .....                                    | 410 |
| E.26.1 Parameter Format .....                                   | 410 |
| E.26.2 MCIP_WDATA Register Format .....                         | 410 |

|        |                                                                                    |     |
|--------|------------------------------------------------------------------------------------|-----|
| E.26.3 | MCIP_READBACK Register Format . . . . .                                            | 410 |
| E.26.4 | Command Usage: Read EN status of all cores . . . . .                               | 411 |
| E.27   | CMD_DEBUG_READ_CMD . . . . .                                                       | 411 |
| E.27.1 | Parameter Format . . . . .                                                         | 411 |
| E.27.2 | MCIP_WDATA Register Format . . . . .                                               | 411 |
| E.27.3 | MCIP_READBACK Register Format . . . . .                                            | 411 |
| E.27.4 | Command Usage: Read last command the multi-core debug unit received . . . . .      | 412 |
| E.28   | CMD_DEBUG_READ_CORE . . . . .                                                      | 412 |
| E.28.1 | Parameter Format . . . . .                                                         | 412 |
| E.28.2 | MCIP_WDATA Register Format . . . . .                                               | 412 |
| E.28.3 | MCIP_READBACK Register Format . . . . .                                            | 413 |
| E.28.4 | Command Usage: Read MCD_CORE register value in the multi-core debug unit . . . . . | 413 |
| E.29   | CMD_GRTC_CLEAR . . . . .                                                           | 413 |
| E.29.1 | Parameter Format . . . . .                                                         | 414 |
| E.29.2 | MCIP_WDATA Register Format . . . . .                                               | 414 |
| E.29.3 | MCIP_READBACK Register Format . . . . .                                            | 414 |
| E.29.4 | Command Usage: Clear the global RTC . . . . .                                      | 414 |
| E.30   | CMD_GRTC_READ_LO . . . . .                                                         | 414 |
| E.30.1 | Parameter Format . . . . .                                                         | 414 |
| E.30.2 | MCIP_WDATA Register Format . . . . .                                               | 414 |
| E.30.3 | MCIP_READBACK Register Format . . . . .                                            | 414 |
| E.30.4 | Command Usage: . . . . .                                                           | 414 |
| E.31   | CMD_GRTC_READ_HI . . . . .                                                         | 414 |
| E.31.1 | Parameter Format . . . . .                                                         | 415 |
| E.31.2 | MCIP_WDATA Register Format . . . . .                                               | 415 |
| E.31.3 | MCIP_READBACK Register Format . . . . .                                            | 415 |
| E.31.4 | Command Usage: Read the global RTC . . . . .                                       | 415 |
| E.32   | CMD_PMU_SET_PMODE . . . . .                                                        | 415 |
| E.32.1 | Parameter Format . . . . .                                                         | 415 |
| E.32.2 | MCIP_WDATA Register Format . . . . .                                               | 416 |
| E.32.3 | MCIP_READBACK Register Format . . . . .                                            | 416 |
| E.32.4 | Command Usage: Set Power Mode of Core . . . . .                                    | 416 |
| E.33   | CMD_PMU_READ_PSTATUS . . . . .                                                     | 416 |
| E.33.1 | Parameter Format . . . . .                                                         | 416 |
| E.33.2 | MCIP_WDATA Register Format . . . . .                                               | 417 |
| E.33.3 | MCIP_READBACK Register Format . . . . .                                            | 417 |
| E.33.4 | Command Usage: Read PMU status register of Specified core . . . . .                | 418 |
| E.34   | CMD_PMU_SET_PCNT . . . . .                                                         | 418 |
| E.34.1 | Parameter Format . . . . .                                                         | 418 |
| E.34.2 | MCIP_WDATA Register Format . . . . .                                               | 418 |
| E.34.3 | MCIP_READBACK Register Format . . . . .                                            | 418 |
| E.34.4 | Command Usage: Set Power-up duration time . . . . .                                | 418 |
| E.35   | CMD_PMU_READ_PCNT . . . . .                                                        | 419 |
| E.35.1 | Parameter Format . . . . .                                                         | 419 |
| E.35.2 | MCIP_WDATA Register Format . . . . .                                               | 419 |
| E.35.3 | MCIP_READBACK Register Format . . . . .                                            | 419 |
| E.35.4 | Command Usage: Read Power-up duration counter value . . . . .                      | 419 |
| E.36   | CMD_PMU_READ_DVFS . . . . .                                                        | 419 |
| E.36.1 | Parameter Format . . . . .                                                         | 419 |

E.36.2 MCIP\_WDATA Register Format .....419

E.36.3 MCIP\_READBACK Register Format .....419

E.36.4 Command Usage: Read performance level for specified core .....420





# List of Tables

|                                                                                 |     |
|---------------------------------------------------------------------------------|-----|
| Table 1-1 Databook Revision History .....                                       | 30  |
| Table 2-1 IP Libraries Included in the Delivery .....                           | 45  |
| Table 2-2 Supported Platforms .....                                             | 46  |
| Table 2-3 APEX Supported Platforms .....                                        | 47  |
| Table 5-1 ICCM0 and ICCM1 Base Address Register .....                           | 80  |
| Table 5-2 Data Cache Auxiliary Registers .....                                  | 85  |
| Table 5-3 Invalidate Data Cache Flags .....                                     | 86  |
| Table 5-4 Effect of the DC flag and LD/ST Instructions on Cache Operation ..... | 89  |
| Table 5-5 Flag Update Conditions .....                                          | 90  |
| Table 5-6 DC_FLSH Control Flags .....                                           | 94  |
| Table 5-7 Advanced Data Cache Auxiliary Registers .....                         | 96  |
| Table 5-8 DC_TAG Control Flags and Update Conditions .....                      | 97  |
| Table 5-9 DC_CTRL Control Flags .....                                           | 98  |
| Table 5-10 SB Flag Update Conditions with Advanced Cache Modes .....            | 99  |
| Table 5-11 Instruction Cache Auxiliary Registers .....                          | 103 |
| Table 5-12 IC_CTRL Control Flags .....                                          | 104 |
| Table 5-13 SB Flag Update Conditions .....                                      | 106 |
| Table 5-14 Advanced Instruction Cache Auxiliary Registers .....                 | 109 |
| Table 5-15 IC_CTRL Control Flags .....                                          | 111 |
| Table 5-16 SB Flag Update Conditions with Advanced Cache Modes .....            | 112 |
| Table 6-1 Conditions Checked in Stack-Checking Scenarios .....                  | 120 |
| Table 8-1 ARC EM Pipeline Stages .....                                          | 131 |
| Table 8-2 Instruction Categories for Timing Calculations .....                  | 133 |

|                                                                                          |     |
|------------------------------------------------------------------------------------------|-----|
| Table 8-3 Number of Instruction Memory Accesses Per Instruction Fetch .....              | 134 |
| Table 8-4 ARC EM Multiplier Design Options.....                                          | 139 |
| Table 9-1 Signals: Instruction Bus Interface Unit with BVCI .....                        | 149 |
| Table 9-2 Signals: Instruction-Bus Interface Unit with AHB .....                         | 150 |
| Table 9-3 Signals: Instruction-Bus Interface Unit with AHB-Lite .....                    | 150 |
| Table 9-4 Signals: Data Bus Interface Unit with BVCI.....                                | 151 |
| Table 9-5 Signals: Data-Bus Interface Unit with AHB.....                                 | 152 |
| Table 9-6 Signals: Data-Bus Interface Unit with AHB-Lite.....                            | 152 |
| Table 9-7 Signals: Combined Instruction- and Data-Bus Interface Unit with AHB-Lite ..... | 153 |
| Table 9-8 Signals: Peripheral-Bus Interface Unit with BVCI .....                         | 153 |
| Table 9-9 Signals: Peripheral-Bus Interface Unit with AHB-Lite.....                      | 154 |
| Table 9-10 AXI and AHB-lite Bus Command Attributes - Kernel Mode .....                   | 155 |
| Table 9-11 AXI and AHB-Lite Bus Command Attributes - Non-Kernel Mode .....               | 155 |
| Table 9-12 Signals: ICCM DMI Unit with BVCI.....                                         | 156 |
| Table 9-13 Signals: ICCM DMI Unit with AHB-Lite .....                                    | 156 |
| Table 9-14 Signals: DCCM DMI Unit with BVCI.....                                         | 157 |
| Table 9-15 Signals: DCCM DMI Unit with AHB-Lite .....                                    | 157 |
| Table 9-16 Signals: Interrupts .....                                                     | 158 |
| Table 9-17 Signals: JTAG Interface .....                                                 | 159 |
| Table 9-18 Signals: JTAG Interface with BVCI .....                                       | 159 |
| Table 9-19 Signals: Halt and Control Interface.....                                      | 160 |
| Table 9-20 Signals: Actionpoints.....                                                    | 161 |
| Table 9-21 RTT Programming Interface .....                                               | 162 |
| Table 9-22 RTT Program Execution Tracking.....                                           | 162 |
| Table 9-23 Actionpoint Status .....                                                      | 163 |
| Table 9-24 Monitor Memory Load Traffic.....                                              | 163 |
| Table 9-25 Monitor Memory Store Traffic.....                                             | 163 |
| Table 9-26 AUX Register Read/Write Monitoring .....                                      | 164 |
| Table 9-27 Core Register Write Port 0 Monitoring.....                                    | 164 |

|                                                                                      |     |
|--------------------------------------------------------------------------------------|-----|
| Table 9-28 Core Register Write Port 0 Monitoring .....                               | 164 |
| Table 9-29 Freeze/Restart Interface .....                                            | 164 |
| Table 9-30 Signals: Clock and Reset .....                                            | 165 |
| Table 9-31 Code Protect Signals .....                                                | 165 |
| Table 10-1 Clock, Reset, and Run Control Signal List .....                           | 167 |
| Table 10-2 Interpretation of sys_halt_r and sys_sleep_r output signals .....         | 170 |
| Table 11-1 Accumulator Operations .....                                              | 173 |
| Table 13-1 .....                                                                     | 181 |
| Table 13-2 Interrupt to External System .....                                        | 183 |
| Table 13-3 Timer CONTROL0/1 Register Field Descriptions .....                        | 191 |
| Table 13-4 Power Status Interface to External System .....                           | 193 |
| Table 13-5 DVFS Interface to External DVFS Controller .....                          | 195 |
| Table 13-6 List of Auxiliary Registers supporting ARConnect .....                    | 196 |
| Table 13-7 Field Descriptions of the MCIP_CMD Register .....                         | 197 |
| Table 13-8 MCIP_READBACK Field Descriptions for CMD_INTRPT_READ_STATUS Command ..... | 199 |
| Table 13-9 DEBUGP Register Field Descriptions .....                                  | 199 |
| Table 13-10 Field Descriptions of the MCIP_SYSTEM_BUILD Register .....               | 200 |
| Table 13-11 Field Description of the MCIP_SEMA_BUILD Register .....                  | 202 |
| Table 13-12 Field Descriptions of the MCIP_MESSAGE_BUILD Register .....              | 202 |
| Table 13-13 Field Descriptions of the MCIP_PMU_BUILD Register .....                  | 203 |
| Table 13-14 ARConnect Command List .....                                             | 203 |
| Table 13-15 Configurations of ARConnect IP .....                                     | 217 |
| Table 14-1 JTAG 1149.1 Registers .....                                               | 220 |
| Table 14-2 JTAG Read and Write Transactions .....                                    | 222 |
| Table 14-3 Host JTAG 1149.7 Interface .....                                          | 233 |
| Table 14-4 Target (CLTAPC) JTAG 1149.7 Interface .....                               | 233 |
| Table 14-5 JTAG 1149.7 Auxiliary Control Signal .....                                | 234 |
| Table 14-6 JTAG 1149.7 Test-Mode Signal .....                                        | 234 |
| Table 14-7 Host JTAG 1149.7 Interface .....                                          | 236 |

|                                                                                           |     |
|-------------------------------------------------------------------------------------------|-----|
| Table 14-8 JTAG 1149.7 Reset Types .....                                                  | 236 |
| Table 14-9 TAP.7 Controller Commands .....                                                | 239 |
| Table 14-10 TAP.7 Controller Registers and Associated Commands .....                      | 240 |
| Table 14-11 Actionpoint Monitoring Targets .....                                          | 248 |
| Table 14-12 Auxiliary Registers for Actionpoints .....                                    | 249 |
| Table 14-13 Key to Auxiliary Register Access Permissions for LR and SR Instructions ..... | 251 |
| Table 14-14 State of Auxiliary Registers on Reset .....                                   | 252 |
| Table 14-15 Actionpoint External Parameter Signals .....                                  | 262 |
| Table 14-16 RTT_COMMAND Bit Field Description .....                                       | 282 |
| Table 15-1 Performance Counters and Features .....                                        | 287 |
| Table 15-2 Countable-Conditions Registers .....                                           | 289 |
| Table 15-3 Logical Conditions .....                                                       | 290 |
| Table 15-4 ISA Conditions .....                                                           | 290 |
| Table 15-5 Cycle-Time Conditions .....                                                    | 292 |
| Table 15-6 Bubbles/Stall Conditions .....                                                 | 293 |
| Table 15-7 Instruction Cache-Related Conditions .....                                     | 294 |
| Table 15-8 Data Cache-Related Conditions .....                                            | 295 |
| Table 15-9 User Flag Conditions .....                                                     | 295 |
| Table 15-10 APEX Conditions .....                                                         | 295 |
| Table 15-11 Performance Counter Registers .....                                           | 297 |
| Table 16-1 UAUX Bus Interface Signals .....                                               | 301 |
| Table 16-2 UAUX Violation Mapping .....                                                   | 302 |
| Table 17-1 Internal Clocks During Sleep State .....                                       | 307 |
| Table 18-1 BusFabric Configuration Parameters .....                                       | 319 |
| Table 18-2 SRAM Controller Configuration Parameters .....                                 | 321 |
| Table 18-3 Clock Controller Configuration Parameters .....                                | 322 |
| Table 18-4 Clock Controller Software Register View .....                                  | 323 |
| Table 18-5 Profiler Configuration Parameters .....                                        | 325 |
| Table 18-6 Profiler Software Register View .....                                          | 327 |

|                                                                                             |     |
|---------------------------------------------------------------------------------------------|-----|
| Table 18-7 Dummy Master Configuration Parameters .....                                      | 330 |
| Table 18-8 Dummy Master Configuration Parameters .....                                      | 331 |
| Table A-1 Differences between ARC 600 and ARC EM Memory-Protection Units .....              | 339 |
| Table C-1 iccm_ram .....                                                                    | 362 |
| Table C-2 iccm1_ram (iccm1_banking = 0) .....                                               | 362 |
| Table C-3 iccm1_ram (iccm1_banking = 1) .....                                               | 363 |
| Table C-4 dccm_ram.....                                                                     | 363 |
| Table C-5 dc_data_ram .....                                                                 | 363 |
| Table C-6 dc_tag_ram.....                                                                   | 364 |
| Table C-7 ic_data_ram .....                                                                 | 364 |
| Table C-8 ic_tag_ram .....                                                                  | 364 |
| Table C-9 smart_ram .....                                                                   | 365 |
| Table C-10 regfile_ram.....                                                                 | 365 |
| Table D-1 ARC UART Control Registers.....                                                   | 377 |
| Table D-2 Field Descriptions for Status Register .....                                      | 381 |
| Table E-1 MCIP_READBACK Register Field Descriptions for CMD_CHECK_CORE_ID Command .....     | 386 |
| Table E-2 MCIP_READBACK Register Field Descriptions for CMD_INTERRUPT_READ_STATUS Command . | 389 |
| Table E-3 MCIP_READBACK Register Field Descriptions for CMD_INTRPT_CHECK_SOURCE Command     | 390 |
| Table E-4 MCIP_READBACK Register Field Descriptions for CMD_SEMA_CLAIM_AND_READ Command ..  | 392 |
| Table E-5 MCIP_READBACK Field Descriptions for CMD_DEBUG_RESET Command.....                 | 404 |
| Table E-6 MCIP_READBACK Register Field Descriptions for CMD_DEBUG_READ_EN Command .....     | 411 |
| Table E-7 MCIP_READBACK Register Field Descriptions for CMD_DEBUG_READ_CORE Command.....    | 413 |
| Table E-8 MCIP_WDATA Register Field Descriptions for CMD_PMU_SET_PMODE Command.....         | 416 |
| Table E-9 MCIP_READBACK Register Field Descriptions for CMD_PMU_READ_PSTATUS Command....    | 417 |
| Table E-10 MCIP_READBACK Register Field Descriptions for CMD_PMU_READ_DVFS Command.....     | 420 |



# List of Figures

|                                                                              |     |
|------------------------------------------------------------------------------|-----|
| Figure 3-1 Overview of ARChitect GUI .....                                   | 52  |
| Figure 5-1 Memory Map .....                                                  | 74  |
| Figure 5-2 Timing Using Banking to Access Fast or Slow Memory .....          | 77  |
| Figure 5-3 ICCM0 and ICCM1 in a 32 bit Configuration .....                   | 79  |
| Figure 5-4 ICCM0 and ICCM1 in a 20-bit Overlapping Configuration .....       | 80  |
| Figure 5-5 Non-Cacheable Region .....                                        | 83  |
| Figure 5-6 DC_IVDC Register .....                                            | 85  |
| Figure 5-7 DC_CTRL Register .....                                            | 86  |
| Figure 5-8 Cache Enabled .....                                               | 88  |
| Figure 5-9 Cache Disabled .....                                              | 88  |
| Figure 5-10 DC_LDL Register .....                                            | 91  |
| Figure 5-11 DC_IVDL Register .....                                           | 93  |
| Figure 5-12 DC_FLSH Register .....                                           | 94  |
| Figure 5-13 DC_FLDL Register .....                                           | 95  |
| Figure 5-14 DC_RAM_ADDR .....                                                | 96  |
| Figure 5-15 DC_TAG for Reads .....                                           | 97  |
| Figure 5-16 DC_TAG for Writes .....                                          | 97  |
| Figure 5-17 DC_DATA .....                                                    | 98  |
| Figure 5-18 DC_CTRL Register .....                                           | 98  |
| Figure 5-19 Indirect Cache-RAM Access Using System Memory Address .....      | 100 |
| Figure 5-20 System Address Format Used for Cache-Controlled RAM Access ..... | 100 |
| Figure 5-21 Direct Cache-RAM Access Using Direct-Cache-RAM Address .....     | 102 |
| Figure 5-22 Direct-Cache-Access Address Register for DC_RAM_ADDR .....       | 102 |

|                                                                                        |     |
|----------------------------------------------------------------------------------------|-----|
| Figure 5-23 IC_IVIC Register .....                                                     | 104 |
| Figure 5-24 IC_CTRL Register .....                                                     | 104 |
| Figure 5-25 Cache Enabled .....                                                        | 105 |
| Figure 5-26 Cache Disabled.....                                                        | 106 |
| Figure 5-27 IC_LIL Register.....                                                       | 107 |
| Figure 5-28 IC_IVIL Register.....                                                      | 108 |
| Figure 5-29 IC_RAM_ADDR Register .....                                                 | 109 |
| Figure 5-30 IC_TAG Register for Reads .....                                            | 110 |
| Figure 5-31 IC_TAG Register for Writes .....                                           | 110 |
| Figure 5-32 IC_DATA Register.....                                                      | 111 |
| Figure 5-33 IC_CTRL Register .....                                                     | 111 |
| Figure 5-34 Indirect Cache RAM Access Using System Memory Address .....                | 113 |
| Figure 5-35 System Address Format Used For Cache Controlled RAM Access .....           | 113 |
| Figure 5-36 Using Direct Cache-RAM Access .....                                        | 114 |
| Figure 5-37 Direct-Cache-Access Address Register for IC_RAM_ADDR .....                 | 115 |
| Figure 6-1 Stack Regions .....                                                         | 119 |
| Figure 6-2 Stack-Checking Scenarios .....                                              | 120 |
| Figure 9-1 Block Diagram Showing Bus Interfaces .....                                  | 147 |
| Figure 9-2 Timing Diagram for the Halt Controls .....                                  | 160 |
| Figure 9-3 Timing Diagram for the Run Controls .....                                   | 161 |
| Figure 10-1 ICCM1 Clocks with ICCM1 Banking .....                                      | 168 |
| Figure 11-1 ARC EM DSP Pipeline.....                                                   | 174 |
| Figure 13-1 System Diagram of two ARC cores integrated with ARConnect .....            | 180 |
| Figure 13-2 JTAG Chain Diagram .....                                                   | 181 |
| Figure 13-3 JTAG Status Register .....                                                 | 190 |
| Figure 13-4 Timer CONTRL0/1 Register .....                                             | 190 |
| Figure 13-5 The MCIP_CMD Register .....                                                | 197 |
| Figure 13-6 MCIP_WDATA Register Format Example for CMD_MSG_SRAM_SET_ADDR Command ..... | 198 |
| Figure 13-7 MCIP_READBACK Register Format for CMD_INTRPT_READ_STATUS Command .....     | 199 |



|                                                                                        |     |
|----------------------------------------------------------------------------------------|-----|
| Figure 13-8 DEBUGP Register. ....                                                      | 199 |
| Figure 13-9 The MCIP_SYSTEM_BUILD Register. ....                                       | 200 |
| Figure 13-10 MCIP_SEMA_BUILD Register. ....                                            | 202 |
| Figure 13-11 The MCIP_MESSAGE_BUILD Register. ....                                     | 202 |
| Figure 13-12 The MCIP_PMU_BUILD Register. ....                                         | 203 |
| Figure 14-1 A JTAG 1149.1 Port with an ARC EM Processor. ....                          | 220 |
| Figure 14-2 Internal Structure of the JTAG 1149.1 Port. ....                           | 224 |
| Figure 14-3 JTAG 1149.1 TAP-Controller State Diagram. ....                             | 225 |
| Figure 14-4 Loading Data into the JTAG 1149.1 Shift Register. ....                     | 226 |
| Figure 14-5 Typical Application for the DesignWare Compact JTAG 1149.7 Core. ....      | 227 |
| Figure 14-6 Compact JTAG 1149.7 Core System-Level Block Diagram. ....                  | 230 |
| Figure 14-7 Conceptual System-Level Block Diagram (Single STL TAP Controller). ....    | 231 |
| Figure 14-8 Conceptual System-Level Block Diagram (Multiple STL TAP Controllers). .... | 232 |
| Figure 14-9 JTAG 1149.7 Top-Level I/O Diagram. ....                                    | 232 |
| Figure 14-10 Compact JTAG 1149.7 Core Functional Block Diagram. ....                   | 235 |
| Figure 14-11 JTAG 1149.7 Example Escape Waveforms. ....                                | 238 |
| Figure 14-12 JTAG 1149.7 EPU Operating States. ....                                    | 239 |
| Figure 14-13 Advanced Protocol (Multiplexed): OScan 1 Format. ....                     | 242 |
| Figure 14-14 Internal Structure of the Host Interface to BVCI Target Module. ....      | 243 |
| Figure 14-15 Loading the Instruction Register (Select-Address Register). ....          | 246 |
| Figure 14-16 Debugger and an ARv2EM Processor. ....                                    | 247 |
| Figure 14-17 System with Four Actionpoints. ....                                       | 253 |
| Figure 14-18 Paired-Actionpoints Example. ....                                         | 257 |
| Figure 14-19 Smart Operation. ....                                                     | 263 |
| Figure 14-20 SMART_BUILD Register. ....                                                | 264 |
| Figure 14-21 SMART_CONTROL Register. ....                                              | 264 |
| Figure 14-22 SMART_DATA Register. ....                                                 | 265 |
| Figure 14-23 SRC_ADDR Value. ....                                                      | 266 |
| Figure 14-24 DEST_ADDR Value Register. ....                                            | 266 |

|                                                                                                   |     |
|---------------------------------------------------------------------------------------------------|-----|
| Figure 14-25 FLAGS Value Register . . . . .                                                       | 266 |
| Figure 14-26 Stack Trace on Reset . . . . .                                                       | 268 |
| Figure 14-27 RTT_ADDRESS Register . . . . .                                                       | 281 |
| Figure 14-28 RTT_DATA Register . . . . .                                                          | 282 |
| Figure 14-29 RTT_COMMAND Register . . . . .                                                       | 282 |
| Figure 15-1 Condition Filter Mode . . . . .                                                       | 289 |
| Figure 16-1 Adding UAUX using ARChitect GUI . . . . .                                             | 300 |
| Figure 16-2 Block Diagram of the UAUX External Bus Interface . . . . .                            | 301 |
| Figure 16-3 Basic UAUX Transaction . . . . .                                                      | 303 |
| Figure 16-4 UAUX Transaction with Waited Cycles . . . . .                                         | 304 |
| Figure 16-5 UAUX Transaction Aborted . . . . .                                                    | 304 |
| Figure 17-1 ARC EM Partition with Resynchronization Modules . . . . .                             | 309 |
| Figure 18-1 System Design Hierarchy . . . . .                                                     | 313 |
| Figure 18-2 BusFabric Block Diagram . . . . .                                                     | 315 |
| Figure 18-3 System Memory Map, CPU Memory Map, and System Memory Map with Default Slave . . . . . | 318 |
| Figure 18-4 SRAM Controller Block Diagram . . . . .                                               | 320 |
| Figure 18-5 Clock Controller Block Diagram . . . . .                                              | 322 |
| Figure 18-6 Clock Controller Register . . . . .                                                   | 323 |
| Figure 18-7 Profiler Block Diagram . . . . .                                                      | 325 |
| Figure 18-8 MSS_PERFCTRL_REG_CTRL Fields Register . . . . .                                       | 329 |
| Figure 18-9 Dummy Master Block Diagram . . . . .                                                  | 330 |
| Figure 18-10 Dummy Slave Block Diagram . . . . .                                                  | 331 |
| Figure D-1 Basic ARC UART Block Diagram . . . . .                                                 | 376 |
| Figure D-2 ARC UART and ARC System Diagram . . . . .                                              | 376 |
| Figure E-1 MCIP_READBACK Register Format for CMD_CHECK_CORE_ID Command . . . . .                  | 385 |
| Figure E-2 MCIP_READBACK Register Format for CMD_INTRPT_READ_STATUS Command . . . . .             | 389 |
| Figure E-3 MCIP_READBACK Register Format for CMD_INTRPT_CHECK_SOURCE Command . . . . .            | 390 |
| Figure E-4 MCIP_READBACK Register Format for CMD_SEMA_CLAIM_AND_READ Command . . . . .            | 392 |
| Figure E-5 MCIP_READBACK Register Format for CMD_MSG_SRAM_SET_ADDR Command . . . . .              | 394 |

|                                                                                         |     |
|-----------------------------------------------------------------------------------------|-----|
| Figure E-6 MCIP_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR Command . . . . .   | 395 |
| Figure E-7 MCIP_READBACK Register Format for CMD_MSG_SRAM_SET_ADDR_OFFSET Command . . . | 396 |
| Figure E-8 MCIP_READBACK Register Format for CMD_MSG_SRAM_READ_ADDR_OFFSET Command.     | 397 |
| Figure E-9 MCIP_READBACK Register Format for CMD_MSG_SRAM_WRITE Command . . . . .       | 397 |
| Figure E-10 MCIP_READBACK Register Format for CMD_MSG_SRAM_READ Command . . . . .       | 401 |
| Figure E-11 MCIP_READBACK Register Format for CMD_MSG_SRAM_READ_INC Command . . . . .   | 402 |
| Figure E-12 MCIP_READBACK Register Format for CMD_DEBUG_RESET Command. . . . .          | 404 |
| Figure E-13 MCIP_READBACK Register Format for CMD_DEBUG_READ_MASK Command . . . . .     | 408 |
| Figure E-14 MCIP_READBACK Register Format for CMD_DEBUG_READ_SELECT Command. . . . .    | 410 |
| Figure E-15 MCIP_READBACK Register for CMD_DEBUG_READ_EN . . . . .                      | 411 |
| Figure E-16 MCIP_READBACK Register for CMD_DEBUG_READ_CMD . . . . .                     | 411 |
| Figure E-17 MCIP_READBACK Register for CMD_DEBUG_READ_CORE . . . . .                    | 413 |
| Figure E-18 MCIP_READBACK Register for CMD_GRTC_READ_LO. . . . .                        | 414 |
| Figure E-19 MCIP_READBACK Register for CMD_GRTC_READ_HI . . . . .                       | 415 |
| Figure E-20 MCIP_WDATA Register for CMD_PMU_SET_PMODE . . . . .                         | 416 |
| Figure E-21 MCIP_READBACK Register for CMD_PMU_READ_PSTATUS . . . . .                   | 417 |
| Figure E-22 MCIP_WDATA Register for CMD_PMU_SET_PCNT . . . . .                          | 418 |
| Figure E-23 MCIP_READBACK Register for CMD_PMU_READ_PCNT. . . . .                       | 419 |
| Figure E-24 MCIP_READBACK Register for CMD_PMU_READ_DVFS . . . . .                      | 419 |



# Preface

This manual provides information about the DesignWare ARC EM processor series.

The information in this manual includes an overview of the product and how to use it, plus information on the major blocks and components, registers, signals, and verification.

## Databook Organization

This databook contains the following chapters:

- [Chapter 1, “Product Overview”](#) provides an introduction to DesignWare ARC processor cores, design goals, the feature set, and an overview of customization.
- [Chapter 2, “Release Notes”](#) provides information about this DesignWare ARC EM release, including deliverables, supported platforms and standards, changes from the previous release, and any known issues.
- [Chapter 3, “Getting Started”](#) gives an overview of the ARChitect configuration tool, how to compile C/C++ or assembly code for ARC EM, and where to go for Synopsys licensing information.
- [Chapter 4, “Configurability”](#) describes the optional components and instructions, how to use preconfigured templates, and multiprocessor support.
- [Chapter 5, “Memory Components”](#) describes the memory map and peripheral space, the debugger view of memory, closely coupled memory, caches, and the instruction-fetch queue.
- [Chapter 6, “Memory-Protection Unit”](#) discusses the stack-checking, code-protection, and memory-protection features available as options for ARC EM processors.
- [Chapter 7, “Interrupts”](#) tells how to configure and enable interrupts, and gives a brief overview of the interrupt architecture.
- [Chapter 8, “Cycle-Timing Characteristics”](#) discusses ARC EM processor timing in general, including exceptions and interrupts.
- [Chapter 9, “External Interfaces”](#) gives an overview of the ARC EM external interfaces and lists the interface signals.
- [Chapter 10, “Clock, Reset, and Run Controls”](#) describes clock-signal requirements, interface considerations, and the reset configuration.
- [Chapter 14, “Debug Features”](#) describes the JTAG port, actionpoints, and SmaRT real-time trace.
- [Chapter 16, “UAUX Interface”](#), UAUX is an optional feature for ARC EM. This optional interface allows user modules containing external auxiliary registers to be connected to processor core.

- [Chapter 15, “Performance Counters”](#), describes the performance monitoring features available in ARC EM.
- [Chapter 12, “Extending the Processor”](#) discusses adding instructions and registers, assembly language support, debugger support, and variable-cycle APEX.
- [Chapter 13, “Multi-Core Option”](#), describes multiple EM Core support to be integrated in a homogenous structure.
- [Chapter 17, “Power-Saving Features”](#) tells how ARC EM provides low power consumption and describes the sleep and halt states.
- [Appendix A, “Converting from ARC 600”](#) discusses hardware and software considerations for migrating from ARC 600 to ARC EM.
- [Appendix B, “Key IP Library Command-Line Options”](#) contains a listing of key ARChitect command-line options specific to ARC EM IP library.
- [Chapter C, “Synthesis and Physical Implementation”](#) gives a brief overview of register-file implementation, integration, and use of DFT-Compiler with reference to the reference design flow documentation included with ARC EM.
- [Appendix D, “ARC UART”](#) provides details on a UART that you can implement for confidence tests.
- The [Glossary](#) provides definitions of general terms.
- An [Index](#) provides hyperlinked reference to key concepts.

## Documentation

ARC EM Includes two documents in Adobe® Acrobat format:

- This *DesignWare ARC EM Databook*
- The *ARCV2 ISA Programmer’s Reference*

The *DesignWare ARC EM Series Bookshelf* (contents.pdf) is the starting point for viewing the documents.

Tools such as ARChitect™ and the MetaWare® Development Toolkit have their own sets of manuals, typically located in their docs/pdf directory.

## Document Revision History

This section tracks the significant documentation changes that occur from release-to-release and during a release.

**Table 1-1 Databook Revision History**

| Version  | Databook Date | Description                                                            |
|----------|---------------|------------------------------------------------------------------------|
| 5756-023 | October 2011  | Initial release                                                        |
| 5756-024 | April 2012    | Added support for DesignWare ARC Floating Point (licensed separately). |

**Table 1-1 Databook Revision History**

| Version  | Databook Date | Description                                                                                                                                                                                                                                                                                                                                                                                           |
|----------|---------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 5756-025 | May 2012      | Updated the following sections for Release 1.0a: <ul style="list-style-type: none"> <li>■ “Deliverables”</li> <li>■ “Documents”</li> <li>■ “Supported Platforms”</li> </ul>                                                                                                                                                                                                                           |
| 5756-026 | October 2012  | <ul style="list-style-type: none"> <li>■ Comprehensive revision to document the features listed in “<a href="#">Changes in Release 1.1</a>” on page 49</li> <li>■ Fixed all documentation STARs to date</li> </ul>                                                                                                                                                                                    |
| 5756-029 | June 2014     | <ul style="list-style-type: none"> <li>■ Added a new chapter for MCIP support</li> <li>■ Added a new chapter for UAUX support</li> <li>■ Added a new chapter for Performance Counters support</li> <li>■ Added a new chapter for DSP support</li> <li>■ Replaced or EIA references to APEX</li> <li>■ Added description for 5D and 7D templates</li> <li>■ Updated description for ARC2MSS</li> </ul> |

## Web Resources

- DesignWare IP product information: <http://www.designware.com>
- Your custom DesignWare IP page: <http://www.mydesignware.com>
- Documentation through SolvNet: <http://solvnet.synopsys.com> (Synopsys password required)
- Synopsys Common Licensing (SCL): <http://www.synopsys.com/keys>

## Customer Support

Customer support is available through SolvNet online customer support and through contacting the Synopsys Technical Support Center.

### Accessing SolvNet

SolvNet includes an electronic knowledge base of technical articles and answers to frequently asked questions about Synopsys tools. SolvNet also gives you access to a wide range of Synopsys online services, which include downloading software, viewing Documentation on the Web, and entering a call to the Support Center.

To access SolvNet:

1. Go to the SolvNet Web page at <http://solvnet.synopsys.com/>.
2. If prompted, enter your user name and password. (If you do not have a Synopsys user name and password, follow the instructions to register with SolvNet.)

If you need help using SolvNet, click SolvNet Help in the Support Resources section.

## Contacting the Synopsys Technical Support Center

If you have problems, questions, or suggestions, you can contact the Synopsys Technical Support Center in the following ways:

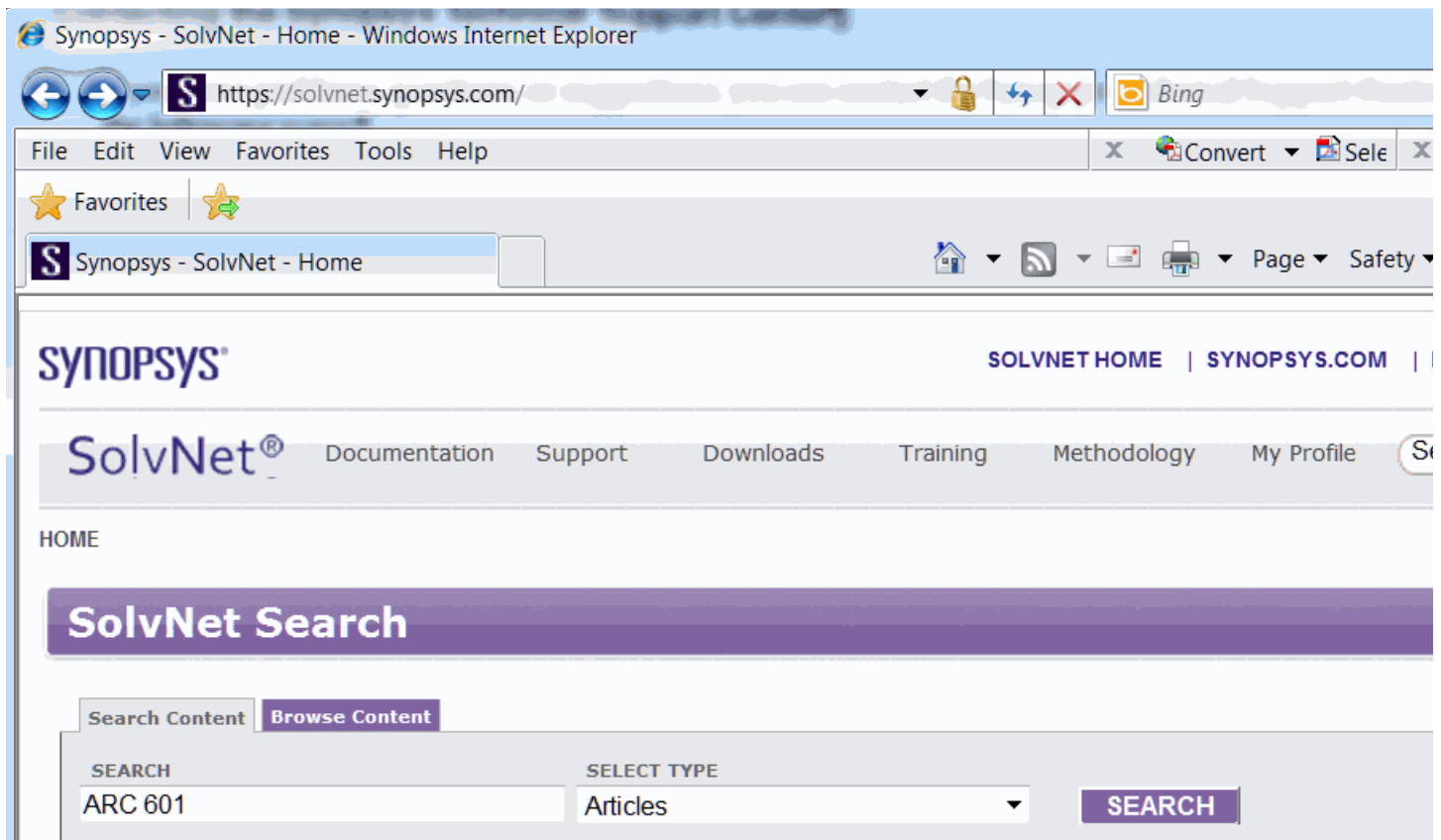
- Open a call to your local support center from the Web by going to <http://solvnet.synopsys.com/> (Synopsys user name and password required), then clicking “Enter a Call to the Support Center.”
- Send an e-mail message to your local support center.
  - E-mail [support\\_center@synopsys.com](mailto:support_center@synopsys.com) from within North America.
  - Find other local support center e-mail addresses at <http://www.synopsys.com/Support/GlobalSupportCenters>.
- Telephone your local support center.
  - Call (800) 245-8005 from within the continental United States.
  - Call (650) 584-4200 from Canada.
  - Find other local support center telephone numbers at <http://www.synopsys.com/Support/GlobalSupportCenters>.

## Accessing ARC Application Notes

SolvNet articles include useful application notes to help you do your job using DesignWare ARC products. To find application notes for your DesignWare ARC product, do the following:



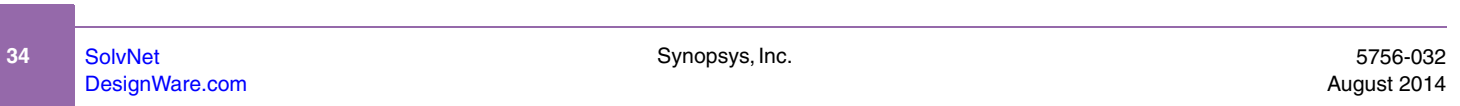
1. Log into SolvNet at <http://solvnet.synopsys.com/>.



2. Enter the name of your product in the **Search** field.
3. Select **Articles** from the **Select Type** drop-down list.
4. Click **Search**.



Application notes are updated frequently, so results can vary.







## 1

# Product Overview

This chapter provides a basic overview of the ARC EM Processor Series.

## 1.1 Introduction to DesignWare ARC Processor Cores

The DesignWare® ARC EM processor series implements the ARCV2™ instruction-set architecture in a highly efficient three-stage pipeline. The ARCV2 instruction-set architecture is the next-generation ISA for the DesignWare ARC family of configurable CPU/DSP cores.

### 1.1.1 Processors in the ARC EM Series

Different ARC EM processor templates are available when you select **File > New Project** in ARChitect to start your design. You can use these templates as-is, or modify them to build a processor fine-tuned to meet your needs.



**Note** The templates available depend on which processor you licensed.

#### 1.1.1.1 em4\_mini: Minimum-Featured ARC EM4 Processor for Sensor Applications

This template features low die area and power consumption, and is suitable for stand-alone sensor interfaces.

- 64 KB address space
- 16-bit program counter (PC) and 8-bit loop counter (LC) registers
- 16 core registers
- 16 KB high-speed closely coupled memories for instructions and data
- AHB-Lite memory bus interfaces for data I/O

#### 1.1.1.2 em4\_dmips: Full featured EM4 Processor

This template is ideal for small, fast controller applications supporting fixed-timing task scheduling.

- 24-bit program counter (PC) and 32-bit loop counter (LC) registers
- 16 core registers

- 16 KB closely coupled memories for instructions and data
- Efficient code-density instructions with fast multiply, divide, and shift hardware
- One 32-bit timer and seven external interrupts with a fast interrupt option
- AHB\_Lite interface to external memory for instructions and data

#### 1.1.1.3 **em4\_dmips\_fpusp: Full-featured EM4 processor with single-precision floating point**

#### 1.1.1.4 **em4\_dmips\_fpuspdp: Full-featured EM4 processor with single-precision floating point with double-precision assist**

#### 1.1.1.5 **em6\_dmips: Full-Featured ARC EM6 Processor**

This template provides the basis for a full-featured small, cached processor for high performance over a large code base.

- 32-bit program counter (PC) and loop counter (LC) registers
- 32 core registers
- 16 KB instruction and data caches
- Efficient code-density instructions
- Two-cycle multiplier with a fast divider and shift hardware
- One 32-bit timer and seven external interrupts with a fast interrupt option
- AHB interface to external memory

#### 1.1.1.6 **em6\_mini: Minimum-Featured ARC EM6 Processor**

#### 1.1.1.7 **em5d\_mini: EM4 processor with DSP for minimal area and high frequency**

This template provides the basis for a full-featured small, cached processor for high performance over a large code base.

- DSP with optimized multiplier and adder with complex instructions enabled
- 24-bit program counter (PC) and loop counter (LC) registers
- 32 core registers
- 32 KB instruction and data caches
- Efficient code-density instructions
- One 32-bit timer
- 8 number 2-level interrupts
- AHB Lite dual interface to external memory

#### 1.1.1.8 **em5d\_nrg: EM4 processor with DSP for low energy consumption**

#### 1.1.1.9 **EM7d\_nrg: EM6 processor with DSP for low energy consumption**

This template provides the basis for a full-featured small, cached processor for high performance over a large code base.

- DSP with optimized multiplier and adder with complex instructions enabled
- 32-bit program counter (PC) and 32-bit loop counter (LC) registers
- 32 core registers
- 8 KB instruction and data caches
- Efficient code-density instructions
- One 32-bit timer
- 8 number 2-level interrupts
- AHB Lite dual interface to external memory

## 1.2 Design Goals

The ARC EM processor series is optimized for use in embedded applications where high performance with minimum power consumption is essential. The cores in the series offer outstanding performance density, delivering 1.77 DMIPS/MHz within a very small footprint and extremely low power consumption.

The small size of the ARC EM processors makes them ideal for applications such as sensors and actuators, memory cards, SSD controllers, 8- and 16-bit microcontroller replacement, and battery-operated products.

## 1.3 Architecture Highlights

The DesignWare ARC EM has numerous advanced features based on the ARCV2 instruction-set architecture (ISA).

- Instructions
  - High-speed multiply, MAC and vector-arithmetic (SIMD) options
  - Single-precision and double-precision floating-point options
  - Freely intermixed 16- and 32-bit instructions
  - User and kernel modes
  - Indexed instructions to save code space
  - ARC Processor Extensions (APEX) to add proprietary instructions
- Registers
  - General-purpose core registers with configurable number of registers
  - Optional register banks
  - Special-purpose auxiliary-register set
- Memory-addressing modes
  - Address-register writeback with pre- and post-index operations
  - Stack-pointer support with multiple push/pop and range checking
  - Scaled data-size addressing mode
  - PC-relative addressing
- Program flow
  - Jumps and branches with optional delay slot
  - Conditional ALU instructions
  - Combined compare-and-branch instructions
  - Zero-overhead loops
- Interrupts and exceptions
  - Maskable pulse or level-triggered external interrupts, configurable from 0 to 240 with up to 16 individually programmable priority levels
  - Automatic save and restore of selected core registers upon entry and exit
  - Optional second core register bank of configurable size



- ❑ Non-maskable exceptions
- ❑ Precise exceptions
- ❑ Memory-privilege exceptions
- ❑ Exception-return instruction
- ❑ Interrupt and exception address-only vectors to save space
- Advanced arithmetic/DSP/SIMD (vector arithmetic) instructions
  - ❑ Code-density options
  - ❑ Bit manipulation and normalization
  - ❑ Byte-level shifts, rotates, and endian reordering
  - ❑ Byte and half-word shifts and rotations
  - ❑ High-speed (one instruction/cycle sustained throughput) multiply, MAC, and SIMD options
  - ❑ Enhanced radix-4 integer division option with a non-blocking graduation protocol
  - ❑ Synchronization support
- Extension packs
  - ❑ Code-density options
  - ❑ Bit manipulation and normalization
  - ❑ Byte-level shifts, rotates, and endian reordering
  - ❑ Byte and half-word shifts and rotations
  - ❑ Integer multiplication options
  - ❑ Two integer division options
- Multiprocessor Support
  - ❑ Support for multiple ARC EM cores to be integrated in a homogeneous structure
- Debug
  - ❑ Start, stop, and single-step the processor
  - ❑ Full visibility of processor state through the debug interface
  - ❑ Breakpoint instruction
  - ❑ Optional hardware breakpoint assist (actionpoints), enables breakpoints even inside ROM code
  - ❑ Optional trace buffer (SmaRT)
  - ❑ Real-Time Trace (RTT)
- Power management
  - ❑ Sleep instruction, supporting multiple sleep modes
  - ❑ Hardware halt control
  - ❑ Power Management Unit (PMU) support through MCIP
- Two 32-bit programmable timers and 64-bit real-time counter (RTC). See [“Timers”](#) on page 68.

## 1.4 ARCV2 Instruction-Set Architecture

The *ARCV2 ISA Programmer's Reference* provides complete information on the ARCV2 instruction-set architecture, including data organization and addressing, register-set details, and complete details of the instructions.

## 1.5 Gate Count

Gate count varies by configuration. A base configuration of ARC EM is 10,000 gates, excluding memories and register file. A typical medium-sized implementation with both instruction and data cache and a 24-bit address space is approximately 22,000 gates (excluding memories and register file).

## 1.6 Customization

Many ARC EM components are optional or customizable, including closely coupled memories, caches, arithmetic components, and extensions.

Optional components can be added using the ARChitect configuration tool (GUI or command line). [Appendix B, “Key IP Library Command-Line Options”](#) contains a listing of key ARChitect command-line options specific to ARC EM IP library. In the ARChitect GUI, options for the currently selected component can be edited; see [“Adding and Configuring Components”](#) on page 53.

You can add instructions and registers using ARC Processor Extensions (APEX). ARChitect integrates the additional logic to support the new instructions and reflects the new capabilities in its **System Design** pane. Changes are also reflected in the processor's build-configuration registers (BCRs), which can be used by embedded software or a debugger to detect the processor configuration.

## 1.7 Fast Configurability

The pre-defined configurable components supplied in IP libraries and the ARChitect tools speed up considerably the task of designing and verifying fully functional RTL code of your core processor, optimized for your specific application. You can customize and configure a system on-the-fly and rebuild it to see the results of your changes in minutes.

## 1.8 Floating Point Extensions

The ARC EM architecture supports an optional IEEE-754-compliant single precision floating-point unit (FPU), for double-precision support a double precision assist package will be provided. The FPU is designed to integrate cleanly into the execute pipeline of the ARC EM core using the APEX interface. It offers better performance than a co-processor implementation.

The FPU has direct access to core registers and all its source-destination dependencies are inter-mixed into the global dependency logic. This high-level of integration results in a more efficient implementation and eliminates unnecessary stall cycles when mixing integer and floating-point instructions.

### Related Topics

- [Chapter 4, “Configurability”](#)
- For more information on the BCRs, see the *ARCV2 ISA Programmer's Reference*.

- For more information on ARChitect, see “[ARChitect Configuration Tool](#)” on page 51 or the *ARChitect User’s Guide* in the *ARChitect Tools Bookshelf*.
- For information on system integration, see [Chapter 18, “Integration and Verification”](#).



# 2

## Release Notes

This chapter provides specific information about this release of the ARC EM series.

### 2.1 About This Release

The processor is delivered as Verilog HDL in a bundle IP library containing other IP libraries and documentation. The HDL can be configured and output from the ARChitect IP configuration tool, along with simple tests and reference scripts for simulation, synthesis, floorplanning, and power analysis (see “[ARChitect Configuration Tool](#)” on page 51). The IP library provides several templates for each core in the series to use as a basis for modification into your own design.

#### 2.1.1 Deliverables

The following key component .iplib files are provided in the IP-library bundle, and can be loaded in ARChitect depending on the licenses purchased.

**Table 2-1 IP Libraries Included in the Delivery**

| Filename       | Description                                                                                                                                                                                                                |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ARCV2EM_bundle | Verilog HDL, simulation scripts, and self-checking test code for ARC EM<br>Includes the ARC Memory Subsystem (MSS), and templates for ARC EM4 and ARC EM6, ARC EM5 and ARC EM7 processor cores in specific configurations. |
| ARCV2_FPGA_RDF | FPGA RDF Libraries.                                                                                                                                                                                                        |

#### 2.1.2 Documents

The following ZIP archives are provided in the IP library bundle.

- **ARC\_EM\_Series\_Bookshelf\_pdf.zip**

Contains this *Databook*, *ARCV2 ISA Programmer's Reference*, *ARC EM Integration Guide*, *RDF Release Notes and RDF User Guide* with technical details and information for programming DesignWare ARC EM processors.

## 2.2 Supported Platforms

The ARC EM IP library is compatible with the following products and platforms:

**Table 2-2 Supported Platforms**

| Supplier        | Product                              | Version                      |
|-----------------|--------------------------------------|------------------------------|
| Synopsys        | ARChitect                            | I-2014.03-SP1-1 or later     |
| Synopsys        | ARC Synopsys RDF                     | 4.0.10 or later              |
| Synopsys        | ARC SEIF RDF                         | 3.0 or later                 |
| Synopsys        | ARC CGR RDF                          | 1.0.5 or later               |
| Synopsys        | ARC SMS RDF                          | 1.0.0 or later               |
| Synopsys        | ARC ASIC RDF                         | 5.0 or later                 |
| Synopsys        | MetaWare Development Toolkit         | J-2014.06 or later           |
| Synopsys        | ARC Debug (CJTAG)(separate download) | 2.1                          |
| Synopsys        | xCAM                                 | 3.2.13 c                     |
| Synopsys        | VCS                                  | 2013.06-4 or later           |
| Microsoft       | Windows 7                            | 2002 Service Pack 2 or later |
| RedHat          | Enterprise Linux                     | 4.x, 5.x,6.x                 |
| Cadence         | IUS                                  | v13.10.006                   |
| Mentor Graphics | ModelSim                             | v10.2 c                      |
| Synopsys        | TCF Library                          |                              |
| Synopsys        | EM Starter Kit                       | v2.0                         |
| Synopsys        | VTOC version                         | v4.3.11 or later             |



### Note

- a. For Cadence simulation, only ncverilog/ncelab/ncsim flow is supported. ncverilog flow is not supported.
- b. Due to modifications applied to the modelsim.ini file by our design flow, use of the MODELSIM environment variable is not supported at this time.
- c. For the Verilator flow described in the APEX Wizard User's Guide, a Verilator package containing the correct versions of Verilator and Perl can be downloaded from <http://www.synopsys.com/IP/ProcessorIP/Pages/designware-arc-opensource.aspx>

The ARC EM APEX is compatible with the following products and platforms:

**Table 2-3 APEX Supported Platforms**

| Supplier                                                                                                                                                                                                             | Product       | Version                 |
|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|---------------|-------------------------|
| GNU                                                                                                                                                                                                                  | GCC           | 4.8-R4 or later (Linux) |
| Microsoft                                                                                                                                                                                                            | Visual Studio | 2008 or later           |
| <a href="http://www.synopsys.com/IP/ProcessorIP/Pages/designware-arc-opensource.aspx">http://www.synopsys.com/IP/ProcessorIP/Pages/designware-arc-opensource.aspx</a><br>(Perl is included in the Verilator Package) | Perl          | v5.8.7 (Windows)        |
| Perl                                                                                                                                                                                                                 | Perl          | v5.8.3 or later (Linux) |
| <a href="http://www.synopsys.com/IP/ProcessorIP/Pages/designware-arc-opensource.aspx">http://www.synopsys.com/IP/ProcessorIP/Pages/designware-arc-opensource.aspx</a>                                                | Verilator     | 3.85                    |
| Sourceware.org                                                                                                                                                                                                       | Cygwin        | 1.7.17 or later         |

**Note**

For the Verilator flow described in the APEX Wizard User's Guide, a Verilator package containing the correct versions of Verilator and Perl can be downloaded from:

<http://www.synopsys.com/IP/ProcessorIP/Pages/designware-arc-opensource.aspx>

## 2.3 Supported Standards

For details on how the standards listed in this section are implemented in ARC EM, see [Chapter 9, "External Interfaces"](#).

### AHB

AMBA 3 AHB Interface Standard - 2008

### AHB-Lite

AMBA 3 AHB-Lite Interface Standard - 2008. The peripheral bus is 100% AHB-Lite compliant.

### BVCI

Virtual Component Interface Standard Version 2 (On-Chip Bus 2 2.0) August 2000

Development Working Group

Member Review Version 24Aug00

OCP International Partnership, [admin@ocpip.org](mailto:admin@ocpip.org)

<http://www.vsi.org/>

### JTAG

IEEE Std 1149.7-2009 and 1149.1-2001

<http://www.ieee.org/>

## 2.4 Release Change Information

### 2.4.1 New in Release 2.0

- Support for UAUX. See [Chapter 16, “UAUX Interface”](#)
- Support for FPU
- Support for Performance Monitoring. See [Chapter 15, “Performance Counters”](#)
- Support for Digital Signal Processing (DSP). See [Chapter 11, “ARC EM DSP Features”](#)
- Improvements to the APEX Interface. A new document, *APEX Databook*, is now included in the DesignWare® ARC EM Series Bookshelf. This APEX databook has complete information on designing and implementing ARC Processor Extensions (APEX). See [Chapter 9, “External Interfaces”](#).
- Low power features for power saving. See [“Power-Saving Modes”](#) on page 306
- Support for RTT. See [“Real Time Trace Interface”](#) on page 161
- Support for Multi-Core IP. See [Chapter 13, “Multi-Core Option”](#)
- Support for `-fpu_dp_assist` in Floating-Point Unit (FPU)
- Removed support for chip ID
- All RDF documents have been merged into two documents, *ARC Synopsys ASIC RDF* and *ARC FPGA Synthesis Flow* and is now part of the ARC EM Bookshelf.

## 2.5 Release History

### 2.5.1 Changes in Release 1.1a

- Bug fixes. See [“Known Issues”](#) on page 49 for the procedure to access the fixed and known issues.
- A build configuration register, `CPROT_BUILD`, is added to the processor configuration. This register indicates if code protection is enabled in the processor. For more information about this register, see the *ARCV2 ISA Programmer's Reference Manual*.
- Generation of TCF file that can be used to configure software tools. When you build a processor using the ARChitect tool, the TCF file is generated and saved in the following location:

`project_directory/build/tool_config`

This TCF file encapsulates into one file processor-specific information such as processor family, core version, register file, endian, and extension options that the MetaWare toolset needs to support a given processor build. For more information about the TCF file, see the *C/C++ Programmer's Guide* for the MetaWare Compiler packaged as part of the MetaWare toolset.

- Updated support for Verilator translation to C++
- A new document, *EIA Cookbook*, is now included in the DesignWare® ARC EM Series Bookshelf. This *EIA Cookbook* describes the basic steps in optimizing software application code for ARC EM family processors. It emphasizes the use of extensions created using Extension Interface Automation (EIA).



## 2.5.2 Changes in Release 1.1

- Revised interrupt architecture with up to 16 priority levels. Interrupt and exception vector table contains target address only. See [Chapter 7, “Interrupts”](#) and the *ARCV2 ISA Programmer’s Reference*.
- Save and restore of multiple registers on interrupt and exception entry and exit. See [Chapter 7, “Interrupts”](#) and the *ARCV2 ISA Programmer’s Reference*.
- Optional register banks. See [“Register File”](#) on page 62.
- One- and two-cycle ICCM. See [Chapter 5, “Memory Components”](#).
- Stack range checking. See [Chapter 6, “Memory-Protection Unit”](#).
- Code protection to prevent unauthorized access. See [Chapter 6, “Memory-Protection Unit”](#).
- Memory protection (MPU). See [Chapter 6, “Memory-Protection Unit”](#) and the *ARCV2 ISA Programmer’s Reference*.
- Compact Two-Wire JTAG interface. See [“Two-Wire JTAG Interface \(IEEE 1149.7\)”](#) on page 227.
- Memory-based SmaRT trace buffer. See [“SmaRT”](#) on page 262.
- Multiple sleep modes. See [“System Level Low Power States”](#) on page 307.
- Radix-4 enhanced divider option. See [“Instruction Set”](#) on page 63 and [“Divide/Remainder Instruction Timing”](#) on page 141.

The hardware ID core version is 0x41. The MetaWare compiler series option is `-arcv2em`, and the MetaWare compiler core option is `-core1`.

## 2.5.3 Changes in Release 1.0a

Added support for DesignWare ARC Floating Point (FPX). Note that FPX must be downloaded and licensed separately.

The hardware ID core version is 0x40. The MetaWare compiler series option is `-arcv2em`, and the MetaWare compiler core option is `-core0`. You must specify both options to compile for this release.

## 2.5.4 Changes in Release 1.0

This is the initial release of DesignWare ARC EM.

The hardware ID core version is 0x40. The MetaWare compiler series option is `-arcv2em`, and the MetaWare compiler core option is `-core0`. You must specify both options to compile for this release.

## 2.6 Known Issues

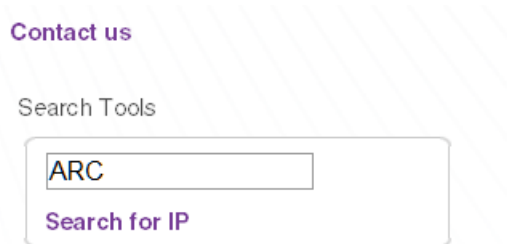
You can find up-to-date lists of known and fixed issues online at [synopsys.com](http://www.synopsys.com).

1. Go to <http://www.synopsys.com>.

- Click the **IP** tab:



- Scroll to **Search Tools**, and search for ARC:



- In the results, click the **STARs** link for your product:



- Enter your SolvNet password if requested.
- For known issues, click the **Open STARs** tab; for issues fixed in this release, click the release number:



# 3

## Getting Started

This chapter explains how to get started using the ARC EM IP library to create your designs.

### 3.1 Introduction

The DesignWare ARC EM processor is provided as an IP library file (.ipLib) that can be opened with the ARChitect configuration tool.

A master IP library contains all the core and support components that are needed to make up a complete system implementation. Extension IP libraries, such as a memory-subsystem library or a design-flow library, add functionality to a master IP library.

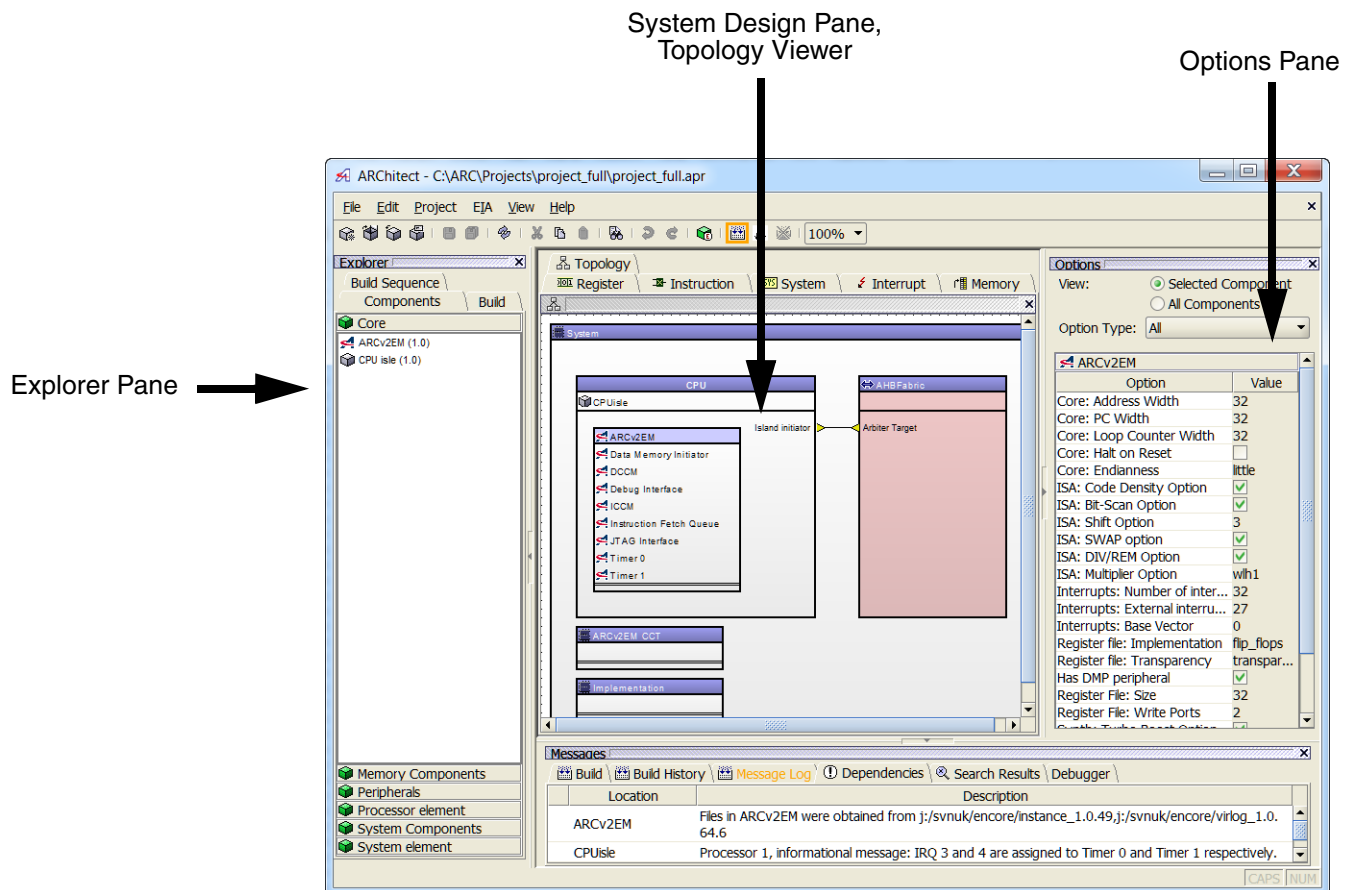
Particular installation instructions for the IP library and any third-party tools are contained in the documentation provided with the IP-library file.

### 3.2 ARChitect Configuration Tool

This section provides an overview of using the ARChitect configuration tool for configuring and building IP with the supplied system-design methodology. The ARChitect toolset provides additional tools as well; see the ARChitect PDF documentation for more information.

The ARChitect tool creates designs from an existing IP library of components. You can configure the components with simulation models and verify them in a standalone environment, and save a project as an instance of the chosen configuration. The interactive user interface provides instant feedback for key parameters of the design and allows quick optimization and architecture exploration.

With the ARChitect GUI, adding components is as easy as dragging them from the **Explorer** pane to the **System Design** pane and selecting options for them in the **Options** pane (see [Figure 3-1](#) on page 52).

**Figure 3-1 Overview of ARChitect GUI**

IP libraries contain all the components — processors, extension components, memory-system components, and peripherals — needed to produce a complete system implementation. Each IP-library component contains all the component data, including HDL source code, testbenches, test code and synthesis scripts.

The IP library itself cannot be edited, but customized extensions can be developed with the APEX tool (see [Extending the Processor](#)).

### 3.2.1 Creating a Project

Install the ARChitect tool using the supplied Linux or Windows installer. You can then launch ARChitect from the Start menu or by typing ARChitect2 in the bin directory (Linux) or ARChitect2.bat in the bin/win32 directory (Windows).



**Note** For 64-bit simulation on Linux, you must change certain environment variables to complete the installation. See the *ARChitect User's Guide* for details.

You need at least one IP library loaded in ARChitect to create a project. To enable an IP library, select **Project > Default Properties > Libraries** and navigate to the IP library.

To create a new project, select **File > New Project**, or click the **New Project** button  and select one of the configuration templates provided.

### 3.2.2 Selecting a Configuration Template

Select a template from the **Templates** list to see a summary of the template configuration in the **Description** pane. A topology view is also shown in the **Topology** viewer. Select the template closest to your design requirements as a starting point.

You can also choose the name of your project, the project directory, and its location by editing the **Name** and **Location** text boxes within this dialog box. The project and all its related files are automatically saved to the project directory created with the chosen name at the chosen location.

After you select the project name, location, and configuration template, create the project by clicking the **Finish** button. The different design views and options become populated with the design information contained within the chosen template, and the **Topology** view displays this design.

Further information on using the ARChitect tool is contained in the ARChitect online help and PDF documentation.

### 3.2.3 Adding and Configuring Components

The **Components** view (in the **Explorer** pane) displays all the components available in the IP libraries you added to your project. More information about a component is displayed when you hover the mouse pointer over the component in the **Components** view.

You can add components into the design by dragging them from the **Components** view to the **Topology** view, or by right-clicking a component in the **Components** view and selecting **Add to Design**.

When a component is selected in the **Topology** view, you can see options for it in the **Options** pane. Tool tips are provided to give more information about each option. To view the tool tip for an option, move the mouse pointer over the option.

Changing the design components and options is reflected in detail in other informational panes of the ARChitect tool. For example, you can examine the impact on gate count, memory sizes, memory address allocation, and other important aspects of the design.

The ARChitect tool suite employs a sophisticated knowledge-based hierarchy-generation algorithm, which connects the components in the topology view in a logical way to form your working design netlist.

Prior to building your design, you can run the Dependencies Checker tool , which compares your design with a knowledge database about design compatibility and interoperability of all the IP library components. This enhances your ability to quickly try different scenarios and optimize your design without having to study an extensive list of requirements for each component.

Error reporting for ARC EM is enhanced with tips on how to fix most errors, so you can immediately rectify the situation.

See the ARChitect online help or PDF documentation for more information on using the ARChitect tool.

### 3.2.4 Choosing Design Flows

The Reference Design Flow (RDF) is provided as a set of extension IP libraries. As new technologies and tools are introduced, alternate design-flow IP libraries can be installed.

To enable the design-flow IP library, select **Project > Project Properties > Paths** and add the design-flows library file (`.iplib`) to the list of IP libraries.

After the library is enabled, particular design-flow options are provided in the Build Sequence and Components views of the Explorer pane. Some flows require several RDF IP libraries or support components. See the *ARC® Synopsys ASIC RDF UserGuide* for detailed information; see [Chapter C, “Synthesis and Physical Implementation”](#) for general information.

### 3.2.5 ARChitect Command Line

Most features of the interactive GUI mode are also available as text commands. The commands can be invoked from the command line or a configuration script. To invoke the ARChitect tool in command-line mode, use the following command:

```
ARChitect2 -cl architect_options ip_library_options
```

- Specify any ARChitect command-line options (*architect\_options*) before the *ip\_library\_options*. To display the available *architect\_options*, use the `ARChitect2 -h` command. These options are also listed in the *ARChitect User's Guide*, viewable in PDF or from the ARChitect Help menu.
- Specify any IP library options (*ip\_library\_options*) after the ARChitect command-line options to configure the core with different or optional features. To display the IP library's command-line options, use the `ARChitect2 -d` command. A summary of the most important IP-library options for ARC EM is included in [Appendix B, “Key IP Library Command-Line Options”](#).

You can specify a configuration template using the `-template` command-line option.

If the ARChitect tool has not enabled the DesignWare ARC EM IP library, you can specify it using the `-libraries` command-line option.

### 3.2.6 Building and Debugging with a TCF File



#### Note

The procedure to set project settings using a TCF file is supported only for 2013.09 or later versions of the Metaware Debugger.

The `.tcf` file is a tool configuration file that is generated when you build a processor using the ARChitect configuration tool using the newer versions of the ARC IP libraries. The TCF file encapsulates into one file the information that the development tools need to support a given processor build. After you have built the processor using the ARChitect tool, this TCF file is created in the following location of your ARChitect processor project folder: `project_directory/build/tool_config/arc.tcf`.



#### Note

The TCF file does not include any APEX information even if you have created APEX in your processor. The unique name for a core will be prefixed before the file name. For example, `AAarc.tcf`. In case of multicore multiple tcf files will be generated, one for each core.

The `.tcf` files are generated by the Tool Configuration Component in your design. For the older projects, this component is automatically instantiated. The Tool Configuration component is delivered in a separate

library that comes with the bundle for the processor. If this library is not present in the bundle with the processor, ARCHitect will fall back to an older version that comes with ARCHitect.

As tools are updated more frequently than processors you may be asked to explicitly load a newer version of the Software Configuration iplib than the iplib provided with the processor.

For more information on the tools, see the *MetaWare Debugger User's Guide*, *MetaWare C/C++ Programmer's Guide* and *MetaWare IDE User's Guide*.

### 3.2.7 Simulation Tool Support

You can use several simulation tools with ARC EM:

- VCS®
- Cadence® Incisive® Design Team Simulator (NC-Sim)
- ModelSim® Verilog®
- DesignWare® ARC® xCAM
- DesignWare® ARC® nSIM (see [“MetaWare Debugger”](#) on page 337)

## 3.3 Compiler and Debugger Support

The DesignWare MetaWare Development Toolkit provides complete support for compiling, assembling, linking, and debugging code for ARC EM.

### 3.3.1 Compiling C/C++ Code for the DesignWare ARC EM Series

To generate code that supports the DesignWare ARC EM Series, use the `-arcv2em` series option and the desired `-core*` option, for example:

```
ccac -arcv2em -core1 main.c task.c setup.s -o main.out
```

Where:

|                       |                                                                                                                                              |
|-----------------------|----------------------------------------------------------------------------------------------------------------------------------------------|
| <code>-arcv2em</code> | Specifies building for a DesignWare ARC EM processor. Additional options are available; see the <i>MetaWare C/C++ Programmer's Guide</i> .   |
| <code>-core1</code>   | Specifies building for DesignWare ARC EM processor version 1.1. See <a href="#">Chapter 2, “Release Notes”</a> for your core-version option. |
| <code>.c files</code> | C-language input files.                                                                                                                      |
| <code>.s files</code> | Assembly language input files. These files are ignored by the compiler and passed to the assembler.                                          |
| <code>-o</code>       | Specifies output executable name.                                                                                                            |
| <code>main.out</code> | Executable output file created by the linker.                                                                                                |

See *MetaWare® Development Toolkit for ARC® Getting Started* for further information on compiler, assembler, and debugger invocation.



### 3.3.2 Assembling for the DesignWare ARC EM Series

The following example generates code from assembly files for v1.1 (-core1):

```
ccac -Hnocrt -Hnolib -arcv2em -core1 main.map main.s -o main.out
```

Where:

|           |                                                                                                                                                |
|-----------|------------------------------------------------------------------------------------------------------------------------------------------------|
| -Hnocrt   | Specifies not to include any C run-time libraries.                                                                                             |
| -Hnolib   | Specifies not to include any MetaWare library code.                                                                                            |
| -arcv2em  | Specifies assembling for an ARC EM processor.                                                                                                  |
| -core1    | Specifies assembling for DesignWare ARC EM processor version 1.1. See <a href="#">Chapter 2, “Release Notes”</a> for your core-version option. |
| map files | MetaWare linker map files to set section addresses.                                                                                            |
| .s files  | Assembly language input files. These files are ignored by the compiler and passed to the assembler.                                            |
| main.out  | Executable output file created by the linker.                                                                                                  |

See *MetaWare® Development Toolkit for ARC® Getting Started* for further information on compiler, assembler and debugger invocation.

### 3.3.3 MetaWare-Debugger Support

The MetaWare debugger is tightly integrated into the configuration toolset and works seamlessly with all supported simulators. The debugger can be used in standalone mode or as part of the integrated simulation environment.

For details on invoking the debugger, see “[MetaWare Debugger](#)” on page 337.

## 3.4 Licensing Your Software

Before running Synopsys tools, you must have installed and configured the Synopsys Common Licensing (SCL) software, retrieved your license key file, and defined the license file environment variable. For detailed information about SCL installation and setup, see the *Synopsys Licensing Quickstart Guide* at the following Web address:

<http://www.synopsys.com/Support/LI/Licensing>

## 3.5 Where to Go from Here

Suggested next steps:

1. Launch ARChitect.
2. Point to the location where you downloaded the IP library. If this is your first IP library, you are prompted to locate it.
3. Choose a template and begin adding and changing components to create your design.

Consult the ARChitect online help or PDF documentation for information on using ARChitect.

# 4

## Configurability

This chapter describes the user-configurable options of the ARC EM processor core.

See [Chapter 18, “Integration and Verification”](#) for information on configuring the simulation environment, including simulation options and additional features associated with the reference design flow.

### 4.1 Overview

The process of configuring the ARC EM processor is tightly coupled to the ARChitect configuration tool. A beginner might prefer to work in the graphical framework (GUI mode), which provides all the information and control options visually with a set of menus and windows. ARChitect also incorporates extensive configuration checking features, which provide immediate feedback and accelerate the process of configuring and optimizing a core to the desired specifications. For information on getting started with ARChitect, see [“ARChitect Configuration Tool”](#) on page 51.

The same configurability options are also available in a command line mode of ARChitect. This mode is usable for an advanced user or for making minor changes to an existing working design. The options in this chapter are described by their command-line name; it is quite simple to find the equivalent option in the GUI. Each GUI option has a brief text description, including any command-line arguments, which may be viewed in a tool tip by placing the mouse pointer over the specific option.

While the goal of this process is to configure and verify an ARC EM processor core for embedding in a larger design, the process is also aimed at configuring a suitable test environment for the standalone core verification. This verification and the back-end flow features make up a sizable part of the configurability options.

When a design is created and configured, ARChitect saves all the command line options of the configuration in the script file `build_configuration.txt`. This file can be later used as an argument file for using ARChitect in command-line mode.



#### Caution

Most optional components add specific auxiliary registers for interacting with the component. When the components are configured, all the hardware and features associated with them are present, and when they are not configured, the associated hardware and features are not present. Trying to access a resource that is not configured might result in unpredictable behavior.

## 4.2 Configuring an ARC EM Processor

The ARC EM processor consists of several functional components. A configuration includes optional components and optional features of the components.

### 4.2.1 Adding an Optional Component

The easiest way to add a component is by dragging a component from the **Components** pane into the **Topology** pane in the ARCHitect GUI. ARCHitect handles this internally using a `create` command and resolves the design hierarchy and connectivity of the component.

### 4.2.2 Specifying Optional Features of Components

You can specify optional features of the components in the Options pane of the ARCHitect GUI or using command-line options. Sections 4.3 and 4.4 describe the most important options available in the IP library.

## 4.3 Options for Build Components



### Caution

This section presents highlights of the most important build options. View the ARCHitect tool tips by hovering over the options, or use command-line option `-d` for arguments, defaults, notes, and important cautions.

### 4.3.1 Instance Features

- `-arcv2em`: specifies that the target core is ARC EM.
- `-arc_num`: specifies a unique number for the core, to be used as part of the core ID. This parameter is also part of the JTAG address of the specific core.
- `-unique_name`: specifies a unique module name.

This option adds a prefix to each module name in the generated RTL to make the names unique. In the case of a multiprocessor template, the prefixes for two CPUs default to AA and BB.



### Note

This option produces unique names at the level of `cpu_isle` and below. It does *not* produce unique names for the reference arbiter and memory system.

### 4.3.2 Implementation Options

- `-addr_size`: determines the address-bus width throughout the core.
- `-pc_size`: determines the addressable code space. This option impacts PC-related hardware. This value cannot exceed `-addr_size`.
- `-lpc_size`: size of the loop-counter register, which changes the maximum number of zero-overhead loop iterations in one instance of a loop. This value cannot exceed the value of `-pc_size`.
- `-byte_order`: the ARC EM architecture supports big- and little-endian byte orientation in memory.

- **-turbo\_boost**: this option offers an implementation trade off between the longest critical path and the maximum clock speed by propagating one or more paths over more than one clock. This is not a multi-cycle path but a synchronous design with more clocked stages. Enabling this option results in a faster achievable clock speed through synthesis. See [“Two-Cycle Branches”](#) on page 138.

The option is useful for applications with a strict requirement for high clock speed. It does not guarantee higher overall performance in all circumstances.

### 4.3.3 Reset Options

- **-halt\_on\_reset**: determines if the processor starts executing instructions immediately upon removal of the reset signal. Some designs require that the ARC EM processor remain in the halted state to allow access for an external host to load memories or initialize other components in an orderly manner.
- **-ic\_disable\_on\_reset**: see [“Instruction Cache”](#) on page 65.

### 4.3.4 Interrupt Options

- **-number\_of\_interrupts**: the total number of interrupts available to the core. Some interrupts are allocated statically to a specific interrupt line (for example, timer interrupts). For more information on Interrupt and register-file options, see *DesignWare ARCv2™ ISA Programmer's Reference Manual*.

This option specifies the highest interrupt line available to the core regardless of interrupt type.

- **-external\_interrupts**: the total number of interrupt pins available for external system components. This parameter must be less than the total number of interrupts.
- **-intvbase\_preset**: the initial address of the interrupt vector table after reset specified in 1 KB blocks (the value is the target address divided by 1024). For example, writing 4 to this field causes executing the reset handler at address 4096. The ARC EM processor starts executing instructions from the 32-bit address at this location (reset handler address) immediately after removal of the reset signal (if the processor is not kept in the halted state).
- **-number\_of\_levels**: the number of interrupt priority levels.
- **-firq\_option**: enables the fast-interrupts option, (priority level 0 interrupts), which uses an alternate register bank (if configured) instead of saving the context to memory.

### 4.3.5 Bus and External Memory

- **-dc\_uncached\_region**: see [“Data Cache”](#) on page 84.
- **-off\_chip\_ram**: specifies that the simulation testbench should include an off-chip memory. Note that this option is for internal usage and by itself does not add a data path to external memory.
- **-mem\_bus\_option**: the ARC EM supports four bus protocols for accessing external memory (BVCI and different versions of AHB), see [Chapter 9, “External Interfaces”](#).

- `-mem_bus_reg_interface`: selects whether the address, command, and data are registered at the memory-bus interface. Select this option to minimize the critical path at high clock speeds.

**Note**

Use caution when configuring the memory bus interface as unregistered BVCI with an instruction-fetch queue. The design of the instruction-fetch queue is optimized for the highest performance through the use of an incremented address rather than a registered address.

If the configuration includes an instruction-fetch queue and a data cache or a data-memory initiator and a unified external memory through an asynchronous on-the-fly external arbiter, the external arbiter logic might create an asynchronous path between the internal prefetch logic and data-request logic.

To resolve this potential issue, use one of the following methods:

- For best performance, the arbiter can remain asynchronous, but you must assure that this on-the-fly arbiter gives priority to the data request over the instruction-fetch request.
- Alternatively, the arbiter can break the potential asynchronous loop by incorporating a two-cycle synchronous arbiter. The arbiter can then register the requests in the first cycle but provide an acknowledge signal only in the next clock cycle. This solution avoids the potential asynchronous loop at the expense of a slightly higher latency.

- `-has_dmp_peripheral`: the option enables the redirection of load/store accesses to one segment (1/16) of the addressable space to a dedicated peripheral bus. This offers high system integration and reduces overall system cost.

Access to the peripheral space is always uncacheable.

- `-per_bus_option`: the ARC EM supports two bus protocols for accessing the peripheral space, when enabled (BVCI and AHB-Lite), see [Chapter 9, “External Interfaces”](#).
- `-per_bus_reg_interface`: selects whether the address, command, and data are registered at the peripheral-bus interface. Select this option to minimize the critical path at high clock speeds.

### 4.3.6 Register File

- `-rgf_impl`: the register file may be synthesized with standard flip-flop cells or specified as a memory macro. If a memory macro is selected, then ARChitect instantiates a simulation model, which must later be replaced with SRAM modules from the vendor’s library. For more information on Interrupt and register file options, see *DesignWare ARCv2™ ISA Programmer’s Reference Manual*.

**Note**

Certain register bank configurations can be implemented only as flip-flops. ARChitect notifies you when such implementation occurs, and you select a macro implementation.

- `-rgf_num_regs`: the ARC EM offers the option to reduce the register file to 16 registers to reduce cost. The application must be compiled specifically for the reduced implementation, consult the *MetaWare C/C++ Programmer’s Guide* and *MetaWare ELF Assembler User’s Guide* for details.

- `-rgf_wr_ports`: the register file may be configured with one or two write ports. The dual write-port implementation adds to cost but might increase performance for certain applications.
- `-rgf_num_banks`: selects the number of register-file banks.
- `-rgf_banked_regs`: selects the number of registers that are replicated in the second register-file bank.

### 4.3.7 Instruction Set

The instruction set of the ARC EM was enhanced over previous DesignWare ARC processors by offering optional instruction packages without the need to customize the design by means of additional APEX instructions and hardware.

Six groups of optional instructions are packaged with the standard core library. These groups were selected based on customer requests and profiling of selected applications and benchmarks. Adding these options may cause a slight or moderate increase in the gate count of the core, so they were not included in the baseline ISA.

Select only the packages and options necessary for your target application.

The *ARCV2 ISA Programmer's Reference* contains a list of optional instructions available in each ISA extension pack.

- `-bitscan_option`: adds normalize, find first/last set instructions.
- `-code_density_option`: adds 16-bit instructions to increase code density; see the *ARCV2 ISA Programmer's Reference*.
- `-div_rem_option`: adds divide and remainder instructions. These are implemented as multi-cycle non-blocking instructions with out-of-order completion. The ARC EM pipeline control handles all aspects of committing these instructions in order and retiring their results when they complete. Two implementation options are available:
  - `radix2`
  - `radix4_enhanced`
- `-mpy_option`: the multiplier ISA option allows selection between several multiplier configurations to trade off performance with silicon area.

Some multiply and divide options share datapath resources to minimize total area. The enhanced radix-4 division option is only available with the faster one and two-cycle multiply options.

Cycle count for the different configurations is explained in [Chapter 8, "Cycle-Timing Characteristics"](#).

- `-has_fpu`: adds the optional IEEE-754 compatible FPU as a component tightly integrated with the integer pipeline. For detailed information all floating-point options and the instructions added for each selection, please refer to the *ARCV2 ISA Programmer's Reference*.
- `-shift_option`: the shift ISA option adds shift and rotation instructions including: ASL/ASR (multiple shifts), ASR8, ASR16, LSL/LSR (multiple shifts), LSL8, LSL16, ROR/ROL, ROL8, ROR8.
- `-swap_option`: the optional SWAP instruction is useful for converting data between big- and little-endian byte orientation.

### 4.3.8 Floating-Point Extensions

The Floating-Point Unit (FPU) provides hardware-accelerated floating point calculations, compliant with the IEEE 754-2008 specification. The FPU becomes an integral part of the ARC EM execution pipeline with full dependency checking and operand bypass capabilities. It is not connected through the extension interface.

#### FPU Features

- Hardware support for single-precision or double-precision operations
- Add, subtract, absolute, negate, compare, and multiply support, including multiply-add and multiply-subtract
- Support for various `float to int`, `int to float` operations
- Optional divide and square-root support
- All other functions supported via the MetaWare floating point emulation library

#### Implementation Targets

- Fully pipelined hardware to allow for single-cycle issue where possible
- Power-saving features on all data paths and intermediate registers

#### Configuration Options

- `-has_fpu: true, false` - Enables MUL/ADD/SUB/ABS/NEG/CMP acceleration along with conversion functions
- `-fpu_dp_assist: true, false` - Enables double precision operations. Only available if `has_fpu` is true.
- `-fpu_div_option: true, false` - Enables additional divide and square-root acceleration.
- `-fpu_fma_option: true, false` - Enables fused multiply-add and multiply-subtract instructions.
- `-fpu_mas_cycles: 1, 2` (1 being default) - Controls whether multiply, add, subtract, and compare take one or two cycles, and whether fused operations take two or four. This option is present to allow higher clock speed at the expense of efficiency.
- `-fpu_div_cycles: 1, 17` - Controls divide and square root gate count. Allows user to trade off area and clock speed for the floating point divide/square root hardware. The 14-cycle version is smaller and will synthesize to a faster clock frequency. (Note that the 14-cycle version will actually take 17 clock cycles from instruction fetch to register writeback).



#### Note

All floating point instructions configured to be more than one cycle will stall a subsequent floating point instruction in the pipeline. This means, for instance, that if the core is configured to implement a 17-cycle floating point divide instruction, only one floating point divide instruction can be executed per 17 clock cycles. Back-to-back floating point instructions will not be pipelined.



- For detailed information about floating-point instructions and registers, see the *ARCV2 ISA Programmer's Reference*.



For the configuration options of DSP, MCIP, Performance Monitoring, see the respective chapters.

### 4.3.9 Closely Coupled Memories

ARC EM processors may include local closely coupled memories for data (DCCM) or code (ICCM). Two ICCM interfaces (ICCM0 and ICCM1) and one DCCM interface are available. Code executing from the closely coupled memories yields the highest performance at the expense of added cost of silicon. A good optimization strategy is to use smaller CCMs for critical code and external memory for all other code and data.

ICCM0 and DCCM have an access time of one clock cycle. ICCM1 is accessed in two clock cycles without any special precautions. You can mix ICCM0 and ICCM1 as required by the application.

ARChitect instantiates simulation models for the CCM components. These components must later be replaced with SRAM modules from the vendor's library.

While the primary use of an ICCM is for code storage, note that it might also be accessed by load/store instructions. This feature can improve integration at a slight performance penalty due to conflicts in accessing the same memory module. This option is very attractive for microcontroller-replacement applications, where integration and cost are strict and compute performance is not very demanding.

- DCCM options:
  - `-dccm_dmi`: enables external access through the DMI (direct memory interface) port
  - `-dccm_size`: size of DCCM memory in bytes
- ICCM0 options:
  - `-iccm0_dmi`: enables external access through the DMI (direct memory interface) port
  - `-icmm0_base`: base address for ICCM0
  - `-iccm0_size`: size of ICCM0 memory in bytes
- ICCM1 options:
  - `-iccm1_dmi`: enables external access through the DMI (direct memory interface) port
  - `-iccm1_base`: base address for ICCM1
  - `-iccm1_banking`: enables memory dual-banking implementation for better performance
  - `-iccm1_size`: size of ICCM1 memory in bytes

See [Chapter 5, “Memory Components”](#) for more details on the CCM size options.

### 4.3.10 Instruction Cache

- `-ic_size`: size in bytes
- `-ic_ways`: degree of associativity
- `-ic_bsize`: line size in bytes

- `-ic_disable_on_reset`: wake-up state of the instruction cache upon removal of reset. See also [“Instruction-Cache Operation on Reset”](#) on page 103.
- `-ic_feature_level`: sets the feature level of the instruction cache; see [“Instruction Cache”](#) on page 103.
- `-ic_pwr_opt_level`: selects power-optimization options in the micro-architecture of the instruction cache.

### 4.3.11 Instruction Fetch Queue

You can add and configure the Instruction Fetch Queue to ARC EM processors using ARChitect GUI. This component has the following options:

- `-ifqueue_size`: Specifies the number of entries in the instruction fetch queue.
- `-ifqueue_burst_size`: Burst size for bus data transfers(32-bit words)

### 4.3.12 Data Cache

- `-dc_size`: size in bytes
- `-dc_ways`: degree of associativity
- `-dc_bsize`: line size in bytes
- `-dc_feature_level`: indicates locking and debug feature level (basic, lock and flush, lock and flush with advanced debug features). See [“Data Cache”](#) on page 84.
- `-dc_uncached_region`: enables the system-address-map uncached region, see [“Data Cache”](#) on page 84. Load/store accesses to this region behave the same as load-direct and store-direct.

Note that the ARC EM also supports an optional peripheral address space. Accesses to this space do not go through the cache and therefore need not be specified as uncachable. These accesses are re-directed to a dedicated peripheral bus.

### 4.3.13 Data Memory Initiator

This component does not have any options.

The component is used to create a data path from the core to the external memory bus in builds without a data cache. This path services all load and store instructions that access memory segments without an internal closely coupled memory (CCM).

### 4.3.14 JTAG Interface

- `-jtag_tclk`: defines clock rate for the JTAG interface. The value cannot exceed one half of the ARC EM core clock speed. This option is listed as an implementation option. When JTAG2Wire optional component is included, the following option is available:
  - `-jtag_node_id` - Set the value of the JTAG 1149.7 TAP controller node ID (eight-bit hexadecimal number). Used with 27 bits of the JTAG device ID to form the 1149.7 Tap-Controller Address (TCA).

The JTAG interface implements a subset of the two-wire and four-wire JTAG protocol to enable complete access to all core resources, including resources in a multiprocessor build. The two-wire JTAG is an optional interface component that complements the standard four-wire JTAG.

### 4.3.15 Debug Interface

This component does not have any options. Include this component in the GUI if you want to access the core resources with a debugger.

The debug port is used to interface external host (or debugger) to the resources available to the ARC EM core, including internal registers and the entire memory space. The debug port is accessed from the JTAG port in the actual design, but may be alternatively connected to a simulation model for verification purposes only (using the `-fast_rascal` option).

### 4.3.16 Timers

Timers are added to a core as a Processor Element in the ARChitect GUI. The following are the timers in the ARC EM core:

- Timer0
- Timer1
- Real-time Counter (RTC)

The processor timers are two independent 32-bit timers and a 64-bit real-time counter (RTC). Timer 0 and timer 1 are identical in operation. The only difference is that these timers are connected to different interrupts. The Timers cannot be included in a configuration without interrupts. Each timer is optional and when present, it is connected to a fixed interrupt; interrupt 16 for timer 0 and interrupt 17 for timer 1. The processor timers are connected to a system clock signal that operates even when the ARCV2-based processor is in the sleep state. The timers can be used to generate interrupt signals that wake the processor from SLEEP state.

The processor timers automatically reset and restart their operation after reaching the limit value. The processor timers can be programmed to count only the clock cycles when the processor is not halted. The processor timers can also be programmed to generate an interrupt or to generate a system reset upon reaching the limit value. The 64-bit RTC does not generate any interrupts. This timer is used to count the clock cycles atomically.

The RTC (real-time counter) has no options. The two timers--timer0 and timer1 have the following options:

- `-timer_0_int_level`: priority level for timer 0 interrupt on reset ; see [“Interrupt Response Times \(Interrupt Latency\)”](#) on page 144.
- `-timer_1_int_level`: priority level for timer 1 interrupt on reset ; see [“Interrupt Response Times \(Interrupt Latency\)”](#) on page 144.

For more information about the Timers, see the *DesignWare ARCV2™ ISA Programmer's Reference Manual*.

### 4.3.17 Code Protection

The ARC EM offers a feature to prevent unauthorized loads and stores from protected regions of the code space. This feature protects memory (code/software) against malware and piracy. See [Chapter 6, “Memory-Protection Unit”](#) for details.

- `-code_protect <true|false>`

Where

- `true` = include code protection

- ❑ false = do not include code protection

#### 4.3.18 Stack Checking

The ARC EM offers real-time Stack overflow and underflow checking. See [Chapter 6, “Memory-Protection Unit”](#) for details.

- -stack\_checking <true|false>

Where

- ❑ true = include stack checking
- ❑ false = do not include stack checking

#### 4.3.19 Memory Protection Unit

The ARC EM Memory Protection Unit (MPU) is an optional component for the ARCV2 EM4 and EM6 processor cores. MPU enables adding access attributes to each of the 16 memory regions. See [Chapter 6, “Memory-Protection Unit”](#) for detailed information on configuring this feature.

- -mpu\_num\_regions: number of configured memory regions (1, 2, 4, 8, or 16)

### 4.4 Optional Components for Debug-Assist

The ARC EM family offers a few debug-assist components to give the user visibility into the core activity and its interaction with external events. These features enhance software development in the actual system environment.

These debug-assist features add to the gate count and cost. In addition, complex breakpoint logic might affect the critical path, so it is advisable to reduce such logic as much as possible.

These optional components may be included for emulation or prototyping and then removed in the process of generating the final netlist. The design avoids any interaction between the debug-assist logic and the pipeline control, so adding and removing these optional components is designed to be seamless. Some software and debugger support is associated with these features, so it is important to thoroughly verify the final netlist after removing the debug components and associated software.

#### 4.4.1 Actionpoints (Advanced Breakpoint Assist)

The ARC EM architecture offers a sophisticated multi-event feature for breakpoints and watchpoints called *actionpoints*. Refer to [“Actionpoints”](#) on page 246 for details. The number of trigger events and the features of the associated logic can be defined at build time.

Note that this hardware can also be used for ROM patching. Traditional code replacement with a software breakpoint is not feasible with ROM code, but hardware actionpoints can be used to program a breakpoint on code address in the ROM space. The exception handler then emulates the replacement code.

- -num\_actionpoints: number of trigger events, see [“Actionpoints”](#) on page 246.
- -aps\_feature: feature level of the Actionpoints system, see [“Actionpoints”](#) on page 246.

#### 4.4.2 SmaRT (Small Real-Time Trace Buffer)

The ARC EM offers a simple trace buffer, which registers in real-time any change (non-sequential) in program-flow. See “SmaRT” on page 262 for details.

- `-smart_stack_entries`: determines the number of entries in the trace buffer.
- `-smart_implementation`:
  - `flip-flop` = FF-based design.
  - `memory` = memory-based design (provides better density for larger trace buffers).

#### 4.4.3 Advanced Real-Time Trace System

The ARC EM processor can be configured with an Real-Time Trace (RTT) producer module which interfaces with the ARC RTT component available in the ARC RTT IP Library.

- `-rtt_producer_type`:
  - `-small` = Trace program flow only
  - `-medium` = Add memory (load/store) access trace
  - `-Full` = Add AUX register read/write and core register write trace

## 4.5 Adding Instructions and More with APEX

You can use the ARC Processor Extensions (APEX) Wizard within the ARChitect configuration tool to add extensions, including instructions, condition-code flags, core registers, and auxiliary registers.

For more information on APEX, see [Chapter 12, “Extending the Processor”](#) or the *APEX Databook* in the *ARM EM Bookshelf*.

## 4.6 Multiprocessor Support

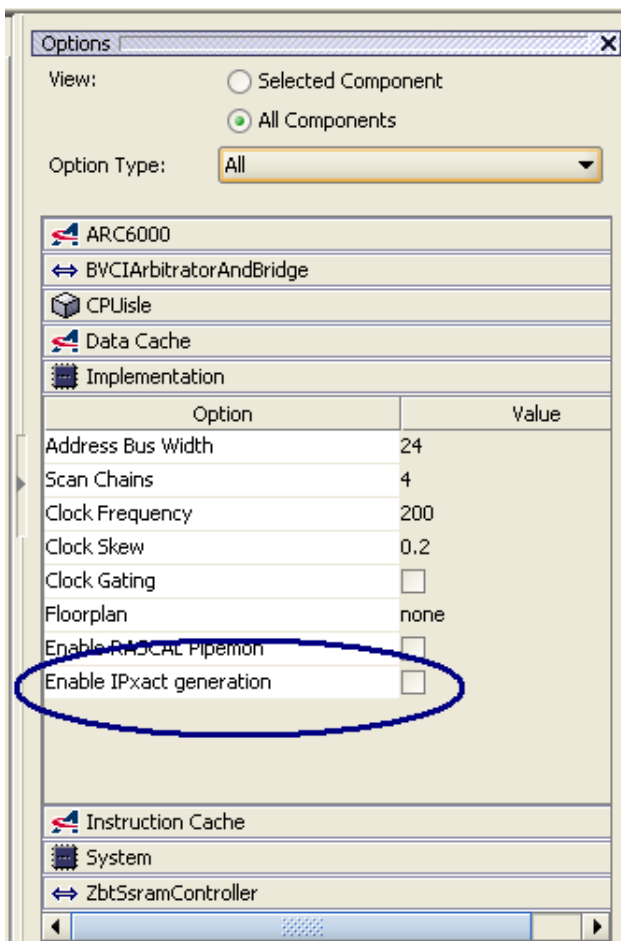
The ARC EM IP library provides support for multiprocessor configurations.



### Note

Multiprocessor designs must assign a different `-arc_num` value to each core.

- ARChitect automatically chains the JTAG port of each processor in a multiprocessor configuration.
- You can generate IP-XACT output by checking the **Enable IPxact** generation option in the **Implementation** component in the ARChitect **Options** pane. This would generate the `ipxact.xml` file in the `build` directory for input to `coreAssembler`.



## 4.7 Other Configuration Features

The ARChitect configuration tool assists in configuring the ARC EM core for your specific application. The standard package also includes extensive support for system integration and verification, including a flexible testbench and basic tests to verify the core in a proven stand-alone environment.

Refer to [Chapter 18, “Integration and Verification”](#) for configurability features of the testbench and other simulation and back-end assist.



# 5

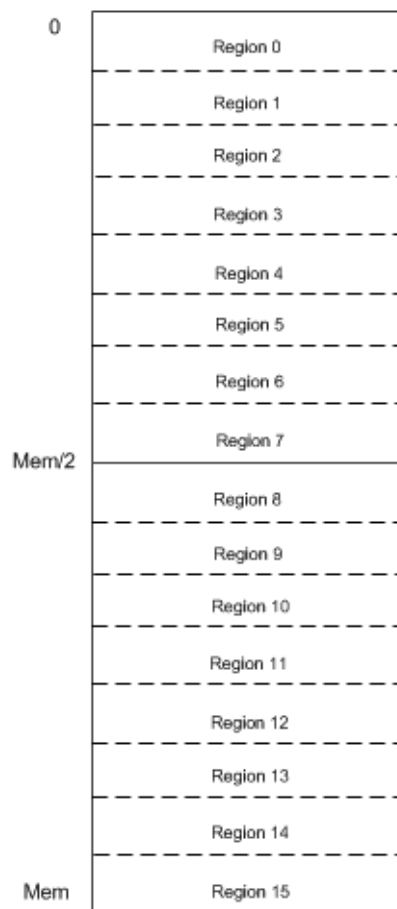
## Memory Components

ARC EM processors can be configured with a variety of memory components to accommodate a wide range of applications and to meet system level constraints and performance targets. The memory system may contain a mix of memory components, each of which may be configured or programmed dynamically according to the desired memory map.

### 5.1 Memory Map

This section describes the software view of the ARC EM memory space. Information in this section is relevant to programmers writing software for the ARC EM processor core.

The ARC EM memory space is divided into 16 equal sections referred to as *regions*. The top four bits of the memory address are used to distinguish each region, as shown in [Figure 5-1](#).

**Figure 5-1 Memory Map**

You can define the size of each region using the parameter `-addr_size` (see [Chapter 4, “Configurability”](#)). For example, with a 32-bit address space (`-addr_size=32`), the size of each region is 256 MB.

This architecture supports separate memory regions for code, data, and peripherals. The mapping and attributes of each region are flexible. Each memory region has one or more of the following attributes:

- Cacheable or non-cacheable

**Note**

For details on the non-cacheable memory locations, see [“Non-Cacheable Memory Locations”](#) on page 83.

- Mapped to a particular physical path or component:
  - Instruction Closely Coupled Memory (ICCM)
  - Data Closely Coupled Memory (DCCM)
  - Memory bus or peripheral bus

Available memory components include the following:

- Instruction Closely Coupled Memory (ICCM)
  - ICCM0 — one-cycle access time
  - ICCM1 — two-cycle access time
- Data Closely Coupled Memory (DCCM)
- Instruction Fetch Queue
- Data memory initiator
- Peripheral data Bus
- Caches
  - Instruction cache
  - Data cache

### 5.1.1 Instruction and Data Closely Coupled Memories

ARC EM supports two types of closely coupled memories: Instruction Closely Coupled Memory (ICCM0, ICCM1) and Data Closely Coupled Memory (DCCM). The ICCM and DCCM are optional, independently configurable in size, and reside on dedicated busses within the processor core. The DCCM is accessed by the processor core through load/store memory references, whereas the ICCM is accessible for instruction fetch as well as load/store references. The ability to perform load and store references to an ICCM makes the creation of a software boot loader very simple.

The CCMs are designed to hold performance critical code and data, providing the processor with fast and predictable access to fixed-size regions of memory, ideal for critical system level routines such as interrupt handlers and other time-sensitive tasks. The CCMs can coexist with other memory components (such as the instruction fetch queue and data-memory initiator) and provide an area efficient method of locking code or data into fast local memory. Power consumption may also be lower when executing from CCMs than from memories on external busses or the caches.

**Note**

DCCM is not reachable if it is mapped to the lower half of the memory space. Ensure that you write only the allowed values into this auxiliary register, as incorrect addresses do not raise any exception or error.

The ICCM0 and DCCM operate at the same clock frequency as the processor core, returning data one clock after a request is issued by the core. The ICCM1 interface is provided for configurations where one additional wait state is needed (such as when executing code from a slower flash memory).

If more than two additional wait states are needed, locate the slower memory component outside the processor core and access it through the instruction-fetch queue (instruction fetches) or data-memory initiator (loads and stores). The external bus protocol can handle wait states, based on the specific bus protocol selected for the configuration.

ICCM1 operates at half the clock frequency of the processor core. ICCM1 may utilize a dual-bank memory to compensate for the slow access time, at a slight overhead in silicon area.

ICCM1 works in combination with the pre fetch logic of the instruction-fetch queue. The words fetched from the ICCM1 are buffered in the instruction-fetch queue. This mechanism takes advantage of the prefetching and buffering capability of the Instruction-Fetch Queue (IFQ) to achieve better performance than a typical two-cycle memory.

Buffering ICCM1 code in the instruction-fetch queue creates a coherency problem for applications which modify the ICCM1 code, such as a debugger inserting a breakpoint instruction. The ARC EM architecture provides means for invalidating the IFQ by writing to the [Invalidate Instruction Cache, IC\\_IVIC](#) register.

If the build has an IFQ, the buffer size available for ICCM1 is equal to IFQUEUE\_SIZE. If the build does not have an IFQ, a fixed buffer size is used to maximize ICCM1 performance.

**Note**

IFQ is not mandatory while using ICCM1, but is optional.

The ARC EM processor core allows other initiators in the system to access its closely coupled memories through a standard target interface (BVCI or AHB-Lite). External accesses are handled with the highest priority. See [Chapter 9, “External Interfaces”](#) for more information about the ARC EM target interface.

### 5.1.1.1 Configuring ICCMs

ICCMs can be configured for different sizes. The base address is also configurable and can be assigned within regions 0 to 7. ICCMs can have an optional DMI port, and optional banking. If banking is enabled, ICCM1 uses two memories of half size, instead of using one memory.

This section describes the configuration options for ICCM0 and ICCM1.

#### Configuring ICCM0

When ICCM0 is present, the following configuration parameters are available:

- iccm0\_region     0..7 Base address
- iccm0\_size       ICCM0 capacity in bytes, power of 2 ( $2^n$ ), minimum 512 bytes
- iccm0\_dmi        false = no DMI port; true = has DMI port

**Note**

When you specify -iccm0\_dmi, a single DMI port is provided for both ICCM0 and ICCM1. However, you must explicitly enable access to it for each ICCM by selecting either -iccm0\_dmi or -iccm1\_dmi, or both. Trying to access an ICCM by an external bus transaction without enabling its DMI option results in RAZ/IOW behavior.

#### Configuring ICCM1

When ICCM1 is present, the following configuration parameters are available:

|                |                                                      |
|----------------|------------------------------------------------------|
| -iccm1_region  | [0..7] Base address                                  |
| -iccm1_banking | [0, 1] Two memory banks for better cycle performance |
| -iccm1_size    | ICCM1 capacity in bytes                              |
| -iccm1_dmi     | false = no DMI port; true= has DMI port]             |

**Note**

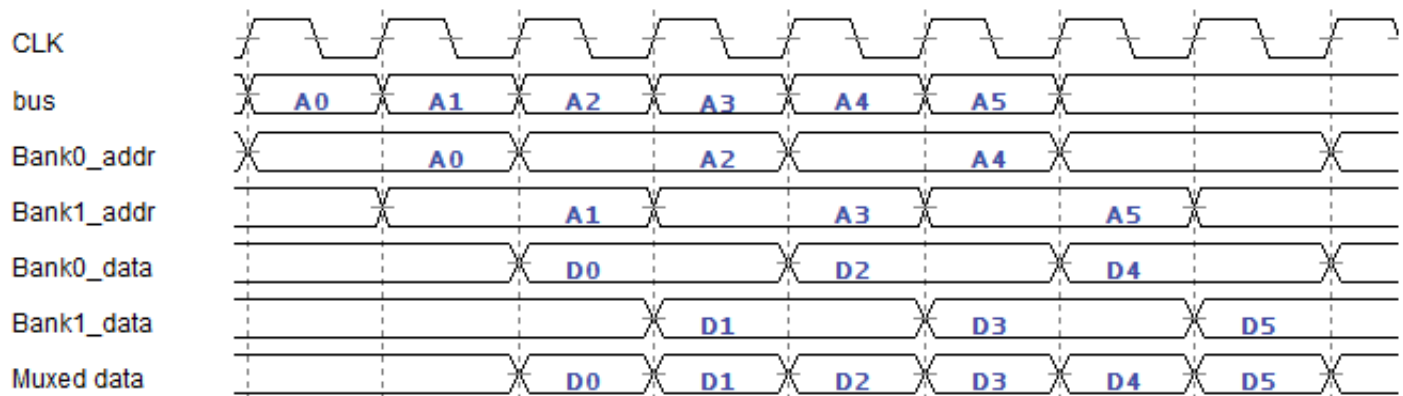
When you specify `-iccm1_dmi`, a single DMI port is provided for both ICCM0 and ICCM1. However, you must explicitly enable access to it for each ICCM by selecting either `-iccm0_dmi` or `-iccm1_dmi`, or both. Trying to access an ICCM by an external bus transaction without enabling its DMI option results in RAZ/IOW behavior

### 5.1.1.2 ICCM1 Banking

Banking can speed up the read throughput from slow memories.

ICCM1 maybe configured with a two-way banking. The timing of both banks is sequenced so that when one bank is accessed, the data from the other bank is ready as shown in [Figure 5-2](#).

**Figure 5-2 Timing Using Banking to Access Fast or Slow Memory**



The banking operation is provided when you specify the `-iccm1_banking` option.

The performance benefits of banking are available only for instruction fetch. Using DMI to access ICCM1 with banking always requires a two-cycle access.

### 5.1.1.3 ICCM Location

The base address of ICCM0 and ICCM1 is always aligned to the start of one of the 16 memory regions. The size of either ICCM could be smaller than the memory region. Configurations with `-addr_size 32` have 16 memory regions of 256 MB. Therefore, the size of the ICCM is always less than the memory region size.

**Note**

An ICCM occupies a full region even if it is smaller than the region size. If a region is accessed above the size of the ICCM which resides in it, then the address would wrap around into an actual ICCM location within that region.

---

In some configurations, the ICCM size may span across one or multiple regions. For example, in a configuration with `addr_size 16`, each memory region is 4 KB. A 16-KB ICCM would occupy four memory regions in this configuration.

You can configure the base address of ICCM0 and ICCM1 at build time using the ARChitect GUI or the following command-line options:

- `-iccm0_base`
- `-iccm1_base`

Depending on your configuration, you might be able to program the base addresses of both ICCMs at run time using the AUX\_ICCM auxiliary register. Keep both ICCMs in the lower half of the address space (regions 0 through 7). An ICCM is not reachable if it is mapped to the upper half of the memory space.

**Caution**

Exercise caution if the ICCM base address is changed dynamically. Ensure that code has been loaded into the ICCM memory before attempting to execute instructions from a newly programmed ICCM region.

**Figure 5-3 ICCM0 and ICCM1 in a 32 bit Configuration**

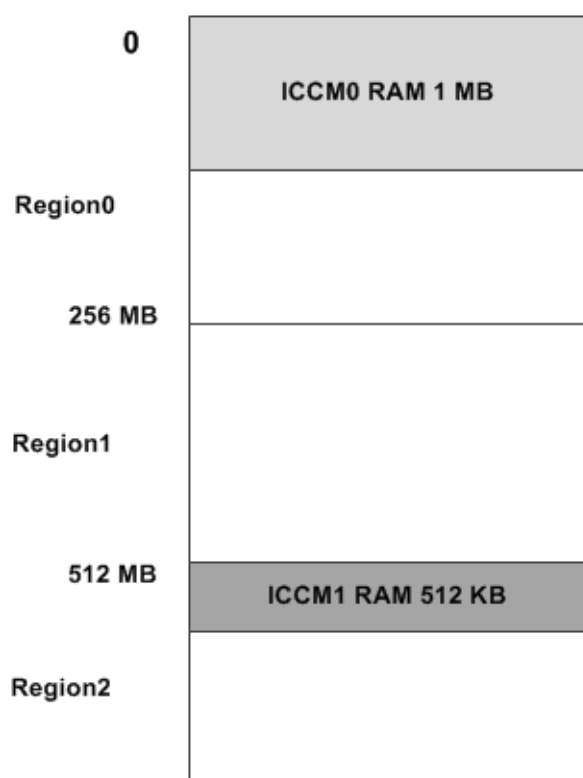
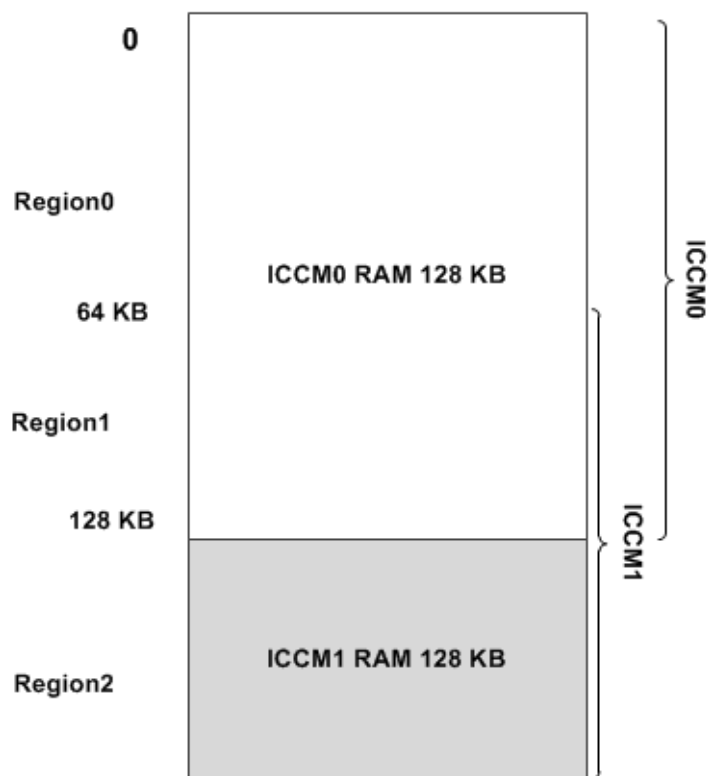


**Figure 5-4 ICCM0 and ICCM1 in a 20-bit Overlapping Configuration**

If ICCM0 and ICCM1 are present in a configuration, care should be taken to properly map the ICCMs so that their addresses do not overlap. If they overlap, ICCM0 gets exclusive access to the regions where they overlap. ICCM0 and ICCM1 can be placed only in regions 0 to 7. The base address of the ICCM must be multiple of its size. This restriction limits the regions where ICCMs can be placed especially when `addr_size` is 16 or 20 bits.

#### 5.1.1.4 Auxiliary Registers

ICCM0 and ICCM1 reside in base regions 0 to 7, determined by the value of the ICCM base address register, `AUX_ICCM`. For proper utilization of the memory space, place ICCMs such that the ICCM addresses do not overlap. If they overlap, ICCM0 prevails.

`AUX_ICCM` (0x208) identifies the base addresses of the ICCM.

**Table 5-1 ICCM0 and ICCM1 Base Address Register**

| Auxiliary Address | Name     | Access Mode | Reset Value | Description       |
|-------------------|----------|-------------|-------------|-------------------|
| 0x00000208        | AUX_ICCM | R/W         | 0x0         | ICCM base address |

For more information about `AUX_ICCM`, see the *ARCV2 ISA Programmer's Reference*.



### 5.1.1.5 DCCM Location

In configurations where all data references are satisfied by DCCM (configurations without data cache or data-memory initiator), the DCCM region always maps to the region starting at Mem/2 (region 8). In configurations where the DCCM co-exists with either a data cache or a data memory initiator component, the DCCM base address may be run-time programmable through the AUX\_DCCM auxiliary register. The DCCM must be mapped to the upper half of the address space (region 8 to region 15). The DCCM is not reachable if it is mapped to the lower half of the memory space.

Like with the ICCM, load or store addresses that fall within the DCCM region always access the DCCM memory. Addresses within the DCCM region but outside the DCCM RAM have their high bits ignored, and therefore wrap around to a location within the DCCM.

Load or store instructions are able to access the whole memory space, including the ICCMs. This feature can be used to support self-modifying code or an ICCM boot loader. It is also useful for basic low-cost control applications, which can share a single ICCM memory for code and data.

**Note**

The DMI port has the highest priority in access to ICCM or DCCM. In case of access conflicts, the processor accesses will stall until DMI is idle.

## 5.1.2 Instruction-Fetch Queue

The Instruction-Fetch Queue (IFQ) is an optional component that pre-fetches instructions using short bursts on the external memory bus. This mechanism helps hide the access latency of individual instruction fetches. This component cannot be used with an instruction cache.

The IFQ is a low-cost alternative to an instruction cache. It reduces the effective access time to code residing in external memory by prefetching it ahead of the processor pipeline.

**Note**

Only IFQ or ICACHE can be present in a build.

### 5.1.2.1 Configuring the Instruction Fetch Queue

You can add the IFQ to your core from the ARChitect GUI. The component has build-time options for queue size (1, 2, 4, 8, or 16 32-bit words) and burst size (1,2,4, or 8 32-bit words). The burst size cannot exceed the queue size.

- `-ifqueue_size` specifies the IFQ depth (number of entries in the IFQ buffer) in 32-bit words.
- `-ifqueue_burst_size` sets the maximum burst size for bus data transfers (in 32-bit words)

**Note**

IFQ fetches ahead whenever possible, consuming appreciable memory bandwidth. Unlike a cache, it does not retain prefetched code on a jump outside the range of code in the buffer. It is therefore important to scale the size of the IFQ for the typical application for best overall performance or optimize it for a critical portion of your code.

### 5.1.2.2 Invalidating IFQ

The instruction-fetch queue stores code similar to an instruction cache and may create a problem for applications which modify the code, such as a debugger inserting a breakpoint instruction. The ARC EM

architecture provides means for invalidating the IFQ by writing register IC\_IVIC. See “[IC\\_IVIC Description](#)” on page 104.

### 5.1.2.3 IFQ Operation

The IFQ generally behaves as follows:

1. The IFQ matches instruction fetch requests against valid entries in the buffer and previously issued prefetch request in progress.
2. If the IFQ finds a match in the buffer for the requested address, the requested instruction code is returned from the buffer.
3. If the requested address is being prefetched, the incoming code is returned to the core when it becomes available from the bus unit.
4. For both cases 2 and 3, the IFQ continues with a prefetch from the next sequential address.
5. If the requested address is not in the buffer or already in prefetch, the IFQ immediately issues a fetch request to the bus, corresponding to the new address.
6. In the absence of any new request from the core, the IFQ performs prefetching from the next sequential instruction address.

### 5.1.3 Data Memory Initiator

The Data Memory Initiator is an optional component of the ARC EM processor core. This component handles load and store references to the external data memory in configurations without a data cache.

The Data Memory Initiator has no buffer and therefore issues only non-burst requests on demand. It offers access to non-critical external data memory without the cost of a cache system.

### 5.1.4 Peripheral Data Bus

The peripheral region is an optional component that allows simple memory-mapped peripherals to be connected to the processor core, using a dedicated bus. The peripheral data bus can be configured as a BVCI or AHB-Lite, independently from the memory bus protocol. The peripheral region can be changed at run time through the DMP\_PERIPHERAL auxiliary register.

The peripheral region is non-cacheable. The component can be added to the core in the ARCHitect GUI. This component has the following option:

- -per\_bus\_reg\_interface: <true | false>

Where

- true = add clocked staging register inside the component.
- false = flow-through

Specify a registered interface to optimize critical path at high clock rates. Low performance implementations may skip the register stage to save area.



#### Caution

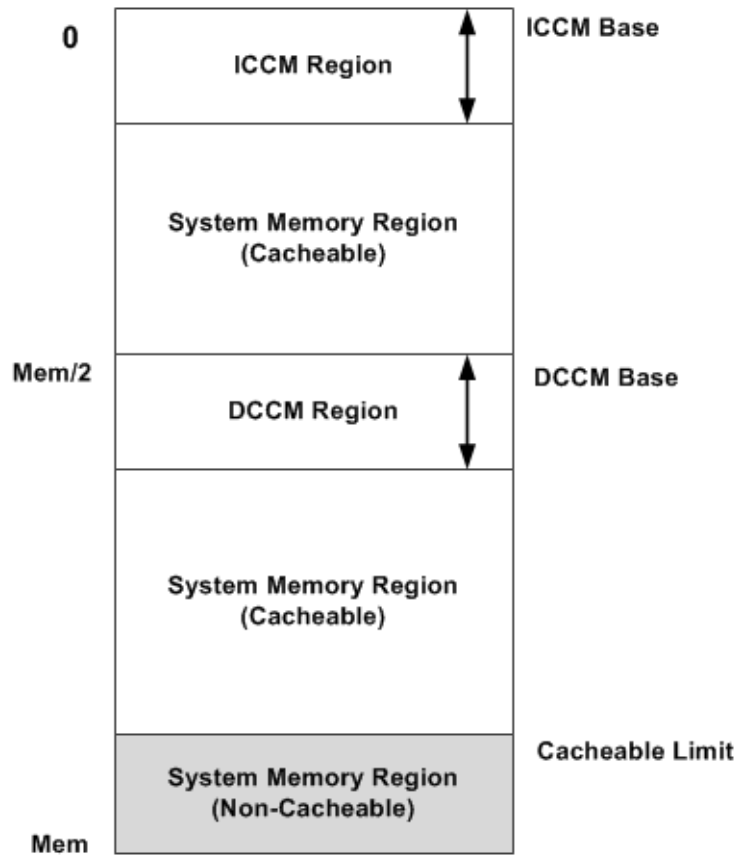
Executing code from the peripheral region may result in undefined behavior.

### 5.1.5 Non-Cacheable Memory Locations

In configurations with a data cache, the memory map may be divided into two parts: cacheable and non-cacheable. The cacheable region always starts at address zero. The AUX\_CACHE\_LIMIT auxiliary register defines the location of the lowest-numbered non-cacheable region. The non-cacheable address space extends from that region to the highest address in memory.

When this feature is enabled, the AUX\_CACHE\_LIMIT auxiliary register points by default to the highest region of memory (region 15), making it non-cacheable.

**Figure 5-5 Non-Cacheable Region**



The non-cacheable region defined by the AUX\_CACHE\_LIMIT auxiliary register only affects data fetches. Instruction fetches are not affected. Data fetches from the non-cacheable region bypass the data cache and are serviced directly from the external memory.

### 5.1.6 Memory-Map Constraints

In summary, the following rules apply:

- The base addresses of the ICCM0, ICCM1, DCCM, and data peripheral are always aligned to the start of one of the 16 memory regions. Each CCM may be smaller than a region or occupy one or more regions.

- ICCM0 and ICCM1 must reside within the lower half or first eight memory regions. The DCCM must reside within the upper half or last eight memory regions. Thus, there is never an overlap between ICCM0 and DCCM nor ICCM1 and DCCM.
- The ICCM0 base address must be a multiple of the ICCM0 size. The same constraint is true for ICCM1 and DCCM.

If two or more component address spaces overlap, the order of precedence is as follows:

1. DCCM region (highest precedence)
2. Peripheral region
3. External memory region

## 5.2 Caches

The ARC EM supports optional data and instruction cache functionality.

This section includes the following subsections:

- [Data Cache](#)
- [Instruction Cache](#)

### 5.2.1 Data Cache

#### 5.2.1.1 Cache Coherency

The ISA provides two variants of the load instruction, `ld` and `ld.di`. The `.di` suffix causes loads and stores to bypass the data cache. MetaWare C/C++ provides the `_Uncached` type qualifier to specify use of the `.di` instruction suffix for loads and stores of a variable.



#### Caution

No hardware mechanism is provided for ensuring cache coherency. Cache coherency is the responsibility of the programmer.

#### 5.2.1.2 Software-Controllable Functions

- [Data Cache Auxiliary Register Summary](#)
- [Invalidate Data Cache, DC\\_IVDC](#)
- [Data Cache Control Register, DC\\_CTRL](#)
- [Lock Data Cache Line, DC\\_LDL](#)
- [Invalidate Data Line, DC\\_IVDL](#)
- [Flush Data Cache, DC\\_FLSH](#)
- [Flush Data Line, DC\\_FLDL](#)

## Data Cache Auxiliary Register Summary

A summary of the data cache auxiliary registers is shown in [Table 5-2](#), followed by a detailed functional description of their operation. All data cache registers can be accessed using the LR and SR instructions or via the debug interface.

**Table 5-2 Data Cache Auxiliary Registers**

| Auxiliary Register Address | Register Name | Access Type | Description            | Initial RESET Value |
|----------------------------|---------------|-------------|------------------------|---------------------|
| 0x47                       | DC_IVDC       | Write Only  | Invalidate Cache       | —                   |
| 0x48                       | DC_CTRL       | Read/Write  | Cache Control Register | 0x0000 00c0         |
| 0x49                       | DC_LDL        | Write Only  | Lock Data Line         | —                   |
| 0x4A                       | DC_IVDL       | Write Only  | Invalidate Data Line   | —                   |
| 0x4B                       | DC_FLSH       | Write Only  | Flush Data Cache       | —                   |
| 0x4C                       | DC_FLDL       | Write Only  | Flush Data Line        | —                   |

## Data Cache Operation on Reset

On reset the data cache is enabled, the last cache operation is set to failed, and the direct cache RAM access is enabled. Furthermore, the data cache is invalidated — all cache lines are invalidated and unlocked, and the tag RAM is cleared.

## Invalidate Data Cache, DC\_IVDC

Address: 0x47

Access: W

**Figure 5-6 DC\_IVDC Register**

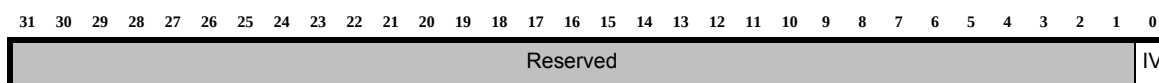


**Table 5-3 Invalidate Data Cache Flags**

| Flag Name | Description                                                                                        | Access Type |
|-----------|----------------------------------------------------------------------------------------------------|-------------|
| IV[0]     | Invalidate Data Cache: Invalidates entire data cache<br>0 – No Action<br>1 – Invalidate Data Cache | W           |

**DC\_IVDC Description**

Writing a 1 to the IV flag (bit 0) in the DC\_IVDC register invalidates and unlocks the entire data cache.

Upon a write to the IV flag, the operation of the invalidate function is determined by the Invalidate Mode (IM) flag that is found in DC\_CTRL (bit 6).

All cache lines are invalidated and unlocked, and each line entry is processed according to the state of the IM flag.

The IM flag (bit 6, DC\_CTRL) controls whether a cache line entry that is marked as being dirty is flushed, invalidated, and unlocked or simply invalidated and unlocked when a DC\_IVDC command is issued.

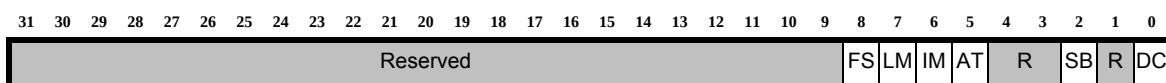
- IM = 0 (Invalidate Only) – Any write to the DC\_IVDC register invalidates all cache-line entries.
- IM = 1 (Invalidate and Flush Dirty Entries) – Any write to the DC\_IVDC register flushes all dirty lines and invalidates the entire data cache. The programmer must be aware that in many cases, setting the IM flag to 1 and initiating a DC\_IVDC command results in an indeterminate cache-cycle time stall. The cycle-time latency is dependent on the number of entries that are dirty and the efficiency of the memory subsystem.

DC\_IVDC is typically used to reset the data-cache contents upon a software process switch.

**Data Cache Control Register, DC\_CTRL**

Address: 0x48

Access: RW

**Figure 5-7 DC\_CTRL Register****DC\_CTRL Control Flags**

| Flag Name | Description                                                                        | Access Type |
|-----------|------------------------------------------------------------------------------------|-------------|
| DC[0]     | Disable Cache: Enables/Disables the cache<br>0 – Enable Cache<br>1 – Disable Cache | R/W         |

|                |                                                                                                                                              |     |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Reserved[1]    | —                                                                                                                                            | —   |
| SB[2]          | Success Bit: Success of last cache operation<br>0 – Last cache operation failed<br>1 – Last cache operation succeeded                        | R/W |
| Reserved[4:3]  | —                                                                                                                                            | —   |
| AT[5]          | Address Debug Type: Used for debug purposes for when accessing cache RAMs.<br>0 – Direct Cache RAM Access<br>1 – Cache Controlled RAM Access | R/W |
| IM[6]          | Invalidate Mode: Selects the invalidate type<br>0 – Invalidate data cache only<br>1 – Invalidate data cache and flush dirty entries          | R/W |
| LM[7]          | Lock Mode: Selects the effect of a flush command on a locked entry<br>0 – Disable flush on locked entry<br>1 – Enable flush on locked entry  | R/W |
| FS[8]          | Flush Status: Status of the data-cache flush mechanism<br>0 - Idle<br>1 – Flush operation in progress                                        | R   |
| Reserved[31:9] | —                                                                                                                                            | —   |

## DC\_CTRL Description

The control flags of the DC\_CTRL are made up of DC (bit 0), AT (bit 5), IM (bit 6), and LM (bit 7). The status flags are the SB flag (bit 3) and FS flag (bit 8).

## Disable Cache (DC[0])

To disable the cache, write a 1 to the DC flag in DC\_CTRL (see [Figure 5-7](#)). Writing a 0 to the DC flag enables the cache. When the data cache is disabled, all load/store requests are sent directly to main memory; therefore the duration of a requested load or store instruction depends on the efficiency of the main memory system. In many cases, the performance of a load/store operation is severely degraded.

[Figure 5-8](#) shows the system configuration when the cache is enabled.

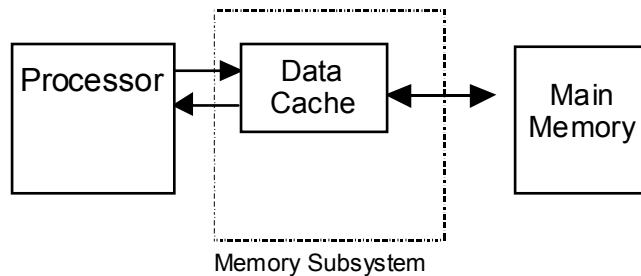
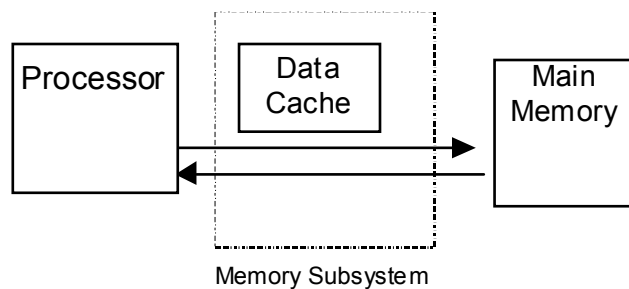
**Figure 5-8 Cache Enabled****DC = 0**

Figure 5-9 shows the system configuration when the cache is disabled.

**Figure 5-9 Cache Disabled****DC = 1****Note**

When the data cache is disabled, it is possible to perform all standard functions as well as all debug operations that are outlined in [“Advanced Features Summary”](#) on page 96. The cache is enabled upon reset (DC=0). Best practice is to flush and invalidate the data cache prior to disabling.

**LD/ST Instructions**

ARC EM provides the capability to enable/disable the data cache through use of the `.di` suffix when performing a load or store operation (LD and ST instruction).

**Caution**

The processor does not provide hardware interlocks between the data cache and data accesses that use the `.di` suffix. To avoid any potential memory-coherency issues, do not mix memory addresses accessed with the `.di` suffix with memory accesses without the `.di` suffix.

The operation of the `.di` cache control directive is ignored if the DC flag is set to '1' (cache disabled). The cache remains disabled irrespective of the type of LD or ST instruction.

[Table 5-4](#) shows the relationship between the DC flag and load/store instructions.



**Table 5-4 Effect of the DC flag and LD/ST Instructions on Cache Operation**

| DC Flag State | Instruction Type                                 | Cache Status        |
|---------------|--------------------------------------------------|---------------------|
| DC = 0        | LD <dest>,<address><br>ST <dest>,<address>       | Data cache enabled  |
| DC = 0        | LD.DI <dest>,<address><br>ST.DI <dest>,<address> | Data cache disabled |
| DC = 1        | LD <dest>,<address><br>ST <dest>,<address>       | Data cache disabled |
| DC = 1        | LD.DI <dest>,<address><br>ST.DI <dest>,<address> | Data cache disabled |

## C Coding Example

The following code illustrates the use of the `.di` suffix in C code. The original C code and compiled results are shown in the following example:

### Example 5-1 C-Code Compilation Results for a Volatile Type Declaration



#### Note

This example illustrates C-code compilation only.

#### C Code

```
#include <stdio.h>

int volatile a;
int b;

int main()
{
    a= 5;
    b= a+3;
}
```

#### Compiled Code

```
a= 5      ;// mov r0,5
          // st.di
r0,<addressA>

b= a+3    ;// ld.di
r0,<addressA>
          // add r0, r0, 3
          // st r0,<addressB>
```

Using the `volatile` type qualifier allows the programmer to declare that a variable could be modified by something other than the process that declared it. Declaration of a `volatile` variable instructs the compiler to generate an `st.di` or `ld.di` for any access to that variable. This ensures that the variable is never cached, allowing processes such as asynchronous interrupt handlers or shared memory architectures to gain access to that data.

**Note**

Volatile data areas and non-cached data areas must be aligned to cache-line boundaries to ensure that memory coherency is maintained. The volatile (and non-cached) memory blocks must be kept separate from the cacheable data memory area.

Memory-mapped peripheral-register accesses must make use of the `.di` suffix to avoid caching, unless the peripheral registers reside in a non-cacheable memory region.

The MetaWare C/C++ toolchain supports the generation of non-cached volatile declarations with the `volatile` qualifier and the `Volatile_cache_bypass` toggle (`.di` extensions). The MetaWare C/C++ toolchain additionally supports the `_Uncached` type qualifier, which does the same thing as the `volatile` qualifier and `Volatile_cache_bypass` toggle, but without involving the semantics of the ANSI `volatile` directive. Note that unofficial tools might not support the generation of non-cached volatile declarations.

## Success Flag (SB[2])

The SB flag indicates the success of a cache operation. Several cache modes use this flag to indicate the success of an operation, and in normal circumstances the programmer does not need to modify the flag contents. The SB flag is updated only when accessing certain registers.

**Table 5-5 Flag Update Conditions**

| Register Name | SB Flag Update |
|---------------|----------------|
| DC_IVDC       | No             |
| DC_CTRL       | No             |
| DC_LDL        | Yes            |
| DC_IVDL       | Yes            |
| DC_FLSH       | No             |
| DC_FLDL       | Yes            |

A 1 indicates a successful operation, while a 0 indicates a failure. When any of the above registers is accessed, the SB flag is updated when the function completes. Accessing the DC\_CTRL register to read the SB flag might result in a processor stall if the operation initiated had not completed. However, the result read back after initiating any of the listed functions reflects the success of that operation.

**Note**

It is possible to write to the SB flag so that when an interrupt occurs, the state of this flag can be saved and then restored on return from the interrupt.

## Address Debug Type (AT[5])

This flag is used in the advanced debug functions. It controls the addressing mechanisms used to access the data cache RAMs. Refer to subsection “[Advanced Register Summary](#)” on page 96 for a detailed description.

## Invalidate Mode (IM[6])

Invalidate mode is used to control the behavior of DC\_IVDC and DC\_IVDL.

Refer to [“Invalidate Data Cache, DC\\_IVDC”](#) on page 85 and [“Invalidate Data Line, DC\\_IVDL”](#) on page 93 for detailed descriptions.

## Lock Mode (LM[7])

The lock mode is used to control the behavior of DC\_LDL, DC\_FLSH, and DC\_FLDL on a locked cache line entry (locked using DC\_LDL).

Refer to [“Lock Data Cache Line, DC\\_LDL”](#) on page 91, [“Flush Data Cache, DC\\_FLSH”](#) on page 94, and [“Flush Data Line, DC\\_FLDL”](#) on page 95 for detailed descriptions.

## Flush Status (FS[8])

The FS flag indicates the state of the flush mechanism. The flag is updated upon reading the DC\_CTRL register. A 0 indicates no flush activity, while a 1 indicates that either the data cache is in a flush state, or is transferring data to main memory.

The FS flag is used to ensure that the data cache is coherent with memory.

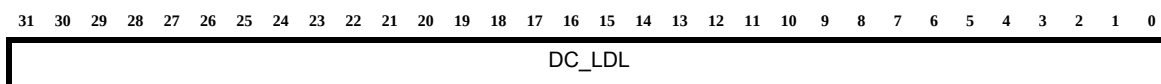
Example of use: Upon entering an interrupt service routine, the initial instructions in the code continually check the state of the FS flag. A one indicates that the cache is not yet coherent with main memory. Data that is read back (`ld .di`) might not reflect the contents of the data cache, while any data written (`st .di`) might be overwritten by the data cache. After the FS flag reads zero, main memory can be safely accessed (fully coherent).

## Lock Data Cache Line, DC\_LDL

Address: 0x49

Access: W

**Figure 5-10 DC\_LDL Register**



## DC\_LDL Description

Writing a system memory address to this register locks the cache line that maps to the given address. The format of the DC\_LDL register is shown in [Figure 5-10](#).

The lock data cache line operation allows any cache line in the data cache to be locked by the programmer. This allows the cache to be used as a high-speed data RAM that has a granularity of the cache-line size. This function can be used to lock critical data such as lookup tables in the cache. This means that the performance is never dependent on the speed of the memory system. The SB flag and FS flag in DC\_CTRL can be checked to verify the completion of a DC\_LDL operation.

When the address is presented to DC\_LDL, the cache checks to see if the requested line is present in the cache. If the line is in the cache and it is dirty, the LM flag found in DC\_CTRL is checked. If LM=1, the line is flushed to memory and subsequently locked. If LM=0, then the line is simply locked.

In the event that the requested line is present in the cache but the line is not dirty, the cache line is re-fetched from memory and is used to replace the original line. This refresh mechanism ensures that the line being locked contains the most up-to-date data from memory.

If the line that is being locked is not present in the data cache, it is simply fetched from memory and placed into the next available line. It is then subsequently locked.

After a line is locked, it cannot be replaced by a cache miss. If the line cannot be locked because all of the ways in the set that the address maps to are already locked, the SB flag in DC\_CTRL is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when a DC\_LDL operation completes successfully.



### Caution

The programmer must be aware that locking all the ways of a set prevents any further areas of memory from being cached if they map to the same set. An access to such a memory location goes directly to external memory.

The `st.di` and `ld.di` instructions always access external memory, even if the same address is locked in the cache.



### Note

If an attempt is made to lock a previously locked line, the line is re-fetched from main memory and locked as normal.

Locked cache lines can be unlocked using the DC\_IVDL register. Refer to [“Invalidate Data Line, DC\\_IVDL”](#) on page 93.

The SB flag is used to indicate a successful lock-line operation. However, if a DC\_LDL causes a line to be flushed to memory, it is important that the FS flag in DC\_CTRL be checked along with the SB flag to ensure that memory is coherent. Perform this check whenever memory coherency is an issue.

## DC\_LDL Code Example

The first instruction initiates an operation to lock a data cache line that loads a cache line at *address* and locks it. This prevents the cache line from being replaced on a cache miss.

```
sr    address, [DC_LDL]    ; Initiate a lock operation at address <address>
```

The last line of code checks to see if the lock operation completed, by checking the SB flag in DC\_CTRL.

```
lr    r3, [DC_CTRL]        ; Access the cache control register
and.f 0, r3, sb_flag        ; Mask out all bits except the SB bit
jeq    not_locked           ; Jump to the 'not locked' function if SB equals zero
...                          ; Success
not_locked:
...                          ; Failure
```

The following code can be used to lock a range of addresses into the cache. This code locks all data longwords between addresses *start\_address* and *end\_address*, verifying that each individual cache-line-lock operation is successful. Note that the line-length boundary is used to increment through a range of addresses.

```

        mov    r0,start_address    ;Set Start Address
        mov    r1,end_address      ;Set End Address
loop:   sr     r0,[DC_LDL]         ;Lock using r0 contents
        lr     r3,[DC_CTRL]       ;Read control register
        and.f  0,r3,sb_flag       ;Was lock successful?
        jeq    not_locked_error   ;Not locked. Goto error
        add    r0,r0,line_length  ;Move onto next line
        sub.f  0,r0,r1            ;Reached the end?
        blt    loop              ;Not complete. Continue

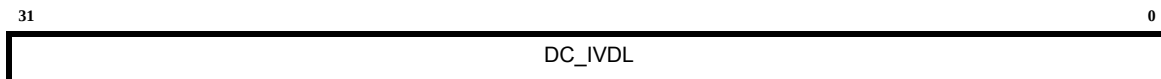
```

## Invalidate Data Line, DC\_IVDL

Address: 0x4A

Access: W

**Figure 5-11 DC\_IVDL Register**



The format of the DC\_IVDL register is shown in [Figure 5-11](#).

## DC\_IVDL Description

When a system-memory address is written to this register, the cache invalidates the line that maps to the given address. If the line is dirty, it might be flushed before invalidation, depending on the IM bit. Locked lines are unlocked and invalidated just like unlocked lines.

The invalidate-data-line (DC\_IVDL) operation allows a cache line to be removed from the cache. The method used to invalidate is selected via the IM flag (bit 6 of DC\_CTRL).

- **IM = 0 (Invalidate Only)** – If the system memory address written to DC\_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked), and the SB flag is set (SB=1, bit 2 DC\_CTRL).
- **IM = 1 (Invalidate and Flush Dirty Entry)** – If the system memory address written to DC\_IVDL is present in the cache, the cache line is invalidated and unlocked (if locked), and if the line is dirty it is flushed to the memory subsystem. Upon completion, the SB flag is set (SB=1, bit 2 DC\_CTRL).

Typical uses are to update a data entry within a table without invalidating the entire cache or use in cache-locking maintenance (such as removing cache-address mappings from the cache). Using this function also aids the update and maintenance of shared memory architectures where addresses within a memory space are shared between multiple logic blocks (such as other processors or peripherals).

The SB flag in DC\_CTRL can be checked to verify the completion of a DC\_IVDL operation.

## DC\_IVDL Code Example

The first instruction initiates an invalidate cache line operation at address table.

```
sr      table,[DC_IVDL]    ;Invalidate 'Table'
```

The last line of code checks to see if the DC\_IVDL operation completed by checking the SB flag in DC\_CTRL.

```

lr      r3, [DC_CTRL]      ;Read control register
and.f   0, r3, sb_flag     ;Check SB flag
jeq     not_invalidated    ;Error. Not Invalidated

```

## Flush Data Cache, DC\_FLSH

Address: 0x4B

Access: W

**Figure 5-12 DC\_FLSH Register**

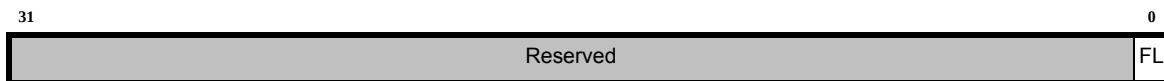


**Table 5-6 DC\_FLSH Control Flags**

| Flag Name | Description                                                                         | Access Type |
|-----------|-------------------------------------------------------------------------------------|-------------|
| FL[0]     | Flush Data Cache:- Flush entire data cache<br>0 – No Action<br>1 – Flush Data Cache | W           |

## DC\_FLSH Description

Writing a 1 to the FL flag in the DC\_FLSH register initiates a flush sequence for the entire data cache. All cache sets are checked sequentially for dirty entries, and entries that contain modified data are flushed to the memory subsystem. Upon completion of a flush sequence the line-dirty bit is cleared.

The operation of a flush command on a locked entry is selected by the LM flag (bit 7) in DC\_CTRL. Two sequences are possible with a locked entry.

- **LM = 0 (Disable Flush on Locked Entry)** – Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache-line that contains a dirty entry is not flushed, keeping the cache line completely locked down.
- **LM = 1 (Enable Flush on Locked Entry)** – Writing a 1 to the FL flag initiates a complete data-cache-flush sequence. A locked cache line that contains a dirty entry is also flushed irrespective of its locked status.

The DC\_FLSH function is used to update main memory with the most recent copy of the data-cache contents. This is very useful for shared-memory architectures, where other processors require updated memory contents.



### Note

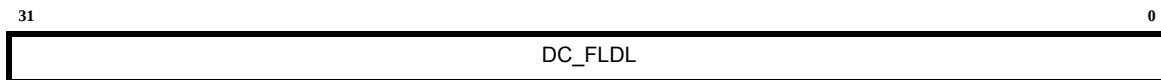
Writing a 0 to the FL flag does not initiate a flush sequence.

## Flush Data Line, DC\_FLDL

Address: 0x4C

Access: W

**Figure 5-13 DC\_FLDL Register**



### DC\_FLDL Description

Writing a system-memory address to DC\_FLDL causes the corresponding cache-line entry to flush (if dirty). When the system address is written to the DC\_FLDL register, the data cache searches for the requested entry. If the entry is present, the line dirty bit is checked. If dirty, the line is copied to the memory subsystem. Upon completion of a flush sequence, the SB flag is updated to indicate a successful operation (SB=1), and the line dirty bit is reset.

If the address specified in DC\_FLDL is not present, the SB flag indicates a failure (SB = 0).

In the event that an entry is locked, the state of the LM flag (bit 7, DC\_CTRL) controls the effect of a DC\_FLDL command on the locked entry.

- **LM = 0 (Disable Flush on Locked Entry)** — A system-memory address written to the DC\_FLDL register initiates a search for that entry. In the event that the requested cache line is locked, the DC\_FLDL command does not generate a flush sequence, even if that entry is dirty. The SB flag indicates a success when the address specified in DC\_FLDL is found.
- **LM = 1 (Enable Flush on Locked Entry)** — A system-memory address written to the DC\_FLDL register initiates a search for that entry. Regardless of the lock status, the DC\_FLDL command generates a flush sequence if that entry is dirty. The SB flag indicates a success (SB=1) upon completion of the flush sequence, and the dirty flag is reset (DT=0).



#### Note

If an entry is not dirty but is present in the cache, the SB flag indicates a success (SB=1).

### 5.2.1.3 Advanced Data-Cache Operation

The data cache contains advanced debug facilities that allow the programmer to interrogate and control the data cache RAM contents.

- [Advanced Features Summary](#)
- [Advanced Register Summary](#)
- [Data Cache External Access Address, DC\\_RAM\\_ADDR](#)
- [Data Cache Tag Access, DC\\_TAG](#)
- [Data Cache Data Access, DC\\_DATA](#)
- [Data Cache Control Register, DC\\_CTRL](#)

## Advanced Features Summary

The data cache allows the programmer to obtain direct access to the internal cache RAMs through either the ARC EM debug interface bus or through LR and SR instructions. This capability provides an invaluable debug environment when using the data cache.

Access to the cache RAMs is achieved through use of the registers outlined in [“Advanced Register Summary”](#).

## Advanced Register Summary

The ARC EM data cache offers powerful cache RAM control. A summary of the advanced cache RAM control registers is shown below.

**Table 5-7 Advanced Data Cache Auxiliary Registers**

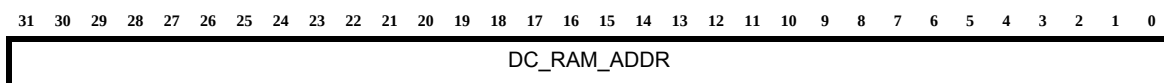
| Auxiliary Register Address | Register Name | Access Type | Description                            | Initial Value |
|----------------------------|---------------|-------------|----------------------------------------|---------------|
| 0x58                       | DC_RAM_ADDR   | Read/Write  | Data Cache External Access RAM Address | 0x0000 0000   |
| 0x59                       | DC_TAG        | Read/Write  | Data cache Tag access                  | 0x0000 0000   |
| 0x5B                       | DC_DATA       | Read/Write  | Data cache Data access                 | Undefined     |

### Data Cache External Access Address, DC\_RAM\_ADDR

Address: 0x58

Access: RW

**Figure 5-14 DC\_RAM\_ADDR**



### DC\_RAM\_ADDR Description

This register points to a location within the cache, to be used in tandem with the DC\_TAG and DC\_DATA registers (refer to subsections [“DC\\_TAG Description”](#) on page 97 and [“DC\\_DATA Description”](#) on page 98).

The interpretation of the register is defined by the AT (Address Debug Type) flag in the DC\_CTRL register. Writing a 0 to the AT flag in DC\_CTRL selects direct cache-RAM access. Writing a 1 to the AT flag selects cache-controlled RAM access.

A detailed description of the AT flag is provided in [“Address Debug Type \(AT\[5\]\)”](#) on page 99.

An SR to the DC\_RAM\_ADDRESS register triggers a read from the data cache and updates the DC\_TAG and DC\_DATA registers. The read is performed according to the AT bit in the DC\_CTRL register.



## Data Cache Tag Access, DC\_TAG

Address: 0x59

Access: RW

**Figure 5-15 DC\_TAG for Reads**



**Figure 5-16 DC\_TAG for Writes**



### Note

Data written into the unused field is disregarded.

**Table 5-8 DC\_TAG Control Flags and Update Conditions**

| Flag Name | Description                                                                                               | Flag Update Conditions                                                                                                             | Access Type |
|-----------|-----------------------------------------------------------------------------------------------------------|------------------------------------------------------------------------------------------------------------------------------------|-------------|
| V[0]      | Valid Flag: Valid flag associated with the cache line<br>0 – Cache line not valid<br>1 – Cache line valid | Programmer updates to DC_TAG<br>DC_IVDL command<br>DC_IVDC command<br>Cache line replacement                                       | R/W         |
| LCK[1]    | Lock Flag: Lock flag associated with the cache line<br>0 – Cache line not locked<br>1 – Cache line locked | Programmer updates to DC_TAG<br>DC_LDL command<br>DC_IVDL command<br>DC_IVDC command                                               | R/W         |
| DT[2]     | Dirty Flag: Dirty flag associated with the cache line<br>0 – Cache line not dirty<br>1 – Cache line dirty | Programmer updates to DC_TAG<br>DC_FLDL command<br>DC_FLSH command<br>DC_IVDL command<br>DC_IVDC command<br>Cache line replacement | R/W         |

## DC\_TAG Description

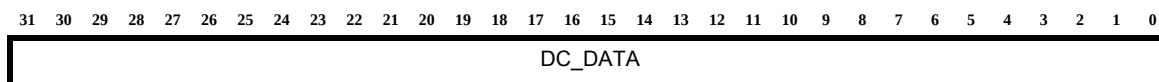
Reading from or writing to this register allows the programmer to access the valid bit, lock bit, dirty bit, and tag of a cache line address specified in the DC\_RAM\_ADDR register. A write transaction to the DC\_RAM\_ADDRESS also triggers an update of the DC\_TAG and DC\_DATA registers.

## Data Cache Data Access, DC\_DATA

Address: 0x5B

Access: RW

**Figure 5-17 DC\_DATA**



### DC\_DATA Description

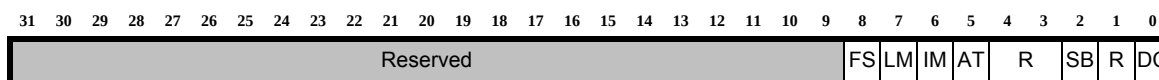
Reading from or writing to the **DC\_DATA** register allows the programmer to access a data from a particular cache line, using the address in **DC\_RAM\_ADDR**. An SR to the **DC\_RAM\_ADDRESS** also triggers an update of the **DC\_TAG** and **DC\_DATA** registers.

## Data Cache Control Register, DC\_CTRL

Address: 0x48

Access: RW

**Figure 5-18 DC\_CTRL Register**



### Note

Reserved bits must be written with zero.

**Table 5-9 DC\_CTRL Control Flags**

| Flag Name     | Description                                                                                                           | Access Type |
|---------------|-----------------------------------------------------------------------------------------------------------------------|-------------|
| DC[0]         | Disable Cache: Enables/Disables the cache<br>0 – Enable Cache<br>1 – Disable Cache                                    | R/W         |
| Reserved[1]   | —                                                                                                                     | —           |
| SB[2]         | Success Bit: Success of last cache operation<br>0 – Last cache operation failed<br>1 – Last cache operation succeeded | R/W         |
| Reserved[4:3] | —                                                                                                                     | —           |

**Table 5-9 DC\_CTRL Control Flags (Continued)**

|                |                                                                                                                                              |     |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------|-----|
| AT[5]          | Address Debug Type: Used for debug purposes for when accessing cache RAMs.<br>0 – Direct Cache RAM Access<br>1 – Cache Controlled RAM Access | R/W |
| IM[6]          | Invalidate Mode: Selects the invalidate type<br>0 – Invalidate data cache only<br>1 – Invalidate data cache and flush dirty entries          | R/W |
| LM[7]          | Lock Mode: Selects the effect of a flush command on a locked entry<br>0 – Disable flush on locked entry<br>1 – Enable flush on locked entry  | R/W |
| FS[8]          | Flush Status: Status of the data cache flush mechanism<br>0 – Idle<br>1 – Flush operation in progress                                        | R   |
| Reserved[31:9] | --                                                                                                                                           | —   |

**DC\_CTRL Description**

Two flags are used in advanced debug operations. These are the SB flag and AT flag.

**Success Flag (SB[3])**

The advanced cache modes use the SB flag to indicate the success of an operation. The SB flag is updated only when accessing certain registers and the address debug mode (AT flag) effects its operation. [Table 5-10](#) indicates the SB flag update conditions.

**Table 5-10 SB Flag Update Conditions with Advanced Cache Modes**

| Register Name | Success Flag Update When AT=0 | Success Flag Update When AT=1 |
|---------------|-------------------------------|-------------------------------|
| DC_RAM_ADDR   | X                             | ✓                             |
| DC_TAG        | X                             | X                             |
| DC_DATA       | X                             | X                             |

**Note**

It is possible to write to the SB flag so that when an interrupt occurs the state of this flag can be saved and eventually restored, upon returning from an interrupt.

The SB flag is only updated upon accessing the registers outlined in [Table 5-10](#). Therefore it is best to perform a flag condition check following an access to these registers.

**Address Debug Type (AT[5])**

The data cache allows the programmer to gain access to the internal RAMs within the cache. The Address Debug Type (AT[5]) flag controls the interpretation of the address in the DC\_RAM\_ADDR register for accessing the internal cache RAMs.

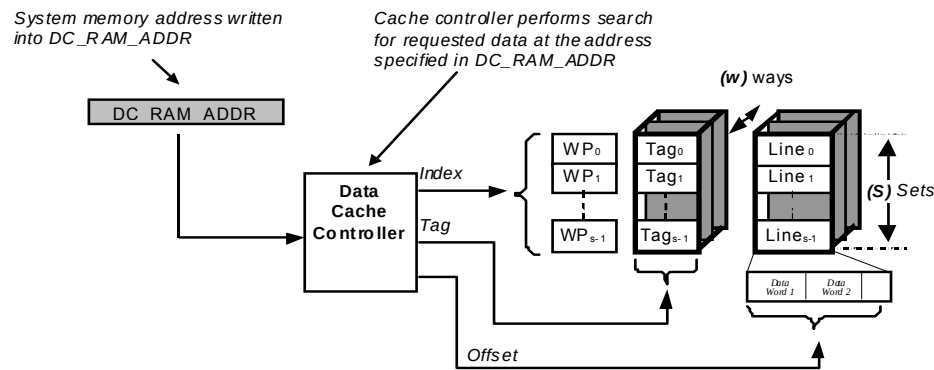
Cache-Controlled RAM Access (AT=1)

Writing a system-memory address into DC\_RAM\_ADDR causes the cache to search for a valid entry. If an entry exists, DC\_TAG and DC\_DATA are updated, and the SB flag is set. The DC\_TAG and DC\_DATA registers can then be read with an LR instruction. You cannot write the cache RAMs in cache-controlled RAM-access mode.

 **Note**

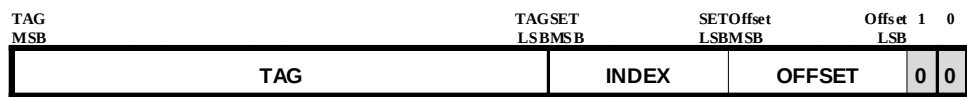
The SB flag is updated by the SR instruction to the IC\_RAM\_ADDR register. A LR instruction never updates the SB flag.

Figure 5-19 Indirect Cache-RAM Access Using System Memory Address



The format of the address that is written into DC\_RAM\_ADDR to search the data cache RAMs is shown in Figure 5-20.

Figure 5-20 System Address Format Used for Cache-Controlled RAM Access



 **Note**

Cache-controlled RAM access is used to search and retrieve a data entry within the data-cache RAMs. Therefore, the contents of DC\_TAG or DC\_DATA cannot be written in this mode. Writing to DC\_TAG or DC\_DATA in this mode generates an instruction error exception. To modify the contents of the data-cache RAMs, the address debug type must be switched to direct cache-RAM access (AT=0).

Example 5-2 Cache-Controlled RAM Access

The following example shows an assembly coding example using the cache-controlled RAM-access mode to find out information about a LD/ST address.

The following code fragment enables cache-controlled RAM access. To do this, the code sets the AT flag in DC\_CTRL.

```
lr r5, [DC_CTRL]      ; Access the cache control register
or r5, r5, at_flag    ; Set the AT flag
sr r5, [DC_CTRL]      ; Write the new mode settings to the control register
```

The next line of code sets up the cache so it reads from DC\_TAG or DC\_DATA retrieve information associated with ls\_addr. Writing ls\_addr to the DC\_RAM\_ADDR register achieves this.

```
mov r0, ls_addr
sr r0, [DC_RAM_ADDR]  ; Prepare to address cache RAMs from address 'ls_addr'
```

The following code reads all three cache RAMs using the address specified in ls\_addr.

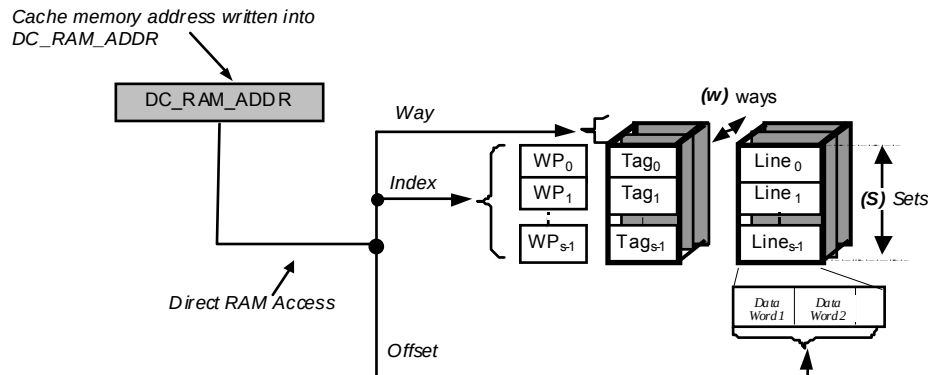
```
lr r1, [DC_TAG]       ; r1 = DC_TAG
lr r3, [DC_DATA]      ; r3 = DC_DATA
```

The next line of code reads the SB flag in the DC\_CTRL register to check if the cache contained an entry for ls\_addr. If the SB flag equals one, then the cache contains a valid entry.

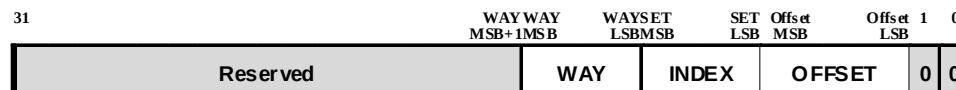
```
lr    r4, [DC_CTRL]   ; Access the cache control register
and.f 0, r4, sb_flag   ; Mask out all bits except the SB flag
jeq    not_in_cache    ; Jump to the 'not in cache' code if SB equals zero
```

### Direct Cache RAM Access (AT=0)

In this mode DC\_RAM\_ADDR is used as a direct index into the cache memory. The cache controller does not perform a search for a valid entry, but simply applies the address specified in DC\_RAM\_ADDR to the cache RAMs. The set field selects the set to be accessed and the way field selects the way to be accessed, identifying a unique line in the cache memory. The offset field is used to locate the data within the chosen line. After the memory location in the cache is specified, registers DC\_TAG and DC\_DATA can be used to read or write the tag RAM or cache data RAM.

**Figure 5-21 Direct Cache-RAM Access Using Direct-Cache-RAM Address**

**Note** For both cache-controlled RAM access and direct cache-RAM access modes, the state of the cache (enabled or disabled) that is controlled by DC[0] in DC\_CTRL does not affect the operation of the address debug functions. The format of the address that is written into DC\_RAM\_ADDR to directly access the data cache RAMs is shown in [Figure 5-22](#)

**Figure 5-22 Direct-Cache-Access Address Register for DC\_RAM\_ADDR**

**Note** Data words in a cache line are word-aligned, and therefore the offset field is written in words (the lower two bits of the address as indicated in [Figure 5-22](#) are written with zero).

### Example 5-3 Direct Cache-RAM Access

The following shows an assembler-coding example using direct cache-RAM access. The following code fragment shows how the direct cache RAM access mode is enabled. To do so, the code clears the AT flag.

```
lr r5,[DC_CTRL]      ; Access the cache control register
bic r5,r5,at_flag    ; Clear the AT flag
sr r5,[DC_CTRL]      ; Write the new mode settings to DC_CTRL
```

The next line of code sets up the DC\_RAM\_ADDR register to access the cache RAMs at address ram\_addr.

```
mov r0,ram_addr
sr r0,[DC_RAM_ADDR] ; Prepare to address cache RAMs from a address 'ram_addr'
```

The next line reads all three cache RAMs directly using the address specified in DC\_RAM\_ADDR with the LR instruction.

```
lr r1,[DC_TAG]      ; r1=TAG_RAM[Address in DC_RAM_ADDR];
```

```
lr r3,[DC_DATA]          ; r3=DATA_RAM[Address in DC_RAM_ADDR];
```

The following code modifies the contents of all the RAMs and writes the new values back into the data-cache RAMs using the SR instruction.

```
add r1,r1,1              ; Modify DC_TAG contents
add r2,r2,1              ; Modify DC_WP contents
add r3,r3,1              ; Modify DC_DATA contents
sr r1,[DC_TAG]           ; Write into DC_TAG using contents of r1
sr r3,[DC_DATA]          ; Write into DC_DATA using contents of r3
```

To read the contents of the cache again to ensure it has been modified, use an SR instruction:

```
sr r0,[DC_RAM_ADDRESS]  ; Prepare to address cache RAMs from address 'ram_addr'
lr r1,[DC_TAG]           ; r1=tag_ram[DC_RAM_ADDRESS];
lr r3,[DC_DATA]          ; r3=data_ram[DC_RAM_ADDRESS];
```

## 5.2.2 Instruction Cache

### 5.2.2.1 Software-Controllable Functions

- [Instruction-Cache-Register Summary](#)
- [Invalidate Instruction Cache, IC\\_IVIC](#)
- [Instruction Cache Control Register, IC\\_CTRL](#)
- [Lock Instruction Cache Line, IC\\_LIL](#)
- [Invalidate Instruction-Cache Line, IC\\_IVIL](#)

#### Instruction-Cache-Register Summary

A summary of the basic instruction-cache registers is shown in [Table 5-11](#). The rest of the section a detailed functional description of their operation. All cache registers can be accessed using LR and SR instructions or via the ARC EM host interface.

**Table 5-11 Instruction Cache Auxiliary Registers**

| Auxiliary Register Address | Register Name | Access Type | Description                 | Initial Reset Value |
|----------------------------|---------------|-------------|-----------------------------|---------------------|
| 0x10                       | IC_IVIC       | Write Only  | Invalidate Cache            | —                   |
| 0x11                       | IC_CTRL       | Read/Write  | Cache Control Register      | 0x0000 0000         |
| 0x13                       | IC_LIL        | Write only  | Lock Instruction Line       | —                   |
| 0x19                       | IC_IVIL       | Write only  | Invalidate Instruction Line | —                   |

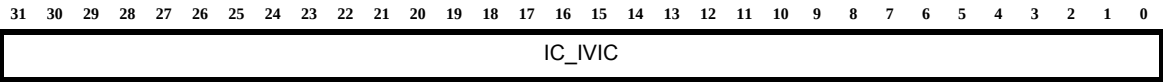
#### Instruction-Cache Operation on Reset

The instruction cache may be enabled immediately after reset, depending on build-configuration option `-ic_disable_on_reset` (see [page 65](#)). If this option is enabled, the last cache operation is set to failed, and the direct cache-RAM access is enabled. Furthermore, the instruction cache is invalidated — all cache lines are invalidated and unlocked, and the tag RAM is cleared.

5.2.3 Invalidate Instruction Cache, IC\_IVIC

Address: 0x10  
Access: W

Figure 5-23 IC\_IVIC Register



IC\_IVIC Description

A write to the IC\_IVIC register invalidates and unlocks the entire instruction cache. Builds with an instruction-fetch queue or ICCM1 also utilize this register for invalidating the buffer content.

This function is initiated by any write to the IC\_IVIC auxiliary register. IC\_IVIC is used to make sure the cache is coherent with memory.

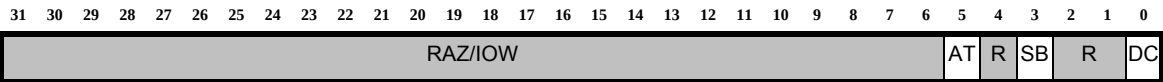


**Caution** All SR accesses to the IC\_IVIC register must be followed by three NOP instructions.

5.2.4 Instruction Cache Control Register, IC\_CTRL

Address: 0x11  
Access: RW

Figure 5-24 IC\_CTRL Register



- Note**
- All reserved bits are RAZ/IOW.
  - R\* indicates RAZ/IOW bits.

Table 5-12 IC\_CTRL Control Flags

| Flag Name | Description                                                                        | Access Type |
|-----------|------------------------------------------------------------------------------------|-------------|
| DC[0]     | Disable Cache: Enables/Disables the cache<br>0 – Enable Cache<br>1 – Disable Cache | R/W         |



**Table 5-12 IC\_CTRL Control Flags**

|                |                                                                                                                                              |     |
|----------------|----------------------------------------------------------------------------------------------------------------------------------------------|-----|
| Reserved [2:1] | RAZ/IOW                                                                                                                                      | —   |
| SB[3]          | Success Bit: Success of last cache operation<br>0 – Last cache operation failed<br>1 – Last cache operation succeeded                        | R/W |
| Reserved [4]   | RAZ/IOW                                                                                                                                      | —   |
| AT[5]          | Address Debug Type: Used for debug purposes for when accessing cache RAMs.<br>0 – Direct Cache RAM Access<br>1 – Cache Controlled RAM Access | R/W |
| Reserved[31:6] | RAZ/IOW                                                                                                                                      | —   |

**IC\_CTRL Description**

The IC\_CTRL register contains the control flags DC (bit 0) and AT (bit 5), and one status flag SB (bit 3).

**Disable cache (DC[0])**

To disable the cache, write a 1 to the DC flag in IC\_CTRL (refer to [Figure 5-24](#)). Writing a 0 to the DC flag enables the cache.

When the instruction cache is disabled, all instruction requests are fetched directly from main memory. This means the duration of an instruction fetch is entirely dependent on the speed of the memory system, and therefore the performance of the processor can be severely degraded.

[Figure 5-25](#) shows the system configuration when the cache is enabled, and [Figure 5-26](#) shows the system configuration when the cache is disabled.

The cache is enabled upon reset (DC=0).

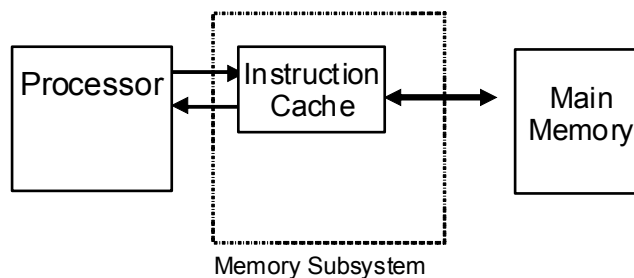
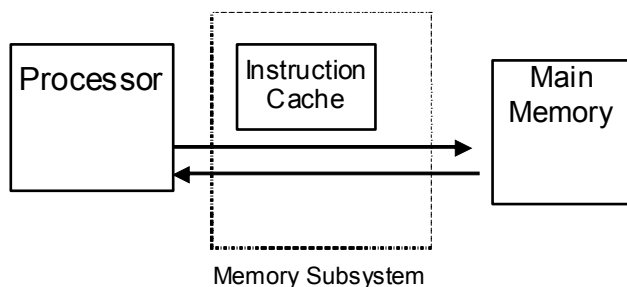
**Figure 5-25 Cache Enabled****DC = 0**

**Figure 5-26 Cache Disabled****DC = 1****Success Flag (SB[3])**

The SB flag indicates the success of a cache operation. Several cache modes use this flag to indicate the success of an operation, and in normal circumstances the programmer does not need to modify the flag contents. The SB flag is updated only when accessing certain registers. [Table 5-13](#) indicates the SB flag update conditions.

**Table 5-13 SB Flag Update Conditions**

| Register Name | SB Flag Update |
|---------------|----------------|
| IC_IVIC       | --             |
| IC_CTRL       | --             |
| IC_LIL        | ✓              |
| IC_IVIL       | ✓              |

**Note**

It is possible to write to the SB flag so that when an interrupt occurs the state of this flag can be saved and then restored upon returning from the interrupt.

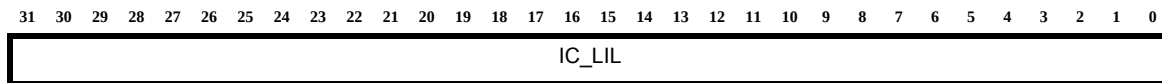
**Address Debug Type (AT[5])**

This flag enables the advanced debug functions in the instruction cache. It controls the addressing mechanisms used to access the instruction-cache RAMs. Refer to [“Address Debug Type \(AT\[5\]\)”](#) on page 112 for a detailed description.

**Lock Instruction Cache Line, IC\_LIL**

Address: 0x13

Access: W

**Figure 5-27 IC\_LIL Register****IC\_LIL Description**

Writing a system-memory address to this register locks the cache line that maps to the given address. Locking allows the cache to be used as a high-speed instruction ROM that has a granularity of the cache-line size.

This function can be used to lock time critical code such as interrupt routines in the cache. This means that the performance is not dependent on the speed of the memory system. The SB flag in IC\_CTRL can be checked to verify the completion of a IC\_LIL operation.

Writing a system memory address to IC\_LIL causes the cache to load the appropriate cache line. The line is then locked so it cannot be overwritten when a cache miss occurs. If the line cannot be locked in the cache because all of the ways in the set that the address maps to are already locked, the SB flag in IC\_CTRL is cleared (SB=0), indicating a cache operation failure. The SB flag is set (SB=1) when an IC\_LIL operation completed successfully. Note that if the programmer attempts to lock a previously locked line, the line is re-loaded and locked as normal.

**Note**

If all lines in a set are locked, the cache performs the requested access direct with main memory.

Line locking using addresses that are mapped to other L1 memory components, such as the ICCM, do successfully lock the line. However, any subsequent instruction fetches still return data from the memory component that has priority, in this case the ICCM.

**Caution**

Be aware that locking all the ways of a set prevents any further areas of memory from being cached if they map to the same set. Line-locking uncachable regions might result in unpredictable memory accesses.

**IC\_LIL Code Examples**

The first instruction initiates a lock instruction cache-line operation that loads a cache line from *program* address and locks it. This prevents the cache line from being replaced on a cache miss.

```
sr    program, [IC_LIL] ; Initiate a lock operation at address 'program'
```

The last line of code checks whether the lock operation completed, by checking the SB flag in IC\_CTRL.

```
lr    r3, [IC_CTRL] ; Access the cache mode register
and.f 0, r3, sb_flag ; Mask out all bits except the SB bit
jeq    not_locked   ; Jump to the 'not locked' cache code function if SB is zero
...                               ; Success
not_locked:
...                               ; Failure
```

The following code may be used to lock a range of addresses into the cache. This code locks all instructions between address *start\_address* and *end\_address*, verifying that each individual cache line lock operation is successful. Note that the line-length boundary is used to increment through a range of addresses.

```

        mov    r0,start_address    ;Set Start Address
        mov    r1,end_address      ;Set End Address
loop:   sr     r0,[IC_LIL]         ;Lock using r0 contents
        lr     r3,[IC_CTRL]       ;Read control register
        and.f  0,r3,sb_flag       ;Was lock successful?
        jeq    not_locked_error   ;Not locked. Goto error
        add    r0,r0,line_length  ;Move onto next line
        sub.f  0,r0,r1            ;Reached the end?
        blt    loop              ;Not complete. Continue

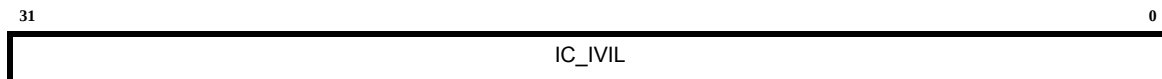
```

## Invalidate Instruction-Cache Line, IC\_IVIL

Address: 0x19

Access: W

**Figure 5-28 IC\_IVIL Register**



### IC\_IVIL Description

When a system memory address is written to this register the cache invalidates a cache line that maps to the given address, including locked lines.

Typical uses are to update an interrupt vector without invalidating the whole cache, or for cache-locking maintenance. The SB flag in IC\_CTRL can be checked to verify the completion of an IC\_IVIL operation.

If the location specified is not in the cache, the SB flag in IC\_CTRL is cleared (SB=0). Otherwise, the SB bit is set (SB=1).



### Caution

All SR accesses to the IC\_IVIL register must be followed by three NOP instructions.

### Example 5-4 Using IC\_IVIL

The first instruction initiates an invalidate cache line operation at address *program*.

```
sr    program,[IC_IVIL]    ;Invalidate 'Program'
```

The last line of code checks to see whether the IC\_IVIL operation completed by checking the SB flag in IC\_CTRL.

```
lr    r3,[IC_CTRL]        ;Read control register
and.f 0,r3,sb_flag        ;Check SB flag
jeq    not_invalidated    ;Error. Not Invalidated

```

#### 5.2.4.1 Advanced Instruction-Cache Operation

The instruction cache contains advanced debug facilities that allow the programmer to interrogate and control the instruction cache RAM contents.

## Advanced Features Summary

The instruction cache allows the programmer to obtain direct access to the internal cache RAMs through either the ARC EM host interface bus or through LR and SR instructions. This capability provides an invaluable debug and test environment when using the instruction cache.

Access to the cache RAMs is provided through the registers outlined in the next subsection.

## Advanced Register Summary

The ARC EM instruction cache offers powerful cache RAM control. A summary of the advanced cache-RAM-control registers is shown below.

**Table 5-14 Advanced Instruction Cache Auxiliary Registers**

| Auxiliary Register Address | Register Name | Access Type | Description                                   | Initial Reset Value |
|----------------------------|---------------|-------------|-----------------------------------------------|---------------------|
| 0x11                       | IC_CTRL       | Read/Write  | Cache control register                        | 0x0000 0000         |
| 0x1A                       | IC_RAM_ADDR   | Read/Write  | Instruction cache external access RAM address | 0x0000 0000         |
| 0x1B                       | IC_TAG        | Read/Write  | Instruction cache tag access                  | 0x0000 0000         |
| 0x1D                       | IC_DATA       | Read/Write  | Instruction cache data access                 | 0x0000 0000         |



### Note

The reset value of the DC bit of the IC\_CTRL register depends on IC\_DISABLE\_ON\_RESET, user-definable option. The IC\_DISABLE\_ON\_RESET build option is zero by default.

## Instruction Cache External-Access Address, IC\_RAM\_ADDR

Address: 0x1A

Access: RW

**Figure 5-29 IC\_RAM\_ADDR Register**



## IC\_RAM\_ADDR Description

This register points to a location within the cache to be used in tandem with IC\_TAG and IC\_DATA registers (refer to subsections “[Instruction-Cache Tag Access, IC\\_TAG](#)” on page 110 and “[Instruction Cache Data Access, IC\\_DATA](#)” on page 111).

The interpretation of the register is defined by the AT (Address Debug Type) flag in the IC\_CTRL register. Writing a 0 to the AT flag in IC\_CTRL selects direct cache-RAM access. Writing a 1 to the AT flag selects cache-controlled RAM access. A detailed description of the AT flag is provided in “[Address Debug Type \(AT\[5\]\)](#)” on page 112. A SR to the IC\_RAM\_ADDR register triggers a read from the instruction cache. The read is performed according to the AT bit in the IC\_CTRL register. An SR to IC\_RAM\_ADDRESS also triggers an update of the IC\_TAG and IC\_DATA registers.

## Instruction-Cache Tag Access, IC\_TAG

Address: 0x1B

Access: RW

**Figure 5-30 IC\_TAG Register for Reads**

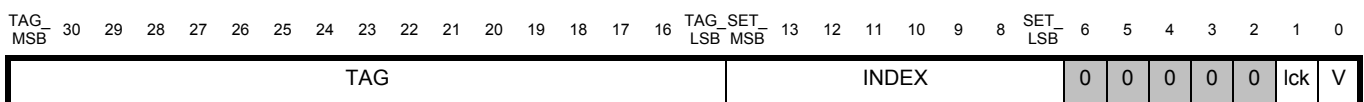
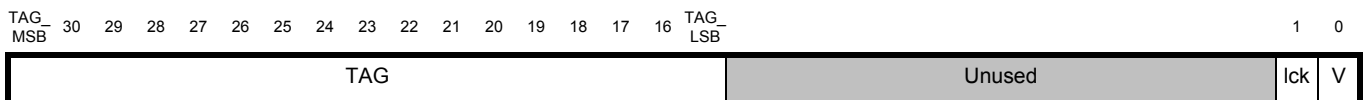


**Figure 5-31 IC\_TAG Register for Writes**



### Note

Data written into the unused field is disregarded.

## IC\_TAG Description

Reading from or writing to this register allows the programmer to access the valid bit, lock bit, and tag of a cache line specified by the IC\_RAM\_ADDR register and the AT flag found in IC\_CTRL. An SR to the IC\_RAM\_ADDRESS register also triggers an update of the IC\_TAG and IC\_DATA registers. The read format of IC\_TAG is shown in [Figure 5-30](#) and the write format in [Figure 5-31](#).



### Caution

Take care when debugging any software that accesses the TAG RAM via the IC\_TAG register.

As part of the host debugger protocol to access memory, an invalidate-cache operation is carried out after any write to memory. This invalidate cache operation updates tag RAMs and therefore modifies any value that has been written to the tag RAM by the software.

An invalidate cache operation goes ahead even if the instruction cache is disabled.

Valid Bit (V[0])

The valid bit (V) indicates whether the data associated with the tag is valid or not. When V=1 the data is valid.

Lock Bit (LCK[1])

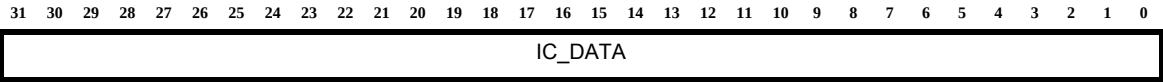
The lock bit (LCK) indicates whether the data associated with the tag is locked. The data entry is locked when LCK=1.

Instruction Cache Data Access, IC\_DATA

Address: 0x1D

Access: RW

Figure 5-32 IC\_DATA Register



IC\_DATA Description

Reading from or writing to the IC\_DATA register allows the programmer to access a data word from a particular cache line, using the address in IC\_RAM\_ADDR. The format of IC\_DATA is shown in [Figure 5-32](#). An SR to the IC\_RAM\_ADDRESS also triggers an update of the IC\_TAG and IC\_DATA registers.

Instruction-Cache Control Register, IC\_CTRL

Address: 0x11

Access: RW

Figure 5-33 IC\_CTRL Register

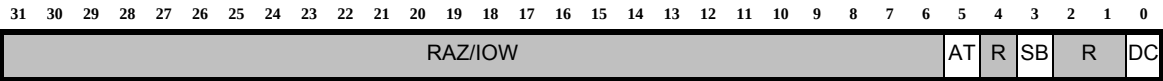


Table 5-15 IC\_CTRL Control Flags

| Flag Name      | Description                                                                        | Access Type |
|----------------|------------------------------------------------------------------------------------|-------------|
| DC[0]          | Disable Cache: Enables/Disables the cache<br>0 – Enable Cache<br>1 – Disable Cache | R/W         |
| Reserved [2:1] | —                                                                                  | —           |

**Table 5-15 IC\_CTRL Control Flags**

|                |                                                                                                                                                |     |
|----------------|------------------------------------------------------------------------------------------------------------------------------------------------|-----|
| SB[3]          | Success Bit:- Success of last cache operation<br>0 – Last cache operation failed<br>1 – Last cache operation succeeded                         | R/W |
| Reserved[4]    | --                                                                                                                                             | --  |
| AT[5]          | Address Debug Type: - Used for debug purposes for when accessing cache RAMs.<br>0 – Direct Cache RAM Access<br>1 – Cache Controlled RAM Access | R/W |
| Reserved[31:6] | —                                                                                                                                              | —   |

**IC\_CTRL Description**

Two flags are used in the advanced debug operations: The SB flag and AT flag. Refer to [Figure 5-24 on page 104](#) for a detailed operational description of the DC, EB and RA flags.

**Success Flag (SB[3])**

The SB flag indicates the success of a cache operation. The advanced cache modes use this flag to indicate the success of an operation. The SB flag is updated only when accessing certain registers, and the address debug mode (AT flag) affects its operation. [Table 5-16](#) indicates the SB flag update conditions.

**Table 5-16 SB Flag Update Conditions with Advanced Cache Modes**

| Register Name | Success Flag Update When AT = 0 | Success Flag Update When AT = 1 |
|---------------|---------------------------------|---------------------------------|
| IC_RAM_ADDR   | --                              | ✓                               |
| IC_TAG        | --                              | --                              |
| IC_DATA       | --                              | --                              |

**Note**

It is possible to write to the SB flag. This means that when an interrupt occurs, the state of this flag can be saved and then restored when returning from the interrupt.

**Address Debug Type (AT[5])**

The instruction cache allows the programmer to access the internal RAMs within the cache. The Address Debug Type (AT[5]) flag controls the interpretation of the address in IC\_RAM\_ADDR when the internal RAMs are accessed.

**Cache-Controlled RAM Access (AT=1)**

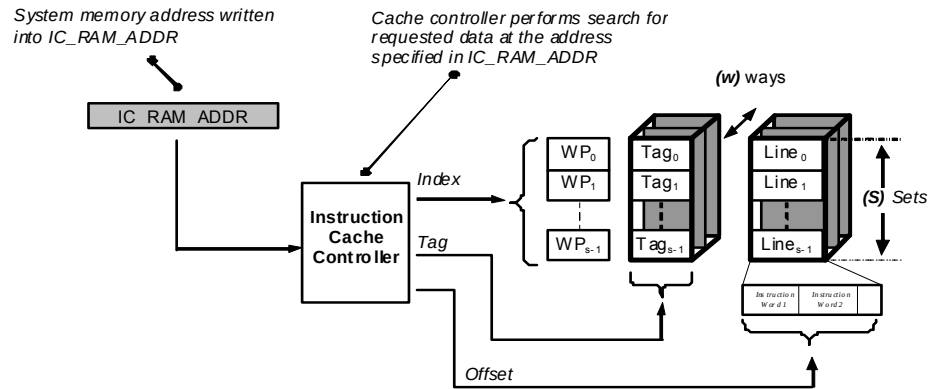
Writing a system memory address into IC\_RAM\_ADDR causes the cache to search for a valid entry. If an entry exists, IC\_TAG and IC\_DATA are updated and the SB flag is set. The IC\_TAG and IC\_DATA registers can then be read with a LR instruction.





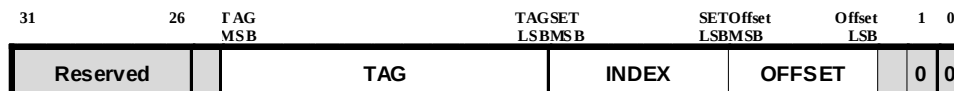
**Note** The SB flag is updated by the SR instruction to the IC\_RAM\_ADDR register. An LR instruction never updates the SB flag.

**Figure 5-34 Indirect Cache RAM Access Using System Memory Address**



The format of the address that is written into IC\_RAM\_ADDR to perform a search in the instruction cache RAMs is shown in [Figure 5-35](#).

**Figure 5-35 System Address Format Used For Cache Controlled RAM Access**



**Note** Cache-controlled RAM Access is used to search and retrieve a data entry within the instruction-cache RAMs. The contents of IC\_TAG or IC\_DATA should not be written to when in this mode. Writing to IC\_TAG or IC\_DATA in cache-controlled mode generates an instruction error exception. To modify the contents of the instruction cache RAMs, the address debug type must be switched to Direct Cache RAM Access (AT=0).

### Example 5-5 Cache Controlled RAM Access Coding

The following shows an assembly coding example using the cache controlled RAM access mode to find out information about a particular PC address.

The following code fragment enables cache controlled RAM accesses. The code simply sets the AT flag in IC\_CTRL.

```

lr   r5, [IC_CTRL]      ; Access the cache control register
or   r5, r5, at_flag    ; Set the AT flag
sr   r5, [IC_CTRL]      ; Write the new mode settings to the control register

```

The next lines set up the cache so that reads from IC\_TAG or IC\_DATA retrieve information associated with pc\_addr. Writing pc\_addr to the IC\_RAM\_ADDR register achieves this.

```
mov  r0,pc_addr
sr   r0,[IC_RAM_ADDR] ;Prepare to address cache RAMs from address 'pc_addr'
```

The following code reads all three cache RAMs using the address specified in pc\_addr.

```
lr    r1,[IC_TAG]      ; r1 = IC_TAG
lr    r3,[IC_DATA]     ; r3 = IC_DATA
```

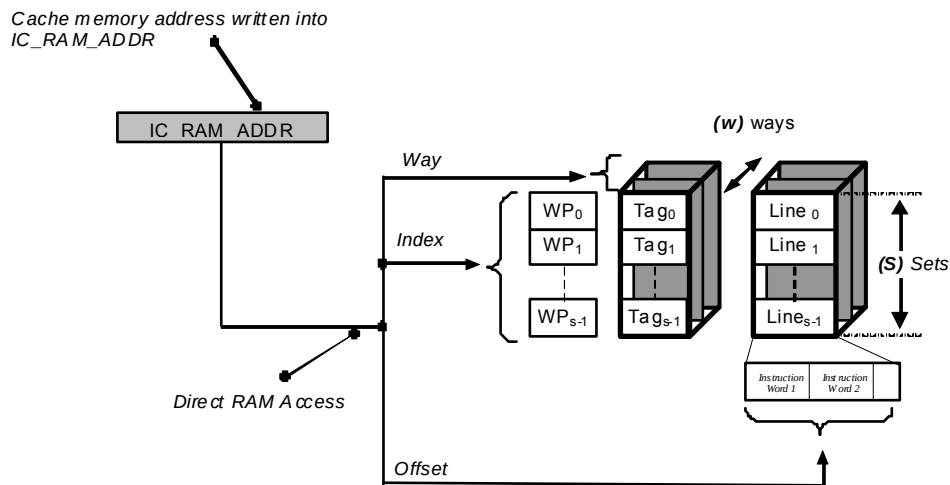
The next line of code reads the SB flag in the IC\_CTRL register to check if the cache contained an entry for pc\_addr. If the SB flag equals one, then the cache contains a valid entry.

```
lr    r4,[IC_CTRL]     ; Access the cache control register
and.f 0,r4,sb_flag     ; Mask out all bits except the SB flag
jeq   not_in_cache     ; Jump to the 'not in cache' code if SB equals zero
```

### Direct Cache-RAM Access (AT=0)

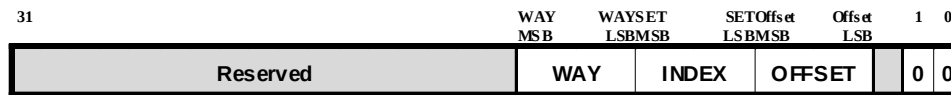
In this mode, IC\_RAM\_ADDR is used as a direct index into the cache memory. The cache controller does not perform a search for a valid entry, but simply applies the address specified in IC\_RAM\_ADDR to the cache RAMs. The set field selects the set to be accessed and the way field selects the way to be accessed, identifying a unique line in the cache memory. The offset field is used to locate one word within the line. After the memory location in the cache is specified, registers IC\_TAG and IC\_DATA can be used to read or write the tag RAM or cache-data RAM.

**Figure 5-36 Using Direct Cache-RAM Access**



**Note** For both cache-controlled RAM access and direct cache-RAM access modes, the state of the cache (enabled or disabled) that is controlled by DC[0] in IC\_CTRL does not affect the address debug functions.

The format of the address that is written into IC\_RAM\_ADDR to access the instruction cache RAMs is shown in [Figure 5-37](#).

**Figure 5-37 Direct-Cache-Access Address Register for IC\_RAM\_ADDR****Note**

Data words in a cache line are word aligned, therefore the offset field is written in words (the lower 2 bits of the address as indicated in [Figure 5-37](#)) are written with zero.

**Example 5-6 Direct Cache RAM Access**

The following code fragment shows how the direct cache RAM access mode is enabled. The code does this by clearing the AT flag.

```
lr r5,[IC_CTRL] ; Access the cache control register
bic r5,r5,at_flag ; Clear the AT flag
sr r5,[IC_CTRL] ; Write the new mode settings to IC_CTRL
```

The next lines set up the IC\_RAM\_ADDR register to access the cache RAMs at address ram\_addr.

```
mov r0,ram_addr
sr r0,[IC_RAM_ADDR] ;Prepare to address cache RAMs from address 'ram_addr'
```

The next lines read all three cache RAMs directly using the address specified in IC\_RAM\_ADDR, by using the LR instruction.

```
lr r1,[IC_TAG] ; r1=tag_ram[IC_RAM_ADDR];
lr r3,[IC_DATA] ; r3=data_ram[IC_RAM_ADDR];
```

The following code modifies the contents of all the RAMs and writes the new values back into the instruction cache RAMs using the SR instruction.

```
add r1,r1,1 ;Modify IC_TAG contents
add r3,r3,1 ;Modify IC_DATA contents
sr r1,[IC_TAG] ;Write into IC_TAG using contents of r1
sr r3,[IC_DATA] ;Write into IC_DATA using contents of r3
```

To read the contents of the cache again to ensure it has been indeed modified, use an SR instruction to IC\_RAM\_ADDR:

```
sr r0,[IC_RAM_ADDR] ;Prepare to address cache RAMs from address 'ram_addr'
lr r1,[IC_TAG] ; r1=tag_ram[IC_RAM_ADDR];
lr r3,[IC_DATA] ; r3=data_ram[IC_RAM_ADDR];
```

## 5.3 Debugger View of Memory

The debugger sees the memory space the same way as the ARC EM processor core. Therefore the debugger has access to all memory components, including the closely coupled memories, external memory and external peripherals. See the *MetaWare Debugger User's Guide* for more information about the debugger.



# 6

## Memory-Protection Unit

ARC EM provides several memory-protection options, which are described in this chapter. These protection options include:

- Stack protection: checks overflow and underflow of reserved stack space
- Code protection: hardware means to block read and write to code space.
- Memory Protection Unit: defines variable regions and assigns access attributes

The different protection schemes may be combined to achieve several levels of protection against malicious or misbehaving code in critical applications.

### 6.1 Stack Checking

This section describes the hardware stack-checking option that monitors stack overflow and underflow. The logic triggers an `EV_PROTV` exception when a violation occurs. Topics covered include configuring this option for your core, enabling the option at run time, and programming the stack range.

#### 6.1.1 Configuring Stack Checking

You can add stack-checking logic to your build by specifying the following option or making the equivalent GUI selection:

```
-stack_checking true
```

When stack checking is configured, the `STACK_REGION_BUILD` build configuration register and the auxiliary registers for specifying the stack memory regions are present.

#### 6.1.2 Build Configuration Register

`STACK_REGION_BUILD` (0xC5) identifies the presence and version of stack checking in the build.

See the *ARCV2 ISA Programmer's Reference* for a register diagram and other register details.

#### 6.1.3 Enabling Stack Checking

The kernel can enable stack checking by setting the `SC` bit (bit 14) in the `STATUS32` register. In user mode, this bit is not writable and returns zero on read. If stack checking is not configured for the core, then this bit is not writable and returns zero on read.

## 6.1.4 Specifying Stack Regions

Specify the memory regions reserved for the stack by loading their addresses into the following auxiliary registers. Separate register pairs define the kernel-mode and user-mode stack regions.



**Note** The stack grows downward.

- USTACK\_TOP (0x260) is the user-mode lower address limit.
- USTACK\_BASE (0x261) is the user-mode upper address limit.
- KSTACK\_TOP (0x264) is the kernel-mode lower address limit.
- KSTACK\_BASE (0x265) is the kernel-mode upper address limit.

These registers can be read and written only in kernel mode. In user mode, the registers are not writable and return zero on read.

All stack operations are performed on 32-bit words, so the lower two bits of the address are ignored; write them as zero.

The number of valid bits is limited to the configured address size of the processor. See the *ARCV2 ISA Programmer's Reference* for register diagrams and other register details.

## 6.1.5 Stack-Checking Operation

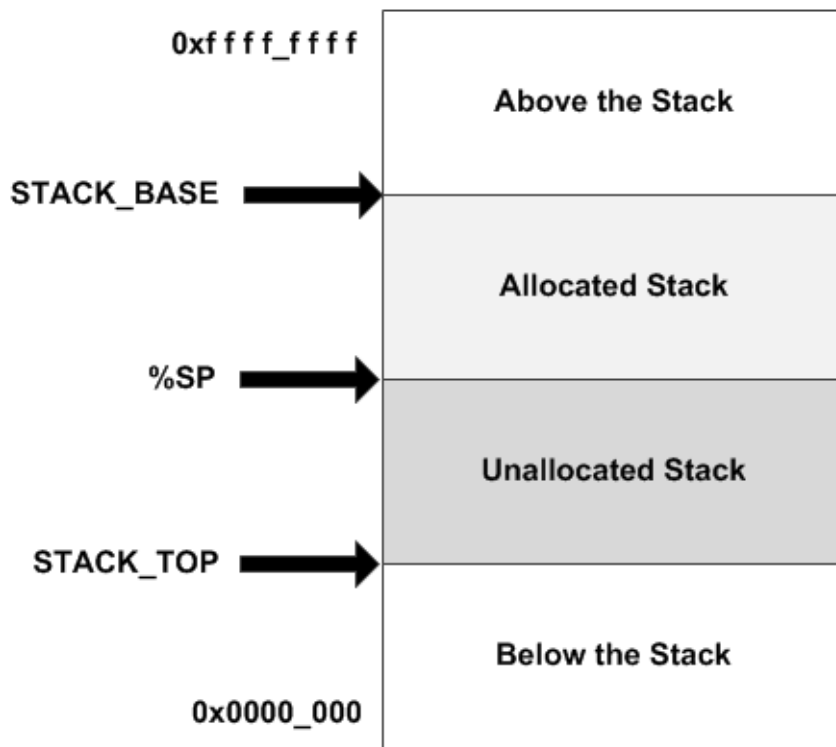
The reserved stack region can be accessed by explicit instructions (such as LD or ST) or by implicit operations (such as an interrupt sequence).

The stack-checking logic differentiates between explicit instructions, which use %SP as the base register and other instructions, which use other addressing modes to access the allocated stack region. This distinction achieves functional safety while allowing common compiler operations to take place.

### 6.1.5.1 Stack Regions

Figure 6-1 shows the stack regions, as defined by the range registers and the architectural stack-pointer register (%SP). The allocated and unallocated regions make up the reserved stack space. The stack grows downward.

- %SP is the architectural stack-pointer register, which usually points to the top of the stack. It is always r28 in ARC EM processors.
- The allocated stack region is defined as all addresses from (and including) %SP up to (but excluding) STACK\_BASE.
- The unallocated stack region is defined as all addresses from (and including) STACK\_TOP up to (but excluding) %SP.
- STACK\_BASE points to the first 32-bit word above the reserved stack space, and hence all addresses greater than or equal to STACK\_BASE are above the stack.
- STACK\_TOP points to the last allocateable word in the stack region, and hence all addresses less than STACK\_TOP are below the stack.

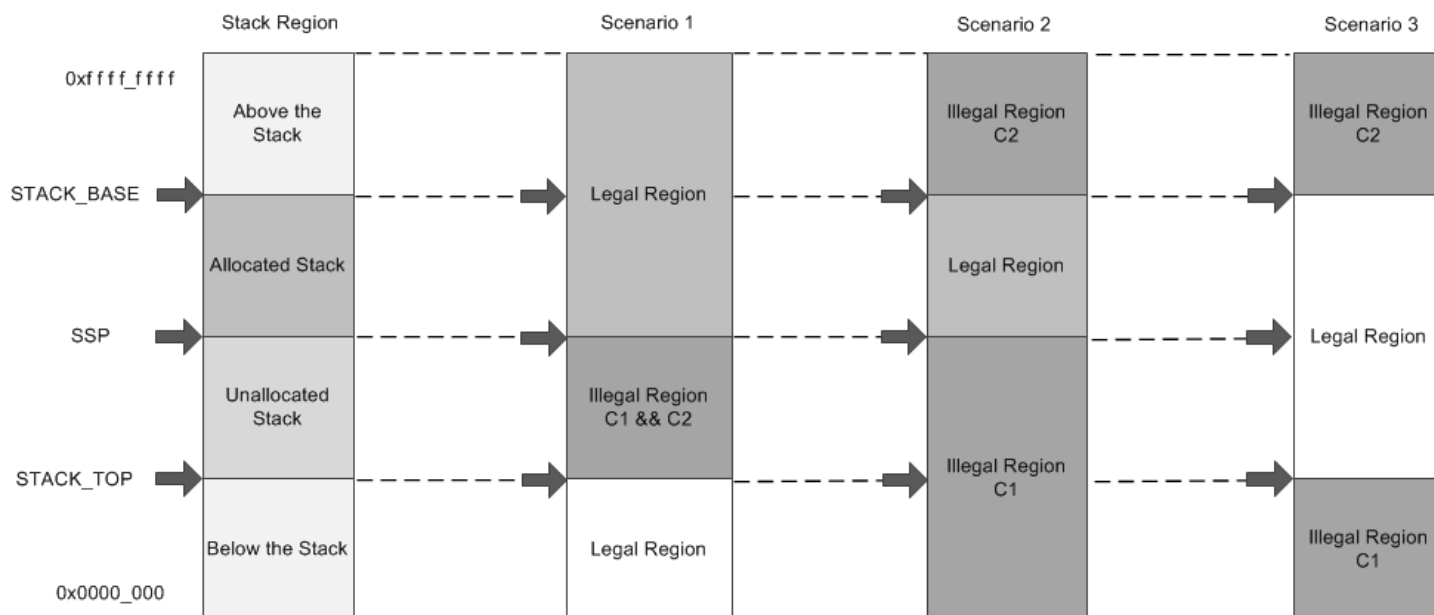
**Figure 6-1 Stack Regions**

#### 6.1.5.2 Stack Access Rules

- Non-%SP based memory references may not reference the unallocated stack region.
- %SP-based memory references that do not modify %SP must be within the allocated stack region.
- %SP-based memory references that modify %SP must be within the reserved (allocated and unallocated) stack space.

Violation of one of these rules when stack checking is enabled results in an `EV_ProtV` exception.

[Figure 6-2](#) and [Table 6-1](#) illustrate the conditions checked in three possible scenarios.

**Figure 6-2 Stack-Checking Scenarios****Table 6-1 Conditions Checked in Stack-Checking Scenarios**

|                 | <b>Scenario 1</b><br><b>Non-%SP-Based Address.</b><br><b>Example: LD r0, [r1];</b> | <b>Scenario 2</b><br><b>%SP-Based, No Auto-Update of %SP</b><br><b>Example: LD r0, [r28];</b> | <b>Scenario 3</b><br><b>%SP-Based, with Auto-Update of %SP</b><br><b>Example: LD.a r0, [r28];</b> |
|-----------------|------------------------------------------------------------------------------------|-----------------------------------------------------------------------------------------------|---------------------------------------------------------------------------------------------------|
| Conditions      |                                                                                    |                                                                                               |                                                                                                   |
| Condition C1    | address < SSP                                                                      | address < SSP                                                                                 | address < STACK_TOP                                                                               |
| Condition C2    | address ≥ STACK_TOP                                                                | address ≥ STACK_BASE                                                                          | address ≥ STACK_BASE                                                                              |
| Condition C3    |                                                                                    | address < STACK_TOP                                                                           |                                                                                                   |
| Error Condition | C1 && C2                                                                           | C1    C2    C3                                                                                | C1    C2                                                                                          |

**Note**

Stack-pointer updates from multi-cycle instructions are not checked for being out of bounds.



## 6.2 Code Protection

This section describes the optional code protection mechanism in the ARC EM processor.

The code protection option is intended to prevent piracy and hacking of code memory by blocking unauthorized loads and stores without impacting processor performance.

### 6.2.1 Configuring Code Protection

You can add code-protection logic to your build by the following option or making the equivalent GUI selection:

```
-code_protection true
```

### 6.2.2 Code Protection Scheme

The ARC EM architecture divides the memory into 16 regions, which can be protected individually.

This feature adds a 16-bit input to the processor core, one bit per region. When the protect bit is set, the processor disables any load or store to the corresponding region. An attempt to access a protected region raises an EV\_ProtV exception.

The protect bits are driven by external logic (such as a register or fuse).

For details on code protect signals, see [Table 9-31, Chapter 9, “External Interfaces”](#). Protect bits can be changed safely only when the processor is halted.



#### Attention

This feature offers only basic protection, and is not a substitute for advanced protection methods. Code residing in external memory is potentially exposed during instruction fetch.

### 6.2.3 Code Protection Behavior

This section describes how the code protection scheme behaves in relation to specific scenarios.

#### 6.2.3.1 Processor Reset

Protection is valid immediately after reset.

#### 6.2.3.2 User and Kernel Mode

The code protection scheme applies to both user and kernel modes.

#### 6.2.3.3 Stack Location

For proper program execution, ensure that the stack is not placed in a code-protected region.

If the stack enters a protected region during automatic context save or restore on interrupt entry or exit, the interrupt is not taken, and an EV\_ProtV exception is raised. If the processor is already in exception mode (STATUS32.AE == 1) then a MachineCheck exception is raised.

#### 6.2.3.4 DMI Access

DMI access violations do not raise code-protection exceptions. Write commands from DMI are ignored, and read transactions return zero data.

### 6.2.3.5 Debug

Code-protection violations do not raise exceptions during debug access. The processor merely ignores load and store instructions in code-protected regions. This scheme allows single-stepping in a code-protected region.

### 6.2.3.6 DFT Scan

Code protection is not available when the core is accessed through the DFT (Design-for-Test) scan chain.

### 6.2.3.7 Code Protection Exception

The EV\_ProTV exception is taken with exception priority 17, the same as other data-access protection violations.

For details on the ECR register, see the *ARCV2™ ISA Programmer's Reference Manual*.

### 6.2.3.8 Coexisting Protection Schemes

Code protection can be used in conjunction with stack checking, MPU or MMU.

If multiple protection schemes raise exceptions simultaneously, they all share the same vector and cause code in the ECR register. However, each protection scheme sets its own distinct bit in the 8-bit parameter field in the ECR register, as follows:

- Code-protection violations set bit 0.
- Stack-checking violations set bit 1.
- MPU permission violations set bit 2.
- MMU permission violations set bit 3.

## 6.3 Memory-Protection Unit

The ARC EM architecture divides the memory into 16 equal-size regions, which may be too coarse for assigning protection attributes.

The optional Memory-Protection Unit (MPU) provides means for defining up to 16 variable-size regions and assigns read, write, and execute attributes per each of them in user and kernel modes. Access violations raise a protection violation exception.

### 6.3.1 Configuring an MPU

The ARC EM MPU is an optional component for ARC EM processor cores.

You can add an MPU using the ARChitect GUI. You can then configure the number of regions by changing the GUI parameter or using the following command line:

```
-mpu_num_regions N
```

Specify the number of memory regions as the argument. Acceptable values are 1, 2, 4, 8, and 16. The default is 8.

### 6.3.2 Enabling the MPU

All MPU registers except MPU\_ECR contain zero after reset. MPU\_ECR contains 0x00060000 after reset. To enable MPU, set MPU\_EN register to 0x40000000.

For MPU register details and exact specifications of the protection attributes, see the *ARCV2 ISA Programmer's Reference*.

### 6.3.3 Overview of MPU Operation

The MPU provides protection by dividing the address space into regions associated with specific attributes such as read, write, and execute. If an attempt is made to access a region for which an associated attribute is not permitted, the ARCV2-based processors raises an EV\_ProTV exception, and prevents the faulting instruction from completing. For details on MPU registers and operation, see the *ARCV2 ISA Programmer's Reference*.



# 7

## Interrupts

ARCV2-based processors support interrupts and exceptions. Apart from memory-error exceptions, most exceptions are synchronous to an instruction and occur at the same place each time a program is executed. Interrupts are asynchronous events, typically generated by unscheduled external input signals. ARCV2-based processors also support software interrupts, which do not have an associated input and therefore are invoked only by software.

Thus ARC EM interrupts can be classified as follows:

- Timer — triggered by one of the optional extension timers
- External — available as input pins to the core
- Software — triggered by software only

Exceptions are typically taken immediately, while interrupts allow certain operations to complete, for example, a multi-cycle multiply operation or an instruction-cache refill operation. The ARCV2 architecture has 16 fixed exceptions assigned to vectors 0 to 15 and 240 interrupts at vectors 16 to 255. The actual exceptions and interrupts implemented in a design depend on the features and parameters chosen during configuration.

The ARC EM interrupt architecture defines a vector table containing contiguous four-byte addresses as each exception or interrupt vector. As up to 256 exceptions and interrupts are supported, the vector table can be up to 1 KB in length. The processor can execute instructions contained within the vector table, except for the reset-vector location (see `-intvbase_preset` option in “[Interrupt Options](#)” on page 61). Do not locate executable code at vector locations used for exceptions or interrupts, as these locations include vectors referencing the appropriate interrupt or exception handlers

**Note**

This chapter provides a brief overview of the interrupt architecture. For complete details on programming interrupt and exception handling, see the *ARCV2 ISA Programmer's Reference*.

For details about minimizing interrupt latency, see [Chapter 8, “Cycle-Timing Characteristics”](#) as well as the FIRQ fast-interrupt option described in the *ARCV2 ISA Programmer's Reference*.

## 7.1 Interrupt Architecture

The ARC EM interrupt unit handles interrupts from the optional timers as well as external interrupt inputs. External interrupts are synchronized to the internal clock and are individually programmable as pulse- or level-sensitive.

The interrupt unit supports up to 16 priority levels and a fast interrupt mechanism (FIRQ) for very-low-latency applications.

- **Fast interrupts** — For the highest-priority interrupts (P0), the processor stores only partial user context (PC and STATUS32 registers) on interrupt entry. This option is intended to avoid memory transactions in the entry and exit sequences for interrupts at level P0.
- **Regular interrupts** — The selected user context is saved and restored to the stack.

For specific information on interrupt entry and exit sequences for fast and regular interrupts, refer to the *ARCo2 ISA Programmer's Reference*.

### 7.1.1 Interrupt-Unit Features

The interrupt unit supports the following features:

- Up to 240 interrupts.
- Up to 16 priority levels: P0 is the highest, and P15 the lowest.
- Individually programmable priority for each interrupt.
- Individually programmable pulse or level sensitivity for each interrupt.
- Automatic save and restore of selected registers upon interrupt entry or exit for fast context switching.
- Context saving to user or kernel stack under program control.
- Optional second core-register bank may be used with fast interrupts (FIRQ) option to minimize latency.
- Ability to disable interrupt levels by setting the priority-level threshold in the STATUS32.E field.
- Interrupts from timer 0 and timer 1 statically assigned to vectors 16 and 17.
- External interrupts dynamically assigned to the lowest available interrupt vector.

### 7.1.2 Interrupt Unit Configuration

The interrupt unit is optional in ARCo2-based processors. To include interrupts in a configuration, add this optional processor element into the core in the ARChitect GUI.

You can customize the interrupt unit by selecting values for the configuration options listed in [Additional Options](#). Up to 240 interrupts can be configured, with up to 16 priority levels. The number of interrupts configured must be at least the sum of the number of timers and the number of external interrupts. Additional software-triggered interrupts may also be included.

The number of interrupts and priority levels selected should not be larger than necessary, as each contributes to the cost in size and power dissipation of the final design.

Interrupts from timer 0 and timer 1, if present, are statically assigned the lowest-numbered interrupt vectors: vector 16 for timer 0, and vector 17 for timer 1. External interrupts, if any, are assigned any non-timer interrupt number, with the lowest available unused interrupt vectors allocated first, followed by software-triggered interrupts, if any, which occupy the highest-numbered interrupt vectors used.

The `-firq_option` adds the fast-interrupt mechanism. When used with the optional second register bank, this option lets the highest-priority level interrupts (P0) avoid saving and restoring context. The result is extremely low latency to start executing the interrupt handler.

Regular (non-FIRQ) interrupts also offer reduced latency by automatically saving selected context in hardware, reducing code size and the time to fetch code for context saving.

## Additional Options

You can use the following options to configure the interrupts.

- `-number_of_interrupts` (0 to 240): Number of interrupts in the interrupt controller
- `-number_of_levels` (0 to 16): Priority levels in the interrupt controller
- `-external_interrupts` (0 to 240): Number of external interrupts
- `-firq_option` (0 or 1): Fast-interrupt option

### 7.1.3 Enabling Interrupts

Interrupts are enabled when the IE bit in the STATUS32 register is set by the SETI instruction or when STATUS32 is restored by an RTIE instruction.

Interrupts can be disabled by clearing the IE bit in STATUS32 using the CLRI instruction or when STATUS32 is restored by an RTIE instruction.

The E field of the STATUS32 register sets a priority-level threshold to selectively disable lower-priority interrupts.

### 7.1.4 Interrupt Priorities

Priority 0 is the highest interrupt priority, and priority 15 is the lowest.

Each configured interrupt has its own priority register. Each priority register has up to four bits. The number of bits is calculated as follows:  $n = \text{ceil}[\log_2(N)]$ , where  $N$  is the number of configured priority levels.

Precedence within a priority level is based on the physical interrupt number, where the lower-numbered interrupt has a higher precedence. After it is accepted, an interrupt may only be preempted by an interrupt at a higher priority level.

### 7.1.5 Pulse-Triggered Interrupts

Each interrupt can be programmed using its IRQ\_TRIGGER register to be either pulse-sensitive or level-sensitive.

ARC EM pulse-triggered interrupts use a synchronous positive-transition capture scheme. When the processor is not in a SLEEP mode, the minimum pulse width for an input interrupts is two clock cycles in duration.

When the ARC EM processor is asleep, the core clock is turned off to reduce dynamic power consumption in the clock tree. Under these circumstances, the interrupt unit does not respond to pulse-sensitive interrupts. Users should therefore ensure that all wake-up interrupts are programmed to be level sensitive.

If a level-sensitive interrupt is asserted when the processor is asleep, and its programmed priority is sufficient to interrupt the sleeping processor, the core clock is turned on to allow the processor to wake up and respond to the interrupt. When the core clock is re-enabled by a level-sensitive interrupt, any pulse-sensitive interrupt line is also enabled. Therefore, any higher-priority pulse-sensitive interrupt that is asserted at that time is taken in preference to the interrupt that turned on the clock.

If a debug operation is initiated when the processor is asleep, it turns on the core clock to enable the debug transaction to access the internal state of the core. At this time, pulse-sensitive interrupts are enabled briefly, and may wake up the processor if one is asserted and is of sufficient priority to wake up the processor.

Therefore, to ensure predictable wake-up behavior it is advised that only level-sensitive interrupts be used as wake-up interrupts.

For additional details on interrupt sensitivity-level programming, see the *ARCV2 ISA Programmer's Reference*.

### 7.1.6 Bank Configuration and Second Register Set

ARC EM processors permit one additional register bank for fast context switching. The number of registers in the additional bank is programmable.

Option `-rgf_num_banks` defines the number of register banks in the processor (1 or 2).

When `-rgf_num_banks` is 2, you can additionally use `-rgf_banked_regs` to define the number of core registers contained in the second bank.

See the *ARCV2 ISA Programmer's Reference* for details on which registers are replicated for different register-file sizes.



#### Note

The register file is always implemented as flip-flops when `-rgf_num_banks` is greater than zero. The currently selected bank number is defined by the three-bit field `STATUS32.RB[2:0]`.

### 7.1.7 Reducing Context-Switching Overhead

The ARC EM interrupt architecture enables fast context switching by automating the save and restore of a partial context on interrupt entry and exit for normal interrupts. For fast interrupts (FIRQ), the save-and-restore sequence is completely omitted. For the fastest interrupt response, assign the most time-critical interrupts to priority level 0 and use the FIRQ option.

#### Partial Context Automatically Saved

The partial context saved for normal interrupts comprises the following registers:



- Context associated with optional features, such as JLI\_BASE, LDI\_BASE and EI\_BASE from the code-density option
- PC
- STATUS32
- An optional set of core register pairs, starting from r0
- Optional BLINK register (r31)
- Optional LP\_COUNT, LP\_START, and LP\_END registers

The AUX\_IRQ\_CTRL register defines the partial context. For further details, see the discussion of interrupt handling in *ARCV2 ISA Programmer's Reference*.

### 7.1.8 Determining Configured Interrupts

To determine which interrupts are configured in a processor, use the following procedure:

1. Disable interrupts.
2. Write the interrupt number to the IRQ\_SELECT register.



The interrupt number must be between 16 and 255. Interrupt vectors less than 16 are reserved for exceptions.

3. Write the interrupt number to the AUX\_IRQ\_HINT register.
4. Read the IRQ\_STATUS.IP bit.

If the IRQ\_STATUS.IP bit is set to 1 for an interrupt number, then that interrupt is configured. If IRQ\_STATUS.IP is set to 0 for an interrupt number, then that interrupt is not configured.

5. Write 0 to the AUX\_IRQ\_HINT register to clear all interrupts.



Repeat Step 2 through Step 5 for each interrupt.

6. Enable interrupts.

## Pseudo code

```
{
CLRI R0                                ;; disable interrupts before manipulating IRQ_SELECT
do {
    write x to the IRQ_SELECT register
    write x to the AUX_IRQ_HINT register
    read from the IRQ_STATUS register    ;;if IRQ_STATUS.P bit is set to 1,
                                        ;;then vector x is defined as an interrupt.
                                        ;;Otherwise vector x is not an interrupt.
                                        ;;If x < 16, then x is an exception
                                        ;;otherwise, x is beyond the interrupt region.
}
write 0 to the AUX_IRQ_HINT register    ;;clear all software interrupt hints
SETI R0                                ;; re-enable interrupts
}
```

## 8

# Cycle-Timing Characteristics

This chapter describes the cycle-level timing behavior of the ARC EM processor core. Information in this chapter can be used to determine the execution time of individual instructions or code sequences.

## 8.1 ARC EM Pipeline Overview

The ARC EM processor has a short, three-stage core pipeline, designed for minimal logical complexity and low power consumption. The stages are listed in [Table 8-1](#), with a summary of their functionality.

**Table 8-1 ARC EM Pipeline Stages**

| Pipeline Stage | Summary                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                         |
|----------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Fetch          | <ul style="list-style-type: none"> <li>Access instruction memories, including instruction cache, ICCM, and IFQ</li> <li>Extract and align instructions that might span individual fetch packets</li> <li>Decode source register addresses</li> <li>Implement zero-overhead loop pre-fetch mechanism</li> <li>Sequence the issuing of micro-operations for multi-cycle instructions</li> </ul>                                                                                                                                                   |
| Execute (XA)   | <ul style="list-style-type: none"> <li>Read source registers from the general-purpose register file</li> <li>Decode each instruction's operator and operand formats</li> <li>Perform all single-cycle ALU operations (arithmetic, logical, shift, etc.)</li> <li>Compute memory addresses for load and store instructions</li> <li>Select the next data-memory operation to be performed</li> <li>Determine the addresses used in auxiliary register read/write operations (LR and SR)</li> <li>Perform branch and jump instructions</li> </ul> |
| Commit (CA)    | <ul style="list-style-type: none"> <li>Prioritize and perform interrupts and exceptions</li> <li>Detect and handle any mis-speculation in the program sequence</li> <li>Access data memories, including data cache, DCCM, and external memory</li> <li>Commit processor state by writing destination registers and updating auxiliary registers</li> </ul>                                                                                                                                                                                      |

### 8.1.1 Instruction Sequencing

In general, the ARC EM pipeline is fully protected, and all restartable exceptions are precise. For this reason, the architectural processor state cannot be updated before an instruction graduates from the commit stage. Any speculative state in the fetch, execute or commit stages is discarded when a pipeline flush is executed.

The ARC EM pipeline defines a sequence of four actions that happen to each instruction as it executes. These are:

1. Issue
  2. Dispatch
  3. Commit
  4. Retire
- An instruction is considered to be *issued* when the fetch stage has passed it to the execute stage, and the instruction is ready to be decoded and executed.
  - An instruction is considered to be *dispatched* when execution of the instruction begins. An instruction that is issued to the execute stage might be delayed in its dispatch by a downstream stall (from the commit stage) or by the non-availability of operands. A dispatched instruction immediately enters the commit stage.
  - An instruction is considered to be *committed* when it graduates successfully from the commit stage. Some instructions might be discarded before they are committed, in which case they have no effect on the architectural state of registers and memory. A committed instruction is guaranteed to update the state according to the semantic definition of that instruction. The update of state is referred to as retiring the instruction.
  - An instruction is considered to have *retired* when the state changes resulting from the successful execution of that instruction have been applied to the architectural state of the processor and its memory system. Retirement for most instructions happens simultaneously with committing. However, some multi-cycle multiplication, division, and extension instructions might retire several cycles after they are committed.

All instructions are issued, dispatched, and committed in strict program order, but the retirement of results may happen out of order when multi-cycle multiplication, division, and extension operations are involved.

Requests to flush the pipeline originate in the commit stage, and can be caused by exceptions, interrupts, mis-speculations or mis-predictions. Flush requests are delayed until all previously committed instructions have retired.

### 8.1.2 Pipeline Hazards

The ARC EM pipeline is capable of executing instructions at a rate of one per clock cycle, by overlapping the fetch, execute and commit stage actions for up to three instructions simultaneously. The peak execution rate of one instruction per cycle is often achieved for well-written code, though in practice the presence of pipeline hazards can reduce the throughput to less than one instruction per clock cycle.

Pipeline hazards have three causes:

- Data dependencies, where information passes from one instruction to another, requiring the receiving instruction to wait for the availability of the new data.

- Control dependencies, where an update to the program counter (PC) means that the instructions behind it in the pipeline should not be executed.
- Structural hazards, where an instruction requires a hardware resource that is currently busy, and must wait.

Other less certain, less frequent events might require the pipeline to be flushed of all instructions. When this happens, the fetch stage restarts execution from after the last committed instruction, which typically leads to a penalty of three clock cycles.

To understand the causes and effects of pipeline hazards, consider the Definition Point (DP) and Use Point (UP) for each item of data that instructions produce or consume. For example, the DP of the destination register associated with an ADD instruction is the commit stage, whereas the UP of all source registers is the execute stage. From this it is clear that the ADD instruction has an effective latency of one cycle; the immediately following instruction can use the result of the ADD instruction without having to wait. This is because the consuming instruction is at the UP in the same clock cycle that the producing instruction is at the DP.

The DP may occur earlier than the point at which a value is retired. This situation indicates that a speculative result is supplied to subsequent instructions through a result-bypass network.

## 8.2 Instruction Timing

The following general rules govern the progression of instructions through the ARC EM pipeline:

1. An instruction that uses data as input cannot move beyond that instruction's UP until the DP for the data it uses is reached by the producing instruction.
2. An instruction that is ready to move forward in the pipeline cannot do so unless the succeeding stage is empty, or the instruction in the succeeding stage is also ready and able to move forward.

These general rules yield a more precise indication of pipeline timing than is possible by ascribing fixed latencies to individual instructions.

Instructions are grouped by timing category, allowing a more concise set of timing rules to be defined.

**Table 8-2 Instruction Categories for Timing Calculations**

| Instruction Category | Instructions                                                                                                                                                                                                                                                                                                                                |
|----------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ALU                  | ■ ADD, ADC, AEX, SUB, SBC, AND, OR, BIC, XOR, MAX, MIN, MOV, TST, CMP, RCMP, RSUB, BSET, BCLR, BTST, BXOR, BMSK, BMSKN, ADD1, ADD2, ADD3, SUB1, SUB2, SUB3, ASL, ASR, LSR, ROL, ROR, RLC, RRC, NOT, NOP, SEXB, SEXH, EXTB, EXTH, NEG, ABS, NORM, NORMH, FFS, FLS, SWAP, SWAPE, ASR16, ASR8, LR, LSL16, LSL8, LSR16, LSR8, ROL8, ROR8, SETcc |
| Load                 | ■ EX, LD, LDH, LDB, LDI, LDI_S, POP_S, PREFETCH,                                                                                                                                                                                                                                                                                            |
| Store                | ■ ST, STH, STB, PUSH_S                                                                                                                                                                                                                                                                                                                      |
| B_flag               | ■ Bcc, BLcc, Jcc, JLcc, LPcc                                                                                                                                                                                                                                                                                                                |
| B_cmp                | ■ BRcc, BBIT0, BBIT1                                                                                                                                                                                                                                                                                                                        |
| Multiop              | ■ ENTER_S, LEAVE_S                                                                                                                                                                                                                                                                                                                          |

**Table 8-2 Instruction Categories for Timing Calculations (Continued)**

| Instruction Category | Instructions                                                                                |
|----------------------|---------------------------------------------------------------------------------------------|
| MPYW                 | ■ MPYW, MPYUW                                                                               |
| MPYL                 | ■ MPY, MPYU                                                                                 |
| MPYMF                | ■ MPY.F, MPYU.F, MPYH, MPYHU, MPYH.F, MPYHU.F                                               |
| DIV                  | ■ DIV, DIVU, REM, REMU                                                                      |
| JLI                  | ■ JLI_S                                                                                     |
| SERIAL               | ■ SR to serializing register, CLRI, FLAG, KFLAG, SYNC, BRK, SWI, SETI, SLEEP, TRAP_S, RTIE, |
| EI_S                 | ■ EI_S                                                                                      |

### 8.2.1 Instruction-Fetch Timing

The time required to fetch and issue a complete instruction depends on the following factors:

- The size of the instruction
- The alignment of the instruction
- The access time of the memory subsystem from which instructions are fetched

ARC EM processors fetch instructions at a rate of up to 32 bits per clock period. Instruction-memory accesses that target ICCM, or that hit in the instruction cache or IFQ, complete in one cycle. Those that experience an instruction cache or IFQ miss take longer.

The number of instruction-memory accesses required for each instruction depends on the size of the instruction, and might also depend on whether the instruction follows sequentially from the previous instruction. A sequential fetch is one that accesses the next memory address in sequence after the previous instruction fetch. The total number of instruction-memory accesses required for each fetch is detailed in [Table 8-3](#).

**Table 8-3 Number of Instruction Memory Accesses Per Instruction Fetch**

| Instruction Bytes | PC Word Aligned? | Sequential Fetch? | Number of memory Accesses Required |
|-------------------|------------------|-------------------|------------------------------------|
| 2                 | Y                | Y                 | 0                                  |
|                   |                  | N                 | 1                                  |
|                   | N                | Y                 | 0                                  |
|                   |                  | N                 | 2                                  |

**Table 8-3 Number of Instruction Memory Accesses Per Instruction Fetch**

| Instruction Bytes | PC Word Aligned? | Sequential Fetch? | Number of memory Accesses Required |
|-------------------|------------------|-------------------|------------------------------------|
| 4                 | Y                | Y                 | 1                                  |
|                   |                  | N                 | 1                                  |
|                   | N                | Y                 | 1                                  |
|                   |                  | N                 | 2                                  |
| 6                 | Y                | Y                 | 2                                  |
|                   |                  | N                 | 2                                  |
|                   | N                | Y                 | 1                                  |
|                   |                  | N                 | 2                                  |
| 8                 | Y                | Y                 | 2                                  |
|                   |                  | N                 | 2                                  |
|                   | N                | Y                 | 2                                  |
|                   |                  | N                 | 3                                  |

The length of time spent by each instruction in the fetch stage is  $\text{MAX}(1, T_f)$ , where  $T_f$  is the sum of the times taken to perform all accesses required to fetch the instruction.

The most costly type of instruction fetch is a misaligned, non-sequential 8-byte instruction, which requires three memory accesses. If each access is in ICCM or is an instruction cache or IFQ hit, then  $T_f = 3$  and such an instruction remains in the fetch stage for three cycles. However, if each memory access is uncached and accesses external memory at a cost of eight cycles per access, then  $T_f = 24$  and this instruction spends 24 cycles in the Fetch stage.

### Examples

- A two-byte ADD\_S instruction is fetched from address 0x00001002, immediately after the previous instruction is issued from address 0x00001000. This implies the third row of the table, indicating no memory access is required. This is because the instruction is still buffered from the previous access to location 0x00001000. Hence,  $T_f = 0$ , and the instruction spends 1 cycle in the Fetch stage.
- A six-byte MOV\_S instruction with long-immediate data is fetched from address 0x00001002, assuming the previous instruction was a branch to that address. According to row 12 of the table, this requires two memory accesses. The first will read from location 0x00001000, to retrieve the half-word at 0x00001002, and the second will read from address 0x00001004 to obtain the long-immediate data.

## 8.2.2 ALU Instruction Timing

The use point for ALU instructions, for all input data, is the execute stage. ALU instructions can define a result register and/or the status flags, if the .f option is enabled for that instruction. The definition point for all ALU result registers and status flags is the commit stage.

Consider the following simple example:

```
ADD    R1, R2, R3
SUB    R4, R4, R1
```

If the ADD instruction is in the commit stage when the SUB instruction is in the execute stage, then R1's defining ADD instruction is at its DP when the SUB instruction is at its UP. Hence, no data hazard is caused by the flow of information from the ADD to the SUB through register R1.

Consider a slightly more complex example:

```
ADD.F   R1, R2, R3
SUB.EQ  R4, R4, R1
```

The DP of the flag result from the ADD.F is again the commit stage, and the UP of the flags in the conditional test of the SUB.EQ is again the execute stage. Hence, this sequence also completes without a data hazard.

In general, sequences of back-to-back ALU operations such as these are able to sustain a throughput of one instruction per cycle without any stalls, provided they can be fetched and issued at that rate.

### 8.2.3 Load and Store instruction Timing

Load and store instructions have two components to their timing characteristics.

- They each have use and definition points for their operands and results.
- The length of time they spend in the commit stage depends on the type of memory they access and the speed of that memory.

The use point for both load and store source register operands is the execute stage, in common with most other instructions. Load and store instructions using the `.a`, `.aw` or `.ab` writeback modes redefine their source address register, with a DP in the commit stage, just as if the instruction performed a regular ADD operation on the source-address register.

Load instructions also define a load destination register, for which the DP is the cycle in which the Load instruction commits, plus *M* cycles, where *M* is the memory-access delay. The memory access delay is one cycle in the case of a DCCM access or a hit in the data cache, but is longer for external-memory accesses or data-cache misses.

Consider the following example:

```
LD      R1, [R2]
SUB     R4, R4, R1
```

In the ideal case, the DP of the load instruction is one cycle later than the earliest time that the SUB instruction could reach its UP. Hence, the SUB instruction might have to wait for one cycle in the execute stage. This situation is referred to as a *load-use stall*.

Load-use stalls can be eliminated by placing an independent instruction between the load and any immediate user of the loaded data, as shown below:

```
LD      R1, [R2]      ;; producer of R1
LD      R3, [R2,4]    ;; independent instruction
SUB     R4, R4, R1    ;; user of R1
```

The independent instruction can also be a Load instruction, as in this example.



If these load instructions are serviced by the DCCM or hit in the data cache, these three instructions execute at a rate of one instruction per cycle. However, if there is a data cache miss, or if the peripheral data region is accessed, then the load remains in the commit stage for an additional  $M$  cycles, where  $M$  is the time required to access memory.  $M$  varies depending on the type of memory, the type of interface, and the speed of external components.

Load operations in the ARC EM processor always block the commit stage for  $M$  cycles. Consider the case of an uncached Load that accesses external memory:

```
LD.DI R1, [R2]
LD     R3, [R2,4]  ;; independent instruction
SUB    R4, R4, R1
```

If the external memory access takes  $M=8$  cycles, then the LD.DI instruction remains in the commit stage for eight cycles, and the following LD instruction remains in the execute stage during this period.

Store instructions also remain in the commit stage for  $M$  cycles, where  $M$  is again the memory system delay. For a DCCM access or a data-cache hit,  $M$  is one. For a Data Cache miss or an external write,  $M$  is larger.

A store followed by a load to the same address creates a data dependency through memory. Similar considerations to those for dependencies through registers also apply. The use point for loading from memory is the execute stage, and the definition point for storing to memory is normally the commit stage. An exception to this rule occurs when a memory location is present in the data cache and an unbroken sequence of loads follows the store, and one of them accesses the stored location. In the case of a sequence of cached loads after a cached store, the committed store data is saved in a buffer until the load sequence is complete. If any load instruction in the sequence tries to access the same location, it stalls for three cycles in the execute stage to allow the pending write to be retired.

## 8.2.4 Branch and Jump Instruction Timing

Instructions in the B\_flag category base their branching decision on value of the status flags (STATUS32 bits ZNCV). Their UP is the execute stage, allowing them to make decisions on the status result of the preceding instruction without incurring a stall. For example, the BEQ instruction in the example below does not incur a stall as a result of the flags being set by the immediately preceding instruction:

```
CMP    R1, R2
BEQ    target
```

B\_cmp operations base their branching decision on the result of an arithmetic comparison performed in the execute stage. Their UP is therefore also the execute stage, though their operands are general-purpose registers and literals rather than flag values.

If a branch or jump instruction specifies a delay-slot instruction (i.e. the .d bit is set) then the delay-slot instruction is always executed regardless of the outcome of the branch or jump condition.

Any modification to PC as a result of taking a branch or jump creates a potential control hazard and means that the fetching of the target instruction cannot begin until the cycle in which the branch instruction is dispatched. If a delay-slot instruction is involved, it can be issued from the fetch stage as the branch is dispatched from the execute stage when the target instruction fetch begins. If no delay-slot instruction is involved, then a taken branch or jump flushes any instruction currently in the fetch stage, as such an instruction would be the wrong instruction to execute next. This creates a *bubble* in the pipeline, between the branch and its target, effectively adding one cycle to the execution time of the branch. For this reason, it is advisable to use delay-slot instructions wherever possible, as they normally eliminate the bubble.

For example, if the following BNE instruction is taken when executed, a gap of at least one cycle occurs between the dispatch of the BNE and the SUB instruction. No instruction commits during the cycle after the BNE commits, and so the BNE effectively takes two cycles to execute in such cases. The empty cycle is normally referred to as a bubble in the pipeline.

```

        ADD.F R1, R2, R3
        BNE   L1
L1:     SUB   R4, R4, 1

```

In contrast, if a delay-slot instruction is involved — in this case a store instruction — then no bubble occurs and the effective execution time of the BNE.D is 1 cycle:

```

        ADD.F R1, R2, R3
        BNE.D L1
        ST    R1, [R5]
L1:     SUB   R4, R4, 1

```

## 8.2.5 Two-Cycle Branches

The decision logic for B\_cmp instructions is more time-consuming than for B\_flag instructions, due to the requirement to perform a comparison before the branch decision is made. When the highest-possible target operation frequency is required, the B\_cmp decision logic can become timing-critical. The ARC EM processor offers a microarchitectural configuration option to reduce the logic-path length associated with B\_cmp instructions, enabling higher operating frequencies to be achieved. This is the `-turbo_mode` option (see “[Implementation Options](#)” on page 60). When this option is specified, the critical path is spread across two clock periods. In addition, B\_cmp instructions incur an additional one-cycle delay from issue to dispatch. Note, however, this path is not a multi-cycle path, and requires no special handling during synthesis.

## 8.2.6 Indexed Execution Timing

The EI\_S instruction initially branches to an instruction selected from a table of instructions, and then branches back to a position immediately after the EI\_S instruction. It therefore has timing characteristics similar to a sequence of two unconditional branches without delay slots. The first branch, to the table instruction, takes two cycles to complete. However, the timing of the return branch is overlapped with the timing of the table instruction itself. Therefore, a single-cycle instruction from the table, such as ADD, has a cycle behavior determined by the return branch, and effectively takes two cycles. Conversely, a multi-cycle table instruction, such as DIV, has a cycle behavior determined by the longer of the two actions, which is in the divide operation itself.



### Note

The cycle behavior of LDI\_S and JL\_S is similar to that of the 32-bit LD and JL.

## 8.2.7 Multiplier Instruction Timing

The ARC EM processor supports a range of different multiplier configurations, each offering a different trade off between speed and complexity.



### Note

Multiplier options wlh4 and wlh5 are not available when the enhanced radix-4 divider option is selected.

Table 8-4 lists the ARC EM Multiplier Design Options.

**Table 8-4 ARC EM Multiplier Design Options**

| Multiply Option Name | Hardware Implementation                          | Instructions Supported                    | Definition Points, by Instruction Group |            |             |
|----------------------|--------------------------------------------------|-------------------------------------------|-----------------------------------------|------------|-------------|
|                      |                                                  |                                           | MPYW                                    | MPY        | MPYM, MPY.F |
| w                    | 16 x 16 multiplier<br>Fully pipelined            | MPYW, MPYUW                               | Commit                                  | -          | -           |
| wlh1                 | 32 x 32 multiplier<br>Fully pipelined (1 stage)  | MPYW, MPYUW,<br>MPY, MPYU,<br>MPYM, MPYMU | Commit                                  | Commit     | Commit      |
| wlh2                 | 32 x 32 multiplier<br>Fully pipelined (2 stages) | MPYW, MPYUW,<br>MPY, MPYU,<br>MPYM, MPYMU | Commit + 1                              | Commit + 1 | Commit + 1  |
| wlh3                 | Two 16 x 16 multipliers<br>Blocking, sequential  | MPYW, MPYUW                               | Commit + 1                              | -          | -           |
|                      |                                                  | MPY, MPYU,                                | -                                       | Commit + 2 | -           |
|                      |                                                  | MPYM, MPYMU                               | -                                       | -          | Commit + 2  |
| wlh4                 | One 16 x 16 multiplier<br>Blocking, sequential   | MPYW, MPYUW                               | Commit + 1                              | -          | -           |
|                      |                                                  | MPY, MPYU,                                | -                                       | Commit + 3 | -           |
|                      |                                                  | MPYM, MPYMU                               | -                                       | -          | Commit + 4  |
|                      |                                                  | MPYM, MPYMU (chained)                     | -                                       | -          | Commit + 1  |
| wlh5                 | One 32x4 multiplier<br>Blocking, sequential      | MPYW, MPYUW                               | Commit + 4                              | -          | -           |
|                      |                                                  | MPY, MPYU,                                | -                                       | Commit + 8 | -           |
|                      |                                                  | MPYM, MPYMU                               | -                                       | -          | Commit + 8  |
|                      |                                                  | MPYM, MPYMU (chained)                     | -                                       | -          | Commit + 1  |

### 8.2.7.1 Multiplier-Operand Demotion

The operand-demotion optimization is automatically applied by all multi-cycle multipliers for which larger operands lead to longer execution times, namely wlh3, wlh4, and wlh5. This converts any 32 x 32 multiplication to an equivalent 16 x 16 operation if both operands of a MPY or MPYU instruction are representable as 16-bit quantities.

If operands can be demoted to 16-bit precision, then the resulting MPY or MPYU instruction assumes the same timing characteristics as the equivalent MPYW or MPYUW instruction for that hardware implementation.

Consider the following example:

```
MOV    R1, 0x00007431
MOV    R2, 0x0000FFFF
MPY    R3, R1, R2          ;; MPY gets demoted automatically to MPYW
```

As both R1 and R2 contain values that can be represented without loss of accuracy as 16-bit values, their product can be computed by a 16 x 16 multiplication operation to produce a 32-bit result. This optimization is detected by the multiplier hardware and requires no software support.

### 8.2.7.2 Chained Multiplications

Multiplier options wlh4, and wlh5 are capable of carrying forward the result of an MPY (or MPYU) to an immediately following MPYM (or MPYMU), if both source operands are registers and source registers are the same for both instructions. This takes advantage of the fact that the MPY (or MPYU) result is known to be identical to the sum of the initial partial products computed by the MPYM (or MPYMU).

When chaining takes place, the DP of the MPY (or MPYU) is unchanged, but the DP of the chained MPYM (or MPYMU) is reduced significantly.

Consider the following example, which is commonly used to compute the 64-bit product of two 32-bit operands:

```
MPY    R3, R1, R2
MPYM   R4, R1, R2          ;; chained operation
```

The presence of the first MPY result within the multiplier means that the second MPYM is able to complete two cycles after the MPY, rather than taking another 5 or 10 cycles.

### 8.2.7.3 Blocking Multiplier Repeat Rates

In the case of multiplier options wlh3, wlh4, and wlh5, the multiplier hardware takes several cycles to compute the result. During this time the multiplier is unable to accept any other multiplication, division, or remainder instruction. Any other instruction that does not have a data or flag dependency with the pending multiply (or divide/remainder) operation may proceed from the execute stage to the commit stage, and might complete ahead of the current multiplier operation.

Writing the multiply result to the register file is synchronized for correct order. The actual writeback can stall another instruction in the commit stage due to resource conflict.

Each multiplier operation blocks the multiplier from accepting a new multiply, divide, or remainder instruction until the result of the first operation has been retired to the register file. This block normally occurs at the definition points shown in the table “[ARC EM Multiplier Design Options](#)” on page 139. Blocked operations remain in the execute stage until the required multi-cycle multiplier unit is free.

## 8.2.8 Divide/Remainder Instruction Timing

When the `-div_rem_option` is `radix2` or `radix4_enhanced`, a bit-serial integer divider is present in the processor, supporting four additional instructions: `DIV`, `DIVU`, `REM` and `REMU`. These instructions compute the signed or unsigned integer division and remainder operations. The DP of all four operations is `commit+33` for the radix-2 divider option and a variable (data-dependent) cycle count for the enhanced radix-4 divider option.

The radix-4 divider is guaranteed to complete the divide operation in `commit+21` cycles, but can sometimes finish in as little as `commit+3` cycles.

### 8.2.8.1 Chained Divide and Remainder Operations

This optimization is implemented only in the radix-2 divider option. The divider computes both the quotient and remainder of the dividend and divisor it is given as operands, regardless of whether a `DIV` or `REM` operation is requested. `DIV` and `DIVU` return the quotient, whereas `REM` and `REMU` return the remainder. `DIV` and `REM` assume signed twos-complement operands, whereas `DIVU` and `REMU` assume unsigned integer operands.

If a second `DIV` or `REM` operation follows a first `DIV` or `REM` operation, the two operands are identical registers for both instructions, and neither register has been modified between the two operations, then the result of the second operation is pre-computed by the first operation. The second operation is then recognized in hardware as chained to the first operation, and the DP is reduced from `commit+33` to `commit+1`.

Consider the following example:

```
DIV  R3, R1, R2    ;; completes at T+33
ADD  R7, R6, R2    ;; completes at T+1
ST   R7, [R1]      ;; completes at T+2
REM  R4, R1, R2    ;; chained operation, completes at T+34
SUB  R9, R9, 1     ;; completes at T+34 if register file has 2 write ports
                        ;; or at T+35 if register file has 1 write port
```

In this example the `REM` instruction reuses the quotient + remainder result from the `DIV`, even though two intervening instructions occur. Furthermore, the `ADD` and `ST` instructions complete well before the `DIV` instruction.

However, consider the following two counter-examples, neither of which permit chaining:

```
DIV R1, R1, R2    ;; DIV modifies one of its source registers, R1
REM R4, R1, R2    ;; REM cannot be chained to DIV, as R1 was modified

DIV R3, R1, R2
LD R1, [R6]      ;; LD modifies one of the DIV source registers, R1
REM R4, R1, R2    ;; REM cannot be chained to DIV, as R1 was modified
```

### 8.2.8.2 Divider Repeat Rates

Each divide or remainder operation blocks the issue of any new multi-cycle multiply, divide, or remainder instruction until the result of the first operation is retired to the register file. This situation normally occurs at the definition points shown in the table “[ARC EM Multiplier Design Options](#)” on page 139. Blocked operations remain in the execute stage until the required multi-cycle multiplier unit is free.

When the `-div_rem_option` is enabled in conjunction with one of the fully pipelined multiplier options (w, wlh1, or wlh2), the divider is implemented independently of the multiplier functional unit, and pending DIV or REM operations do not block MPY operations.

### 8.2.9 Multi-Operation Instruction Timing

The ARCV2 instruction-set architecture provides instruction extensions to implement a complete function prologue or epilogue sequence as a single 16-bit encoded instruction. These `ENTER_S` and `LEAVE_S` instructions might perform a variable number of stack push or pop operations, frame pointer repositioning, and in the case of `LEAVE_S` might also transfer control back to the caller. The timing of these instructions therefore depends on the number of push or pop operations, and on the timing of each memory read or write access.

The rules for computing the instruction timing of `ENTER_S` and `LEAVE_S` are:

1. Every `ENTER_S` and `LEAVE_S` instruction spends 2 cycles in the execute stage.
2. In addition, if the instruction specifies  $i$  memory operations, each of those operations occupies the Commit stage for  $M_i$  cycles, where  $M_i$  is the time required to perform the memory read or write associated with an equivalent `POP_S` or `PUSH_S` instruction to the specified memory address.
3. If  $i \neq 0$ , then the instruction takes an additional cycle to update the stack pointer.
4. If the frame pointer is manipulated by the `ENTER_S` or `LEAVE_S` instruction, that also takes an additional cycle.
5. Finally, a `LEAVE_S` instruction specifying return jump requires one extra cycle if  $i \neq 0$ , or two extra cycles if  $i = 0$ .

### 8.2.10 Extension Instruction Timing

Extension instructions defined via ARCHitect's ARC Processor Extensions (APEX) interface can be of the following types:

- Non-blocking
- Blocking, and of fixed duration
- Blocking, and of variable duration

All APEX instructions have their use point at the execute stage, in common with baseline ALU operations.

#### 8.2.10.1 Non-blocking APEX Instruction Timing

Non-blocking APEX instructions are fully pipelined and do not hold up succeeding instructions unless they stall due the non-availability of their operands. The definition point for a non-blocking extension instruction depends on the latency of each specific extension instruction, and is `commit + latency - 1`.

All non-blocking extension instructions that have a latency greater than two are allowed to complete out-of-order, in a similar way to multi-cycle multiply or divide operations.

#### 8.2.10.2 Blocking APEX Instruction Timing

Blocking APEX instructions have their UP at the execute state and their DP at the commit stage. However, a blocking APEX instruction with latency  $L$  remains in the execute stage for  $L$  cycles.

If the latency of a blocking APEX instruction is set to 0, then it becomes a self-timed extension instruction. Through a simple handshake protocol, the extension instruction logic informs the processor when the latency period has expired, at which point the instruction is permitted to progress to the commit stage. The timing of such instructions is therefore implementation-dependent.

For more information on the APEX features, see *ARC EM APEX Databook*.

### 8.2.11 Serializing Auxiliary-Register Writes

Writes to certain auxiliary registers must take effect from the instruction immediately after the SR instruction that updates those registers. This is enforced in the ARC EM processor by designating such auxiliary registers as *serializing*, and ensuring that a pipeline flush takes place immediately after any SR to a serializing register commits.

The set of serializing auxiliary registers is:

- LP\_START
- LP\_END
- JLI\_BASE
- All actionpoint auxiliary registers
- All auxiliary registers related to data cache
- All auxiliary registers related to instruction cache and IFQ
- All auxiliary registers related to MPU

Three cycles are added to the execution time whenever a serializing auxiliary register is written.

### 8.2.12 Serializing Control Instructions

A few instructions require a pipeline flush whenever they execute. Any speculative calculations immediately following these instructions in the pipeline might base their calculations on stale data, and must therefore be discarded.

The set of serializing control instructions is:

- RTIE
- SYNC
- BRK or BRK\_S
- TRAP\_S
- SWI or SWI\_S
- SLEEP
- FLAG instructions that halt the processor
- LP or LPcc instructions that have a loop body of eight bytes or less.

Three cycles are added to the execution time whenever a serializing control instruction is executed.



### 8.2.13 Miscellaneous Instruction Delays

The first instruction fetch after the last instruction of a zero-overhead loop is delayed whenever a previously issued, but not yet committed, instruction specifies the serializing LP\_COUNT register as a destination.

All LR instructions are assumed to be potential readers of the status flags at their point of dispatch. This is because the actual auxiliary address is unknown at the dispatch point (the execute stage) and might be STATUS32. All SR instructions are assumed to be potential writers of the status flags for the same reason.

For this reason, an LR instruction following a DIV.F instruction (for example) is delayed in the execute stage until the DIV.F instruction has retired its flag result. Similarly, an SR instruction following a DIV.F instruction is delayed by the same amount, in order to ensure that its write to STATUS32 is not subsequently overwritten by the earlier-issued DIV.F instruction when it retires.

## 8.3 Exception Timing

Exceptions are normally associated with the instruction that raised the exception, and normally take place at the point in time when the instruction would otherwise commit. Exception processing requires the following steps:

1. Wait until all previously committed instructions have retired their outstanding results, and all in-progress cache misses and memory operations are completed.
2. Flush the pipeline and restart instruction fetching from the exception-vector location.

If no committed instructions that have not yet retired their results are outstanding, then step 1 can be skipped. The delay involved at step 1 depends on the latency of any outstanding committed multi-cycle operations, or the remaining memory-access or cache-refill time.

Flushing the pipeline in step 2 takes one additional cycle.

Jumping to the exception handler requires a single instruction-fetch cycle, as it involves fetching an aligned four-byte entry-point address for the exception vector (see [Table 8-3](#) for details). An implicit jump to the entry point of the exception handler is dispatched in the cycle after it is issued, and in the following cycle, the first instruction fetch from the exception handler begins.

## 8.4 Interrupt Response Times (Interrupt Latency)

Interrupt response time and its predictability are often a critical factor in the performance of real-time systems. The minimum interrupt response time for ARC EM processors is five cycles, and when the processor is suitably configured, this minimum response time can be achieved consistently. This response time is measured from the rising edge of a re-synchronized external interrupt to the completion of the first instruction after the interrupt.

The best practices to achieve the minimum response time are as follows:

- Ensure that memory references are serviced within the core, and do not require external bus requests. This means using instruction closely coupled memory (ICCM) and data closely coupled memory (DCCM) rather than instruction and data caches, or ensuring that cache accesses reference blocks that are locked in cache and are therefore guaranteed to hit.



- Avoid configurations with multi-cycle multiply and divide implementations. A multi-cycle multiply or divide operation that sets the status flags (ZNCV), such as `mpy.f`, must be completed before an interrupt can be taken. This introduces a small quantity of uncertainty into the response time, in cases where an interrupt coincides with a flag-setting multi-cycle multiply or divide.

In addition to the 5-cycle minimum response time of the processor to interrupts, most systems also insert two cycles of delay between the assertion of an external asynchronous interrupt line and the assertion of the corresponding processor interrupt line, in order to synchronize the interrupt signal to the processor clock. To further minimize the time required to execute the first instruction of the interrupt handler, it is advisable to align the start of the interrupt handler to a four-byte boundary.

The ARC EM offers a fast-interrupt (FIRQ) feature to optimize overall context switching for the highest priority interrupt level. The FIRQ option can be utilized with a second register bank to further reduce interrupt overhead. For further details, see [Chapter 7, “Interrupts”](#).

Also ensure that the processor is not operating with interrupts disabled. Interrupts are disabled immediately after an exception is taken, and interrupts of the same priority or lower are disabled when an interrupt is taken. It is therefore the responsibility of software to save any critical processor state and then re-enable interrupts in a timely way.



#### Note

The behavior of interrupt entry and exit is programmed through the `AUX_IRQ_CTRL` register. This design allows the kernel to define how much of the register context is saved automatically and avoids the need to use `push_s` & `pop_s` instructions to save and restore registers to the stack on interrupt entry and exit.

However, interrupts may be slowed down when using this feature. If the processor configuration includes fast interrupts and one of these fast interrupts occurs immediately after a standard IRQ, then the fast IRQ is held off until the auto-save sequence is completed by the hardware. In the case of software stacking of registers, as soon as the fast IRQ occurs it can be serviced (immediately after the current `push_s` or `pop_s` instruction retires).

## 8.5 Cache Miss Penalties

When an instruction or data cache miss occurs, the associated instruction fetch or load/store operation will perceive an additional cycle cost, which is referred to as the cache miss penalty. When a cache miss occurs, the processor waits until the cache miss has been serviced before continuing to execute instructions. In configurations with both Instruction and Data caches, it is possible for misses to occur simultaneously in both caches. In principle these are handled independently inside the core, and can therefore be serviced simultaneously. However, the external bus may arbitrate at some level between the instruction and data busses, and this may serialize overlapping instruction and data cache refills.

The magnitude of each cache miss penalty depends on the configuration of the cache and the speed of memory external to the core. In ARC EM the movement of data between caches and external memory is implemented using burst transfers on the BVCI, AHB, or AHB-lite bus configured with the core. Each 32-bit element of a burst transfer contains  $B/4$  individual 4-byte data transfer operations, where  $B$  is the configured cache block size in bytes.

Typically, a burst transfer begins with an address phase, during which the core emits a memory address and performs a read or write request. The ARC EM core will present a registered burst request on the instruction or data bus one cycle after the originating Instruction Cache or Data Cache miss takes place. Depending on contention on the bus, a response may be received by the core in the same cycle, or it may be delayed by a dynamically varying number of cycles. If the delay between presenting the request and the bus

acknowledging the first 4-byte transfer is A cycles, then the complete transfer will take T cycles, assuming each elemental transfer takes 1 cycle. Thus, T is given by:

$$T = A + (B/4)$$

If the bus protocol, and the external memory system, permit individual elements of the burst to be delayed, then any such external delays must be added to T. For example, when accessing a slow external memory, the acknowledgement for each element of the transfer may take perhaps 2 cycles each. So, in general, if the sustained rate of transfer is one 32-bit word every N cycles across the bus, then T will instead be given by:

$$T = A + (B/4*N)$$

In general, the initial latency of a burst transfer may depend on whether it is a read or write, so we may distinguish these burst delays using AR and AW, though these may be identical in some systems.

### 8.5.1 Instruction Cache Miss Penalties

The instructions in an instruction cache are never modified by the processor, and therefore when an instruction cache line is replaced it can be simply overwritten. The cache miss penalty for an instruction cache is therefore simply the time required to perform a burst read request of BI bytes, where BI is the block size of the instruction cache. The instruction cache miss penalty will therefore be TI

$$TI = AR + (BI/4*N)$$

## 8.6 Multi-cycle Floating Point Instructions

All multi-cycle floating point instructions stall a dependent instruction or another floating point instruction, but do not stall a non-dependent instruction in the any other pipeline.

# 9

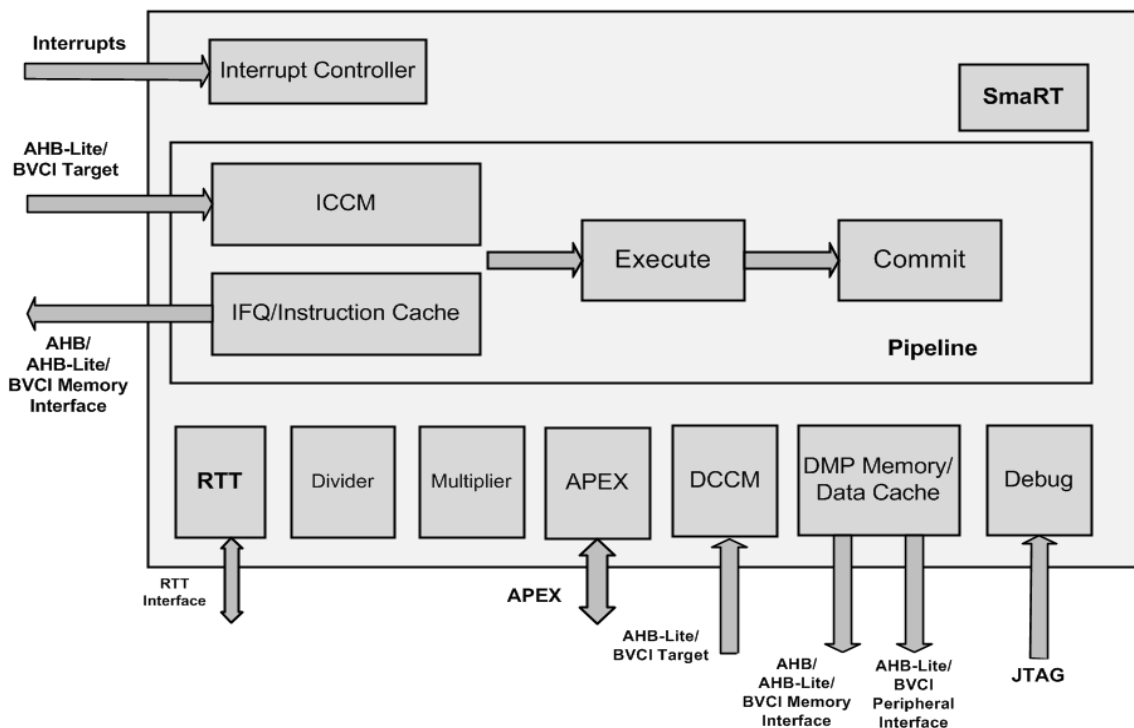
## External Interfaces

This chapter lists the interface signals for the ARC EM processor core.

### 9.1 Bus Interfaces

The ARC EM processor series uses standard protocols for its bus interfaces (see “Supported Standards” on page 47 for references). Figure 9-1 on page 147 provides an overview of the bus interfaces.

**Figure 9-1 Block Diagram Showing Bus Interfaces**



### 9.1.1 Memory Interface Types

The ARC EM processor can be configured to support one of the following protocols on its memory interface:

- AHB: The processor is configured with two AHB master interfaces. One interface handles instruction fetches and the other interface handles data (load/store) accesses.
- AHB-Lite single: The processor is configured with a single AHB-Lite master interface. This interface handles both instruction fetches and data (load/store) accesses.
- AHB-Lite dual: The processor is configured with two AHB-Lite master interfaces. One interface handles instruction fetches and the other interface handles data (load/store) accesses.
- BVCI: The processor is configured with two BVCI master interfaces. One interface handles instruction fetches and the other interface handles data (load/store) accesses.

**Note**

The instruction fetch master interface is only present in configurations with an Instruction cache or an instruction fetch queue. Likewise, the data access master interface is only present in configurations with a data cache or data memory initiator.

The memory interface is configured as either a registered or unregistered interface. When it is configured as a registered interface, the inputs and output signals are registered.

This option adds two cycles to the latency of the bus system: one cycle to send out the command information and one to register the incoming data. These flops are removed if the interface is configured as unregistered.

The unregistered interface is the better choice in systems where the memory interface inputs/outputs are not in the critical timing path.

For specific limitations on use of the unregistered BVCI memory interface, see Note in [Bus and External Memory](#) section, [Chapter 4, “Configurability”](#).

You can configure the memory-interface bus protocol and choose registered or unregistered in the ARChitect GUI or by specifying the `-mem_bus_option` and `-mem_bus_reg_interface` command-line options. For more details, see [Appendix B, “Key IP Library Command-Line Options”](#).

## 9.1.2 Instruction-Bus Interface Unit

This interface handles instruction-fetch requests from the instruction cache or the instruction fetch queue. Requests from this interface are always 32-bit-aligned. The size of the burst depends on the cache-line size or the depth of the instruction fetch queue. The bus protocol of this interface is configurable at build time.

The registered and unregistered options are available for all bus interfaces.



### Note

When configured with the AHB or AHB-Lite protocols, this interface never generates BUSY transfers.

All burst requests are always locked.

- BVC I Master ([Table 9-1](#))
- AHB Master ([Table 9-2](#))
- AHB-Lite Master ([Table 9-3](#))

**Table 9-1 Signals: Instruction Bus Interface Unit with BVC I**

| Signal      | Width          | Direction from/to Core | Notes                        |
|-------------|----------------|------------------------|------------------------------|
| ibu_cmdval  | 1              | Output                 |                              |
| ibu_eop     | 1              | Output                 |                              |
| ibu_address | 16, 20, 24, 32 | Output                 | Configurable using ARChitect |
| ibu_be      | 4              | Output                 |                              |
| ibu_cmd     | 2              | Output                 |                              |
| ibu_wdata   | 32             | Output                 |                              |
| ibu_plen    | 8              | Output                 |                              |
| ibu_rspack  | 1              | Output                 |                              |
| ibu_wrap    | 1              | Output                 |                              |
| ibu_contig  | 1              | Output                 |                              |
| ibu_cmdack  | 1              | Input                  |                              |
| ibu_rspval  | 1              | Input                  |                              |
| ibu_rdata   | 32             | Input                  |                              |
| ibu_reop    | 1              | Input                  |                              |
| ibu_rerr    | 1              | Input                  |                              |

**Table 9-2 Signals: Instruction-Bus Interface Unit with AHB**

| Signal          | Width          | Direction from/to Core | Notes                        |
|-----------------|----------------|------------------------|------------------------------|
| ibu_ahb_hbusreq | 1              | Output                 | Not AHB-Lite                 |
| ibu_ahb_hgrant  | 1              | Input                  | Not AHB-Lite                 |
| ibu_ahb_hlock   | 1              | Output                 |                              |
| ibu_ahb_hwrite  | 1              | Output                 |                              |
| ibu_ahb_haddr   | 16, 20, 24, 32 | Output                 | Configurable using ARChitect |
| ibu_ahb_htrans  | 2              | Output                 |                              |
| ibu_ahb_hsize   | 3              | Output                 |                              |
| ibu_ahb_hburst  | 3              | Output                 |                              |
| ibu_ahb_hprot   | 4              | Output                 |                              |
| ibu_ahb_hready  | 1              | Input                  |                              |
| ibu_ahb_hresp   | 2              | Input                  |                              |
| ibu_ahb_hrdata  | 32             | Input                  |                              |
| ibu_ahb_hwdata  | 32             | Output                 |                              |

**Table 9-3 Signals: Instruction-Bus Interface Unit with AHB-Lite**

| Signal         | Width          | Direction from/to Core | Notes                        |
|----------------|----------------|------------------------|------------------------------|
| ibu_ahb_hlock  | 1              | Output                 |                              |
| ibu_ahb_hwrite | 1              | Output                 |                              |
| ibu_ahb_haddr  | 16, 20, 24, 32 | Output                 | Configurable using ARChitect |
| ibu_ahb_htrans | 2              | Output                 |                              |
| ibu_ahb_hsize  | 3              | Output                 |                              |
| ibu_ahb_hburst | 3              | Output                 |                              |
| ibu_ahb_hprot  | 4              | Output                 |                              |
| ibu_ahb_hready | 1              | Input                  |                              |
| ibu_ahb_hresp  | 1              | Input                  |                              |
| ibu_ahb_hrdata | 32             | Input                  |                              |
| ibu_ahb_hwdata | 32             | Output                 |                              |

### 9.1.3 Data-Bus Interface Unit

This interface handles data memory references (loads and stores) from the data cache or data memory initiator components. When the processor core is configured with a data cache, this interface generates burst transfers with the size of a data-cache line. When the processor core is configured with a data memory initiator component, this interface always generates single (non-burst) transfers. The bus protocol of this interface is configurable at build time.

- BVC I Master ([Table 9-4](#))
- AHB Master ([Table 9-5](#))
- AHB-Lite Master ([Table 9-6](#))



#### Note

When configured with the AHB or AHB-Lite protocols, this interface never generates BUSY transfers.

All burst requests are always locked.

**Table 9-4 Signals: Data Bus Interface Unit with BVC I**

| Signal      | Width          | Direction from/to Core | Notes                        |
|-------------|----------------|------------------------|------------------------------|
| dbu_cmdval  | 1              | Output                 |                              |
| dbu_cmd_dbg | 1              | Output                 |                              |
| dbu_eop     | 1              | Output                 |                              |
| dbu_address | 16, 20, 24, 32 | Output                 | Configurable using ARChitect |
| dbu_be      | 4              | Output                 |                              |
| dbu_cmd     | 2              | Output                 |                              |
| dbu_wdata   | 32             | Output                 |                              |
| dbu_plen    | 8              | Output                 |                              |
| dbu_rspack  | 1              | Output                 |                              |
| dbu_wrap    | 1              | Output                 |                              |
| dbu_contig  | 1              | Output                 |                              |
| dbu_cmdack  | 1              | Input                  |                              |
| dbu_rspval  | 1              | Input                  |                              |
| dbu_rdata   | 32             | Input                  |                              |
| dbu_reop    | 1              | Input                  |                              |
| dbu_rerr    | 1              | Input                  |                              |

**Table 9-5 Signals: Data-Bus Interface Unit with AHB**

| Signal          | Width          | Direction from/to Core | Notes                        |
|-----------------|----------------|------------------------|------------------------------|
| dbu_ahb_hbusreq | 1              | Output                 |                              |
| dbu_ahb_hgrant  | 1              | Input                  |                              |
| dbu_ahb_hlock   | 1              | Output                 |                              |
| dbu_ahb_hwrite  | 1              | Output                 |                              |
| dbu_ahb_haddr   | 16, 20, 24, 32 | Output                 | Configurable using ARChitect |
| dbu_ahb_htrans  | 2              | Output                 |                              |
| dbu_ahb_hsize   | 3              | Output                 |                              |
| dbu_ahb_hburst  | 3              | Output                 |                              |
| dbu_ahb_hprot   | 4              | Output                 |                              |
| dbu_ahb_hready  | 1              | Input                  |                              |
| dbu_ahb_hresp   | 2              | Input                  |                              |
| dbu_ahb_hrdata  | 32             | Input                  |                              |
| dbu_ahb_hwdata  | 32             | Output                 |                              |

**Table 9-6 Signals: Data-Bus Interface Unit with AHB-Lite**

| Signal         | Width          | Direction from/to Core | Notes                        |
|----------------|----------------|------------------------|------------------------------|
| dbu_ahb_hlock  | 1              | Output                 |                              |
| dbu_ahb_hwrite | 1              | Output                 |                              |
| dbu_ahb_haddr  | 16, 20, 24, 32 | Output                 | Configurable using ARChitect |
| dbu_ahb_htrans | 2              | Output                 |                              |
| dbu_ahb_hsize  | 3              | Output                 |                              |
| dbu_ahb_hburst | 3              | Output                 |                              |
| dbu_ahb_hprot  | 4              | Output                 |                              |
| dbu_ahb_hready | 1              | Input                  |                              |
| dbu_ahb_hresp  | 1              | Input                  |                              |
| dbu_ahb_hrdata | 32             | Input                  |                              |
| dbu_ahb_hwdata | 32             | Output                 |                              |



### 9.1.4 Combined Instruction- and Data-Bus Interface Unit

The ARC EM processor core offers as an option a single AHB-Lite interface for both instructions and data. This interface can be used in low-end systems to help reduce the overall silicon area. The bus protocol of this interface is configurable at build time.

[Table 9-7](#) lists the interface signals.

**Table 9-7 Signals: Combined Instruction- and Data-Bus Interface Unit with AHB-Lite**

| Signal     | Width          | Direction from/to Core | Notes                        |
|------------|----------------|------------------------|------------------------------|
| ahb_hlock  | 1              | Output                 |                              |
| ahb_hwrite | 1              | Output                 |                              |
| ahb_haddr  | 16, 20, 24, 32 | Output                 | Configurable using ARChitect |
| ahb_htrans | 2              | Output                 |                              |
| ahb_hsize  | 3              | Output                 |                              |
| ahb_hburst | 3              | Output                 |                              |
| ahb_hprot  | 4              | Output                 |                              |
| ahb_hready | 1              | Input                  |                              |
| ahb_hresp  | 1              | Input                  |                              |
| ahb_hrdata | 32             | Input                  |                              |
| ahb_hwdata | 32             | Output                 |                              |

### 9.1.5 Peripheral-Bus Interface Unit

This optional interface allows simple memory-mapped peripherals to be connected to the processor core. This interface can coexist with the data bus interface unit. This interface handles requests from the memory region programmed as the peripheral region (see [Chapter 5, “Memory Components”](#)). The bus protocol of this interface is configurable at build time.

- BVCi Master ([Table 9-8](#))
- AHB-Lite Master ([Table 9-9](#))

**Table 9-8 Signals: Peripheral-Bus Interface Unit with BVCi**

| Signal               | Width | Direction from/to Core | Notes |
|----------------------|-------|------------------------|-------|
| dbu_per_bvci_cmdval  | 1     | Output                 |       |
| dbu_per_bvci_cmd_dbg | 1     | Output                 |       |
| dbu_per_bvci_eop     | 1     | Output                 |       |

**Table 9-8 Signals: Peripheral-Bus Interface Unit with BVCI (Continued)**

| Signal               | Width       | Direction from/to Core | Notes                        |
|----------------------|-------------|------------------------|------------------------------|
| dbu_per_bvci_address | 16,20,24,32 | Output                 | Configurable using ARCHitect |
| dbu_per_bvci_be      | 4           | Output                 |                              |
| dbu_per_bvci_cmd     | 2           | Output                 |                              |
| dbu_per_bvci_wdata   | 32          | Output                 |                              |
| dbu_per_bvci_plen    | 8           | Output                 |                              |
| dbu_per_bvci_rspack  | 1           | Output                 |                              |
| dbu_per_bvci_wrap    | 1           | Output                 |                              |
| dbu_per_bvci_contig  | 1           | Output                 |                              |
| dbu_per_bvci_cmdack  | 1           | Input                  |                              |
| dbu_per_bvci_rspval  | 1           | Input                  |                              |
| dbu_per_bvci_rdata   | 32          | Input                  |                              |
| dbu_per_bvci_reop    | 1           | Input                  |                              |
| dbu_per_bvci_rerr    | 1           | Input                  |                              |

**Table 9-9 Signals: Peripheral-Bus Interface Unit with AHB-Lite**

| Signal             | Width          | Direction from/to Core | Notes                        |
|--------------------|----------------|------------------------|------------------------------|
| dbu_per_ahb_hlock  | 1              | Output                 |                              |
| dbu_per_ahb_hwrite | 1              | Output                 |                              |
| dbu_per_ahb_haddr  | 16, 20, 24, 32 | Output                 | Configurable using ARCHitect |
| dbu_per_ahb_htrans | 2              | Output                 |                              |
| dbu_per_ahb_hsize  | 3              | Output                 |                              |
| dbu_per_ahb_hburst | 3              | Output                 |                              |
| dbu_per_ahb_hprot  | 4              | Output                 |                              |
| dbu_per_ahb_hready | 1              | Input                  |                              |
| dbu_per_ahb_hresp  | 1              | Input                  |                              |
| dbu_per_ahb_hrdata | 32             | Input                  |                              |
| dbu_per_ahb_hwdata | 32             | Output                 |                              |

## 9.1.6 Bus Command Attributes

The AXI and AHB-Lite initiator interfaces include the standard bus command attributes for cacheable and protection modes. Table 9-10 and Table 9-11 define the attribute values for the different transactions issued by the ARC EM processor core in the kernel mode and in the non-kernel mode.

**Table 9-10 AXI and AHB-lite Bus Command Attributes - Kernel Mode**

|              | Description | Kernel Mode (or Vectors or Prologue/Epilogue) |                       |            |                   |             |             |
|--------------|-------------|-----------------------------------------------|-----------------------|------------|-------------------|-------------|-------------|
|              |             | Icache Instruction Fetch                      | IFQ Instruction Fetch | D\$ Refill | Dcache Write-back | Uncached LD | Uncached ST |
| ahb_hprot[3] | Cacheable   | 0                                             | 0                     | 0          | 0                 | 0           | 0           |
| ahb_hprot[2] | Bufferable  | 0                                             | 0                     | 0          | 0                 | 0           | 0           |
| ahb_hprot[1] | Privileged  | 1                                             | 1                     | 1          | 1                 | 1           | 1           |
| ahb_hprot[0] | Data        | 0                                             | 0                     | 1          | 1                 | 1           | 1           |

**Table 9-11 AXI and AHB-Lite Bus Command Attributes - Non-Kernel Mode**

|              | Description | Non-Kernel Mode          |                              |            |                   |             |             |
|--------------|-------------|--------------------------|------------------------------|------------|-------------------|-------------|-------------|
|              |             | Icache Instruction Fetch | non Icache Instruction Fetch | D\$ Refill | Dcache Write-back | Uncached LD | Uncached ST |
| ahb_hprot[3] | Cacheable   | 0                        | 0                            | 0          | 0                 | 0           | 0           |
| ahb_hprot[2] | Bufferable  | 0                        | 0                            | 0          | 0                 | 0           | 0           |
| ahb_hprot[1] | Privileged  | 0                        | 0                            | 0          | 0                 | 0           | 0           |
| ahb_hprot[0] | Data        | 0                        | 0                            | 1          | 1                 | 1           | 1           |

## 9.1.7 Direct Memory Interface Unit

When configured with closely coupled memories, the ARC EM processor core allows other initiators in the system to access the CCMs through a standard target interface. This interface may be used to preload the CCMs while the core is fully functional and running. External accesses during execution are handled with the highest priority. One dedicated interface exists for each closely coupled memory. The bus protocol of the target is configurable at build time and must be the same as the external memory-initiator interface.

The clock ratio between the Direct Memory Interface (external AHB master) and the core clock must be 1:1.



### Note

Both single and incremental burst requests are supported by the target interface.

- ICCM Direct Memory Interface (DMI):

- BVCi Slave ([Table 9-12](#))
- AHB-Lite Slave ([Table 9-13](#))
- DCCM DMI:
  - BVCi Slave ([Table 9-14](#))
  - AHB-Lite Slave ([Table 9-15](#))

**Table 9-12 Signals: ICCM DMI Unit with BVCi**

| Signal            | Width          | Direction from/to Core | Notes                        |
|-------------------|----------------|------------------------|------------------------------|
| iccm_bvci_cmdval  | 1              | Input                  |                              |
| iccm_bvci_cmdack  | 1              | Output                 |                              |
| iccm_bvci_address | 16, 20, 24, 32 | Input                  | Configurable using ARChitect |
| iccm_bvci_be      | 4              | Input                  |                              |
| iccm_bvci_cmd     | 4              | Input                  |                              |
| iccm_bvci_wdata   | 32             | Input                  |                              |
| iccm_bvci_eop     | 1              | Input                  |                              |
| iccm_bvci_rspval  | 1              | Output                 |                              |
| iccm_bvci_rdata   | 32             | Output                 |                              |
| iccm_bvci_reop    | 1              | Output                 |                              |
| iccm_bvci_rerr    | 1              | Output                 |                              |
| iccm_bvci_rspack  | 1              | Input                  |                              |

**Table 9-13 Signals: ICCM DMI Unit with AHB-Lite**

| Signal          | Width          | Direction from/to Core | Notes                        |
|-----------------|----------------|------------------------|------------------------------|
| iccm_ahb_hwrite | 1              | Input                  |                              |
| iccm_ahb_haddr  | 16, 20, 24, 32 | Input                  | Configurable using ARChitect |
| iccm_ahb_htrans | 2              | Input                  |                              |
| iccm_ahb_hsize  | 3              | Input                  |                              |
| iccm_ahb_hburst | 3              | Input                  |                              |
| iccm_ahb_hready | 1              | Input                  |                              |
| iccm_ahb_hresp  | 1              | Output                 |                              |

**Table 9-13 Signals: ICCM DMI Unit with AHB-Lite (Continued)**

| Signal               | Width | Direction from/to Core | Notes |
|----------------------|-------|------------------------|-------|
| iccm_ahb_hrdata      | 32    | Output                 |       |
| iccm_ahb_hwdata      | 32    | Input                  |       |
| iccm_ahb_hsel        | 1     | Input                  |       |
| iccm_ahb_hready_resp | 1     | Output                 |       |

**Table 9-14 Signals: DCCM DMI Unit with BVCI**

| Signal            | Width          | Direction from/to Core | Notes                        |
|-------------------|----------------|------------------------|------------------------------|
| dccm_bvci_cmdval  | 1              | Input                  |                              |
| dccm_bvci_cmdack  | 1              | Output                 |                              |
| dccm_bvci_address | 16, 20, 24, 32 | Input                  | Configurable using ARChitect |
| dccm_bvci_be      | 4              | Input                  |                              |
| dccm_bvci_cmd     | 2              | Input                  |                              |
| dccm_bvci_wdata   | 32             | Input                  |                              |
| dccm_bvci_eop     | 1              | Input                  |                              |
| dccm_bvci_rspack  | 1              | Input                  |                              |
| dccm_bvci_rspval  | 1              | Output                 |                              |
| dccm_bvci_rdata   | 32             | Output                 |                              |
| dccm_bvci_reop    | 1              | Output                 |                              |
| dccm_bvci_rerr    | 1              | Output                 |                              |

**Table 9-15 Signals: DCCM DMI Unit with AHB-Lite**

| Signal          | Width       | Direction from/to Core | Notes                        |
|-----------------|-------------|------------------------|------------------------------|
| dccm_ahb_hwrite | 1           | Input                  |                              |
| dccm_ahb_addr   | 16,20,24,32 | Input                  | Configurable using ARChitect |
| dccm_ahb_htrans | 2           | Input                  |                              |
| dccm_ahb_hsize  | 3           | Input                  |                              |
| dccm_ahb_hburst | 3           | Input                  |                              |

**Table 9-15 Signals: DCCM DMI Unit with AHB-Lite (Continued)**

| Signal               | Width | Direction from/to Core | Notes |
|----------------------|-------|------------------------|-------|
| dccm_ahb_hready      | 1     | Input                  |       |
| dccm_ahb_hresp       | 1     | Output                 |       |
| dccm_ahb_hrdata      | 32    | Output                 |       |
| dccm_ahb_hwdata      | 32    | Output                 |       |
| dccm_ahb_hsel        | 1     | Input                  |       |
| dccm_ahb_hready_resp | 1     | Output                 |       |

## 9.2 Interrupts Interface

The ARC EM processor core can be configured to have up to 240 external interrupts. The pin names of external interrupts reflects the vector number for each interrupt and ranges from 16 to 255. When internal timers are configured, Interrupts 16 and 17 are reserved for the internal timers. In this case, the corresponding interrupt signal does not appear as an external interrupt input.

The ARC EM processor core accepts up to 240 external interrupts. Mapping of these external interrupt pins to devices requesting interrupts is not pre-defined and is very flexible. Interrupt pins 16 and 17 maybe reserved for the internal timers, when the timers are present.

**Table 9-16 Signals: Interrupts**

| Signal    | Width | Direction from/to Core | Description                                                                               |
|-----------|-------|------------------------|-------------------------------------------------------------------------------------------|
| irq_16_a  | 1     | Input                  | irq_16_a is an external interrupt if timer 0 is not present.                              |
| irq_17_a  | 1     | Input                  | irq_17_a is an external interrupt if timer 1 is not present.                              |
| irq_<N>_a | 1     | Input                  | Range <N> is 18 to 255, depending on the number of Interrupts configured using ARChitect. |

## 9.3 Test and Debug Interface

This interface complies with the IEEE 1149.1-2001 JTAG specifications. This interface can also be used with the 2-wire JTAG configuration of IEEE 1149.7 System Test Logic (STL) specifications. This interface can be converted into full 2-wire JTAG by using an IEEE 1149.7 tap controller. The BVCI interface is also available for debugging.

- JTAG Slave ([Table 9-17](#))
- BVCI Slave ([Table 9-18](#))

**Table 9-17 Signals: JTAG Interface**

| Signal      | Width | Direction from/to Core |
|-------------|-------|------------------------|
| test_mode   | 1     | Input                  |
| jtag_trst_n | 1     | Input                  |
| jtag_tck    | 1     | Input                  |
| jtag_tms    | 1     | Input                  |
| jtag_tdi    | 1     | Input                  |
| jtag_tdo    | 1     | Output                 |

**Table 9-18 Signals: JTAG Interface with BVC1**

| Signal      | Width | Direction from/to Core |
|-------------|-------|------------------------|
| dbg_cmdval  | 1     | Input                  |
| dbg_cmdack  | 1     | Output                 |
| dbg_address | 32    | Input                  |
| dbg_be      | 4     | Input                  |
| dbg_cmd     | 2     | Input                  |
| dbg_wdata   | 32    | Input                  |
| dbg_eop     | 1     | Input                  |
| dbg_rspval  | 1     | Output                 |
| dbg_rdata   | 32    | Output                 |
| dbg_reop    | 1     | Output                 |
| dbg_rerr    | 1     | Output                 |
| dbg_rspack  | 1     | Input                  |

## 9.4 Halt and Run Controls

This interface allows external logic to control the running/halted state of the processor through two pairs of request-and-acknowledgment handshake protocols. One pair controls halting while the other controls running after a halt. For an explanation of how these signals work to control the halt the processor and set it running, see [“External Halt and Run Control Signals”](#) on page 168.

Mapping this pair of signals on General Purposes Input Output (GPIO) registers can be used as a means for multiprocessor debugging.

Table 9-19    Signals: Halt and Control Interface

| Signal         | Width | Direction from/to Core | Description                      |
|----------------|-------|------------------------|----------------------------------|
| arc_halt_req_a | 1     | Input                  | Asynchronous halt request        |
| arc_halt_ack   | 1     | Output                 | Core acknowledge to halt request |
| arc_run_req_a  | 1     | Input                  | Asynchronous run request         |
| arc_run_ack    | 1     | Output                 | Core acknowledge to run request  |

Figure 9-2 shows the timing for the halt controls.

Figure 9-2    Timing Diagram for the Halt Controls

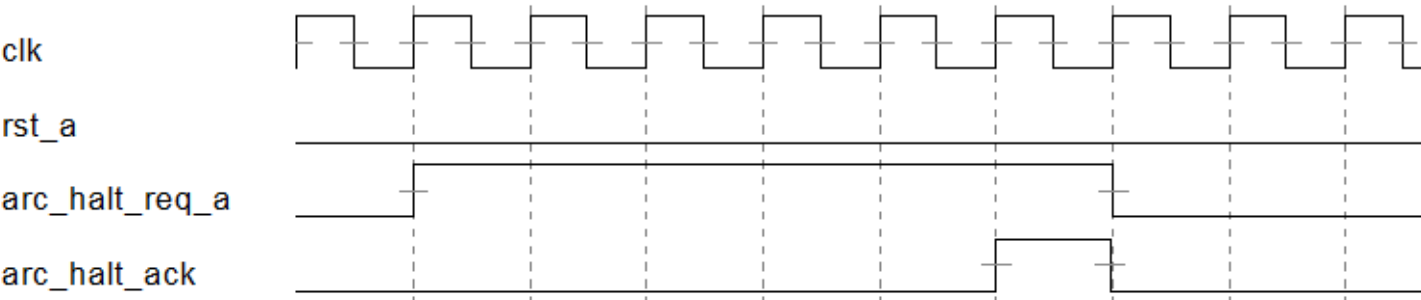
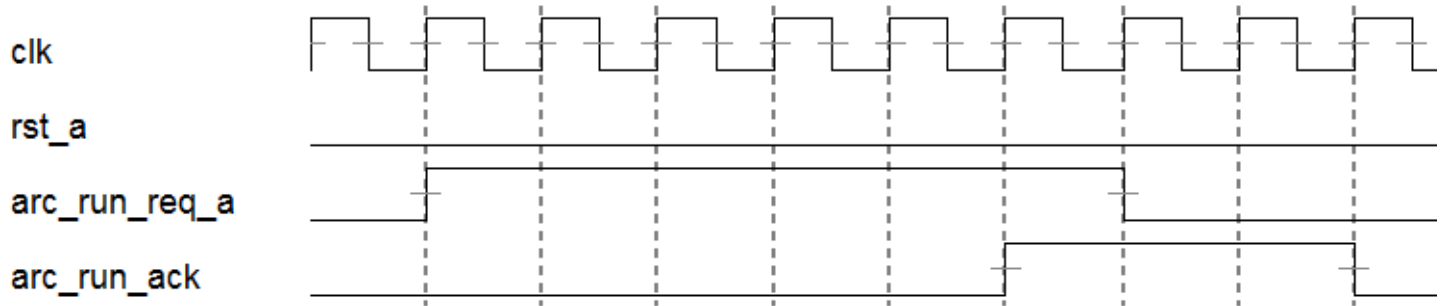




Figure 9-3 shows the timing for the run controls.

**Figure 9-3 Timing Diagram for the Run Controls**



## 9.5 Actionpoint Signals

When fully featured actionpoints are configured, an additional set of inputs is present at the external interface. There are two 32-bit data inputs, `ap_param0` and `ap_param1`, each of which has a read and write access indicator (`ap_param0_read`, `ap_param0_write`, `ap_param1_read` and `ap_param1_write`). All actionpoint external inputs are assumed to be synchronous with respect to the core clock.

The read and write access indicators allow an external agent to inform the actionpoints matching logic that an external read or write event occurred in association with the respective parameter value, typically an address or data value. An actionpoint can be programmed to trigger on either a read or a write event, and external actionpoints are triggered on the assertion of the corresponding read or write access indicator, provided the parameter matches the value programmed into the actionpoints logic. In practice, the read and write access indicators can be used to indicate any desired event, as defined by external logic.

**Table 9-20 Signals: Actionpoints**

| Signal                       | Width | Direction from/to Core | Description                 |
|------------------------------|-------|------------------------|-----------------------------|
| <code>ap_param0</code>       | 32    | Input                  | External value actionpoints |
| <code>ap_param0_read</code>  | 1     | Input                  | Enable read actionpoints    |
| <code>ap_param0_write</code> | 1     | Input                  | Enable write actionpoints   |
| <code>ap_param1</code>       | 32    | Input                  | External value actionpoints |
| <code>ap_param1_read</code>  | 1     | Input                  | Enable read actionpoints    |
| <code>ap_param1_write</code> | 1     | Input                  | Enable write actionpoints   |

## 9.6 Real Time Trace Interface

The ARC EM processor core can be configured with a Real Time Trace (RTT) producer component to interface to an external RTT module. The RTT subsystem monitors the processor execution, read transactions and write transactions to memory and the Auxiliary registers, as well as writes to core registers. Separate busses are provided to report such transactions, which can happen simultaneously.

The subsystem consists of a RTT processor interface within the ARC EM processor (`alb_rtt_interface`), and an external ARC RTT hardware IP block residing in the ARC RTT IP Library. The RTT module queues and prepares the trace information for transmission using a Nexus interface to a debug host.

### 9.6.1 Real-Time Trace Interface Signals

**Table 9-21 RTT Programming Interface**

| Signal                   | Width | Direction from/to Core | Description         |
|--------------------------|-------|------------------------|---------------------|
| <code>rtt_addr_r</code>  | 10    | Output                 | RTT program address |
| <code>rtt_drd_r</code>   | 32    | Input                  | RTT read data       |
| <code>rtt_dwr_r</code>   | 32    | Output                 | RTT write data      |
| <code>rtt_read_a</code>  | 1     | Output                 | RTT read pulse      |
| <code>rtt_write_a</code> | 1     | Output                 | RTT write pulse     |

**Table 9-22 RTT Program Execution Tracking**

| Signal                        | Width | Direction from/to Core | Description                        |
|-------------------------------|-------|------------------------|------------------------------------|
| <code>rtt_inst_commit</code>  | 1     | Output                 | Instruction has committed          |
| <code>rtt_inst_addr</code>    | 32    | Output                 | Instruction address (pc)           |
| <code>rtt_cond_valid</code>   | 1     | Output                 | Commit instruction was conditional |
| <code>rtt_cond_pass</code>    | 1     | Output                 | Condition code test passed         |
| <code>rtt_branch_taken</code> | 1     | Output                 | Instruction was a branch           |
| <code>rtt_branch_indir</code> | 1     | Output                 | Branch was indirect                |
| <code>rtt_branch_taddr</code> | 32    | Output                 | Target address of branch/execute   |
| <code>rtt_dslot</code>        | 1     | Output                 | Branch has delay slot              |
| <code>rtt_exception</code>    | 1     | Output                 | Exception has been taken           |
| <code>rtt_interrupt</code>    | 1     | Output                 | Interrupt has been taken           |
| <code>rtt_enter_sleep</code>  | 1     | Output                 | Core has entered sleep mode        |
| <code>rtt_zd_loop</code>      | 1     | Output                 | Zero-delay loop has been taken     |
| <code>rtt_debug</code>        | 1     | Output                 | Core has entered debug mode        |
| <code>rtt_process_id</code>   | 8     | Output                 | Current value of PID register      |
| <code>rtt_pid_wr_en</code>    | 1     | Output                 | Write enable for PID register      |

**Table 9-22 RTT Program Execution Tracking (Continued)**

| Signal          | Width | Direction from/to Core | Description                      |
|-----------------|-------|------------------------|----------------------------------|
| rtt_ss_halt     | 1     | Output                 | Single-step event                |
| rtt_sw_bp       | 1     | Output                 | Software breakpoint hit          |
| rtt_hw_bp       | 1     | Output                 | Hardware breakpoint hit          |
| rtt_hw_exc      | 1     | Output                 | Hardware error (memory error)    |
| rtt_sleep_mode  | 1     | Output                 | Enter sleep state event          |
| rtt_dbg_halt    | 1     | Output                 | In halt state                    |
| rtt_rst_applied | 1     | Output                 | Processor reset has been applied |

**Table 9-23 Actionpoint Status**

| Signal     | Width | Direction from/to Core | Description                      |
|------------|-------|------------------------|----------------------------------|
| rtt_wp_hit | 1     | Output                 | Actionpoint has been hit         |
| rtt_wp_val | 8     | Output                 | Actionpoints triggered (pending) |
| rtt_sw_bp  | 1     | Output                 | Software breakpoint hit          |

**Table 9-24 Monitor Memory Load Traffic**

| Signal          | Width    | Direction from/to Core | Description                      |
|-----------------|----------|------------------------|----------------------------------|
| rtt_mem_ld      | 1        | Output                 | Load instruction committed       |
| rtt_mem_ld_addr | 32       | Output                 | Memory address                   |
| rtt_mem_ld_data | 32 or 64 | Output                 | 32-bit or 64-bit Write/Read data |
| rtt_mem_ld_sz   | 2        | Output                 | Size of transfer                 |
| rtt_mem_ld_dreg | 6        | Output                 | Destination register for load    |

**Table 9-25 Monitor Memory Store Traffic**

| Signal          | Width    | Direction from/to Core | Description                        |
|-----------------|----------|------------------------|------------------------------------|
| rtt_mem_st      | 1        | Output                 | Store instruction committed        |
| rtt_mem_st_addr | 32       | Output                 | Memory address                     |
| rtt_mem_st_data | 32 or 64 | Output                 | 32-bit or 64-bit memory write data |

**Table 9-25 Monitor Memory Store Traffic (Continued)**

| Signal        | Width | Direction from/to Core | Description      |
|---------------|-------|------------------------|------------------|
| rtt_mem_st_sz | 2     | Output                 | Size of transfer |

**Table 9-26 AUX Register Read/Write Monitoring**

| Signal       | Width | Direction from/to Core | Description                  |
|--------------|-------|------------------------|------------------------------|
| rtt_aux_addr | 32    | Output                 | Auxiliary register address   |
| rtt_aux_lr   | 1     | Output                 | Committed instruction was LR |
| rtt_aux_sr   | 1     | Output                 | Committed instruction was SR |
| rtt_aux_dwr  | 32    | Output                 | Write data of SR             |
| rtt_aux_drd  | 32    | Output                 | Read data of LR              |

**Table 9-27 Core Register Write Port 0 Monitoring**

| Signal           | Width | Direction from/to Core | Description                           |
|------------------|-------|------------------------|---------------------------------------|
| rtt_cr_addr0     | 6     | Output                 | Core register address                 |
| rtt_cr_wr_en_lo0 | 1     | Output                 | Core register write enable: dwr[31:0] |
| rtt_cr_dwr_lo0   | 32    | Output                 | Core register write data lo           |

**Table 9-28 Core Register Write Port 0 Monitoring**

| Signal           | Width | Direction from/to Core | Description                           |
|------------------|-------|------------------------|---------------------------------------|
| rtt_cr_addr1     | 6     | Output                 | Core register address                 |
| rtt_cr_wr_en_lo1 | 1     | Output                 | Core register write enable: dwr[31:0] |
| rtt_cr_dwr_lo1   | 32    | Output                 | Core register write data lo           |

**Table 9-29 Freeze/Restart Interface**

| Signal     | Width | Direction from/to Core | Description               |
|------------|-------|------------------------|---------------------------|
| rtt_freeze | 1     | Input                  | Request processor to stop |

## 9.7 Clock and Reset

The ARC EM core has a single functional clock, `clk`. The processor reset signal `rst_a` is asynchronous and active high.

The ARC EM processor is designed to accept an asynchronously asserted `rst_a` signal. However, the `rst_a` signal should be rescinded synchronously with respect to `clk`, to meet setup and hold time requirements between the falling edge of `rst_a` and the rising edge of `clk`. The synchronization of the reset signal must take place outside the processor core. Reset should be applied for a minimum of four processor clock cycles. See also [Chapter 10, “Clock, Reset, and Run Controls”](#).

**Table 9-30 Signals: Clock and Reset**

| Signal                      | Width | Direction from/to Core | Notes                         |
|-----------------------------|-------|------------------------|-------------------------------|
| <code>clk</code>            | 1     | Input                  | Core clock                    |
| <code>rst_a</code>          | 1     | Input                  | Asynchronous reset            |
| <code>debug_reset</code>    | 1     | Output                 | Reset request from debug host |
| <code>watchdog_reset</code> | 1     | Output                 | Watchdog reset request        |

For an explanation of how the `debug_reset` and `watchdog_reset` signals can be used to reset the processor, see [“Debug Reset and Watchdog Reset”](#) on page 169.

## 9.8 Code Protect

For brief description on Code Protection, see section [Code Protection](#), in [Chapter 6, “Memory-Protection Unit”](#).

[Table 9-31](#) lists the width and directions of Code Protect signals.

**Table 9-31 Code Protect Signals**

| Signal                   | Width | Direction from/to Core | Notes                               |
|--------------------------|-------|------------------------|-------------------------------------|
| <code>protect [0]</code> | 1     | Input                  | Code protection switch for Region 0 |
| <code>protect [1]</code> | 1     | Input                  | Code protection switch for Region 1 |
| <code>protect [2]</code> | 1     | Input                  | Code protection switch for Region 2 |
| <code>protect [3]</code> | 1     | Input                  | Code protection switch for Region 3 |
| <code>protect [4]</code> | 1     | Input                  | Code protection switch for Region 4 |
| <code>protect [5]</code> | 1     | Input                  | Code protection switch for Region 5 |
| <code>protect [6]</code> | 1     | Input                  | Code protection switch for Region 6 |
| <code>protect [7]</code> | 1     | Input                  | Code protection switch for Region 7 |
| <code>protect [8]</code> | 1     | Input                  | Code protection switch for Region 8 |

**Table 9-31 Code Protect Signals**

| Signal       | Width | Direction from/to Core | Notes                                |
|--------------|-------|------------------------|--------------------------------------|
| protect [9]  | 1     | Input                  | Code protection switch for Region 9  |
| protect [10] | 1     | Input                  | Code protection switch for Region 10 |
| protect [11] | 1     | Input                  | Code protection switch for Region 11 |
| protect [12] | 1     | Input                  | Code protection switch for Region 12 |
| protect [13] | 1     | Input                  | Code protection switch for Region 13 |
| protect [14] | 1     | Input                  | Code protection switch for Region 14 |
| protect [15] | 1     | Input                  | Code protection switch for Region 15 |

# 10

## Clock, Reset, and Run Controls

This chapter describes the interfaces and signal requirements for the provision of clock and reset signals, and for external control over when the processor should be running or in a halted state. For tables of signals on each external interface, see [Chapter 9, “External Interfaces”](#).

### 10.1 Signal Interface

The processor interface for clock and reset consists of the first two input signals listed in [Table 10-1](#).

**Table 10-1 Clock, Reset, and Run Control Signal List**

| Signal                | Direction | Description                                          |
|-----------------------|-----------|------------------------------------------------------|
| clk                   | Input     | Processor clock                                      |
| rst_a                 | Input     | Processor reset                                      |
| sys_halt_r            | Output    | Indicates that processor is halted                   |
| sys_sleep_r           | Output    | Indicates that processor is sleeping                 |
| arc_halt_req_a        | Input     | Request to halt the processor                        |
| arc_halt_ack          | Output    | Acknowledgement that processor is halted             |
| arc_run_req_a         | Input     | Request to start processor running from halted state |
| arc_run_ack           | Output    | Acknowledgement that processor is running            |
| sys_sleep_mode_r[2:0] | Output    | Indicates the sleep mode entered by the processor    |

### 10.2 Clock-Signal Requirements

All ARC EM processors are implemented as fully static designs, in which all synchronous elements in the processor are clocked on the positive edge of the `clk` signal.

The clock net is not explicitly buffered, as the clock tree is expected to be synthesized during place-and-route.

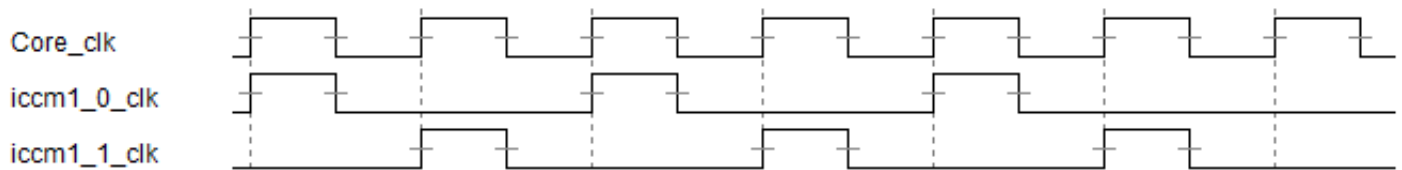
The achievable clock frequency depends on the chosen processor configuration, the sizes and configuration of caches and CCMs, and the silicon implementation technology. For further details on the sleep, halt, and run states, refer to [Chapter 17, “Power-Saving Features”](#).

### 10.3 ICCM1 Clock Details

ICCM1 clocks are generated by the ARC EM core and use clock gating. ICCM1 is clocked at half the frequency of the core clock. This is achieved by enabling the core clock on alternate cycles using a clock gate.

When ICCM1 banking is configured, two separate clocks of opposite phase are used to clock the two banks of ICCM1.

**Figure 10-1 ICCM1 Clocks with ICCM1 Banking**



### 10.4 Reset Signal Requirements

The processor reset signal `rst_a` is asynchronous, and active high. The ARC EM processor is designed to accept an asynchronously asserted `rst_a` signal. The `rst_a` input is re-timed within the internal module, `reset_ctrl`, such that the processor reset is de-asserted synchronously with respect to `clk`, in order to meet setup and hold time requirements between the falling edge of `rst_a` and the rising edge of `clk`. This extends the internal core reset by up to two clock cycles so external bus activity such as debug or DMI requests must not be applied to the CPU until at least two clocks after the external `rst_a` input is de-asserted. Reset should be applied for a minimum of four processor clock cycles.

The `rst_a` signal is not explicitly buffered, as synthesis tools are expected to automatically buffer the `rst_a` net.

### 10.5 External Halt and Run Control Signals

External logic can control the running or halted state of the processor through two pairs of request and acknowledgment signals; one controls halting, and the other controls restarts after a halt.

- The `arc_halt_req_a` signal is asserted by external logic that wishes to halt the ARC EM processor. This signal may be asserted asynchronously with respect to the processor clock. When the processor is ready to halt, it enters the halted state in response to the assertion of `arc_halt_req_a`, and asserts the `arc_halt_ack` acknowledgement signal. After the acknowledgment is received, the external logic may rescind the `arc_halt_req_a` signal, and the processor remains halted.
- The `arc_run_req_a` signal is asserted by external logic that wishes to bring the ARC EM processor out of the halted state to resume execution. This signal may be asserted asynchronously with respect to the processor clock. In response to the assertion of `arc_run_req_a`, the processor resumes execution and asserts the `arc_run_ack` signal. After the acknowledgment is granted, the external logic may rescind the `arc_run_req_a` signal.



The acknowledgement signals `arc_halt_ack` and `arc_run_ack` are synchronous with respect to the processor clock. External logic might need to synchronize them to its own internal clock, if it is operating in its own clock domain.

- The delay from assertion of `arc_halt_req_a` to the assertion of `arc_halt_ack` depends on the state of the processor at the time of the request. The processor completes any outstanding operations to which it is already committed before it halts. These operations may include the completion of external memory transactions or multi-cycle instructions that are committed but have not yet retired their result. When an external device asserts `arc_halt_req_a`, ARC EM asserts `arc_halt_ack`. When an external device deasserts `arc_halt_req_a`, ARC EM deasserts `arc_halt_ack`.
- The delay from assertion of `arc_run_req_a` to the assertion of `arc_run_ack` is four processor clock cycles. If the clock source is gated or disabled (by the external logic), then it must be restored before `arc_run_req_a` is asserted. When an external device asserts `arc_run_req_a`, ARC EM asserts `arc_run_ack`. When an external device deasserts `arc_run_req_a`, ARC EM deasserts `arc_run_ack`.

## 10.6 Debug Reset and Watchdog Reset

The `debug_reset` signal is present in the port list of the core whenever a debug interface is configured. It is an active-high output that is asserted in response to a debug host writing 1 to either bit 1 or bit 0 of the `DB_RESET` register in the debug interface.

This signal remains asserted while bit 1 or bit 0 of `DB_RESET` is asserted. The `DB_RESET` register is cleared when the core is reset.

The `watchdog_reset` signal is present in the `cpu_top` port list when at least one timer is configured. It is an active-high output that is asserted in the clock cycle after one of the timer `COUNT` registers reaches its corresponding `LIMIT` value, provided the `W` bit of the associated timer `CONTROL` register is set to 1. Either of the two timers can trigger this condition. After the `watchdog_reset` signal is asserted, it remains asserted until the core is reset.

The `debug_reset` and `watchdog_reset` signals are typically interpreted as reset requests from the core. External logic is responsible for asserting `rst_a` in response to the assertion of either of these signals, if debug or watchdog reset functionality is required.

The enclosing system may choose to ignore these two output signals if debug or watchdog reset functionality is not required.

## 10.7 Reset Configuration

ARC EM processors can be configured to be in either the halted or running state upon removal of the reset signal. To configure, use the `-halt_on_reset` option in `ARCHitect`.

- When `-halt_on_reset` is set to true, the processor is in the halted state immediately after reset. Bringing the processor out of the halted state requires either an external debugger or host, or external logic using the `arc_run_req_a` signal described in [“External Halt and Run Control Signals”](#).
- When `-halt_on_reset` is set to false, the processor begins executing code from the reset-vector base address immediately after coming out of reset. System designers must ensure that valid code is placed in memory at that address prior to rescinding `rst_a`. Valid code can be provided by placing external ROM at the reset-vector address, or by pre-loading ICCM at the reset vector location via the external ICCM interface.

## 10.8 Run Status Signals

Two output signals give a continuous indication of whether the processor is running, sleeping, or halted. These signals are `sys_halt_r` and `sys_sleep_r`, and their interpretation is given in [Tables 10-2](#).

**Table 10-2** Interpretation of `sys_halt_r` and `sys_sleep_r` output signals

| <code>sys_halt_r</code> | <code>sys_sleep_r</code> | Processor state |
|-------------------------|--------------------------|-----------------|
| 0                       | 0                        | Running         |
| 1                       | -                        | Halted          |
| 0                       | 1                        | Sleeping        |

See also [Chapter 17, “Power-Saving Features”](#).

# 11

## ARC EM DSP Features

This chapter describes the Digital Signal Processing (DSP) features of the ARC EM processor series, which are available with the ARC EM IP library.

The `DSP assist` option can be used to minimize the processor loading when executing entry-level DSP applications, such as basic audio processing. However, it is not a replacement of dedicated DSP hardware, which is still required for high-performance applications.

The DSP extensions of the ARC EM architecture are fully supported by the compiler and can be utilized in compilations of non-DSP code, producing dense and fast code. ARC EM DSP includes the following features:

- [Fractional Data Types](#)
- [Rounding](#)
- [Saturation](#)
- [Vector Operations](#)
- [Complex Operations](#)
- [Butterfly Acceleration](#)
- [Accumulator Operations](#)

### 11.1 Fractional Data Types

Support for signed number in the range  $1 > n \geq -1$ . Fractional data types are denoted as Q7, Q15 and Q31, in which the integer represents the number of fractional bits and the sign bit is implicit; EM DSP supports the Q15 and Q31 data types.

### 11.2 Rounding

Rounding occurs in case the result of an instruction lacks precision to return all lower significant bits. For example, a right-shift instruction, shifting the value `0xdeadbeef` right by four bits; the non-rounding truncating instruction `ASRS` yields the result `0xfdeadbee`; the rounding version of the arithmetic shift `ASRSR` on the other hand yields the result `0xfdeadbef`, adding  $\frac{1}{2}$  LSB to the result to account for the `0xf` value that got shifted out on the right side.

In general rounding can be thought of as adding  $\frac{1}{2}$  LSB to a result, followed by setting all bits below the LSB to 0. For example converting a 32-bit fractional value to a 16-bit fractional value involves adding  $0x00008000$  ( $= \frac{1}{2}$  LSB) to the 32-bit fractional before setting the lower 16 bits of the result to 0.

Two rounding modes are supported:

- [Rounding Up](#)
- [Convergent Rounding](#)

The rounding mode can be selected using the RM bit in the DSP\_CTRL register. The rounding mode bit applies to all fractional instructions.

### 11.2.1 Rounding Up

Rounding up involves adding  $\frac{1}{2}$  LSB to the result before truncating, as described in [Rounding](#).

### 11.2.2 Convergent Rounding

Convergent rounding is also known as “rounding to nearest even.” In most cases convergent rounding yields the same result as rounding up. The only exception is when the  $\frac{1}{2}$  LSB is set and all lower significant bits ( $\frac{1}{4}$  LSB ...) are cleared; in this case convergent rounding only rounds up if the LSB is set (odd), yielding a result that has the LSB cleared (even).

Over multiple instructions convergent rounding avoids the error introduced by always rounding 0.5 up. This is especially relevant for filters with an infinite response (IIR), where rounding up would lead to an offset.

## 11.3 Saturation

Saturation is only supported on signed arithmetic instructions. Saturation occurs as a result of an arithmetic overflow. Instead of modulo wrapping the result as in non-saturating instructions the result clips to the maximum or minimum value without changing the sign bit of the result.

For example adding two 32-bit numbers  $0xdeadbeef$  and  $0x80000000$  results in a 33-bit result  $0x15eadbeef$ , too wide to represent in a 32-bit register. In this example a non-saturating instruction ADD.F returns  $0x5eadbeef$  and sets the overflow bit in the STATUS32 register; the saturating instruction ADDS.F on the other hand clips the result to the minimum negative value  $0x80000000$ , setting the overflow flag as well as the sticky saturation flag in DSP\_CTRL.

## 11.4 Accumulator Operations

ARC EM includes four auxiliary registers storing the state of the accumulators: ACC0\_LO, ACC0\_HI, ACC0\_GLO, and ACC0\_GHI. Vector multiply and multiply accumulate operations affect the state of all these registers; the low vector lane state gets reflected in the ACC0\_LO and ACC0\_GLO registers; the high vector lane state gets reflected in the ACC0\_HI and ACC0\_GHI registers. Wide accumulator operations store their state in the ACC0\_LO, ACC0\_HI, and ACC0\_GHI registers; the ACC0\_GLO register does not change its value on wide operations. For more information on accumulator operations, refer *ARC ISA Programmer's Reference Guide*.

**Table 11-1 Accumulator Operations**

| Accumulator Operations |                                                      |
|------------------------|------------------------------------------------------|
| Mnemonic               | Description                                          |
| ASLACC                 | Required for efficient normalization of accumulator  |
| ASLSACC                | Required for efficient normalization of accumulator  |
| FLAGACC                | Required for testing accumulator                     |
| GETACC                 | Required for casting and spilling of the accumulator |
| DIVACC                 | Efficiency avoiding emulation                        |
| SQRTACC                | Efficiency of complex arithmetic, avoiding emulation |
| NORMACC                | Required for efficient normalization of accumulator  |
| SETACC                 | Required for restoring the accumulator after a spill |

## 11.5 Vector Operations

Low and high accumulators used separately, For more information on vector operations, refer *ARC ISA Programmer's Reference Guide*.

## 11.6 Complex Operations

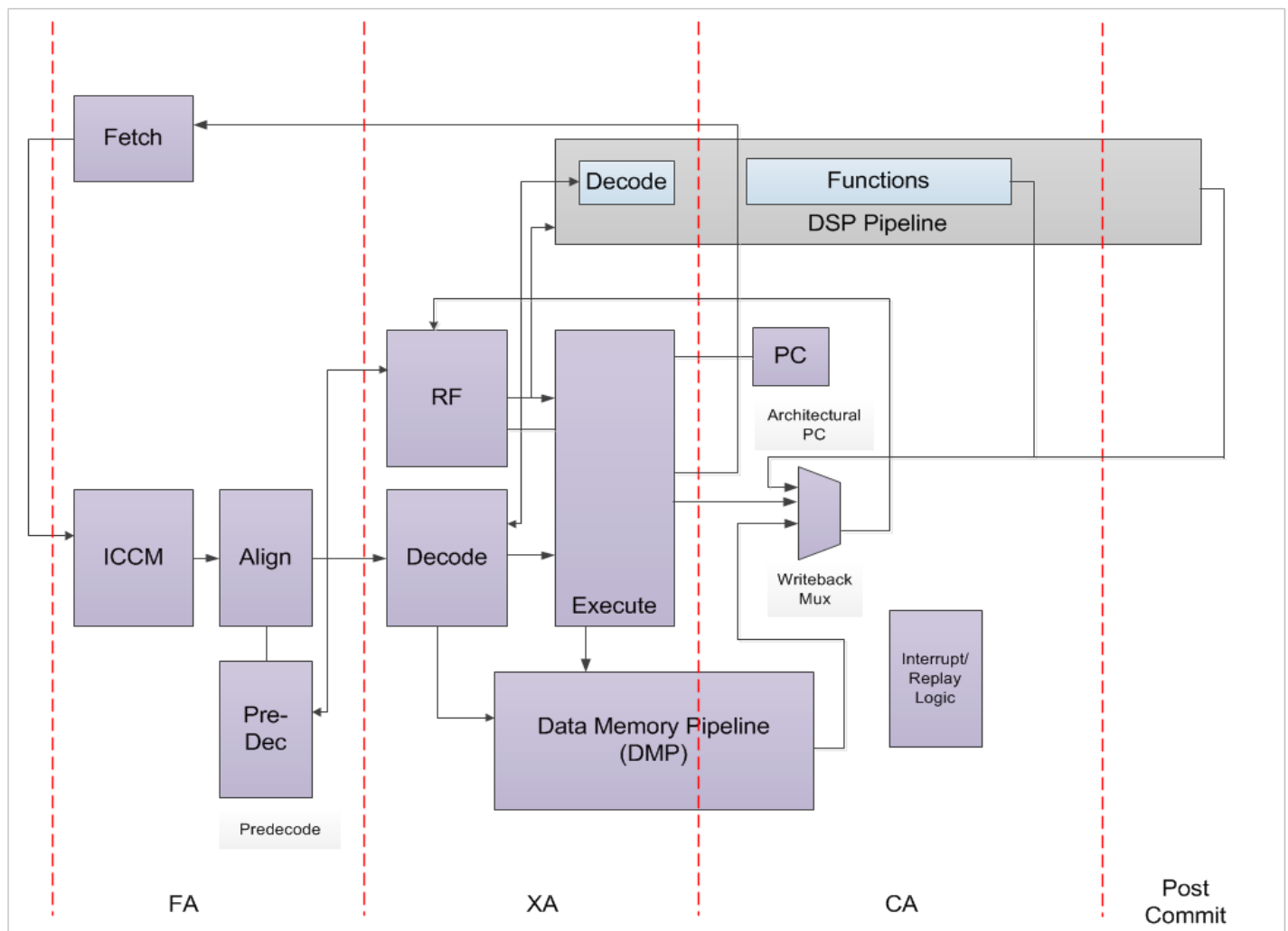
For 16-bit complex multiply instructions the four multipliers calculate the four 16x16 partial products. The complex multiply function subtracts the product of the two imaginary parts; the complex conjugated multiply subtracts the product of one of the imaginary\*real products. For more information on complex operations, refer *ARC ISA Programmer's Reference Guide*.

## 11.7 Butterfly Acceleration

ARC EM features two instructions to accelerate FFT butterfly computations. The butterfly instructions always round based on the rounding mode from the DSP control auxiliary register (DSP\_CTRL). Optionally, the butterfly instructions will support scaling of the results. Scaling is controlled by the DSP\_FFT\_CTRL.SC bit.

## 11.8 ARC EM DSP Pipeline Behavior

ARC EM DSP extends the baseline ARC EM with a DSP pipeline that encapsulates all DSP related instructions.

**Figure 11-1 ARC EM DSP Pipeline**

DSP spans the Execute (XA) and Commit (CA) stages, and two post-commit stages:

- In the XA stage the DSP pipeline decodes DSP instructions and flop the operands to the pipeline, to prevent the inputs to the pipeline from toggling on non-DSP instructions.
- In the CA stage the DSP pipeline performs the saturating addition, subtraction, and shift operations as well as the multiply-accumulate operation.
- In the post-commit stage the DSP pipe writes back ALU results and performs rounding and saturation for MAC operations.
-

## 11.9 Instruction Timing

For the 32x32 datapath configuration all instructions are issued at a rate of one instruction per clock cycle. Given the three stage DSP MAC pipeline, all MPY/MAC operations as well as the accumulator operations have a latency of three cycles for the 32x32 datapath configuration.

All ALU (add/sub/shift) operations always have two cycles latency. In case of an explicit read after implicit accumulator write hazard then the pipeline will stall for one cycle. Implicit DSP accumulator read after implicit write hazards will not cause any bubble.

### 11.10 ARC EM DSP Configurations

| Option Name     | ARCHitect Option | Option Values        | Description                                                                                                                |
|-----------------|------------------|----------------------|----------------------------------------------------------------------------------------------------------------------------|
| Implementation  | -dsp_impl        | inferred   optimized | The datapath components may be inferred from Verilog or optimized using carry-save components for timing closure purposes. |
| Complex support | -dsp_complex     | no   yes             | Include complex arithmetic, single cycle complex multiply, conjugated multiply, and FFT butterfly.                         |





# 12

## Extending the Processor

DesignWare ARC EM family processors provide an interface for adding *ARC Processor Extensions (APEX)* using the ARChitect tool's APEX Wizard. You can use the APEX interface to expand the processor architecture by adding proprietary instructions, core and auxiliary registers with special access modes, condition codes, and flags. You can also add signals to integrate external devices such as third-party IP with your extension. This feature can make data from the external IP available directly to processor instructions without going through a bus.

The additions to the processor are organized into functional blocks called *APEX*. An *APEX* can contain one or more instructions, core or auxiliary registers, condition codes, or signals. APEX are stored in an extension library and can be added to a design using drag-and-drop. The APEX interface handles the integration automatically.

### Advantages of APEX

APEX let you add custom instructions into the existing framework of a processor without worrying about control logic or how the instruction is handled. You design an independent piece using a wizard with just a few lines of Verilog.

Using APEX lets you bring in third-party IP or your own previous design and make it directly accessible by processor instructions with no need to build busses or other hardware. For example, a standalone DSP co-processor can be integrated directly at the processor level without busses and the associated delays and pipeline issues.

Proprietary functions implemented as APEX are also much harder to hack, making your code more secure.

### APEX Features

- *Extension instructions* implement a custom function in hardware to be added to the processor, for example a custom adder to alleviate a bottleneck in application code.
- *Extension core registers* can be added to the core. These extension registers are available to all instructions and are typically used for information that changes often or must be accessed quickly.
- *Extension auxiliary registers* are accessible in software using the LR and SR instructions and are typically used for information that does not change often and does not need to be accessed quickly.
- *Condition codes* can be used for conditional execution depending on the value of a processor flag or combination of flags.

- *Signals* added using APEX propagate to the processor's core boundary, resulting in additional boundary ports. They can be input, output, or bidirectional.

### For More Information

For more information, see the *APEX Databook* provided with your processor documentation. The *APEX Databook* provides complete information on the following:

- APEX development flow
- APEX Wizard
- Processor APEX interface
- Optimizing applications and eliminating bottlenecks using APEX

# 13

## Multi-Core Option

The ARC EM processor cores can be integrated into a multi-core system with an optional ARConnect IP to accommodate a wide range of applications, such as divide-and-conquer parallel applications or pipelined stream applications.

The ARConnect IP is a collection of system components to provide efficient inter-core interconnections and manipulations in multi-core scenarios.

The system components in ARConnect IP include:

- Inter-core Communication Components
  - Inter-core Interrupt Unit
  - Inter-core Semaphore Unit
  - Inter-core Message
- Multi-core Debug-assist Unit
- Global Real-Time Counter Unit
- Power Management Unit

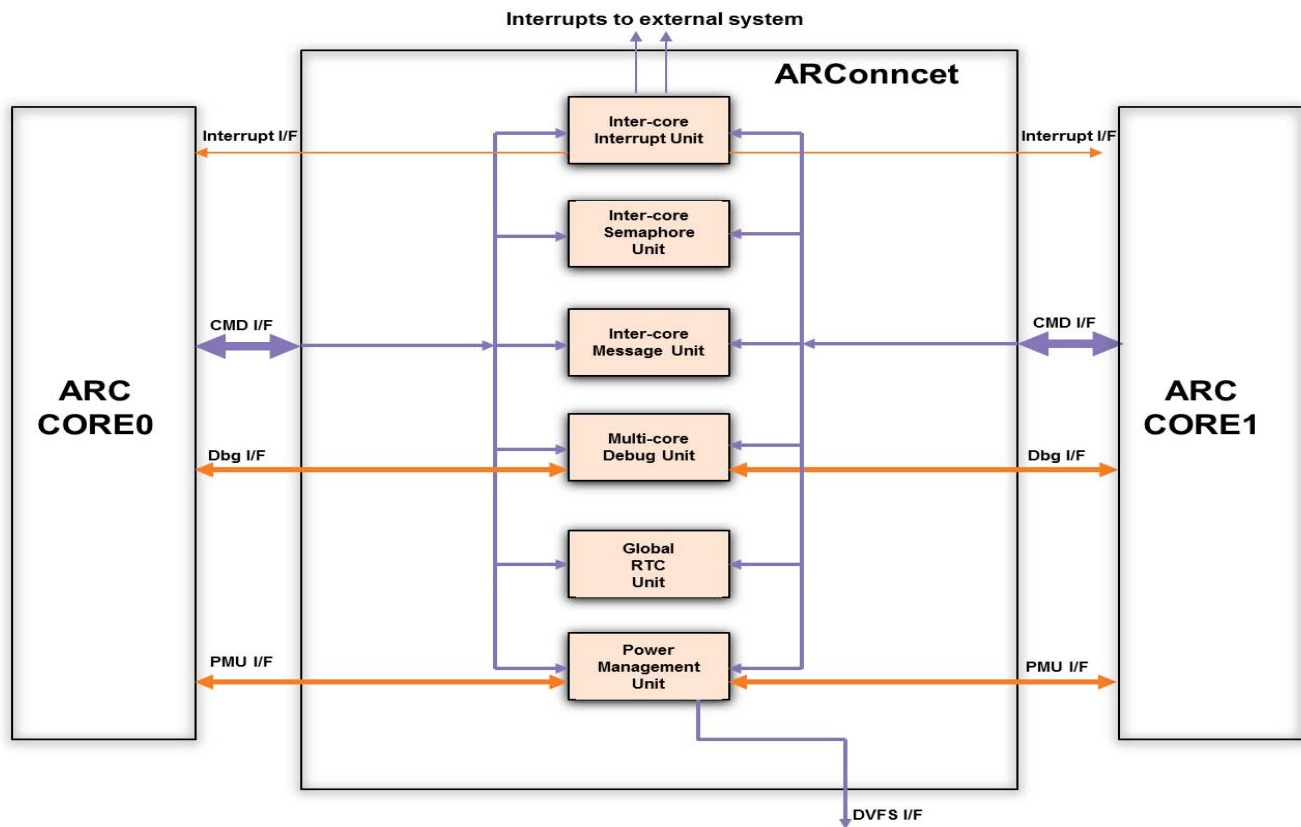


The ARConnect IP can also be integrated with single-core system, for example, a single EM core can be configured with ARConnect Power Management Unit to reduce the power consumption.

## 13.1 Multi-Core Integrated with ARConnect IP

Figure 13-1 shows two ARC cores integrated with ARConnect IP.

Figure 13-1 System Diagram of two ARC cores integrated with ARConnect



The ARConnect IP in Figure 13-1 connects two ARC cores. The processor cores connected to ARConnect are numbered with CORE ID from CORE 0 to CORE N-1 (depend on the number of cores)

Each core can send operation requests (encoded to a set of ARConnect commands) or data to ARConnect components by writing auxiliary registers, MCIP\_CMD and MCIP\_WDATA; and the read-back data is received in the auxiliary registers, MCIP\_READBACK. For example, each core can use SR instruction to write CMD\_CHECK\_CORE\_ID command to MCIP\_CMD, and then use LR instruction to read MCIP\_READBACK, to check out its CORE ID.

The introductions of auxiliary registers, MCIP\_CMD, MCIP\_WDATA and MCIP\_READBACK can be referred in section Table 13-6.

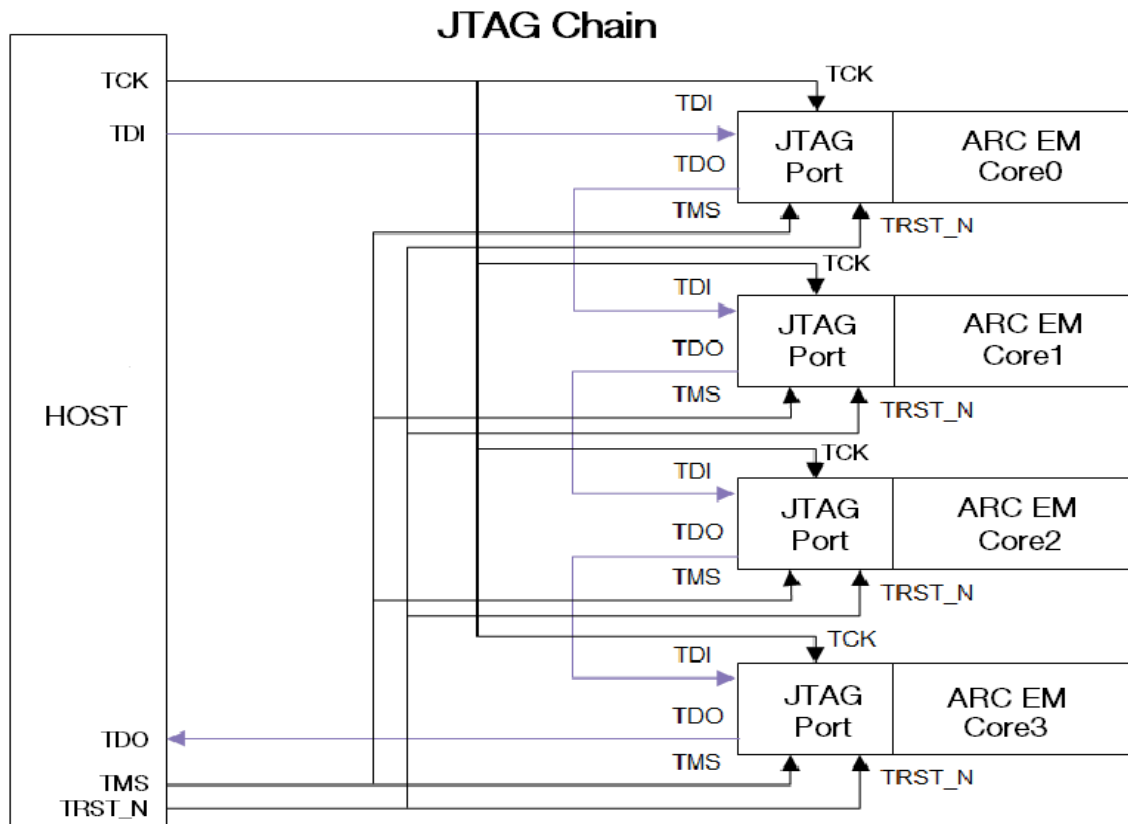
The full list of ARConnect commands can be referred in Table 13-14

To accommodate debugging of each core in the multi-core system, the JTAG port of the processor core can be connected to form a JTAG chain.

A four-core system with a JTAG chain is shown in Figure 13-2. The connection layout is as follows:

- TDI host signal is connected to the TDI signal of Core0.

- TDO signal of Core0 is connected to the TDI signal of Core1.
- TDO signal of Core1 is connected to the TDI signal of Core2.
- TDO signal of Core2 is connected to the TDI signal of Core3.
- TDO signal of Core3 is connected to the TDO signal of the host.
- TCK, TMS and TRST\_N signals of host are connected JTAG port of each ARC core.

**Figure 13-2 JTAG Chain Diagram**

ARConnect have dedicated clock signals. Table 16-1 lists the full description of the reset, test\_mode and clock ports.

**Table 13-1**

| Signal Name    | Directions | Bus Width | Description                                     |
|----------------|------------|-----------|-------------------------------------------------|
| mcip_clk       | Input      | 1         | The clock signal of ARConnect.                  |
| rst_a          | Input      | 1         | External reset signal, asynchronously asserted. |
| mcip_test_mode | Input      | 1         | Test mode signal of ARConnect.                  |

| Signal Name           | Directions | Bus Width | Description                                                                                                                                                                                                                                                 |
|-----------------------|------------|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mcip_<core_pfx>clk_en | Input      | 1         | No integer clock ratio supported, so it should be always tied to 1'b1 to guarantee the functional correctness. The <core_pfx> prefix represents the unique signal prefix name of each processor core when integrated with ARConnect in a multi-core system. |

## 13.2 Introduction of ARConnect Components

### 13.2.1 Inter-Core Interrupt Unit

ARConnect Inter-core Interrupt Unit support inter-core interrupts and acknowledgements between any two cores to handle asynchronous communication events. Each core can utilize specific ARConnect Command to manipulate Inter-core Interrupt Unit. The ARConnect commands can be referred in [Table 13-14](#).

#### 13.2.1.1 Features

Each ARC EM processor (interrupt initiator) can use inter-core interrupt unit:

- Generates pulse-sensitive interrupts to external system.
- Generates level-sensitive interrupts to any other core.
- Reads the status (acknowledged or not acknowledged by the interrupt receiver) of the outstanding interrupts.

Each ARC core (interrupt receiver) can use Inter-core interrupt unit to:

- Check the source of interrupts generated from which interrupt initiator(s).
- Generates interrupt acknowledges to any other core.

#### 13.2.1.2 Generating Inter-core Interrupts

Each core can generate interrupts to other core by using ARConnect command `CMD_INTRPT_GENERATE_IRQ`. The interrupts generated by Inter-core Interrupt Unit from other initiator cores to a receiver core are combined to a single interrupt line and connected to `irq19_a` of the receiver core.

Each core can generate interrupts to external system by using ARConnect command `CMD_INTRPT_GENERATE_IRQ`. The interrupts to external system through Inter-core Interrupt Unit are indicated by separated interrupt lines. So for an N-cores system, there are N interrupt lines generated from Inter-core Interrupt Unit to the external system. Table 16-1 lists the full description of the interrupt interface from Inter-core Interrupt Unit to external system.

**Table 13-2 Interrupt to External System**

| Signal Name                | Directions | Bus Width | Description                                                                    |
|----------------------------|------------|-----------|--------------------------------------------------------------------------------|
| mcip_irq_core<pfx>_to_host | Output     | 1         | Pulse-sensitive interrupt to external system from core with prefix <core_pfx>. |

**Note**

The <core\_pfx> prefix represents the unique signal prefix name of each processor core when integrated with ARConnect in a multi-core system.

The interrupt to external system is pulse-sensitive (asserts for one cycle of the ARConnect IP clock domain), therefore, Clock Domain Crossing (CDC) logic is needed to sample the pulse if the clock domain of the external system is asynchronous to ARConnect IP clock domain.

**13.2.1.3 Check Inter-core Interrupt Sources**

Since the Inter-core Interrupts from other cores to a given core are combined to a single interrupt line, each core acting as an interrupt receiver can use ARConnect command CMD\_INTRPT\_CHECK\_SOURCE to check which core or cores the interrupt is generated by. The MCIP\_READBACK register returns a vector value, each bit of which represents the interrupt generation status of one core. If there are multiple interrupt-initiating cores generating interrupts to the same interrupt receiver core, the core can choose to service them one by one or all of them together in the interrupt service routine for different application scenarios.

**13.2.1.4 Acknowledge Inter-core Interrupts**

As an interrupt receiver, each core can use ARConnect command CMD\_INTRPT\_GENERATE\_ACK to generate interrupt acknowledgments to the interrupt-initiating cores in the system.

**13.2.1.5 Check acknowledge Status**

As an interrupt initiator, each core can use ARConnect command CMD\_INTRPT\_READ\_STATUS to check the interrupt acknowledge status from it to other cores. The MCIP\_READBACK register returns the interrupt status of the specific core. A value of 1 indicates that there is an outstanding interrupt, while the value of 0 indicates that there is no outstanding interrupt or a former interrupt has been serviced (acknowledged).

As an interrupt initiator, it is recommended to check the status of a previous interrupt it generated before generating another interrupt(s) to the same interrupt receiver core. This is to ensure that the previous interrupt is not overridden.

**13.2.2 Inter-Core Semaphore Unit**

ARConnect Inter-core Semaphore Unit support light weight semaphores to aid inter-core communications for common shared resources between multiple cores. Each core can utilize specific ARConnect Command to manipulate Inter-core Semaphore Unit. The ARConnect commands can be referred in [Table 13-14](#).

### 13.2.2.1 Features

- Support up to thirty-two semaphores and the number of semaphores is build-time configurable.
- Each core can claim a semaphore from Inter-core Semaphore Unit, but the mutual exclusion is provided among all cores, that is, if one core claimed and obtained the ownership of a semaphore, other cores cannot claim the same semaphore.
- Each core can release an obtained semaphores to allow the semaphore available for other cores.
- Each core can compulsively release a semaphore that is obtained by other cores.
- Round-robin arbitration scheme is adopted to make sure the mutual-exclusively granting semaphore to requesting cores with randomized fairness.

### 13.2.2.2 Claim Semaphores

Each core in the can use ARConnect command `CMD_SEMA_CLAIM_AND_READ` to claim a semaphore and verify its ownership. The index of semaphore is set using the parameter field of the command. `MCIP_READBACK` register returns the claiming result. A returned value of 1 indicates that the semaphore is successfully owned by the core; otherwise, the semaphore is not available and has been claimed by other cores.

### 13.2.2.3 Release Owned Semaphores

If a core has already obtained a particular semaphore, to release the semaphore and allow it to be used by other cores, the core can use ARConnect command `CMD_SEMA_RELEASE` to release its ownership. The index of a semaphore is set using the parameter field of the command.

### 13.2.2.4 Release Semaphores of Other Cores

In some applications, each core can use ARConnect command `CMD_SEMA_FORCE_RELEASE` to compulsively release a semaphore that has been occupied by other cores in the system and make it available. The index of semaphore is set using the parameter field of the command.

## 13.2.3 Inter-Core Message Unit

ARConnect Inter-core Message Unit implemented a small size, low-latency SRAM (single-port) to provide efficient message passing among cores or supports the synchronizing of the tasks between any paired cores in the multi-core system at low latency. Each core can utilize specific ARConnect Command to manipulate Inter-core Message Unit. The ARConnect commands can be referred in [Table 13-14](#).

### 13.2.3.1 Features

- Each core can write or read Message SRAM to send or receive messages.
- Round-robin arbitration scheme is adopted if two or more cores want to read from or write to the Single-port SRAM simultaneously.
- The size of the SRAM is build-time configurable, supporting 128, 256, 512, 1K, 2K, and 4K Bytes.
- The data width for each SRAM read/write operation is fixed at 32 bits.



Message SRAM is not mapped to the memory address space, therefore, cannot be accessed by Load and Store instructions.

**Note**

The SRAM address occupies 4 KB of space even if the configured size is smaller than 4 KB. If the address is accessed above the size of SRAM, then the address wraps around into the actual SRAM location.

The read or write access to the SRAM can be accomplished by an immediate address set into the MCIP\_CMD register accompanied with corresponding SRAM read/write commands. In order to facilitate consecutive SRAM access operations with regular address strides, incremental mode is also supported to read or write SRAM by setting the start address and offset address.

## 13.2.4 Multi-Core Debug Unit

ARConnect Multi-core Debug Unit provides additional debug assist features in multi-core scenarios. Each core can utilize specific ARConnect Command to manipulate Multi-core Debug Unit. The ARConnect commands can be referred in [Table 13-14](#).

### 13.2.4.1 Features

Each core can use Multi-core Debug Unit to:

- Selectively Halt any combination of other cores simultaneously.
- Selectively Run or restart (after previous halt) any combination of other cores simultaneously.
- Selectively Reset any combination of other cores simultaneously.

Each core use Multi-core Debug Unit to:

- Conditionally halt any core based on a halt occurring in one or more of the other cores

**Note**

Only one core can access the multi-core debug unit at one time. If two or more cores access it simultaneously, the hardware result is unpredictable. Software need to guarantee no access conflict occurs.

### Selective run/halt/reset

To support selective run/halt/reset, different Multi-core Debug commands are used to generate run/halt/reset requests to target cores.

The reset command only generates resets to the cores, and does not affect other components in the system. Typical scenarios for using the reset command include those where the state of the core being reset is known by the Debugger or by the core generating the reset. For example, if Core1 is the master CPU of the system,

it may halt Core2, and then reprogram the memory for Core2. Core1 can then issue a reset to Core2 to clear all retained state from the previous program and start Core2 running the new program.

**Note**

The reset signal generated by the multi-core debug mechanism is not interlocked with the core pipeline in any way, that is, it is up to the software to ensure that it is safe to issue a reset signal from the multi-core debug unit prior to doing so.

This should be done by halting the core to be reset, then ensuring there are no outstanding bus transactions in progress. Once the core is fully inactive, the reset signal can be safely issued. Failure to do this could result in the pipeline locking up.

The target core could be under sleep or lower power mode when it receives the halt, run or reset request.

**Selective Halt Request:**

- If the core is already under halt state, the halt request will be masked and not be sent to target core.
- If the core is already under sleep or power down mode, the halt request will keep asserting until the target core acknowledged the halt request (after waking up or powering up). The target core will be halted after acknowledged the halt request.

**Selective Run Request:**

- If the core is already under halt state, the run request will restart the target core first and then the target core starts to run.
- If the core is already under sleep state, the run request will keep asserting until the target core acknowledged the run request (after waking up). The target core will be unhalted after acknowledged the run request.
- If the core is already under power down mode, the run request will power up the target core first and then the target core starts to run.

**Selective Reset Request:**

- If the core is already under halt or sleep state, the reset request will reset the target core.
- If the core is already under power down mode, the run request will power up the target core first and then reset the target core.

**Conditional Halt**

To support conditionally-halts, the Multi-core Debug Unit contains internal MASK and SELECT registers to control which cores are observed for triggering and to select which cores are controlled by the result.

The internal MASK register provides four bits per core to select any of the following events from the core to selectively generate a global halt pulse if that core halts:

- Core global H bit (H): corresponding to the H field of the STATUS32 register.
- Actionpoint halt (AH): corresponding to the AH field of the DEBUG register.
- Software breakpoint halt (BH): corresponding to the BH field of the DEBUG register.
- Self halt (SH): corresponding to the SH field of the DEBUG register.

The internal SELECT register selects the processor cores to be halted by this global halt pulse.

Multi-core debug commands are used to set and read the internal MASK and SELECT registers. Any or all of the MASK registers can be set with a single command.

**Note**

A core cannot conditionally halt itself. For example, a four-core system has its multi-core debug unit configured to halt all cores whenever a self-halt occurs in any core. If Core2 halts, Core1, Core3 and Core0 are conditionally halted by the debug unit as a result. Core2 has already been halted at this point and cannot be halted by the debug unit again. In this case, the EH bit in the DEBUG registers of Core1, Core3, and Core0 are set, but the EH bit in the DEBUG register of Core2 is not set.

## 13.2.5 Global Real-Time Counter (GRTC)

ARConnect Global Real-Time Counter Unit implemented a 64-bit free-running Global Real-Time Counter (GRTC) which can be used as a central timing reference when multiple cores are under different clock domains. Each core can utilize specific ARConnect Command to manipulate Global RTC Unit. The ARConnect commands can be referred in [Table 13-14](#).

### 13.2.5.1 Features

- Each core can read the value of GRTC
- Each core can clear the value of GRTC

Since the data bus width between the core and the ARConnect IP is 32 bits, the read sequence is the following:

1. Read the low 32-bit of the counter
2. Read the high 32-bit of the counter

Reading the low 32-bit of the counter triggers a snapshot of current moment and buffer the higher 32-bit of GRTC. Each core has its private snapshot buffer registers.

## 13.2.6 Power Management Unit

ARConnect Power Management Unit provides the capability to individually shut down the power of different power domains and dynamically regulate the supply voltage and clock frequency (DVFS) of the whole multi-core system, to reduce the power consumption. Each core can utilize specific ARConnect Command to manipulate Power Management Unit. The ARConnect commands can be referred in [Table 13-14](#).

### 13.2.6.1 Features

Power Management:

- Divide multi-core system into different power domains
- Control different power domains to be powered-down or up with different approaches
- Provide each power domain's status information visible to software and interface to external system.

- Provide reference UPF flow to infer isolation cells, power switches, etc.

DVFS:

- Control the whole multi-core system to execute at one of four performance levels (PLs). The highest performance level is to achieve highest performance. The lowest performance level is to achieve lowest power consumption.



**Note**

Only one core can access the power management unit at one time. If two or more cores access it simultaneously, the hardware result is unpredictable. Software need to guarantee no access conflict occurs.

### 13.2.6.2 Power Domain Partition

The entire system consists of:

- a. Each individual processor cores
  - b. MP (Multi-core power domain)
    - i. ARConnect IP
  - c. RTT(if configured on)
1. Each core is divided into 3 power domains:
    - a. Core Always On domain, including logics of:
      - i. RTC
      - ii. Timer0, timer1, watchdog timer (if configured on)
      - iii. Interrupt re-sync
      - iv. Core re-sync
      - v. JTAG port
      - vi. Clock control
      - vii. Reset control logic
      - viii. pmu\_ctrl logic
    - b. Core Power domain 2 (PD2):
      - i. Architecturally visible memories (i.e. CCMs if configured)
    - c. Core Power domain 1 (PD1):
      - i. Except PD1 and PD2 logics, other logics and architecturally invisible memories (e.g. Caches)
  2. MP is divided into 3 power domains:
    - a. MP Always On domain
      - i. ARConnect internal programmable registers (e.g. SELECT, MASK register in Multi-core Debug Unit, etc.)
      - ii. ARConnect Power Management Unit
      - iii. ARConnect Global-RTC Unit

- b. MP Power Domain 2 (PD2)
  - i. Architecturally visible memories (i.e. message SRAM in Inter-core Message Unit if configured)
- c. MP Power domain 1 (PD1):
  - i. Except PD1 and PD2 logics, other logics and architecturally invisible
- 3. RTT put in single power domain
  - a. RTT Power domain 1 (PD1)
    - i. Entire RTT (including logistics and memories)

### 13.2.6.3 Power Down Modes

The PMU provides two types of power down modes, PM1 and PM2, to shut off the power supply of logics or memories located at power domain1 and power domain2.

#### 13.2.6.3.1 Core PM1 and PM2 modes

##### Core PM1 mode:

- For each core, entering PM1 mode shuts down power supply of PD1. Architecturally visible memories in core PD2 enter into Deep Sleep (if supported by the memory cells) status.



**Note**

In PM1 mode, any bus access to memories in deep sleep status will result in a bus error.

##### Core PM2 mode:

- For each core, entering PM2 mode shuts down power supply of both core PD1 and PD2.

#### Approaches for entering into PM1 and PM2 modes

- Each core can be powered down by this core itself executing SLEEP instruction. When the core is idle, in order to reduce the static power consumption software can execute a SLEEP instruction to enter into a specific sleep mode. The operand bits ([7:5]) of the SLEEP instruction indicate different power down modes, value 6 indicate PM1 mode, value 7 indicate PM2 mode. sys\_sleep\_mode\_r[2:0] output vector exposed power down modes to PMU, and PMU will shut down different power domains accordingly.



**Note**

The SLEEP instruction with power down mode is treated as a NOP instruction when single-stepping is used by the debugger.

- Each core can be powered down by programming PMU. PMU contain PSTATUS registers for individual cores, which can be programmed by using ARConnect command CMD\_PMU\_SET\_PMODE to set power down modes, and PMU will shut down different power domains of this core accordingly.



**Note**

The hardware does not guarantee the interlocks - software needs to make sure the target core should be in halted state before being powered-down by another core. For example, if core1 wants to program PMU to power down core2, software is responsible to use some halt mechanism (multi-core debug unit, or self-halt) to make sure Core2 has been halted first, and then program PMU to power down Core2.

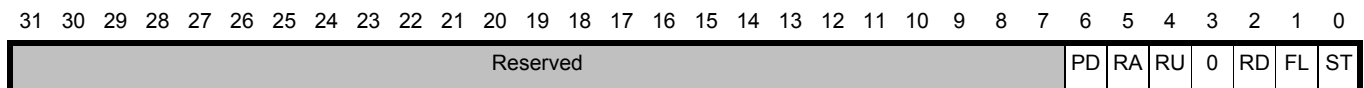
**Save-and-Restore mechanism:**

- Prior to powering down the core, software must save all required architectural status to external memory before powered-down and restore them back after the core powered up again. For example, STATUS32, contents of all general registers, contents of CCMs, etc. may need be saved to external memory before powered-down by software using Store instructions, after powered up all of those contents need to be restored back by software using Load instructions.
- The contents under always on domain do not need to be saved-and-restored, e.g. all ARConnect internal programmable registers, each core's RTC, timer0/1, etc.

**Extended register fields when PMU is configured:**

When PMU is configured on, a field named PD is added to each core's JTAG status register (0x8). The PD bit will be set when the debug transaction failed due to the core is powered down (in this case, the FL bit is not set), and the following fields will return constant data.

- stalled flag (ST) is 0
- failure flag (FL) is 0
- ready flag (RD) is 1
- run flag (RU) is 0
- reset applied (RA) is 0

**Figure 13-3 JTAG Status Register**

Timers and RTC are under the always on domain. Software can control the Timers or RTC to continue run or halt when a core powered down.

1. RTC can be halted by programming the AUX\_RTC\_CTRL.E bit.
2. A field named PD is added to the timer0/1 auxiliary registers (CONTROL0/1, 0x22 and 0x101) when PMU is configured on. Timer0/1 can be halted by setting this field on.

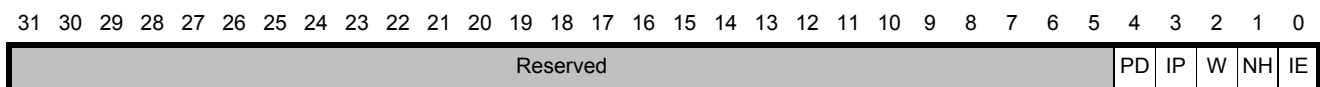
**Figure 13-4 Timer CONTRL0/1 Register**

**Table 13-3 Timer CONTROL0/1 Register Field Descriptions**

| Field | Bits | Description                                                                                                                                                                                           |
|-------|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| PD    | [4]  | Power down behavior<br>1: Timer continues to count when the core is in power down mode.<br>0: Timer clock will be gated to save power.<br>This is only present when core is configured with ARConnect |

**13.2.6.3.2 MP PM1 and PM2 Modes****MP PM1 mode:**

- For MP, entering PM1 mode shuts down power supply of MP PD1. Architecturally visible memories in MP PD2 enter into Deep Sleep (if supported by the memory cells) status

**Note**

In PM1 mode, any bus access to memories in deep sleep status will result in a bus error.

**MP PM2 mode:**

- For MP, entering PM2 mode shuts down power supply of both MP PD1 and PD2.

**Approaches to enter into MP PM1 or PM2 modes:**

- MP can be powered down by programming PMU. In order to power down the entire system to save more power consumptions, software can power down MP besides to power down all cores. PMU contain PSTATUS registers for MP, which can be programmed by using ARConnect command CMD\_PMU\_SET\_PMODE to set power down modes for MP. After the MP PSTATUS is programmed with specific power modes, it should be noted only when all cores have entered into power down mode (either PM1 or PM2) and all outstanding transactions in the MP have been completed, then PMU shut down MP; When any core is going to be powered up by any power-up event, PMU will power up MP first.

**13.2.6.3.3 RTT PM1 Mode****RTT PM1 mode:**

- For RTT, entering PM1 mode shuts down power supply of RTT PD1.

**Approaches to enter into RTT PM1 mode:**

- RTT can be powered down by programming PMU. Software can power down RTT when RTT is idle to save power consumptions. PMU contain PSTATUS registers for RTT, which can be programmed by using ARConnect command CMD\_PMU\_SET\_PMODE to set power down modes for RTT. After the RTT PSTATUS is programmed with specific power modes, PMU shut down RTT immediately.

**Note**

The hardware does not guarantee the interlocks - software needs to make sure there are no pending transactions in RTT before powering it down.

### 13.2.6.4 Power Up Events

#### 13.2.6.4.1 Core Power-up Events

Each processor core will be powered up at the following cases:

- Powered up by interrupts to the core. Each interrupt meet all of following conditions can power up the powered-down core:
  - a. Interrupts are globally enabled (set in STATUS32.IE).
  - b. This interrupt is enabled (set in IRQ\_ENABLE).
  - c. The priority level of the interrupt is higher or equal to the interrupt priority threshold of the core (set in STATUS32.E[3:0])
- Powered up by external reset. The external reset will reset all logics in the multicore system, including PMU itself and all power domains in:
  - a. Individual cores
  - b. MP
  - c. RTT

When the external reset is applied, the control signals to isolation cells and power switches are disabled and the processor core is powered up (if it is powered down).



#### Note

The external reset should be applied with enough clock cycles for the power rails stabilization, the exact cycles count depends on different technology processes determined by the system integrator.

- Powered up by the watchdog\_reset or wdt\_reset of the core. The watchdog\_reset or wdt\_reset outputs are generated from timers in the core. PMU will power up the core when it detected its watchdog\_reset or wdt\_reset asserted. And watchdog\_reset will reset entire core.
- Powered up by selective reset. Selective reset is generated by Multi-core Debug Unit. PMU will power up the core when it detected its selective reset request. And selective reset will reset entire core.
- Powered up by run request generated by external run request or selective run request. Each core has an external run request port arc\_run\_req\_a. The selective run is generated by Multi-core Debug Unit. PMU will power up the core when it detected any kind of run request.
- Powered up by programming the PMU. PMU contain PSTATUS registers for individual cores, which can be programmed by using ARConnect command CMD\_PMU\_SET\_PMODE to set power up mode, and PMU will power on this core accordingly.
- Powered up by Debugger programming the DEBUGP register. DEBUGP register is an optional auxiliary register of each core which only present when PMU is configured on. The Debugger can set the PUT bit of the DEBUGP register to power up the core. The layout of DEBUGP can be referred in [DEBUGP Register](#).

After powered up, the powered-down domain in core will be applied with a reset to make sure all flip-flops and states return to the default values after powered up. An internal power-up duration counter PCNT is implemented in PMU to control the applied reset cycles after each domain powered up. PCNT can be programmed by CMD\_PMU\_SET\_PCNT command.



If the processor core wants to resume its program execution to the exact states before it powered-down, the Save-and-Restore mechanism should be adopted.

#### 13.2.6.4.2 Power Up Events for the MP Domain

MP only can be powered down when all the cores have been powered down. MP will be powered-up at following cases:

- Powered up by an external reset. The external reset will reset all logics in the multicore system.
- Powered up when any core is going to be powered up, because PMU always powers up the MP first.
- Powered up by programming the PMU. PMU contain PSTATUS registers for MP, which can be programmed by using ARConnect command CMD\_PMU\_SET\_PMODE to set power up mode for MP, and PMU will power on MP accordingly.

After powered up, the powered-down domain in MP will be applied with a reset. PMU internal power-up duration counter PCNT also control the applied reset cycles after MP powered up.

#### 13.2.6.4.3 Power Up Events for RTT

RTT can be powered down entirely when it is idle. It will be powered-up at following cases:

- Powered up by an external reset. The external reset will reset all logics in the multicore system.
- Powered up by programming the PMU. PMU contain PSTATUS registers for RTT, which can be programmed by using ARConnect command CMD\_PMU\_SET\_PMODE to set power up mode for RTT, and PMU will power on RTT accordingly.

After power up, the powered-down domain in RTT will be applied with a reset. PMU internal power-up duration counter PCNT also control the applied reset cycles after RTT powered up.

#### 13.2.6.4.4 Interface indicating Power Status

The power mode of each domain is ported as outputs and can be used to monitor the power status at SoC level. A full description of the power status interface from PMU is shown in [Table 13-4](#)

**Table 13-4 Power Status Interface to External System**

| Signal Name     | Direction | Bus Width | Description                                                                                                          |
|-----------------|-----------|-----------|----------------------------------------------------------------------------------------------------------------------|
| <core_pfx>pmode | Output    | 2         | Core current power mode of core with prefix <core_pfx><br>2'h00: Power on mode<br>2'h01: PM1 mode<br>2'h10: PM2 mode |
| mp_pmode        | Output    | 2         | Current power mode of MP<br>2'h00: Power on mode<br>2'h01: PM1 mode<br>2'h10: PM2 mode                               |

| Signal Name | Direction | Bus Width | Description                                                          |
|-------------|-----------|-----------|----------------------------------------------------------------------|
| rtt_pmode   | Output    | 2         | Current power mode of RTT<br>2'h00: Power on mode<br>2'h01: PM1 mode |

**Note**

The `<core_pfx>` prefix represents the unique signal prefix name of each processor core when integrated with ARConnect in a multi-core system.

### 13.2.6.5 Dynamic Voltage and Frequency Scaling

Software can use specific ARConnect command to dynamically regulate the supply voltage and clock frequency of the entire multi-core system according to its workloads.

The entire multi-core system is able to run at four performance levels (PLs). PL value 0 indicates the highest performance level, and the entire multi-core system executes at the highest clock frequency with the maximum supply voltage. PL value 3 indicates the lowest performance level, and the entire multi-core system executes at the slowest clock frequency with the lowest supply voltage.

PMU DVFS need to co-work with an external DVFS controller at SoC system level. PMU outputs vector signals to demonstrate the PL value of the entire multi-core system, so that the external DVFS controller at SoC system level can regulate the power supply and frequency of the entire multi-core system according to its performance level.

A full description of the DVFS interface to external DVFS controller is shown in [Table 13-5](#)

**Table 13-5 DVFS Interface to External DVFS Controller**

| Signal Name           | Direction | Bus Width | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
|-----------------------|-----------|-----------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mcip_pmu_dvfs_v_req   | Output    | 1         | <p>Voltage regulation request. 4-phase handshake protocol is adopted.</p> <ol style="list-style-type: none"> <li>1. PMU assert up mcip_pmu_dvfs_v_req signal to send voltage regulation request to external DVFS controller.</li> <li>2. The external DVFS controller respond by asserting up mcip_pmu_dvfs_v_ack_a after the voltage regulation is completed.</li> <li>3. PMU de-assert mcip_pmu_dvfs_v_req after seeing mcip_pmu_dvfs_v_ack_a asserted.</li> <li>4. The external DVFS de-assert mcip_pmu_dvfs_v_ack_a after seeing mcip_pmu_dvfs_v_req de-asserted.</li> </ol>  |
| mcip_pmu_dvfs_v_ack_a | Input     | 1         | Acknowledgement signal for voltage request                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                        |
| mcip_pmu_dvfs_f_req   | Output    | 1         | <p>Frequency regulation request. 4-phase handshake protocol is adopted.</p> <ol style="list-style-type: none"> <li>1) PMU assert up mcip_pmu_dvfs_f_req signal to send frequency regulation request to external DVFS controller.</li> <li>2) The external DVFS controller respond by asserting up mcip_pmu_dvfs_f_ack_a after the frequency regulation is completed.</li> <li>3) PMU de-assert mcip_pmu_dvfs_f_req after seeing mcip_pmu_dvfs_f_ack_a asserted.</li> </ol> <p>The external DVFS de-assert mcip_pmu_dvfs_f_ack_a after seeing mcip_pmu_dvfs_f_req de-asserted.</p> |

| Signal Name           | Direction | Bus Width | Description                                                                                                                                                 |
|-----------------------|-----------|-----------|-------------------------------------------------------------------------------------------------------------------------------------------------------------|
| mcip_pmu_dvfs_f_ack_a | Input     | 1         | Acknowledgement signal for frequency request.                                                                                                               |
| mcip_pmu_dvfs_data    | Output    | 2         | Performance level requested to external DVFS controller<br>1) 2'h00: Request PL0<br>2) 2'h01: Request PL1<br>3) 2'h10: Request PL2<br>4) 2'h11: Request PL3 |

- To increase the performance level of the entire multi-core system, apply the following sequence to regulate its voltage and frequency:
  1. The PMU sends out a voltage regulation request to the external DVFS controller.
  2. The external DVFS controller responds to the request and increases the supply voltage according to the value of PL.
  3. The PMU sends out a frequency regulation request to the external DVFS controller.
  4. The external DVFS controller responds to the request and increases the clock frequency according to the value of PL.
- To decrease the performance level of the entire multi-core system, apply the following sequence to regulate its voltage and frequency:
  5. The PMU sends out a frequency regulation request to the external DVFS controller.
  6. The external DVFS controller responds to the request and decreases the clock frequency according to the value of PL.
  7. The PMU sends out a voltage regulation request to the external DVFS controller.
  8. The external DVFS controller responds to the request and decreases the supply voltage according to the value of PL.

## 13.3 Programming Model of ARConnect

### 13.3.1 Auxiliary Registers Supporting ARConnect

Table 13-6 summarizes the auxiliary registers defined in the ARC cores to support ARConnect IP. The definition of access permissions is from the *ARCv2ISA Programmer's Reference Manual*. Any attempt to access in user mode raises a privilege violation exception. Writing to a read-only register raises an illegal exception.

**Table 13-6 List of Auxiliary Registers supporting ARConnect**

| Name                       | Address | R/W | Description |
|----------------------------|---------|-----|-------------|
| <b>Auxiliary Registers</b> |         |     |             |

| Name                                 | Address | R/W | Description                                                                                                                                      |
|--------------------------------------|---------|-----|--------------------------------------------------------------------------------------------------------------------------------------------------|
| MCIP_CMD                             | 0x600   | R/W | Command register. Set commands and parameters for inter-processor communication with the ARConnect component.                                    |
| MCIP_WDATA                           | 0x601   | R/W | Written data register. Set data to be written into the ARConnect component corresponding to the command set in the MCIP_CMD register             |
| MCIP_READBACK                        | 0x602   | R   | Data read back register from the ARConnect component according to the read commands set in the MCIP_CMD register                                 |
| DEBUGP                               | 0x610   | N/A | Only present when PMU is configured on, and only can be accessed by the Debugger when core is powered down to control the power state of a core. |
| <b>Build Configuration Registers</b> |         |     |                                                                                                                                                  |
| MCIP_SYSTEM_BUILD                    | 0x0D0   | R   | Multi-core system configuration build register                                                                                                   |
| MCIP_SEMA_BUILD                      | 0x0D1   | R   | Inter-core semaphore unit configuration build register                                                                                           |
| MCIP_MESSAGE_BUILD                   | 0x0D2   | R   | Inter-core message unit passing configuration build register                                                                                     |
| MCIP_PMU_BUILD                       | 0x0D3   | R   | Power Management Unit configuration build register.                                                                                              |

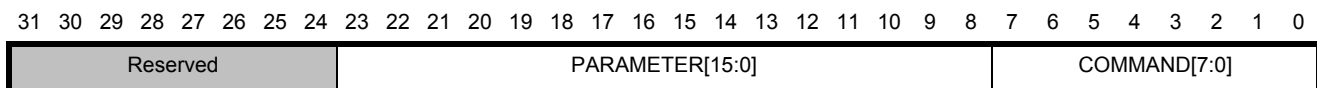
If processor core is configured with ARConnect, each core will contain a group of ARConnect auxiliary registers defined in [Table 13-6](#).

ARConnect components contain some internal programmable registers which cannot be programmed by LR/SR instructions directly but can be accessed by using ARConnect commands indirectly. All the ARConnect internal programmable registers are reset to zeros by default.

### 13.3.1.1 MCIP\_CMD

The MCIP\_CMD register is used to send commands to the ARConnect IP. For example, ARC core can write specific command to MCIP\_CMD register through SR instruction and then the core pass this command to ARConnect (to generate interrupts to other cores, or pass messages to other cores, etc.). ARC core can also read MCIP\_CMD register using an LR instruction to check the content of last written command. [Figure 13-5](#) shows the layout of the MCIP\_CMD register.

**Figure 13-5 The MCIP\_CMD Register**



[Table 13-7](#) lists the field descriptions of the MCIP\_CMD register.

**Table 13-7 Field Descriptions of the MCIP\_CMD Register**

| Field   | Bits  | Description                                                                                                                  |
|---------|-------|------------------------------------------------------------------------------------------------------------------------------|
| COMMAND | [7:0] | Command to ARConnect components. The list of commands is described in the <a href="#">Table 13-14 ARConnect Command List</a> |

**Table 13-7 Field Descriptions of the MCIP\_CMD Register**

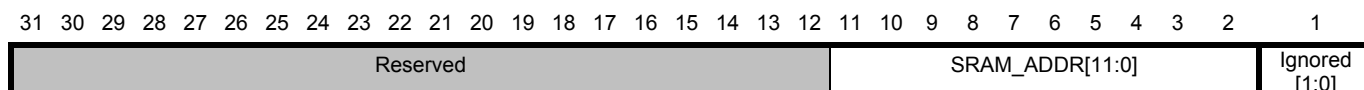
| Field     | Bits   | Description                                                                                                          |
|-----------|--------|----------------------------------------------------------------------------------------------------------------------|
| PARAMETER | [23:8] | Parameter of command to ARConnect components. Any parameter that is required by a command is supplied in this field. |

**13.3.1.2 MCIP\_WDATA**

The MCIP\_WDATA register is used to store data to be passed from the core to the ARConnect with various ARConnect commands.

The layout of the MCIP\_WDATA register depends on the particular command.

For example, the data to be written into the MCIP\_WDATA register before the CMD\_MSG\_SRAM\_SET\_ADDR command is shown in [Figure 13-6](#). The address of the message SRAM is set and then passed to the Inter-core Message Unit for subsequent read/write SRAM operations.

**Figure 13-6 MCIP\_WDATA Register Format Example for CMD\_MSG\_SRAM\_SET\_ADDR Command**

### 13.3.1.3 MCIP\_READBACK

The MCIP\_READBACK register returns read data subsequent to all ARConnect related read commands. The layout of the MCIP\_READBACK register depends on the particular inter-core communication command. This register is read-only; writes to it are ignored and raise an illegal-instruction exception.

For example, the MCIP\_READBACK register returns the interrupt status of the inter-core interrupt after the interrupt initiator issued CMD\_INTRPT\_READ\_STATUS command

**Figure 13-7 MCIP\_READBACK Register Format for CMD\_INTRPT\_READ\_STATUS Command**

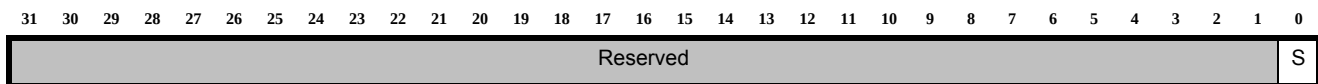


Table 13-8 lists the field descriptions of the MCIP\_READBACK register for the CMD\_INTRPT\_READ\_STATUS Command.

**Table 13-8 MCIP\_READBACK Field Descriptions for CMD\_INTRPT\_READ\_STATUS Command**

| Field | Bits | Description                                                                                                                                                                                                                                                                                                                        |
|-------|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S     | [0]  | Interrupt Status of the specified Inter-core Interrupt<br>Indicates that the interrupted core has serviced the inter-core interrupt. <ul style="list-style-type: none"> <li>0 = there is no outstanding interrupt or a former interrupt has been serviced (acknowledged)</li> <li>1 = there is an outstanding interrupt</li> </ul> |

### 13.3.1.4 DEBUGP Register

The DEBUGP register only present when PMU is configured on. The Debugger can set the PUT bit of the DEBUGP register to power up the core. Other bits of DEBUGP register record the power status of core for the information to Debugger. If it is not accessed by the Debugger, the Illegal Instruction exception is raised. If the Debugger accesses this register while the core is powered up, the access is ignored.

**Figure 13-8 DEBUGP Register**

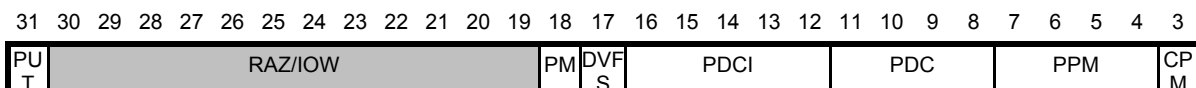


**Table 13-9 DEBUGP Register Field Descriptions**

| Field | Bits  | Description                                                                                            |
|-------|-------|--------------------------------------------------------------------------------------------------------|
| CPM   | [3:0] | Current power mode of the core. Read-only.<br>4'h0: Power on mode<br>4'h1: PM1 mode,<br>4'h2: PM2 mode |

**Table 13-9    DEBUGP Register Field Descriptions**

| Field | Bits    | Description                                                                                                                              |
|-------|---------|------------------------------------------------------------------------------------------------------------------------------------------|
| PPM   | [7:4]   | Previous power mode of the core. Read-only.<br>4'h0: Power on mode,<br>4'h1: PM1 mode,<br>4'h2: PM2 mode                                 |
| PDC   | [11:8]  | Power down cause. Read-only.<br>4'h0: SLEEP instruction<br>4'h1: PMU status register set via core                                        |
| PDCI  | [16:12] | CORE ID that causes power down. Read-only.                                                                                               |
| PM    | [17]    | Whether Power Management feature is supported or not for PMU. Read-only.<br>0x0 = No power management,<br>0x1 = Power management present |
| DVFS  | [18]    | Whether DVFS feature is supported for PMU or not. Read-only.<br>0x0 = No DVFS,<br>0x1 = DVFS present                                     |
| PUT   | [31]    | Power up trigger bit. Write-only.<br>Write 1 into this bit to trigger the power up request to PMU. Always read as zero.                  |

**13.3.1.5    MCIP\_SYSTEM\_BUILD**

MCIP\_SYSTEM\_BUILD is an optional auxiliary register in the ARC core family. If the ARC processor is not configured with ARConnect IP, reading this register returns zero.

Figure 13-9 shows the layout of the MCIP\_SYSTEM\_BUILD register.

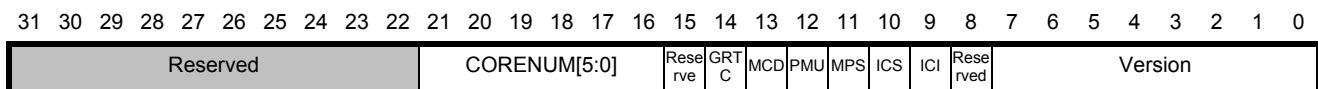
**Figure 13-9    The MCIP\_SYSTEM\_BUILD Register**

Table 13-10 lists the field description of the MCIP\_SYSTEM\_BUILD Register.

**Table 13-10    Field Descriptions of the MCIP\_SYSTEM\_BUILD Register**

| Field                      | Bits  | Description                                                                                                                                       |
|----------------------------|-------|---------------------------------------------------------------------------------------------------------------------------------------------------|
| Version                    | [7:0] | The ARConnect version. Currently 0x02                                                                                                             |
| Inter-Core Interrupt (ICI) | [9]   | <ul style="list-style-type: none"> <li>0 = No inter-core interrupt-unit supported</li> <li>1 = Inter-core interrupt-unit are supported</li> </ul> |
| Inter-Core Semaphore (ICS) | [10]  | <ul style="list-style-type: none"> <li>0 = No inter-core semaphore unit supported</li> <li>1 = Inter-core semaphore unit are supported</li> </ul> |



**Table 13-10 Field Descriptions of the MCIP\_SYSTEM\_BUILD Register**

| Field                       | Bits    | Description                                                                                                                               |
|-----------------------------|---------|-------------------------------------------------------------------------------------------------------------------------------------------|
| Message Passing SRAM (MPS)  | [11]    | <ul style="list-style-type: none"> <li>0 = No inter-core message unit supported</li> <li>1 = Inter-core message unit supported</li> </ul> |
| Power Management Unit (PMU) | [12]    | <ul style="list-style-type: none"> <li>0 = No power management unit supported</li> <li>1 = Power management unit is supported</li> </ul>  |
| Multi-Core Debug (MCD)      | [13]    | <ul style="list-style-type: none"> <li>0 = No multi-core debug unit supported</li> <li>1 = Multi-core debug unit is supported</li> </ul>  |
| Global RTC (GRTC)           | [14]    | <ul style="list-style-type: none"> <li>0 = No Global RTC</li> <li>1 = Global RTC is supported</li> </ul>                                  |
| CORE_NUMBER (CORENUM)       | [21:16] | <ul style="list-style-type: none"> <li>Number of cores connected to ARConnect. e.g., 6'h02: 2 cores; 6'h04: 4 cores</li> </ul>            |

**13.3.1.6 MCIP\_SEMA\_BUILD**

MCIP\_SEMA\_BUILD is an optional auxiliary register in the ARC EM processor family. If inter-core semaphores are not configured, reading the MCIP\_SEMA\_BUILD register returns zero.

Figure 13-10 shows the layout of the MCIP\_SEMA\_BUILD register.

**Figure 13-10 MCIP\_SEMA\_BUILD Register**

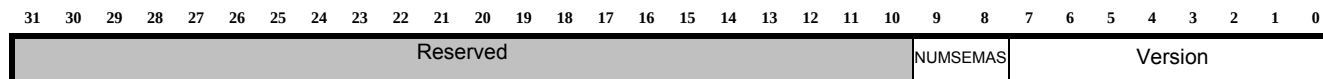


Table 13-11 lists the field descriptions of the MCIP\_SEMA\_BUILD register.

**Table 13-11 Field Description of the MCIP\_SEMA\_BUILD Register**

| Field    | Bits  | Description                                                                                                                                    |
|----------|-------|------------------------------------------------------------------------------------------------------------------------------------------------|
| Version  | [7:0] | Inter-Core semaphore unit version.                                                                                                             |
| NUMSEMAS | [9:8] | <ul style="list-style-type: none"> <li>2'b00 = 8 semaphores</li> <li>2'b01 = 16 semaphores (default)</li> <li>2'b10 = 32 semaphores</li> </ul> |

### 13.3.1.7 MCIP\_MESSAGE\_BUILD

MCIP\_MESSAGE\_BUILD is an optional auxiliary register in ARC cores. If inter-core message passing is not configured reading the MCIP\_MESSAGE\_BUILD register returns zero.

Figure 13-11 shows the layout of the MCIP\_MESSAGE\_BUILD register.

**Figure 13-11 The MCIP\_MESSAGE\_BUILD Register**

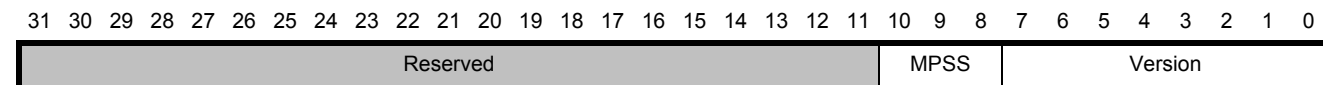


Table 13-12 lists the field description of the MCIP\_MESSAGE\_BUILD register.

**Table 13-12 Field Descriptions of the MCIP\_MESSAGE\_BUILD Register**

| Field                            | Bits   | Description                                                                                                                                                                                                                   |
|----------------------------------|--------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| Version                          | [7:0]  | Inter-Core message-passing unit version.                                                                                                                                                                                      |
| Message Passing SRAM Size (MPSS) | [10:8] | <ul style="list-style-type: none"> <li>3'b000 = 128 Bytes;</li> <li>3'b001 = 256 Bytes;</li> <li>3'b010 = 512 Bytes (default);</li> <li>3'b011 = 1K Bytes;</li> <li>3'b100 = 2K Bytes;</li> <li>3'b101 = 4K Bytes;</li> </ul> |

### 13.3.1.8 MCIP\_PMU\_BUILD

MCIP\_PMU\_BUILD is an optional auxiliary register in ARC core. If the Power Management Unit is not supported in the system reading the MCIP\_PMU\_BUILD register returns zero.

Figure 13-12 shows the layout of the MCIP\_PMU\_BUILD register.

**Figure 13-12 The MCIP\_PMU\_BUILD Register**

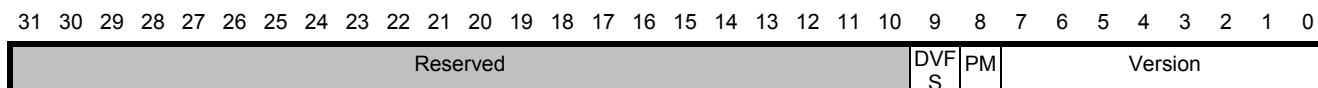


Table 13-13 lists the field description of the MCIP\_PMU\_BUILD register.

**Table 13-13 Field Descriptions of the MCIP\_PMU\_BUILD Register**

| Field                                    | Bits  | Description                                                           |
|------------------------------------------|-------|-----------------------------------------------------------------------|
| Version                                  | [7:0] | PMU version. Currently set to 0x01                                    |
| Power Management (PM)                    | [8]   | 0x0 = No power management<br>0x1 = Power management present (default) |
| Dynamic Voltage Frequency Scaling (DVFS) | [9]   | 0x0 = No DVFS<br>0x1 = DVFS present (default)                         |

### 13.3.2 ARConnect Command List

The full list of ARConnect commands in Table 13-4. The details of each command can be referred in Appendix E: [ARConnect Commands](#) for more details.

**Table 13-14 ARConnect Command List**

| Command | Name                    | Description                                                                                    | Component                 |
|---------|-------------------------|------------------------------------------------------------------------------------------------|---------------------------|
| 0x00    | CMD_CHECK_CORE_ID       | Read the CORE ID of this core                                                                  | N/A                       |
| 0x01    | CMD_INTRPT_GENERATE_IRQ | Generate interrupts from one core to other cores in the system.                                | Inter-core Interrupt Unit |
| 0x02    | CMD_INTRPT_GENERATE_ACK | Generate interrupt acknowledge from this interrupt receiver core to interrupt initiator core.  |                           |
| 0x03    | CMD_INTRPT_READ_STATUS  | Read the status of an interrupt sent from interrupt initiator core to interrupt receiver core. |                           |
| 0x04    | CMD_INTRPT_CHECK_SOURCE | Read the source of inter-core interrupt set by other cores in the system.                      |                           |

**Table 13-14 ARConnect Command List**

| Command | Name                    | Description                                                                               | Component                 |
|---------|-------------------------|-------------------------------------------------------------------------------------------|---------------------------|
| 0x11    | CMD_SEMA_CLAIM_AND_READ | Claim the ownership of a semaphore and then check its status.                             | Inter-core Semaphore Unit |
| 0x12    | CMD_SEMA_RELEASE        | Release the ownership of a semaphore and make it available for other cores in the system. |                           |
| 0x13    | CMD_SEMA_FORCE_RELEASE  | Force release of a semaphore that is owned by another core and make it available.         |                           |

**Table 13-14 ARConnect Command List**

| Command | Name                          | Description                                                                                                                                                                                   | Component               |
|---------|-------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------------|
| 0x21    | CMD_MSG_SRAM_SET_ADDR         | Set up a SRAM address for subsequent message passing SRAM read/write operations.                                                                                                              | Inter-core Message Unit |
| 0x22    | CMD_MSG_SRAM_READ_ADDR        | Read the last updated internal SRAM address register of this core.                                                                                                                            |                         |
| 0x23    | CMD_MSG_SRAM_SET_ADDR_OFFSET  | Set the address offset value for subsequent message passing SRAM read/write operations in incremental mode.                                                                                   |                         |
| 0x24    | CMD_MSG_SRAM_READ_ADDR_OFFSET | Read the address offset value of this core set by a previous CMD_MSG_SRAM_READ_ADDR_OFFSET command.                                                                                           |                         |
| 0x25    | CMD_MSG_SRAM_WRITE            | Write data into message passing SRAM. The SRAM address is set by the CMD_MSG_SRAM_SET_ADDR command.                                                                                           |                         |
| 0x26    | CMD_MSG_SRAM_WRITE_INC        | Write data into message passing SRAM. The SRAM address is incremented and updated according to the current SRAM address and the offset value set by the CMD_MSG_SRAM_SET_ADDR_OFFSET command. |                         |
| 0x27    | CMD_MSG_SRAM_WRITE_IMM        | Write data into message passing SRAM. The address of SRAM is specified by Parameter [11:0] field accompanied with the command.                                                                |                         |
| 0x28    | CMD_MSG_SRAM_READ             | Read data from message passing SRAM. The SRAM address is set by the CMD_MSG_SRAM_SET_ADDR command.                                                                                            |                         |
| 0x29    | CMD_MSG_SRAM_READ_INC         | Read data from message passing SRAM. The SRAM address is incremented and updated according to the current SRAM address and the offset value set by the CMD_MSG_SRAM_SET_ADDR_OFFSET command.  |                         |
| 0x2A    | CMD_MSG_SRAM_READ_IMM         | Read data from message passing SRAM. The SRAM address is specified by the Parameter [11:0] field.                                                                                             |                         |
| 0x31    | CMD_DEBUG_RESET               | Generate reset to the core.                                                                                                                                                                   |                         |
| 0x32    | CMD_DEBUG_HALT                | Selective halt any combination of core(s).                                                                                                                                                    |                         |

**Table 13-14 ARConnect Command List**

| Command | Name                  | Description                                                                                                                                                                                                      | Component                   |
|---------|-----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-----------------------------|
| 0x33    | CMD_DEBUG_RUN         | Selective run any combination of core(s).                                                                                                                                                                        | Multi-Core debug Unit       |
| 0x34    | CMD_DEBUG_SET_MASK    | Set the MASK register in the multi-core debug unit to shield some halt events from specified core(s).                                                                                                            |                             |
| 0x35    | CMD_DEBUG_READ_MASK   | Read the internal MASK register value in the multi-core debug unit.                                                                                                                                              |                             |
| 0x36    | CMD_DEBUG_SET_SELECT  | Set internal SELECT register in the multi-core debug unit to trigger a conditional halt to specified core(s).                                                                                                    |                             |
| 0x37    | CMD_DEBUG_READ_SELECT | Read the internal SELECT register value in the multi-core debug unit.                                                                                                                                            |                             |
| 0x38    | CMD_DEBUG_READ_EN     | Read the EN flags of all cores.                                                                                                                                                                                  |                             |
| 0x39    | CMD_DEBUG_READ_CMD    | Read the last-written internal MCD_CMD register in the multi-core debug unit.                                                                                                                                    |                             |
| 0x3A    | CMD_DEBUG_READ_CORE   | Read the last-written internal MCD_CORE register in the multi-core debug unit. The MCD_CORE register is updated t by the value of MCIP_WDATA register ahead of some multi-core debug related read/write commands |                             |
| 0x41    | CMD_GRTC_CLEAR        | Clear the value of global run-time counter.                                                                                                                                                                      | Global RTC Unit (GRTC)      |
| 0x42    | CMD_GRTC_READ_LO      | Read the low 32-bit snapshot value of Global RTC                                                                                                                                                                 |                             |
| 0x43    | CMD_GRTC_READ_HI      | Read the high 32-bit value of global RTC                                                                                                                                                                         |                             |
| 0x51    | CMD_PMU_SET_PMODE     | Set the power mode for the specified core(s)                                                                                                                                                                     | Power Management Unit (PMU) |
| 0x52    | CMD_PMU_READ_PSTATUS  | Read the power mode status register of the specified core                                                                                                                                                        |                             |
| 0x53    | CMD_PMU_SET_PCNT      | Set power up duration register                                                                                                                                                                                   |                             |
| 0x54    | CMD_PMU_READ_PCNT     | Read power up duration register                                                                                                                                                                                  |                             |
| 0x55    | CMD_PMU_SET_DVFS      | Set DVFS performance level for the whole multi-core system.                                                                                                                                                      |                             |
| 0x56    | CMD_PMU_READ_DVFS     | Read the DVFS performance level for the whole multi-core system.                                                                                                                                                 |                             |

**Note**

The commands sent to ARConnect are ignored under the following conditions, and no illegal command or error information is generated from the hardware.

- If one ARConnect component unit (such as an inter-core interrupt unit) is not configured during build-time, the commands related to this unit will be ignored.
- If undefined commands are sent to the ARConnect component, these undefined commands will be ignored.
- For commands that do not require a parameter, it is recommended to set the parameter field to 0; for commands that require a parameter, it is recommended to set the value of parameter in the valid range. In both cases, if the parameter is set beyond the scope or with an invalid value, the hardware result is unpredictable.

Software must guarantee the correct usage of commands for ARConnect under different configurations.

### 13.3.3 ARConnect Command Usage Examples

This section introduces the examples of instruction sequences utilizing ARConnect commands for different applications.

#### 13.3.3.1 Inter-core Interrupt Examples - Scenario 1

Core1 generates inter-core interrupt to Core2. Core2 receives and serves the interrupt, and then generates interrupt acknowledgement to Core1.

In this case, Core1 is the interrupt initiator core. To avoid overriding the previous interrupt, it is recommended to check the interrupt status before Core1 generates a new interrupt to Core2. A segment program is shown in [Example 13-1](#).

#### Example 13-1 Inter-Core Interrupt Example Scenario 1 - Core1

|                                                               |                                                                                                           |
|---------------------------------------------------------------|-----------------------------------------------------------------------------------------------------------|
| <code>chk status:</code>                                      | Check the interrupt status of Core2, so that previous interrupt is not overridden.                        |
| <code>SR 0x00000203, [MCIP_CMD]</code>                        | Sends <code>CMD_INTRPT_READ_STATUS</code> command to check the status of interrupt it generated to Core2. |
| <code>LR r1, [MCIP_READBACK]</code><br><code>CMP r1, 1</code> | The interrupt status is read back from <code>MCIP_READBACK</code> command.                                |
| <code>BNEQ chk_status</code><br>...                           | Exit the loop until the corresponding status bit is cleared.                                              |
| <code>SR 0x00000201, [MCIP_CMD]</code>                        | Sends <code>CMD_INTRPT_GENERATE_IRQ</code> command to Core2 to generate inter-core interrupt.             |

Core2 is the interrupt receiver core, a segment program of its interrupt handler is shown in [Example 13-2](#)

**Example 13-2 Inter-Core Interrupt Example Scenario 1 - Core2**

|                                              |                                                                                                                                                                                                                                                                                                              |
|----------------------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SR r2, [MCIP_CMD]                            | The content of r2 is 0x00000004, indicates sending CMD_INTRPT_CHECK_SOURCE command to check the source of interrupt generated by which core(s).                                                                                                                                                              |
| LR r4, [MCIP_READBACK]                       | The vector value is read back from MCIP_READBACK register to core register of r4.                                                                                                                                                                                                                            |
| AND.F 0, r4, 0x00000002<br>BEQ gen_ack_to_c1 | Check whether the bit corresponding to Core1 is asserted                                                                                                                                                                                                                                                     |
| gen_ack_to_c1:<br>SR r5, [MCIP_CMD]          | The content of r5 is 0x00000102, indicates sending CMD_INTRPT_GENERATE_ACK to Core1.<br>Note: Make sure the interrupt has been de-asserted before exit the interrupt handler, for example, by using CMD_INTRPT_CHECK_SOURCE command again in a loop until the initiator core have de-asserted the interrupt. |

**13.3.3.2 Inter-core Interrupt Examples - Scenario2**

Core0, Core1 and Core2 generate the inter-core interrupt to Core3. Core3 serves the interrupt and generates interrupt acknowledgment to Core0, Core1 and Core2, respectively.

In this case, Core0, Core1 and Core2 are the interrupt initiators. The segment program of each core can be the same as scenario1. [Example 13-3](#) shows this scenario.

**Example 13-3 Inter-Core Interrupt Example Scenario 2 - Core0, Core1, Core2**

|                                                |                                                                                    |
|------------------------------------------------|------------------------------------------------------------------------------------|
| chk_status:                                    | Check the interrupt status of Core3, so that previous interrupt is not overridden. |
| SR 0x00000303, [MCIP_CMD]                      | Sends CMD_INTRPT_READ_STATUS command.                                              |
| LR r1, [MCIP_READBACK]<br>CMP.F r1, 0x00000001 | The interrupt status is read back from MCIP_READBACK .                             |
| BEQ chk_status                                 | Exit the loop until the corresponding status bit is cleared.                       |
| SR 0x00000301, [MCIP_CMD]                      | Sends CMD_INTRPT_GENERATE_IRQ command to Core3 to generate inter-core interrupt.   |
| ...                                            |                                                                                    |

Core3 is the interrupt receiver core, a segment program of its interrupt handler is shown in [Example 13-4](#).



**Example 13-4 Inter-Core Interrupt Related Commands Scenario 2 - Core3**

serve\_irq:

```
SR r0, [MCIP_CMD]
```

The content of r0 is 0x00000004, indicates sending CMD\_INTRPT\_CHECK\_SOURCE command to check the source of interrupt generated by which core(s).

```
LR r1, [MCIP_READBACK]
```

The vector value is read back from MCIP\_READBACK register to core register of r1.

```
CMP r1, 0
```

Check whether all interrupts have been served.

```
BEQ end_serve_irq
```

```
...
```

```
MOV r2, 1
```

Set the start value 1 for index control.

serve\_irq\_for\_each\_core:

```
MOV r3, r1
```

Move the interrupt source vector to a temp register r3.

```
SUB r10, r2, 1
```

Generate the shift\_loop control value.

shift\_loop:

```
CMP r10, 0
```

```
BEQ gen_ack_to_each_core
```

If equal, go to generate interrupt acknowledges.

```
LSR r3, r3
```

Shift right operation of interrupt source vector.

```
SUB r10, r10, 1
```

```
BAL shift_loop
```

gen\_ack\_to\_each\_core:

```
AND.F 0, r3, 0x00000001
```

Check the value of bit0 for shifted interrupt source vector.

```
BEQ incr_index
```

If the AND result is zero, do nothing and increment the index; otherwise, go on, generate interrupt, and acknowledge to the core with specific index value.

```
...
```

```
SR r11, [MCIP_CMD]
```

```
...
```

Instructions to generate command value and store into r11. In this case, the content of r11 can be one of "0x00000002/0x00000102/0x00000202" to send CMD\_INTRPT\_GENERATE\_ACK to Core0, Core1 and Core2 respectively.

incr\_index:

```
ADD r2, r2, 1
```

```
CMP r2, MCIP_CORE_NUMBER
```

```
BLS serve_irq_for_each_core
```

If the index value is less than or equal to number of cores in the system, go on to check the interrupt status.

```
end_serve_irq:
```

```
RTIE
```

Note: Make sure the corresponding bit of interrupt source has been cleared before exit the handler.

### 13.3.3.3 Inter-Core Semaphore Examples

#### 13.3.3.3.1 Scenario 1

In this scenario, the Core1 claims the ownership of semaphore8. After sending some commands, the Core1 releases the semaphore8.

##### Example 13-5 Inter-Core Semaphore Example 1 - Core1

```
claim:  SR 0x00000811,
[MCIP_CMD]
```

Core1 sends CMD\_SEMA\_CLAIM\_AND\_READ command to claim for the ownership of semaphore8.

```
LR r0, [MCIP_READBACK]
```

The claim result is read back from MCIP\_READBACK.

```
CMP r0, 1
```

Check whether the claim is successful.

```
BNEQ claim
```

If it is a failure, then go back to claim the semaphore8. Otherwise, continue with the claim.

```
...
...
```

```
SR 0x00000812, [MCIP_CMD]
```

Send CMD\_SEMA\_RELEASE command to release semaphore8 and allow it to be available for other cores

#### 13.3.3.3.2 Scenario 2

Core2 claims the ownership of semaphore8, but it has been occupied by other cores. Core2 compulsively releases semaphore8, and then claims the ownership of it.

##### Example 13-6 Inter-Core Semaphore Example 2 - Core1

```
claim:  SR 0x00000811, [MCIP_CMD]
```

Core2 sends CMD\_SEMA\_CLAIM\_AND\_READ command to claim the ownership of semaphore8.

```
LR r0, [MCIP_READBACK]
```

The claim result is read from MCIP\_READBACK to core register r0.

```
CMP r0, 1
```

Check whether the claim is successful.

```
BNEQ force_release
```

If it is a failure, then release the semaphore8. Otherwise, continue with the claim.

```
...
...
```

```
force_release:
```

```
SR 0x00000813, [MCIP_CMD]
```

Core2 sends CMD\_SEMA\_FORCE\_RELEASE command to compulsively release semaphore8 and allow it to be available.

BA claim

...

### 13.3.3.4 Inter-Core Message Passing Examples

#### 13.3.3.4.1 Scenario

Several messages are needed to be passed between Core1 and Core2. Core1 is the message producer and Core2 is the message consumer.

Messages are written into and read from message passing SRAM. Address 0x00 in the message passing SRAM is used to transfer communication flag between Core1 and Core2.

#### Example 13-7 Inter-Core Message Passing Example Scenario - Core1

|                             |                                                                                                                                                                                          |
|-----------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| SR 0x064, [MCIP_WDATA]      | Set the message passing SRAM address to be 0x064                                                                                                                                         |
| SR 0x00000021, [MCIP_CMD]   | Send CMD_MSG_SRAM_SET_ADDR command to ARConnect                                                                                                                                          |
| SR -4, [MCIP_WDATA]         | Set the SRAM address offset value to be -4 (a negative value).                                                                                                                           |
| SR 0x00000023, [MCIP_CMD]   | Send CMD_MSG_SRAM_SET_ADDR_OFFSET command.                                                                                                                                               |
| ...                         |                                                                                                                                                                                          |
| SR r1 [MCIP_WDATA]          | r1 contains the data to be written into message passing SRAM.                                                                                                                            |
| SR 0x00000026, [MCIP_CMD]   | Send CMD_MSG_SRAM_WRITE_INC command. The data in the MCIP_WDATA register is written into address 0x064. After that, the SRAM address increments to 0x060 (current address+offset value). |
| SR r2, [MCIP_WDATA]         | r2 contains the data to be written into the SRAM                                                                                                                                         |
| SR 0x00000026, [MCIP_CMD]   | Send CMD_MSG_SRAM_WRITE_INC command. The data in the MCIP_WDATA register is written into address 0x060. After that, the SRAM address will increment to 0x05c.                            |
| SR 0x55555555, [MCIP_WDATA] | After producing message, set address 0x00 to a particular value 0x55555555 to tell Core2 message is ready.                                                                               |
| SR 0x00000027, [MCIP_CMD]   | Send CMD_MSG_SRAM_WRITE_IMM command, the SRAM address is specified in parameter field (0x00).                                                                                            |

#### Example 13-8 Inter-Core Message Passing Example Scenario - Core2

...

Other tasks, then waiting messages passed from core1.

|                                                   |                                                                                                                                                                                                                                 |
|---------------------------------------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>wait_flag: SR 0x0000002a, [MCIP_CMD]</code> | Send CMD_MSG_SRAM_READ_IMM command                                                                                                                                                                                              |
| <code>LR r0, [MCIP_READBACK]</code>               | The data is read back from MCIP_READBACK                                                                                                                                                                                        |
| <code>CMP r0, 0x55555555</code>                   | Check whether Core1 has produced the message.                                                                                                                                                                                   |
| <code>BNEQ wait_flag</code>                       | If failed, go back to wait message.                                                                                                                                                                                             |
| <code>SR 0x050, [MCIP_WDATA]</code>               | Set the message passing SRAM address to be 0x050                                                                                                                                                                                |
| <code>SR 0x00000021, [MCIP_CMD]</code>            | Send CMD_MSG_SRAM_SET_ADDR command.                                                                                                                                                                                             |
| <code>SR 4, [MCIP_WDATA]</code>                   | Set the SRAM address offset value to be 4 (a positive value).                                                                                                                                                                   |
| <code>SR 0x00000023, [MCIP_CMD]</code>            | Send CMD_MSG_SRAM_SET_ADDR_OFFSET command                                                                                                                                                                                       |
| <code>SR 0x00000029, [MCIP_CMD]</code>            | Send CMD_MSG_SRAM_READ_INC command to read content in address 0x00 the data will be read out to MCIP_READBACK register from SRAM address 0x050. After that, the address will increment to 0x054 (current address+offset value). |
| <code>LR r1, [MCIP_READBACK]</code>               | Data read from address 0x050 is stored to core register r1.                                                                                                                                                                     |
| <code>SR 0x00000029, [MCIP_CMD]</code>            | Send CMD_MSG_SRAM_READ_INC command, The data will be read out to MCIP_READBACK register from address 0x054. After that, the SRAM address will increment to 0x058.                                                               |
| <code>LR r2, [MCIP_READBACK]</code>               | Data read from address 0x054 is stored to core register r2                                                                                                                                                                      |

## 13.3.4 Multi-Core Debug Examples

### 13.3.4.1 Conditionally Halt Cores Scenario

Core0 will occur a breakpoint halt during the execution. Debugger can program the multi-core debug mechanism to trigger a conditional halt to other cores when breakpoint halt occurring in Core0.

The below example shows a segment program of Core2 if the debugger programs the multi-core debug unit through Core2.

#### Example 13-9 Conditionally Halt Cores Scenario

|                                          |                                                                                                                                                                                         |
|------------------------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <code>SR 0x0000000e, [MCIP_WDATA]</code> | Core1, Core2 and Core3 are selected for following debug command.                                                                                                                        |
| <code>SR 0x00000036, [MCIP_CMD]</code>   | Send CMD_DEBUG_SET_SELECT command to Multi-core debug unit, the cores selected by previous SR instruction is halted by the global halt pulse.                                           |
| <code>SR 0x00000001, [MCIP_WDATA]</code> | Core1 is selected for following debug commands.                                                                                                                                         |
| <code>SR 0x00000434, [MCIP_CMD]</code>   | Send CMD_DEBUG_SET_MASK command to Multi-core debug unit, Parameter[7:0] is 4'b0100 to indicate the mask value of SH, BH, AH and H bits to control the generation of global halt pulse. |

### 13.3.4.2 Selectively Halt Cores Scenario

Core0 programs the multi-core debug unit to selectively halt other cores in the system.

#### Example 13-10 Selectively Halt Cores Scenario

|                             |                                                                                                                |
|-----------------------------|----------------------------------------------------------------------------------------------------------------|
| SR 0x0000000e, [MCIP_WDATA] | Core1, Core2, and Core3 are selected for following debug command.                                              |
| SR 0x00000032, [MCIP_CMD]   | Send CMD_DEBUG_HALT command to Multi-core debug unit, the cores selected by previous SR instruction is halted. |

### 13.3.4.3 Selectively Run Cores Scenario

All cores have already halted. The debugger programs the multi-core debug unit to run all cores in the system.

#### Example 13-11 Selectively Run Cores Scenario

|                             |                                                                                                                   |
|-----------------------------|-------------------------------------------------------------------------------------------------------------------|
| SR 0x0000000f, [MCIP_WDATA] | Core0 to Core3 are selected for following debug command.                                                          |
| SR 0x00000033, [MCIP_CMD]   | Send CMD_DEBUG_RUN command to multi-core debug unit, the cores selected by previous SR instruction starts to run. |

### 13.3.4.4 Selectively Reset Cores Scenario

Core0 is the master CPU of two core system. This is used for reprogramming the Core1.

#### Example 13-12 Selectively Reset Cores Scenario

wait\_core1\_halt:

|                                                     |                                                          |
|-----------------------------------------------------|----------------------------------------------------------|
| SR 0x00000038, [MCIP_CMD]<br>LR r0, [MCIP_READBACK] | Send CMD_DEBUG_READ_EN command to Multi-core debug unit. |
| AND.F 0, r0, 0x00000002<br>BNZ wait_core1_halt      | Check EN bit of Core1.                                   |
|                                                     | Re-allocate the program of Core1.                        |
| SR 0x00000002, [MCIP_WDATA]                         | Core1 is selected for following debug command.           |
| SR 0x00000031, [MCIP_CMD]                           | Send CMD_DEBUG_RESET command to reset Core1.             |

## 13.3.5 Power Management Related Examples

### 13.3.5.1 Scenario 1

Core0 enters PM1 mode by SLEEP instruction, and then is powered up by its Timer0 interrupt.

Prior to powering down the PD1 (Power Domain 1) of the Core0, all required architectural status should be saved in external memories in order to resume program execution after it exits from the PM1 mode. But PM1 mode doesn't shut off the power of CCMs, there is no need to save data in CCMs of Core0, if exist.

When Core0 is powered up and exited from PM1 mode, it will execute the program started from the reset handler. And it needs to restore the necessary information saved before.

**Note**

If the core is powered down with PM2 mode, CCMs will lost contents after powered down, so the contents of CCMs should be also saved to external memories before powering down, and restored back after powering up.

**Example 13-13 Enters PM1 Mode by SLEEP Instruction and Exits by Timer0 Interrupt Scenario - Core0**

|                                                              |                                                                                                              |
|--------------------------------------------------------------|--------------------------------------------------------------------------------------------------------------|
| <code>_reset:</code>                                         | Reset exception vector base                                                                                  |
| <code>.long _start</code>                                    |                                                                                                              |
| <code>...</code>                                             |                                                                                                              |
| <code>irq16: .long timer0_handler</code>                     | Timer0 (irq16) interrupt vector base                                                                         |
| <code>...</code>                                             |                                                                                                              |
| <code>test_start:</code>                                     |                                                                                                              |
| <code>.text</code>                                           |                                                                                                              |
| <code>.global _start</code>                                  |                                                                                                              |
| <code>_start:</code>                                         | On-reset start point of the program                                                                          |
| <code>...</code>                                             |                                                                                                              |
| <code>SR 0x00000052, [MCIP_CMD]</code>                       | Send CMD_PMU_READ_PSTATUS command to check the power status of Core0                                         |
| <code>LR r0, [MCIP_READBACK]</code>                          | The PSTATUS value is read back from MCIP_READBACK register to core register of r0                            |
| <code>AND.F 0, r0, 0x000000f0</code>                         | Check previous power mode of Core0.                                                                          |
| <code>BNZ restore_status</code>                              | If its previous status is one of power down mode, branch to <i>restore_status</i> for data restoring process |
| <code>...</code>                                             |                                                                                                              |
| <code>SR 0x00000038, [MCIP_WDATA]</code>                     | Set the value of power-up duration counter                                                                   |
| <code>SR 0x00000053, [MCIP_CMD]</code>                       | Send CMD_PMU_SET_PCNT command to PMU                                                                         |
| <code>...</code>                                             |                                                                                                              |
| <code>CLRI 0</code>                                          | Configure Time0 relative interrupt settings                                                                  |
| <code>SR 0X00000010, [IR_SELECT]</code>                      |                                                                                                              |
| <code>SR 0X00000000, [IRQ_TRIGGER]</code>                    |                                                                                                              |
| <code>SR 0X00000000,</code><br><code>[IRQ_PRIORITY]</code>   |                                                                                                              |
| <code>FLAG 0x0000001e</code>                                 |                                                                                                              |
| <code>SR 0X00000001, [IRQ_ENABLE]</code>                     |                                                                                                              |
| <code>SETI 0</code>                                          | Power down preparation- save necessary architectural status of processor, such as STATUS32 register          |
| <code>...</code>                                             |                                                                                                              |
| <code>...</code>                                             |                                                                                                              |
| <code>SR 0X00000000,</code><br><code>[TIMER0_CONTROL]</code> | Configure Timer0 auxiliary registers                                                                         |

```
SR 0X00004000,
[TIMER0_LIMIT]

SR 0X00000011,
[TIMER0_CONTROL]

SR 0X00000000,
[TIMER0_COUNT]
...
```

```
SLEEP 0xd8
...
```

Core0 enters PM1 mode by SLEEP instruction

timer0\_handler:

```
SR 0X00000000,
[TIMER0_CONTROL]
RTIE
```

Timer0 (irq16) interrupt handler

### 13.3.5.2 Scenario 2

In a multi-core system, Core0 uses CMD\_PMU\_SET\_DVFS command to regulate the performance level of whole system according to application workloads.

#### Example 16-20 Dynamic Frequency and Voltage Scaling Scenario - Core0

```
_reset:
.long _start

...
...
```

Reset exception vector base

```
test_start:
.text
.global_start
```

```
_start:
...
```

On-reset start point of the program

```
SR 0x00000055, [MCIP_CMD]
...
```

Send CMD\_PMU\_SET\_DVFS command to set system performance level to Level0

### 13.3.6 ARConnect Configurability

Each component in ARConnect IP can be configured individually.



Table 13-15 lists the configuration options supported in ARConnect.

**Table 13-15 Configurations of ARConnect IP**

| Components                | Configuration Option | Value Range                     | Description                                                                                                                                      |
|---------------------------|----------------------|---------------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------|
| Inter-core Interrupt Unit | -mcip_has_intrpt     | 0,1                             | <ul style="list-style-type: none"> <li>0 = No inter-core interrupt</li> <li>1 = Has inter-core interrupt</li> </ul>                              |
| Inter-core Semaphore Unit | -mcip_has_sema       | 0,1                             | <ul style="list-style-type: none"> <li>0 = No inter-core semaphore</li> <li>1 = Has inter-core semaphore</li> </ul>                              |
| Inter-core Semaphore Unit | -mcip_sema_num       | 8,16,32                         | Number of semaphores for inter-core semaphore unit                                                                                               |
| Inter-core Message Unit   | -mcip_has_msg_sram   | 0,1                             | <ul style="list-style-type: none"> <li>0 = No message unit</li> <li>1 = Has message unit</li> </ul>                                              |
| Inter-core Message Unit   | -mcip_msg_sram_size  | 128, 256, 512, 1024, 2048, 4096 | Bytes of the message SRAM                                                                                                                        |
| Multi-core Debug Unit     | -mcip_has_debug      | 0,1                             | <ul style="list-style-type: none"> <li>0 = No multi-core debug assist</li> <li>1 = Has multi-core debug assist</li> </ul>                        |
| Global RTC Unit           | -mcip_has_grtc       | 0,1                             | <ul style="list-style-type: none"> <li>0 = No global RTC unit</li> <li>1 = Has global RTC unit</li> </ul>                                        |
| Power Management Unit     | -mcip_has_pmu        | 0,1                             | <ul style="list-style-type: none"> <li>0 = No power management unit</li> <li>1 = Has power management unit</li> </ul>                            |
| Power Management Unit     | -mcip_pmu_has_pm     | 0,1                             | <ul style="list-style-type: none"> <li>0 = PMU does not support power management unit</li> <li>1 = PMU supports power management unit</li> </ul> |
| Power Management Unit     | -mcip_pmu_has_dvfs   | 0,1                             | <ul style="list-style-type: none"> <li>PMU does not support DVFS functionality</li> <li>PMU supports DVFS functionality</li> </ul>               |



# 14

## Debug Features

This chapter describes the debug features of the ARC EM processor series, which are available with the ARC EM IP library. These features include the following:

- JTAG port for accessing core resources from an external host
- Actionpoints — a sophisticated multi-event breakpoint-watchpoint system
- SmarT — Small Real-Time Trace Buffer
- RTT — Real Time Trace

### 14.1 Host Access via JTAG

This section describes how JTAG is used to control the processor and access various resources:

- Processor-internal resources:
  - Core registers
  - Auxiliary registers
  - Closely coupled memories
  - Caches
- Processor-visible external resources:
  - System memory
  - Peripherals

For generic interface information such as JTAG signal descriptions and a summary of electrical behavior, see [Chapter 9, “External Interfaces”](#).

The JTAG module contains logic for communicating with the ARC EM processor and its memory system. The host interacts with this module via high-level commands that specify transaction parameters. For details on how to access specific resources or how host-debug functions are implemented, see the section on the host in the *ARCV2 ISA Programmer's Reference*.

The module supports the IEEE 1149.1 four-wire JTAG interface and the two-wire IEEE 1149.7 Compact JTAG interface (see [“Supported Standards”](#) on page 47). The Compact JTAG (1149.7) IP is an adapter you can add as a separate component in front of one or more 1149.1 JTAG ports of the processor.

For more information, see the following:

- [“Four-Wire JTAG Interface \(IEEE 1149.1\)”](#)
- [“Two-Wire JTAG Interface \(IEEE 1149.7\)”](#) on page 227

The two-wire interface is an adapter outside the existing JTAG interfaces; adding it does not change the programming model of any underlying 1149.1 JTAG interfaces. However, adding the two-wire interface does add another layer of Compact JTAG (1149.7) registers accessible only through special 1149.7 control protocols (described in [“Two-Wire JTAG Interface \(IEEE 1149.7\)”](#) on page 227).

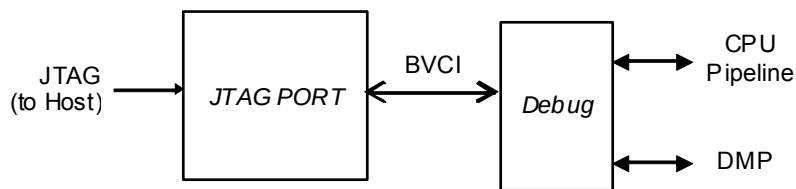
## 14.2 Four-Wire JTAG Interface (IEEE 1149.1)

### 14.2.1 JTAG 1149.1 Port

The block diagram shown in [Figure 14-1](#) shows how the JTAG communications port integrates within an ARC EM system. The port is linked to the host interface of the ARC EM processor and to system memory via the data-memory pipeline.

The JTAG port is also known as *TAP* (Test Access Port).

**Figure 14-1 A JTAG 1149.1 Port with an ARC EM Processor**



### 14.2.2 JTAG 1149.1 Programmer's Model

The JTAG module includes the internal registers as shown in [Table 14-1](#). The host can perform a read or write transaction to a memory location or an ARC EM register using these internal registers. Six of the seven registers are collectively referred to as data registers (IEEE STD 1149.1). The remaining register is the instruction register, which plays a central role in accessing all debug-interface registers.

**Table 14-1 JTAG 1149.1 Registers**

| Value | Code | JTAG Register                | Type        |
|-------|------|------------------------------|-------------|
| N/A   | N/A  | INSTRUCTION REGISTER         | Instruction |
| 0x8   | 1000 | JTAG STATUS REGISTER         | Data        |
| 0x9   | 1001 | TRANSACTION COMMAND REGISTER | Data        |

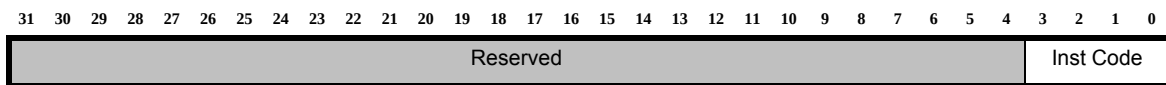
**Table 14-1 JTAG 1149.1 Registers**

| Value | Code | JTAG Register    | Type |
|-------|------|------------------|------|
| 0xA   | 1010 | ADDRESS REGISTER | Data |
| 0xB   | 1011 | DATA REGISTER    | Data |
| 0xC   | 1100 | IDCODE REGISTER  | Data |
| 0xF   | 1111 | BYPASS REGISTER  | Data |

The registers are described in the following sections.

#### 14.2.2.1 The JTAG 1149.1 Instruction Register

The instruction register is used to access all data registers. Each data register is addressed by a unique 4-bit instruction code.



The correct code must be written into the instruction register to access the required data register. The instruction shift register is loaded with 0x1 in capture-IR, so that external circuitry, by looking for a transition when in shift-IR, can detect a stuck-at fault in the JTAG chain.



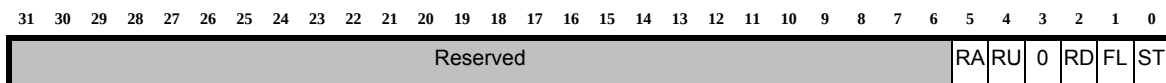
#### Note

Because of this behavior, which is specified in IEEE 1149.1, the current contents of the instruction register cannot be read by the external circuitry. The register itself is initialized to point to the IDCODE register in test-logic-reset.

In addition to accessing the data registers (which are documented in sections [14.2.2.2](#) to [14.2.2.7](#)), the instruction register is also used to select a test sequence to apply to the device.

#### 14.2.2.2 The JTAG 1149.1 Status Register (Instruction Code 0x8)

The JTAG status register is read-only and is used by the host device to obtain important information on the state of the JTAG module or the result of debug operations. The bits of the register are assigned as follows.



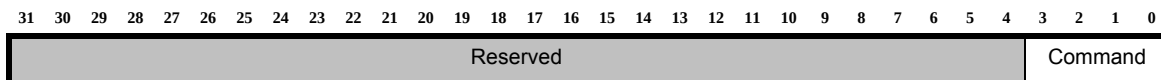
Each field in the JTAG status register reflects the following information:

- Bit 0 - The stalled (ST) flag indicates that the current transaction is still in progress.
- Bit 1 - When true, the failure (FL) flag indicates that a read (or write) has failed. For example, this flag is set if an access to an unimplemented core occurs. The failure flag is cleared automatically when a new transaction starts.
- Bit 2 - The ready (RD) flag indicates whether the JTAG module is available to accept another transaction command. This flag is set when a transaction has just completed or when the JTAG module is idle.

- Bit 3 - Always zero.
- Bit 4 - The run (RU) flag is set to one when the ARC EM processor is running. This bit is cleared to zero when the processor is halted. This flag is the inverse of the halt bit in the ARC EM STATUS32 register.
- Bit 5 - The Reset Applied (RA) flag is used by the debug host to determine that a target reset has occurred. The RA flag is set to 1 on a hard reset of the CPU and can be written only by the host.
- Bits 31 to 6 - Reserved.

### 14.2.2.3 The JTAG 1149.1 Transaction-Command Register (Instruction Code 0x9)

The transaction-command register is used to specify the communication transaction to be performed.



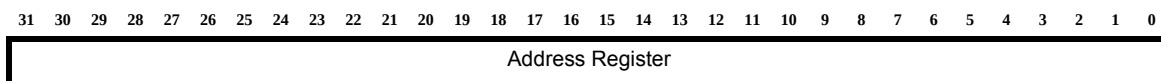
The JTAG module supports several different accesses or transactions, which are shown in [Table 14-2](#) with their associated encodings.

**Table 14-2 JTAG Read and Write Transactions**

| Value | Code | Communication Transaction                       |
|-------|------|-------------------------------------------------|
| 0x0   | 0000 | Write to a memory location.                     |
| 0x1   | 0001 | Write to an ARC EM core register.               |
| 0x2   | 0010 | Write to an ARC EM auxiliary register.          |
| 0x3   | 0011 | NOP. The register is initialized to this value. |
| 0x4   | 0100 | Read from a memory location.                    |
| 0x5   | 0101 | Read from an ARC EM core register.              |
| 0x6   | 0110 | Read from an ARC EM auxiliary register.         |

### 14.2.2.4 The JTAG 1149.1 Address Register (Instruction Code 0xA)

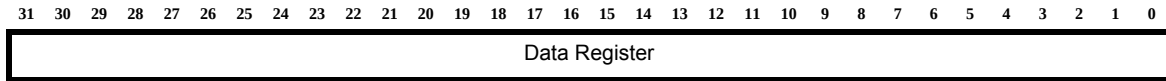
The address register is used to supply the address for read and write transactions to the ARC EM registers and system memory.



Accesses to memory must always be given in bytes. Access to the ARC EM internal registers is specified by their register numbers. The value contained in this register is automatically incremented by four (a memory access) or one (a register access) when a read or write transaction completes. This feature is used to save cycle time when downloading or uploading a stream of data. For this reason, the register does not need to be rewritten with the next address value.

### 14.2.2.5 The JTAG 1149.1 Data Register (Instruction Code 0xB)

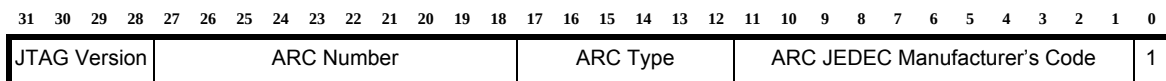
The data register performs two functions. When data is written to this register, it is placed into a write buffer that drives two write data buses, one for the ARC EM host interface and another for the memory arbitrator interface. This bus is used to specify the data contents to be written when performing a write transaction.



A read buffer is selected to read this register. The read buffer stores data retrieved from the target device during a read transaction. The appropriate read data bus (arbitrator or host interface) is selected according to which device the host is accessing.

### 14.2.2.6 The JTAG 1149.1 IDCODE register (instruction code 0xC)

The IDCODE register is used by the JTAG emulator to identify the core as an ARC EM core. The IDCODE register is selected when a reset is applied to the JTAG module.



Each field in the IDCODE register reflects the following information:

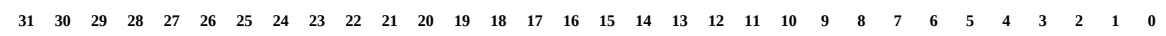
- Bits 31 to 28 - These bits define the version of the JTAG module. Currently set to the value 0x1.
- Bits 27 to 18 - Set this field to the number of ARC processors in the system. Use the same value as the corresponding field in the IDENTITY auxiliary register, if there are fewer than 256 ARC processors in the system.
- Bits 17 to 12 - This field is set to 0x04 to identify the processor type as ARC EM.
- Bits 11 to 1 - This field contains the manufacturer's code assigned by JEDEC, encoded as specified in the IEEE 1149.1-2001 standard. The manufacturer's code is 0x58 in group five, so this field encodes to 0100 101 1000.
- Bit 0 - This field is fixed at 1, as specified in the IEEE 1149.1-2001 standard. This field is used with the previous field to allow automatic discovery when the chain is initialized.

JEDEC never assigns the manufacturer's code 0x7F in group zero. The JTAG emulator can therefore shift the 32-bit IDCODE 0x000000FF into TDI at the beginning of the chain after reset. The standard specifies that upon receiving a TCK when in the reset-test-logic state, the instruction register must be initialized to point to the IDCODE register if this register exists, and to the BYPASS register otherwise.

In capture-DR, the shift register is loaded with this 32-bit code if IDCODE is in the instruction register, and with a single bit of zero if BYPASS is. Thus the external circuitry can examine TDO in shift-DR and determine whether this TAP controller has only a one-bit bypass register (TDO = zero), or a 32-bit IDCODE register (TDO = 1). By shifting through and examining until 0x000000FF appears, the emulator can uniquely identify the number and kind of devices in the chain.

### 14.2.2.7 The JTAG 1149.1 Bypass Register (Instruction Code 0xF)

The bypass register is required as part of the IEEE STD 1149.1 standard.





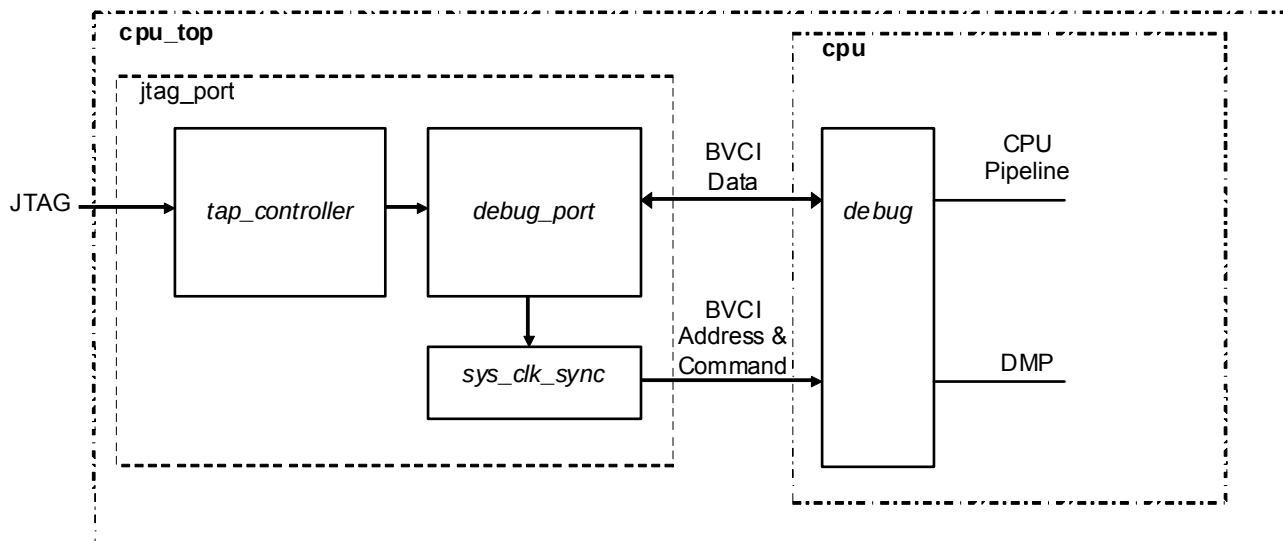
When this register is selected, the serial data input (TDI) is connected to the serial data output (TDO) through this register. The data on TDI is passed to TDO on the rising edge of the JTAG clock TCK when in shift-DR, and the register is initialized to 0 in capture-DR. In all other states, the data in this register is held.

### 14.2.3 The JTAG 1149.1 TAP Controller

The Test Access Port (TAP) controller is central to the operation of the JTAG module as shown in [Figure 14-2](#). All internal register accesses are performed serially using the TAP controller. An accompanying block, the debug port, performs the majority of internal tasks (accessing internal JTAG registers) and external tasks (performing BVCi transactions). The debug port and the TAP controller are clocked by TCK, and a separate module synchronizes the BVCi initiator signals to the system clock.

On the other side of the BVCi Debug Interface, the CPU debug module contains the address, data, command, and status registers, and handles the host interface and DMP transactions. This allows you to optionally replace the JTAG port with a custom interface to an external debugger or to another processor in a master-slave configuration.

**Figure 14-2 Internal Structure of the JTAG 1149.1 Port**

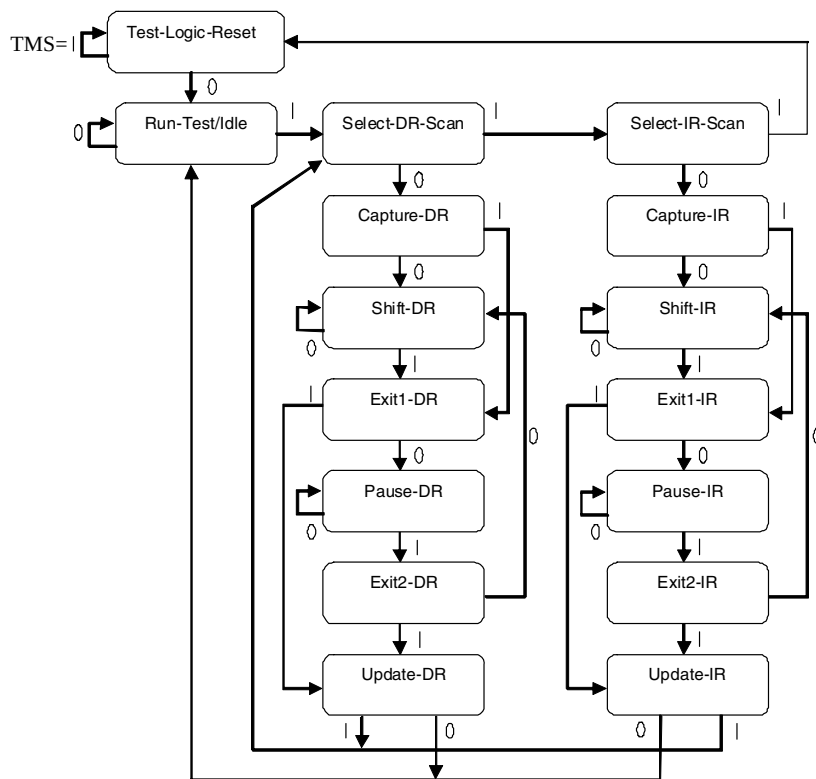


The TAP controller is an internal state machine that is controlled entirely by the host using the TMS and TCK interface signals. The controller is used to indirectly initiate communication transactions and access the internal JTAG registers. The state machine consists of 16 states, as shown in [Figure 14-3](#) on page 225.



## 14.2.4 The JTAG 1149.1 TAP Controller State Machine

**Figure 14-3 JTAG 1149.1 TAP-Controller State Diagram**



Each state contains at least one entry point with two possible exit paths. A state transition is performed on the rising edge of TCK. The decision to determine the exit path is made according to the logic level of TMS.

The test-logic-reset state is used to initialize all internal JTAG registers and control signals to default contents and inactive logic levels. The state is entered immediately when TRST\* is asserted. In addition, this state can also be entered (regardless of the current state) at any time during operation by holding TMS high and applying a maximum of five clock pulses on TCK.

The run-test/idle state is always preceded by the test-logic-reset, update-DR, and update-IR states on the rising edge of TCK when TMS is low. This state is employed to initiate a read/write access or place the JTAG module in the idle state. The read/write access defined by the address, data, and command registers only occurs once on entry to run-test/idle.

The remaining section of the state diagram (in [Figure 14-3](#)) contains two state sequence structures that are used to access all internal JTAG registers. Registers can be written to or read from serially using the TDI and TDO signals along with the TCK and TMS signals. Both structures are identical, but (as denoted by the mnemonics IR and DR) one structure is used to access the instruction register and the other dedicated solely to accessing all data registers.

An internal JTAG register is accessed by placing the TAP controller into the appropriate scan structure (select-DR-scan or select-IR-scan). The data contents of the selected register are loaded into a shift register in the capture-XR state. The state is then entered from the select-XR-scan state by pulling TMS low and applying a clock pulse on TCK. Capture occurs when the capture-XR state is exited.

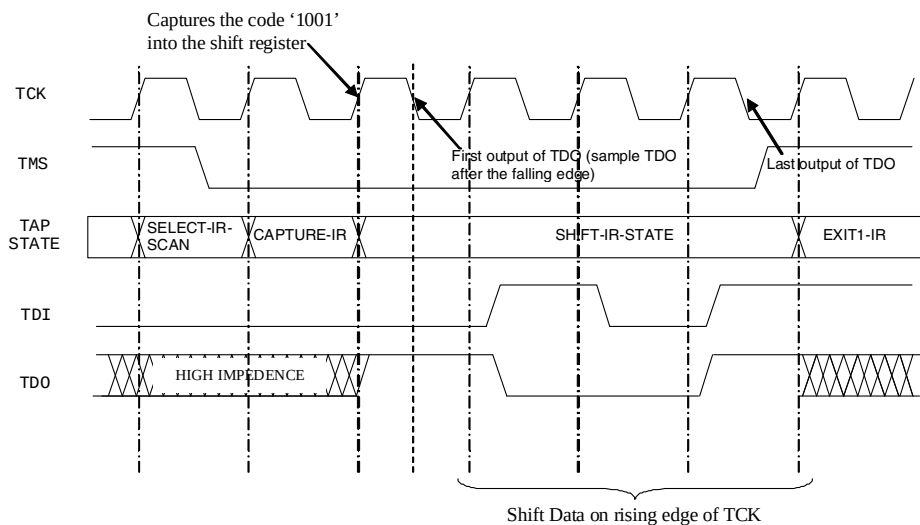
The shift register is used to shift data into the chain from TDI (write phase), simultaneously shifting data out of the chain at TDO (read phase).

By holding TMS low and applying a second clock pulse on TCK, the TAP controller goes into the shift-XR state. The TAP controller remains in the shift-XR state when TMS is held low. This state allows data to be loaded serially (least-significant bit first) into the shift register. The TDI signal is always sampled on the rising edge of TCK, starting on the second entry into the shift-XR state. The data shifted out are placed on TDO on the falling edge of TCK starting on the first entry into the shift-XR state. The last sample of TDI is always performed when exiting the shift-XR state. For instance, when the instruction register contains the BYPASS instruction, a zero is loaded into the one-bit bypass register on the clock that exits capture-DR and enters shift-DR. At this point, the zero appears on TDO. On the next clock, TDI appears on TDO, and data continue to be shifted through until the final TDI is shifted to TDO on the clock that exits shift-DR and enters exit1-DR.

After the data are finally shifted in or out of the shift register, the selected JTAG register is updated with the shift register contents when the TAP controller is placed into the update-XR state. Updating occurs on the falling edge of TCK after the state is entered.

The remaining states pause-XR and exit2-XR are used to stall the shift process if the data to be shifted in cannot be presented in time for the next rising edge of TCK (assuming a continuous frequency on TCK).

**Figure 14-4 Loading Data into the JTAG 1149.1 Shift Register**



The timing diagram in [Figure 14-4](#) illustrates the concept of shifting data into the shift register using TCK, TMS, and TDI. The diagram illustrates the host device writing to the instruction register with the four-bit value 1010, thus selecting the address register. In the capture stage the four-bit value 0001 is loaded into the shift register ready to be shifted out. The instruction register is not updated with the shift register contents (and the address register is not selected) until the update-IR state is entered. Following the update-IR state the select-DR-scan state structure is entered to access the address register.

During the capture-IR phase the four-bit value 0001 is always loaded into the shift register, regardless of the instruction-register contents. The first two least-significant bits aid in diagnosing faults along the IEEE bus.

## 14.3 Two-Wire JTAG Interface (IEEE 1149.7)

The ARC two-wire JTAG interface is a macro that implements the IEEE 1149.7 Standard for a test-access port that offers reduced pins and enhanced functionality.

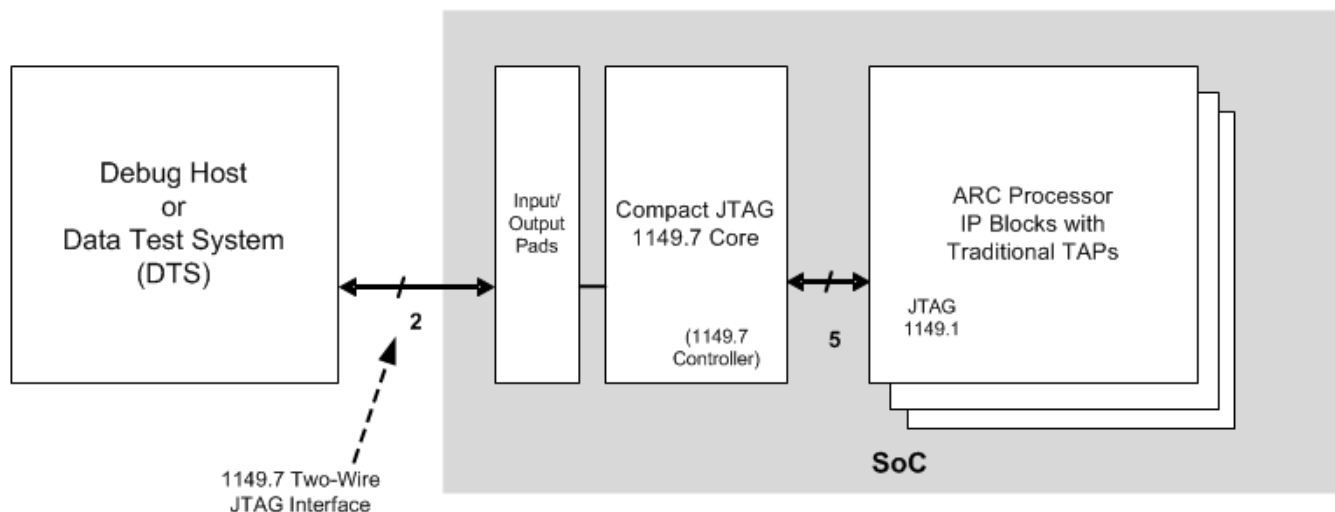
The 1149.7 Standard does not replace but *extends* the 1149.1 JTAG Standard. This IP functions as an interface between an existing 1149.1 JTAG target system port and a 1149.7 host, providing two-wire interface capability, among other enhancements, without the need to change the existing 1149.1 on-chip target infrastructure either in terms of hardware interface or programming model.

The Compact JTAG 1149.7 Core is compliant with the IEEE 1149.7 *Standard for Reduced-Pin and Enhanced-Functionality Test Access Port and Boundary-Scan Architecture*, revision 2009.

The Compact JTAG 1149.7 Core IP is implemented as an ARC IP library compatible with the ARChitect configuration tool to seamlessly add 1149.7 functionality to DesignWare ARC processor cores. The ARChitect-generated design can be integrated with any one or more 1149.1 JTAG target ports within an IC (including ARC Processor cores and/or other IP blocks).

Figure 14-5 shows a typical application for the Compact JTAG 1149.7 Core.

**Figure 14-5 Typical Application for the DesignWare Compact JTAG 1149.7 Core**



### 14.3.1 Compact JTAG 1149.7 Core Features

- The Compact JTAG 1149.7 Core has the following features:
- Provides a two-pin interface to one or more existing on-chip 1149.1 TAP controllers
- Supports Series Scan or Star-4/Star-2 topologies of interconnected 1149.7 controllers
- Implements IEEE 1149.7-2009, Classes T0 - T4
  - Class T0 maintains IEEE 1149.1-specified behavior while supporting multiple TAPs on a single chip or System in a Package
  - Class T1 adds support for the IEEE 1149.7 command protocol, and generation of test reset

- ❑ Class T2 adds a chip-level super bypass with the capability to bypass a chip's system test logic, resulting in a one-bit bypass for both instruction- and data-register paths
  - ❑ Class T3 adds support for connecting TAP.7 controllers in a four-wire star topology
  - ❑ Class T4 adds support for two-pin operation along with two-wire star topology
- Efficient transfer of test and debug information
- Implemented with 1149.7 specified sub-modules:
  - ❑ RSU: Reset and Selection Unit
  - ❑ EPU: Extended Protocol Unit
  - ❑ APU: Advanced Protocol Unit
- Supports T4 mandatory scan formats JScan0-3, MScan, and OScan0-1
- Implements T4 mandatory registers, commands, and related functions
- Implements Test Reset (TRESET) to reset the target-system test logic
- Supports multiple ARC core configurations (multiple on-chip 1149.1 TAP controllers)
  - ❑ Utilizes existing daisy-chain topology support
- Defaults to 1149.1 mode (4/5 wire JTAG mode)
  - ❑ Switches to two-wire mode with appropriate DTS signaling on TCK, TMS only
- Core Area:
  - ❑ Approximately 2500 gates
  - ❑ Soft core supporting Synopsys Reference Design Flows (RDFs) for ASIC and FPGA
- Selection Directives (SSDs) are *not* supported
- Falling-edge sampling of JTAG inputs TMS, [TDI] controlled by the SREDGE register is not supported; the SREDGE register must be set to 1.

### 14.3.2 Adding the Compact JTAG 1149.7 Core

The Compact JTAG 1149.7 Core IP is implemented as Verilog HDL in a bundle IP library. You can add it to an existing JTAG module in your design using the ARChitect configuration tool.

The following DesignWare Compact JTAG 1149.7 Core parameter is selectable using the ARChitect GUI or command line.

- `-jtag_node_id` - Set the value of the TAP controller node ID (eight-bit hexadecimal number). Used with 27 bits of the JTAG device ID to form the 1149.7 Tap-Controller Address (TCA).

```
-jtag_node_id 0
```

### 14.3.3 Compact JTAG 1149.7 Synthesis Constraints

In a build generated by ARChitect using the ARC Reference Design Flow (RDF) IP libraries, synthesis constraints for the CPU subsystem design — including the Compact JTAG component — are located in the `build/constraints` directory as an SDC file. The Compact JTAG Core does not alter the TCK restriction

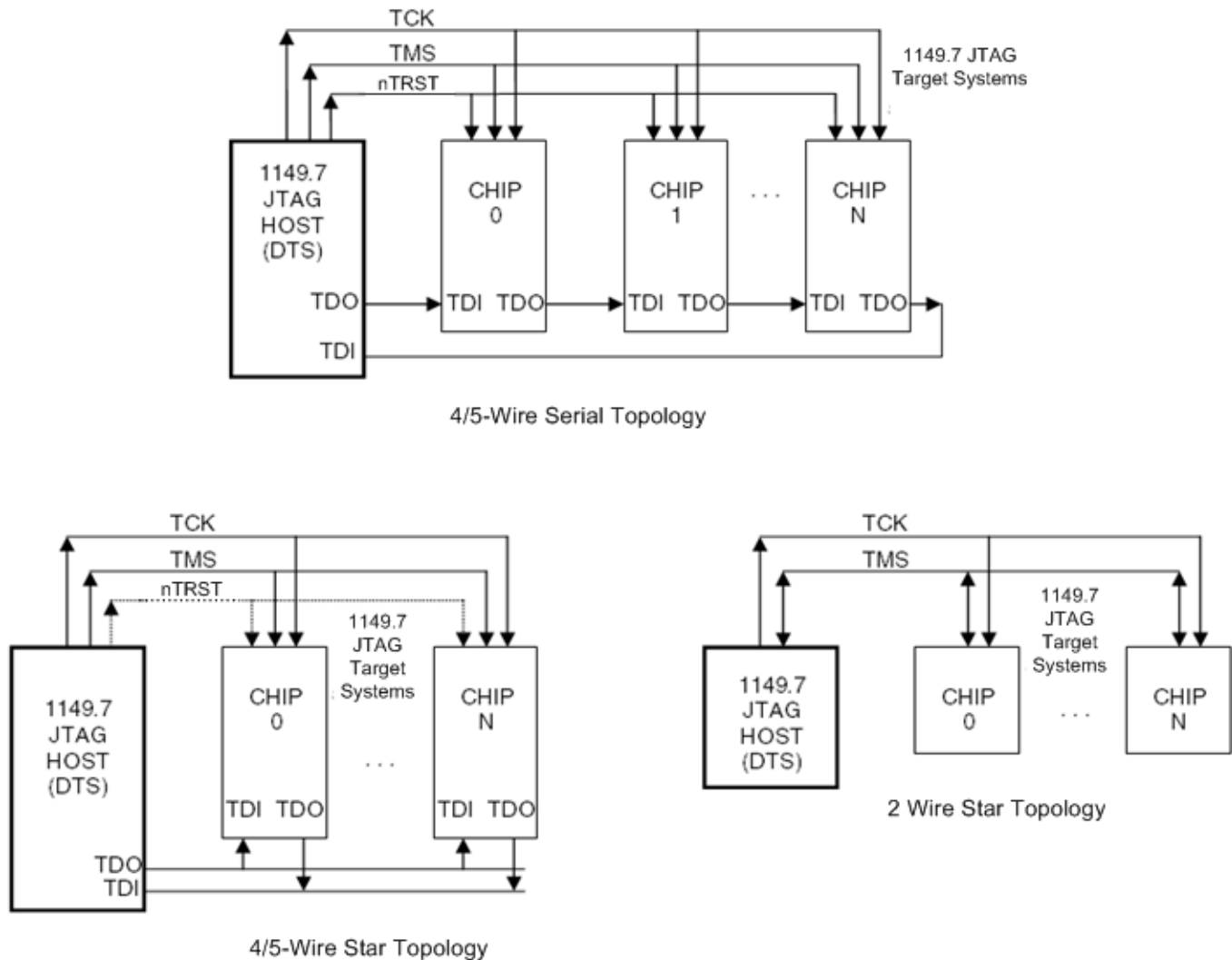
of the standard 1149.1 JTAG port; the TCK clock frequency must be less than half of the CPU clock, and defaults to 10 MHz. The Maximum TCK frequency is dependent on the physical system components including the cable, connector, pod, and ASIC or FPGA implementation, and is expected to be well in excess of 10 MHz.

#### **14.3.4 Compact JTAG 1149.7 Core System-Level Overview**

The Compact JTAG 1149.7 Core (TAP.7) adapts the IEEE 1149.7 2-wire JTAG protocol to the traditional 1149.1 four- or five-wire protocol. It serves as a chip-level resource and can serve multiple cores within the chip. The Compact JTAG controller supports class levels T0 - T4 to provide two-wire JTAG capability (among other capabilities).

#### **14.3.5 JTAG 1149.7 System-Level Block Diagrams**

Previous generations of the JTAG Standard allowed only for a serial connection of chip-level or intra-chip TAP controllers. This situation sometimes resulted in very long and complex scan paths that limited the overall performance of test or debug operations. The 1149.7 Standard provides for star (parallel) connection of 1149.7 TAPs as well as serial connection.

**Figure 14-6 Compact JTAG 1149.7 Core System-Level Block Diagram**

Additionally, the standard allows for a DTS connection to support a hierarchy of TAPs arranged in technology branches that can be a mix of 1149.7 and 1149.1 (two-wire and four-wire) technologies as well as topology branches comprising series and star topologies. Most systems have a single technology branch with a small number of target-system TAPs.

### 14.3.6 JTAG 1149.7 Chip-Level Block Diagrams

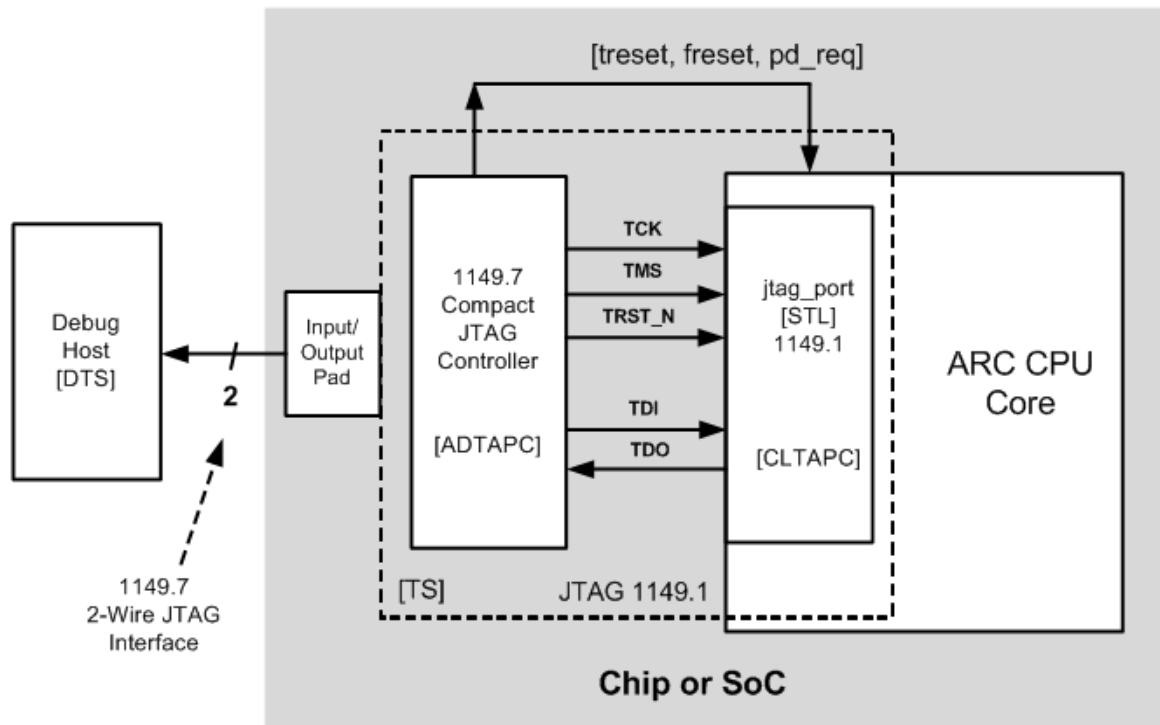
The 1149.7 Standard refers to the JTAG host as the Data Test System (DTS) and the target device as the Target System (TS). A typical Target System has a single Compact JTAG 1149.7 Core interfacing to the external JTAG port and one or more 1149.1 TAPs built into IP macros, referred to as the System Test Logic (STL).

While the 1149.7 Standard does not preclude multiple 1149.7 TAPs within a chip, the use of wire-ANDed signaling in the 1149.7 specifications makes such an arrangement less attractive for on-chip use. A better approach is to use a chip-level (1149.1) TAP or CLTAPC dedicated to select and control multiple 1149.1 on-chip TAPs.

### 14.3.6.1 JTAG 1149.7 Single TAP Controller STL

The controller (two-wire adapter) includes an independent TAP-controller state machine (ADTAPC or adapter TAP controller) that is identical in state definitions but separate from any existing TAP controllers within IP blocks within a chip, or a chip-level TAP controller CLTAPC, which may be added for controlling other TAPs within the chip.

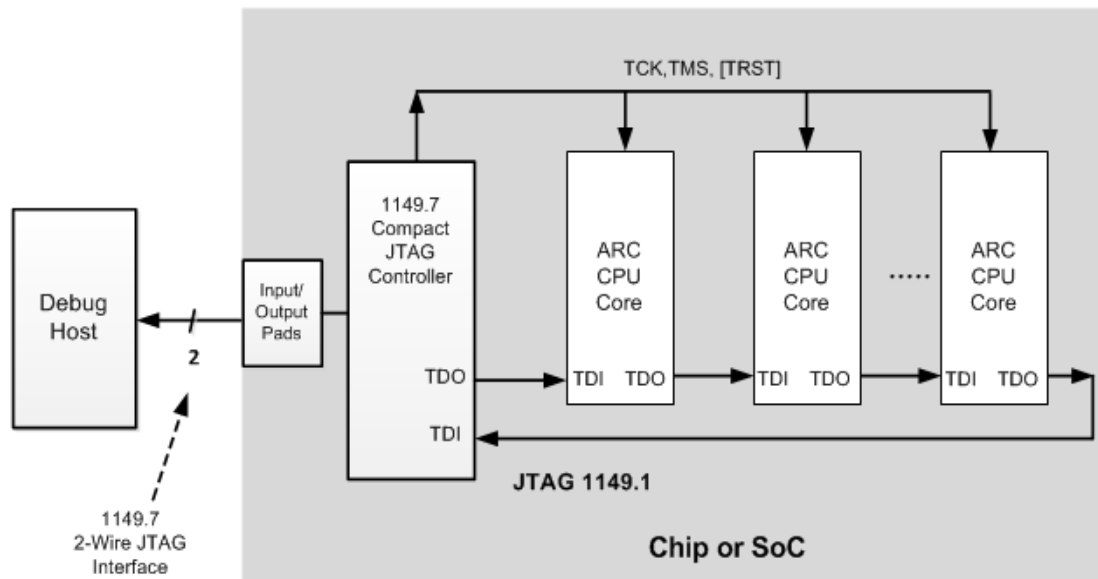
**Figure 14-7 Conceptual System-Level Block Diagram (Single STL TAP Controller)**



### 14.3.6.2 JTAG 1149.7 Multiple STL TAP controllers

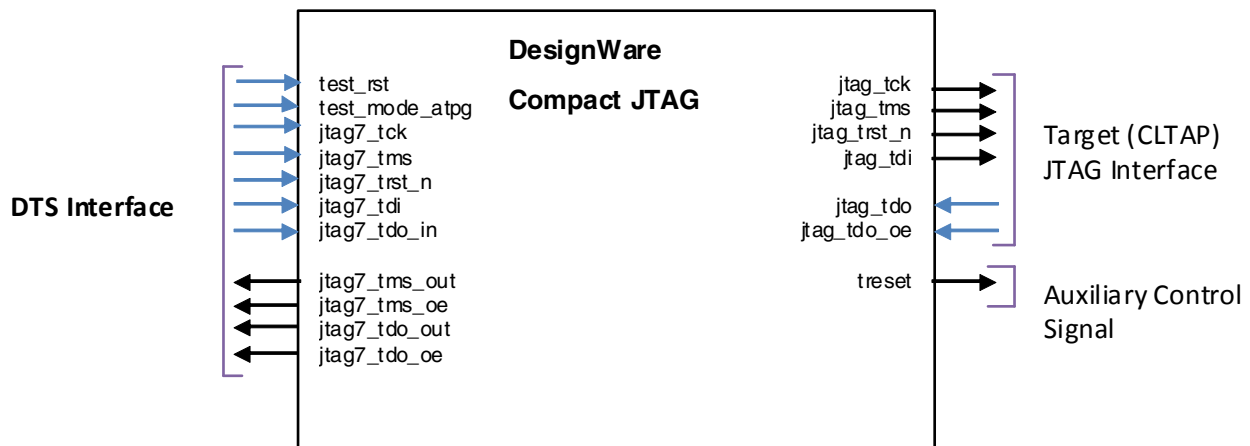
In the 1149.7 Standard, a chip-level TAP controller (CLTAPC) is depicted as one method to control multiple on-chip 1149.1 TAP controllers by selecting one of them and aiding chip-level power management, etc. In the ARC implementation of multiple cores, a CLTAPC is not used and the 1149.7 controller directly interfaces with one or more CPU cores arranged in a serial daisy-chain topology as shown in [Figure 14-8](#).

As in earlier versions of the ARC processor cores, multiple CPUs can be debugged through a single JTAG port, either through a 1149.7 adapter or directly through a 1149.1 4/5 wire JTAG port.

**Figure 14-8 Conceptual System-Level Block Diagram (Multiple STL TAP Controllers)**

### 14.3.7 JTAG 1149.7 Signals

Figure 14-9 provides an overview of the Compact JTAG 1149.7 Core top-level signals.

**Figure 14-9 JTAG 1149.7 Top-Level I/O Diagram**

#### 14.3.7.1 JTAG 1149.7 Signal Descriptions

This section describes the top-level signals.

All digital signals are active high unless stated otherwise. Signals prefixed with jtag7\_ are synchronous to jtag7\_tck, while signals prefixed with jtag\_ are synchronous to jtag\_tck unless stated otherwise.

The output test clock jtag\_tck is derived from the input test clock jtag7\_tck, and output edges have a small delay (flip-flop-clock-to-output delay) from input to output. They are not asynchronous to each other.



### 14.3.7.2 Host JTAG 1149.7 Signals

Table 14-3 describes the host JTAG interface signals used with the Compact JTAG 1149.7 Core.

**Table 14-3 Host JTAG 1149.7 Interface**

| Signal         | I/O | Used In Mode               | Description                                                                                                                                                                                                                         |
|----------------|-----|----------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| jtag7_tck      | I   | 2-wire / 4-wire            | TCK, Test Clock input. The JTAG host drives the DTS interface signals TDI and TMS on the falling edge of this clock signal. Both rising and falling edges of this clock are used by the Compact JTAG 1149.7 Core to sample signals. |
| jtag7_tms      | I   | 2-wire / 4-wire            | TMS, Test Mode Select input                                                                                                                                                                                                         |
| jtag7_tms_out  | O   | 2-wire                     | TMS Output Value, Test Mode Select output value, to be driven on TMS I/O pad when jtag7_tms_oe is asserted.                                                                                                                         |
| jtag7_tms_oe_n | O   | 2-wire                     | TMS Output Enable, Test Mode Select output I/O pad drive enable                                                                                                                                                                     |
| jtag7_tdi      | I   | 4-wire (Star-4)            | TDI, Test Data Input (unused in two-wire mode)                                                                                                                                                                                      |
| jtag7_tdo      | O   | 4-wire (Star-4)            | TDO, Test Data Output (unused in two-wire mode)                                                                                                                                                                                     |
| jtag7_tdo_oe   | O   | 4-wire (Star-4)            | TDO, Test Data Output I/O pad drive enable (unused in two-wire mode)                                                                                                                                                                |
| jtag7_tdo_in   | I   | 4-wire (Star-4)            | TDO, Test Data Output I/O pad input value (may be tied low in two-wire mode)                                                                                                                                                        |
| jtag7_trst_n   | I   | 4-wire (Star-4) (Optional) | TRST_N, Test Reset input (Active-Low, Asynchronous)<br>Optional connection to host; may be tied to active-low power-on reset within target                                                                                          |

### 14.3.7.3 Target CLTAPC JTAG 1149.7 Signals

Table 14-4 describes the CLTAPC JTAG interface signals.

**Table 14-4 Target (CLTAPC) JTAG 1149.7 Interface**

| Signal   | I/O | Used In Mode    | Description                       |
|----------|-----|-----------------|-----------------------------------|
| jtag_tck | O   | 2-wire / 4-wire | JTAG TCK, Test Clock input.       |
| jtag_tms | O   | 2-wire / 4-wire | JTAG TMS, Test Mode Select output |
| jtag_tdi | O   | 2-wire / 4-wire | JTAG TDI, Test Data               |

**Table 14-4 Target (CLTAPC) JTAG 1149.7 Interface**

| Signal      | I/O | Used In Mode               | Description                                                                                                                                                                                                                                                                          |
|-------------|-----|----------------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| jtag_tdo    | I   | 2-wire / 4-wire            | JTAG TDO, Test Data Input                                                                                                                                                                                                                                                            |
| jtag_tdo_oe | I   | 2-wire / 4-wire            | JTAG TDO, Test Data Output I/O pad drive enable Input                                                                                                                                                                                                                                |
| jtag_trst_n | O   | 2-wire / 4-wire (Optional) | JTAG TRST_N, Test Reset input (Active-Low, Asynchronous)<br>Combination of jtag7_trst_n, test_rst, rst_n, treset or a type-3 reset (defined in “ <a href="#">JTAG 1149.7 Reset Types</a> ” on page 236), including resets generated from escapes, check packets, and delay elements. |

**14.3.7.4 JTAG 1149.7 Auxiliary Control Signal**

[Table 14-5](#) describes the auxiliary control signal.

**Table 14-5 JTAG 1149.7 Auxiliary Control Signal**

| Signal | I/O | Used In Mode    | Description                                   |
|--------|-----|-----------------|-----------------------------------------------|
| treset | O   | 2-wire / 4-wire | 1149.7 Test Reset Output. To reset STL logic. |

**14.3.7.5 JTAG 1149.7 Test-Mode Signal**

[Table 14-6](#) describes the test-mode signal.

**Table 14-6 JTAG 1149.7 Test-Mode Signal**

| Signal         | I/O | Used In Mode    | Description                                                                                                  |
|----------------|-----|-----------------|--------------------------------------------------------------------------------------------------------------|
| test_mode_atpg | I   | 2-wire / 4-wire | Test Mode Enable. Used by ATPG scan test tools. Set to logic 1 to enable test mode, logic 0 for normal mode. |

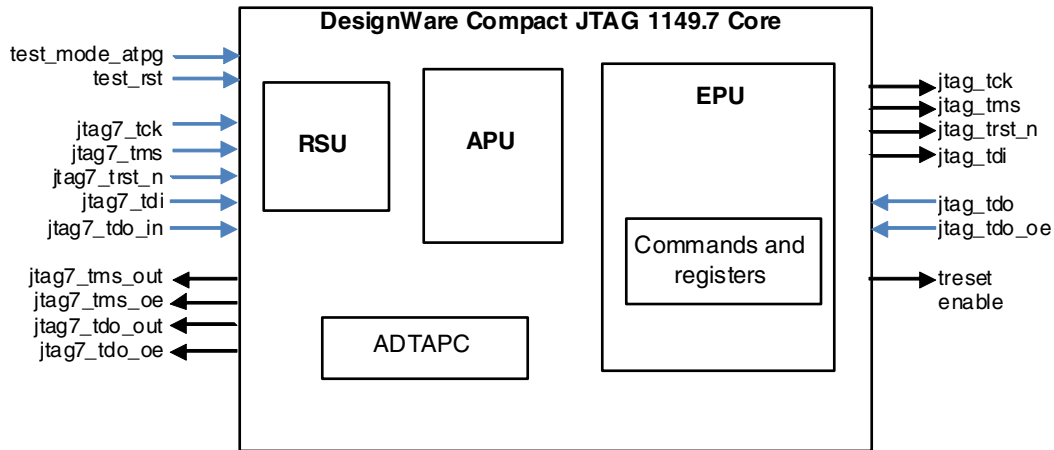
**14.3.8 JTAG 1149.7 Functional Description: Overview**

The TAP.7 architecture is designed to interoperate with TAP.1 by using state sequences and signaling that are transparent to TAP.1 controllers. New TAP.7 functionality is invoked by control protocols that enable selection and deselection of individual TAPs and selective use of 1149.7 advanced protocols (multiplexed two-wire).

The Compact JTAG 1149.7 Core has the following main blocks:

- Reset and Selection Unit (RSU)
- Advanced Protocol Unit (APU)
- Extended Protocol Unit (EPU)

[Figure 14-10](#) shows a functional block diagram of the Compact JTAG 1149.7 Core.

**Figure 14-10 Compact JTAG 1149.7 Core Functional Block Diagram**

### 14.3.9 JTAG 1149.7 Operating Modes

The following key concepts encompass the major operating states of the TAP.7 adapter:

- Online (Selected) / Offline (Deselected)
- Coupled / Decoupled (ADTAPC coupling with CLTAPC/STL)
- Control Levels entered through Zero-Bit Scan (ZBS) sequences
- EPU operating states defined by control register bits
  - RUNNING
  - SUSPENDED
  - INHIBITED

These operating modes are described briefly in the sections below and in more detail in the 1149.7 Standard.

### 14.3.10 JTAG 1149.7 Initialization

The 1149.7 Standard specifies six reset types and four startup options. These types and options work together to handle many scenarios and system configurations and are described in detail in the standard. The descriptions presented here are at a high level and conceptual.

The DesignWare Compact JTAG 1149.7 Core is configured to the startup option IEEE 1149.1-Compliant (shaded in [Table 14-7](#)). Thus, the IP can be used in 1149.1 four-wire mode by default.

**Table 14-7 Host JTAG 1149.7 Interface**

| Startup Option                  | Before ADTAPC Placed Online? | Default Characteristics            |                |          |        |                         |
|---------------------------------|------------------------------|------------------------------------|----------------|----------|--------|-------------------------|
|                                 |                              | ADTAPC Placed Online with Power-Up | CLTAPC Coupled | SGC Reg. | SCNFMT | Initial TMS signal bias |
| IEEE 1149.1-Compliant           | n/a                          | Yes                                | Yes            | 1        | JScan0 | Pull-up                 |
| IEEE 1149.1-Compatible          | n/a                          | Yes                                | No             | 0        | JScan1 | Pull-up                 |
| IEEE 1149.1-Protocol Compatible | n/a                          | Yes                                | No             | 0        | JScan1 | Pull-up                 |
| Offline-at-Startup              | Yes                          | No                                 | No             | 0        | JScan1 | No biasing              |
|                                 | No                           | Yes                                | No             | 0        | JScan1 | No biasing              |

### 14.3.11 JTAG 1149.7 Startup Options

An 1149.7 target system (TS) powers up online (in conventional 1149.1 four-wire protocol, also known as standard protocol). The target system can be brought offline, using a selection/deselection sequence in a 1149.7 control protocol. An offline 1149.7 TAP (the ADTAPC) is parked, meaning that its TAPC state machine does not change state in response to standard protocol signaling on TCK/TMS inputs (as it normally does); instead, a special startup sequence is used to bring an 1149.7 TAP online.

The Compact JTAG 1149.7 Core supports selection via Selection Escapes and does not support Selection Alerts. Note the distinction between selection/deselection and coupling/decoupling; the 1149.7 Standard is not consistent in this terminology, but in this document, *selection* refers to the connection between the DTS and an ADTAPC while *coupling* refers to the connection between the ADTAPC and the CLTAPC.

### 14.3.12 JTAG 1149.7 Reset Types

The 1149.7 Standard specifies six reset types to initialize selected subsets of the TAP.7 controller and/or downstream JTAG port test logic (STL). [Table 14-8](#) outlines the source and the scope of each reset type.

**Table 14-8 JTAG 1149.7 Reset Types**

| Reset Type | Sync (neg. edge tck) | Sources                         | Reset ADTAPC | Reset TAP7 Regs | Reset TAP7 SMs | Reset CLTAPC and STL (sys_rst) | Affect STL Selection (esc det) | Affect ADTAP Selection Logic (off / online) |
|------------|----------------------|---------------------------------|--------------|-----------------|----------------|--------------------------------|--------------------------------|---------------------------------------------|
| 5          | Sync                 | TRESET Register (treset output) | No           | No              | No             | Yes                            | No                             | No                                          |
| 4          | Sync                 | Test Logic Reset State (ADTAPC) | No           | Yes             | All except csm | No                             | No                             | Yes (not initialized)                       |

**Table 14-8 JTAG 1149.7 Reset Types**

| Reset Type | Sync (neg. edge tck)      | Sources                                                                                  | Reset ADTAPC | Reset TAP7 Regs | Reset TAP7 SMs | Reset CLTAPC and STL (sys_rst) | Affect STL Selection (esc det) | Affect ADTAP Selection Logic (off / online) |
|------------|---------------------------|------------------------------------------------------------------------------------------|--------------|-----------------|----------------|--------------------------------|--------------------------------|---------------------------------------------|
| 3          | Sync (single clock width) | - Reset Escape<br>- Check Packet, CP_RS0 Directive<br>- Delay Element, DLY_RS0 Directive | Yes          | Yes             | All            | Yes                            | Yes                            | Yes                                         |
| 2          | Async                     | Chip level reset (rst_a)                                                                 | Yes          | Yes             | All            | Yes                            | Yes                            | Yes                                         |
| 1          | Async                     | TRST_N pin (jtag7_trst_n input)                                                          | Yes          | Yes             | All            | Yes                            | Yes                            | Yes                                         |
| 0          | Async                     | Ext. TAP7 pwr mgmnt logic (test_rst input)                                               | Yes          | Yes             | All            | Yes                            | Yes                            | Yes                                         |

### 14.3.13 JTAG 1149.7 Bypass Mode

The Compact JTAG 1149.7 Core, when configured with a four wire host JTAG interface (option `jtag_2wire_only=false`), can be operated in four-wire mode or two-wire mode. The four-wire mode is referred to as bypass mode, during which the Compact JTAG 1149.7 Core is essentially transparent for the conventional 1149.1 Standard protocol, passing it directly from the host JTAG port to the target CPU debug JTAG port. The bypass mode is the default after a power-up or hard reset (as well as any of Type0 - 4 reset).

### 14.3.14 JTAG 1149.7 Selection/Deselection

The Compact JTAG 1149.7 Core supports selection via Selection Escapes and does not support Selection Alerts. Scan Selection Directives, special TAP controller state sequences, are also used in 1149.7 to select and deselect TAPs in pause states; this arrangement allows a star topology branch to emulate a conventional series topology network (using series-equivalent scans). SSDs as a method of selection and deselection (parking) are *not* supported by the Compact JTAG 1149.7 Core.

#### 14.3.14.1 Escapes

According to the 1149.7 standard, "An Escape overlays control information onto the information normally transferred with the TCK(C) and TMS(C) signals without changing the normal information conveyed in a bit period. A TAP.7 Controller with an RSU interprets the count of the number of TMS(C) edges while TCK(C) is a logic 1 as one of four Escapes. Each Escape has a different function and TMSC edge count."

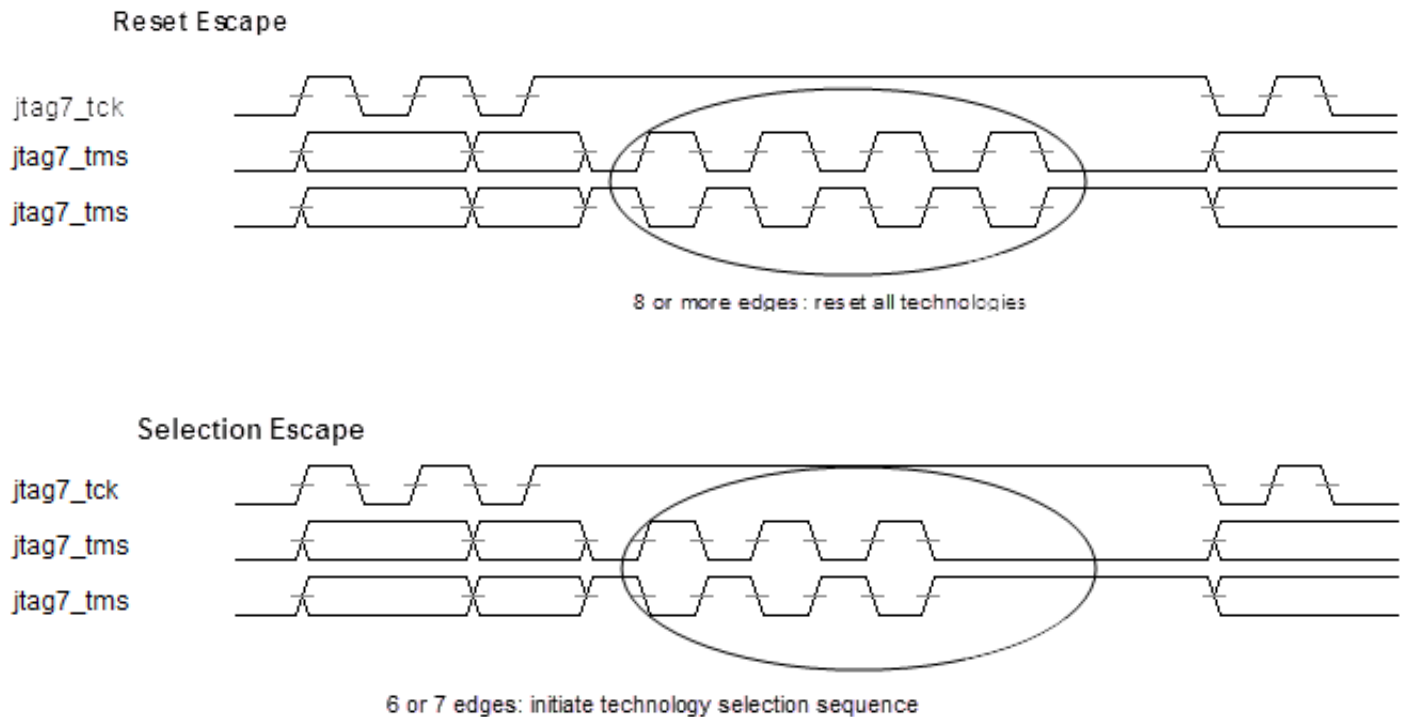
Escapes:

- Custom (2 or 3 edges): Ends scan and transport transfers

- Deselection (4 or 5 edges): Initiates the deselection of online technologies
- Selection (6 or 7 edges): Initiates a Selection Sequence to place the ADTAPC online or offline
- Reset (8 or more edges): Resets all technologies (generates a Type-3 Reset)

The DTS creates an escape by generating one or more TMS(C) edge pairs while the TCK(C) signal is a logic one level (1), as shown in [Figure 14-11](#).

**Figure 14-11 JTAG 1149.7 Example Escape Waveforms**



#### 14.3.14.2 Scan-Selection Directives (Not Supported)

The Compact JTAG 1149.7 Core does not support scan-selection directives.

#### 14.3.14.3 Scan-Topology Training Sequence

The Compact JTAG adapter has the capability to discover the scan topology (Series, Star-4, and Star-2) in which it is deployed during a scan-topology training sequence generated by the DTS. This process updates the TOPOL register within the TAP.7 that qualifies selection of technology branches during a selection sequence.

#### 14.3.14.4 JTAG Device ID with ARCV2EM Target

The 1149.1 JTAG IDCODE value read when used with the ARC EM CPU core is 0x200444b1.

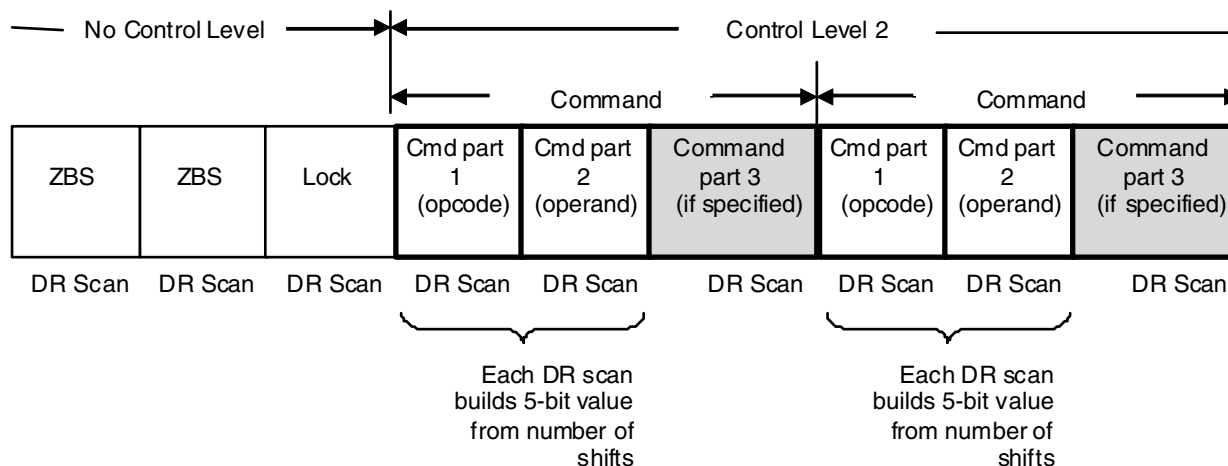
#### 14.3.15 JTAG 1149.7 Control Levels

Apart from the signaling used for escapes (selection, deselection, reset, etc.), another new 1149.7 protocol transparent to the 1149.1 CLTAPC and EMTAPs is the use of Zero-Bit-Scan (ZBS) sequences to enter various control levels. Level two (a ZBS count of two, locked) provides access to TAP.7 registers to hold control,

status, and configuration information. At this control level the STL (including CLTAPC) is decoupled, and the scan path is managed internal to the TAP.7 controller.

Figure 14-12 illustrates the scan sequence for entering control level two and executing three-part commands.

**Figure 14-12 JTAG 1149.7 EPU Operating States**



### 14.3.16 JTAG 1149.7 Command Processing

The Compact JTAG 1149.7 Core has a portfolio of global and local registers used to convey the TAP.7 capabilities and control selection/deselection, discovery/training, and scan formats/timing. Registers are accessed through two-part or three-part TAP.7 Controller Commands available at Control Level 2 (see “JTAG 1149.7 Control Levels” on page 238).

TAP.7 commands are DR scans of ADTAPC while locked at control level two, also referred to as a CR (Control Register) scan. Global registers may also be loaded during a selection sequence.

Table 14-9 lists the commands that are implemented in this version of the DesignWare Compact JTAG 1149.7 Core.

**Table 14-9 TAP.7 Controller Commands**

| Opcode                | Mnemonic | Description                 |
|-----------------------|----------|-----------------------------|
| <b>Store Commands</b> |          |                             |
| 0x00                  | STMC     | Store Miscellaneous Control |
| 0x01                  | STC1     | Store Conditional one-bit   |
| 0x02                  | STC2     | Store Conditional two-bits  |
| 0x03                  | STFMT    | Store Format                |
| 0x04                  | STPST    | NOT IMPLEMENTED (T5 only)   |
| 0x05                  | STTESTM  | NOT IMPLEMENTED             |

**Table 14-9 TAP.7 Controller Commands (Continued)**

| Opcode                    | Mnemonic | Description               |
|---------------------------|----------|---------------------------|
| <b>Select Commands</b>    |          |                           |
| 0x06                      | MCM      | Make Cond. Group Member   |
| 0x07                      | MSC      | Make Scan Group Candidate |
| <b>Scan Commands</b>      |          |                           |
| 0x08                      | SCNB     | Scan Bit                  |
| 0x09                      | SCNS     | Scan String               |
| <b>Enumerate Commands</b> |          |                           |
| 0x0A                      | CIDA     | Controller ID Allocate    |
| 0x0B                      | CIDD     | Controller ID De-allocate |
| <b>Reserved Commands</b>  |          |                           |
| 0x0C – 0x1F               | Reserved | Reserved                  |

### 14.3.17 TAP.7 Registers and Programming Model

Table 14-10 lists the registers and associated commands that are implemented in the DesignWare Compact JTAG 1149.7 Core.

**Table 14-10 TAP.7 Controller Registers and Associated Commands**

| Register                 | R/W – G/L | Width | Default Value (rst type; see Legend) | Assoc. Cmd | Description                                                                                                                   |
|--------------------------|-----------|-------|--------------------------------------|------------|-------------------------------------------------------------------------------------------------------------------------------|
| <b>Control Registers</b> |           |       |                                      |            |                                                                                                                               |
| ECL                      | W-G       | 1     | 0 [0-4]                              | STMC       | <b>Exit Control Level</b> — Writing to this register zeros the ZBS count and unlocks the control level.                       |
| SUSPEND                  | W-G       | 1     | 0 [0-4]                              | STMC       | <b>SUSPEND</b> — Writing to this register zeros the ZBS count, unlocks the ZBS count, and suspends the use of control levels. |
| ZBSINH                   | W-G       | 1     | 0 [0-4]                              | STMC       | <b>ZBSINH</b> — Writing to this register inhibits ZBS detection.                                                              |
| SCNFMT                   | W-G       | 5     | DF [0-4]                             | STFMT      | Scan Format                                                                                                                   |
| SSDE                     | W-G       | 1     | 0 [0-4]                              | STMC       | SSD Enable                                                                                                                    |
| DLYC                     | W-G       | 2     | 0 [0-4]                              | STMC       | Delay Control                                                                                                                 |
| RDYC                     | W-G       | 2     | 0 [0-4]                              | STMC       | Ready Control                                                                                                                 |



Table 14-10 TAP.7 Controller Registers and Associated Commands (Continued)

| Register                   | R/W – G/L | Width | Default Value (rst type; see Legend) | Assoc. Cmd       | Description                                                                                                                                                                     |
|----------------------------|-----------|-------|--------------------------------------|------------------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| <b>Options Registers</b>   |           |       |                                      |                  |                                                                                                                                                                                 |
| TOPOL                      | W-L       | 2     | DT [0-3]<br>NC [4]                   | STC2             | Topology                                                                                                                                                                        |
| FRESET                     | R/W-G     | 1     | 0 [0-4]                              | STC1             | <b>Functional Reset</b> — Requests system logic to generate a functional reset.<br>0: A functional-reset request is not asserted.<br>1: A functional-reset request is asserted. |
| TRESET                     | W-L       | 1     | 0 [0-4]                              | STC1             | <b>Test Reset</b> — Asserts a test reset to the STL.<br>0: No reset.<br>1: Reset the STL by asserting jtag_trst_n and setting jtag_tms signal to 1.                             |
| <b>Select Registers</b>    |           |       |                                      |                  |                                                                                                                                                                                 |
| CGM                        | W-L       | 1     | 0 [0-4]                              | STMC MCM<br>SCNB | <b>Conditional Group Member</b> — Specifies whether the EPU is a member of the Conditional Group<br>0: Not a Conditional Group member<br>1: Conditional Group member            |
| SGC                        | W-L       | 1     | DSGM [0-4]                           | STMC MSC<br>SCNB | Scan Group Candidate                                                                                                                                                            |
| SREDGE                     | W-L       | 1     | 0 [0-4]                              | STC1             | Falling-edge sampling <i>not supported</i> ; this register must be set to 1.                                                                                                    |
| <b>Enumerate Registers</b> |           |       |                                      |                  |                                                                                                                                                                                 |
| CID                        | W-L       | 4     | 0 [0-4]                              | CIDA             | Controller ID                                                                                                                                                                   |
| CIDI                       | W-L       | 1     | 0 [0-4]                              | CIDD             | Controller ID Invalid                                                                                                                                                           |
| <b>Read-Only Registers</b> |           |       |                                      |                  |                                                                                                                                                                                 |
| RDBACK0                    | R-L       | 32    | n/a                                  | SCNS             | Read-back 0                                                                                                                                                                     |
| CNFG0                      | R-L       | 32    | n/a                                  | SCNS<br>SCNB     | Configuration Register 0                                                                                                                                                        |

Legend:

NC is No change.

DF is Default Scan Format.

DSGM is Default Scan Group Membership.

DT is Default Topology.

### 14.3.18 JTAG 1149.7 Controller ID Allocation and Deallocation

Because the 1149.7 Standard supports star configurations where multiple TAPs are connected to shared signals, a method to address individual TAPs is necessary so TDO data can be supplied by a selected TAP without drive conflicts. The addressing is provided by the use of a 35-bit Tap Controller Address (TCA) composed of a device ID and a node ID. To reduce the addressing overhead, a scheme to assign short (four-bit) aliases for connected TAPs is specified; the alias is referred to as a Controller ID (CID), and instructions supporting allocation and deallocation of CIDs are defined.

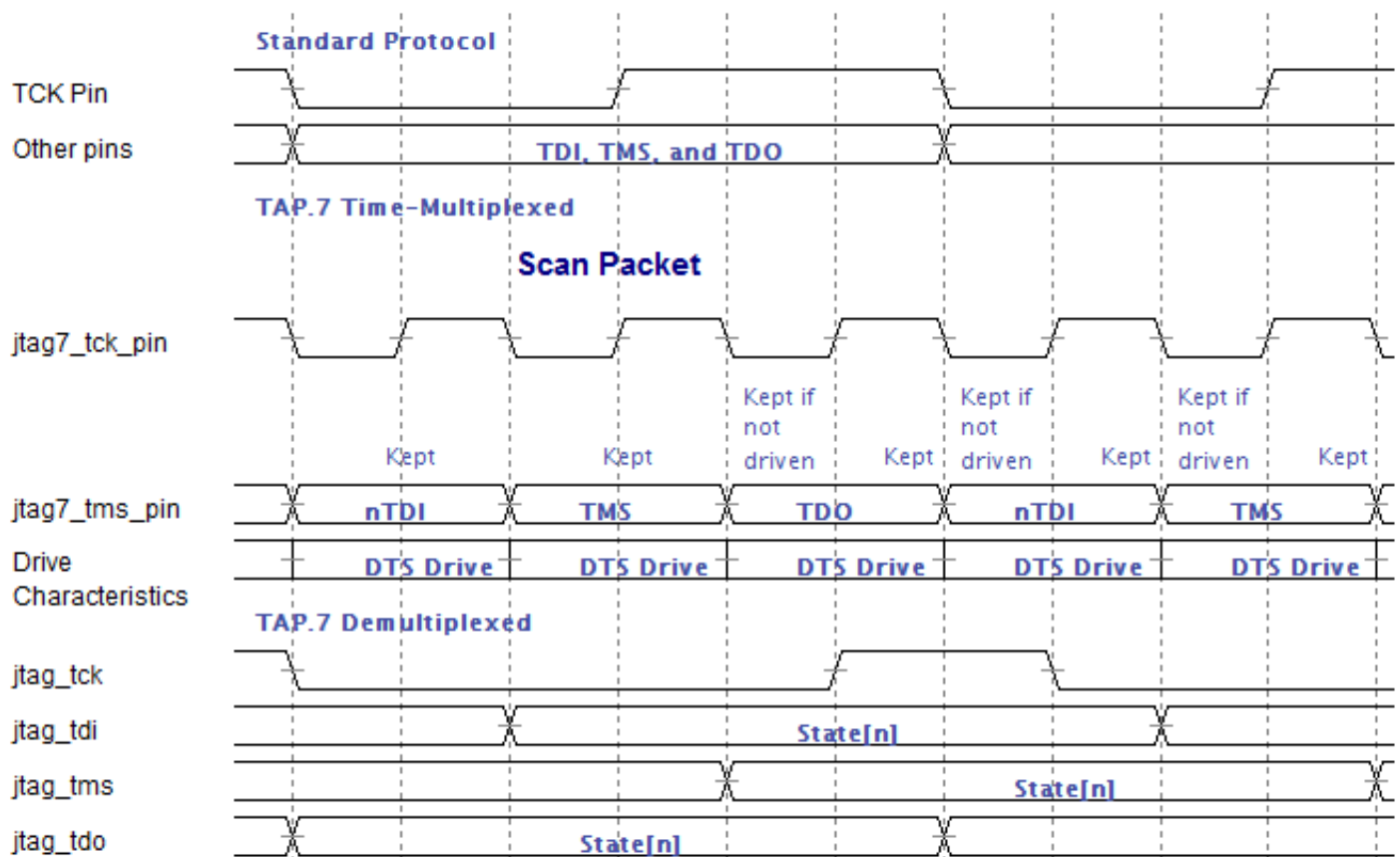
CIDs are allocated either explicitly by the DTS (directed method) or by a voting scheme (arbitration) using wire-ANDed TDO data (TDO pin for four-wire configurations or TDO time slot in the multiplexed MScan format in two-wire configurations); this method is referred to as the undirected method. These operations are described in the 1149.7 Standard.

### 14.3.19 JTAG 1149.7 Scan Formats

The Compact JTAG 1149.7 Core supports all the scan formats defined as mandatory by the 1149.7 Standard; these formats include JScan0-3, OScan0-1, and MScan.

Figure 14-13 illustrates the main scan format (OScan1) used for two-wire operation.

**Figure 14-13 Advanced Protocol (Multiplexed): OScan 1 Format**



## 14.4 The Debug Port

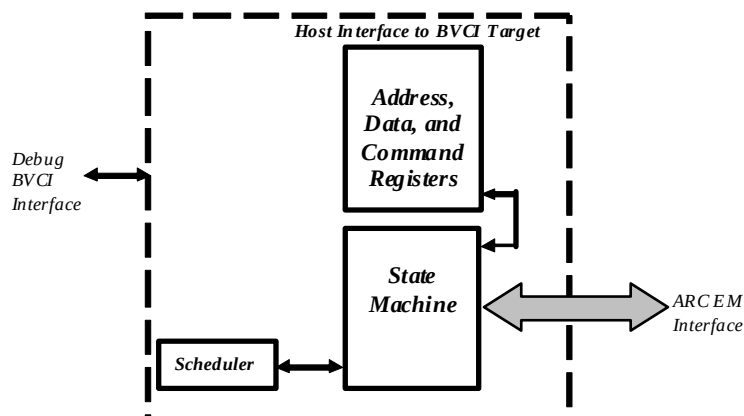
The debug-port module contains all the registers specific to the JTAG interface. This includes the instruction and data shift registers, the instruction register itself, the bypass register, and the IDCODE register (see “JTAG 1149.1 Programmer's Model” on page 220). It also contains a restricted BVCi initiator, which, in conjunction with the system-clock synchronization module, is responsible for access to the address, data, transaction command, and status registers, and for initiation of read and write accesses.

Note that the debug BVCi interface is completely separate from the memory BVCi interfaces. The address space of the debug BVCi interface contains only six valid addresses in its stock configuration: the addresses of the address, data, transaction command, and status registers, along with two special addresses which, when written with any data, cause the read/write access to be initiated and the address, data, and command registers to be reset, respectively.

## 14.5 The Host Interface to BVCi Target

The address, data, status, and transaction-command registers are in the host-interface-to-BVCi target module (as shown in Figure 14-14). It contains a state machine, which performs all read and write bus transactions that are supported by the JTAG module, if a valid code is in the transaction-command register. Access is initiated by placing the TAP controller into the run-test/idle state, which causes the debug port to write the `do_cmd` address on the BVCi interface. This request is then fed to the state machine. The scheduler is responsible for verifying transaction requests and providing the mechanism that allows the host device and the JTAG module to maintain a strict synchronizing relationship.

**Figure 14-14 Internal Structure of the Host Interface to BVCi Target Module**



The scheduler verifies a transaction request from the host device by checking the value contained in the transaction-command register. The transaction-request signal is asserted only when the transaction-command register contains a valid code and a write to the `do_cmd` address occurs on the debug BVCi interface. The scheduler asserts a transaction-request signal to start the defined bus transaction. The BVCi target always responds to all commands in a single cycle. It is the responsibility of the debugger to maintain synchronization by polling the status register for the READY bit after an access starts.

## 14.6 Setting Up Read/Write Transactions

The following are the stages of defining and initiating read and write accesses:

- [Setting up a Write Access to the ARC EM Processor or Memory](#)
- [Accessing the Status Register](#)
- [Setting up a Read Access from the ARC EM Processor or Memory](#)

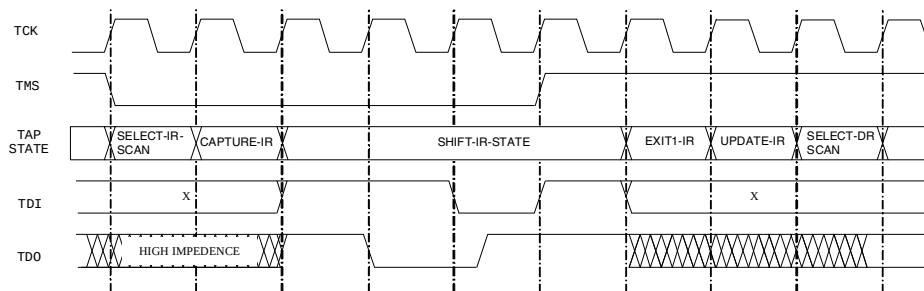
### 14.6.1 Setting up a Write Access to the ARC EM Processor or Memory

A write access requires placing the TAP controller into the test-logic-reset state to reset the JTAG module. Reset initializes all the internal JTAG registers to default values and all interface control signals to inactive logic levels. This initialization process is performed by asserting TRST\* or by holding TMS high and applying a maximum of five clock pulses on TCK. This ensures that when the run-test/idle state is entered a bus transaction is not triggered from a valid code that might already be present in the transaction command register.

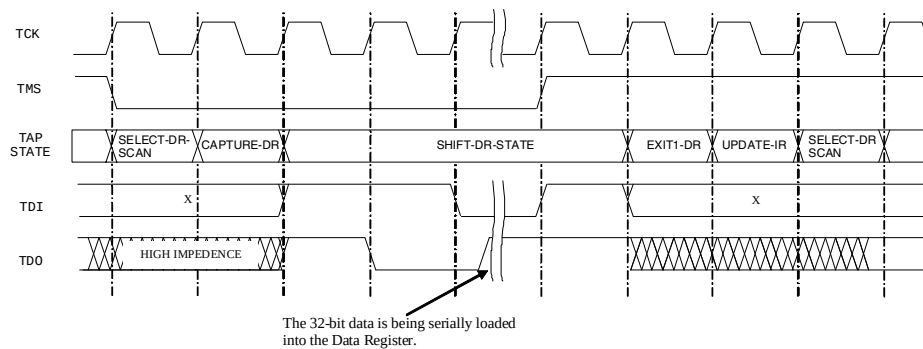
The reset procedure is not required for every read and write access, and is performed only following a system reset. The next stage is to set up the transaction-parameter registers. These registers include the address register, data register, and transaction-command register.

The address and data registers are loaded with the appropriate values so that the write access can be performed on the ARC EM processor or to memory. The sequence is as follows:

1. The instruction register is loaded with the code for accessing the data register. This is accomplished by entering the select-IR-scan state and updating the instruction register.
2. The code 1011 for the data register is serially loaded in the shift-IR state. The timing diagram for writing the instruction register with the code 1011 to select the data register is shown below:



3. The select-DR-scan state is then selected to serially load in the data to be employed by the write data bus when write access is performed. This action is shown in the timing diagram below:



4. The address register is now accessed by loading the code 1010 into the instruction register. The TAP controller then enters the select-DR-scan state structure, and the data to be used are serially loaded in on the address bus.
5. The last stage is writing the transaction-command register with the code instructing the JTAG module to perform a write transaction to memory or the ARC EM registers.

After all the transaction parameters are set up, the write transaction is started by placing the TAP controller into the run-test/idle state.

To obtain information about the transaction, the JTAG Status register is accessed. Because this is a read-only register, the signal supplied on TDI is ignored when the register contents are shifted out through TDO. The appropriate bit fields are then checked to verify a write transaction was performed.

## 14.6.2 Accessing the Status Register

The JTAG Status register is accessed in the following way: The first bit in the status register is shifted out to determine whether the JTAG module is stalled. If the stalled bit is set, then the select-DR-scan state structure is exited and returned to later. This means a requested transaction is already underway.

If the JTAG module is not stalled, then the second bit is shifted out of the status register to determine whether that transaction has failed. If the failed bit is set, then the select-DR-scan state structure is exited.

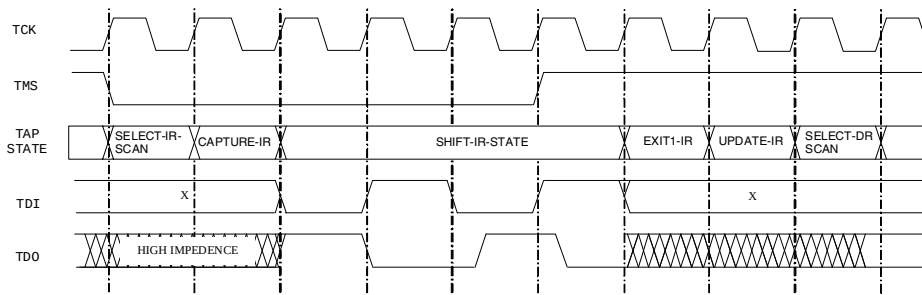
If the failed bit is not set, then the ready bit is shifted out to determine whether the transaction has completed or not. If the ready bit is set, then the transaction has finished and a new transaction can be started. If the ready bit is not set, then the select-DR-scan state structure is exited and the above procedure is repeated.

The PC\_SEL bit is an optional bit that does not need to be shifted out.

## 14.6.3 Setting up a Read Access from the ARC EM Processor or Memory

Setting up a read transaction follows almost the same procedure as setting up a write transaction.

The address register is accessed by writing the four-bit code 1010 into the instruction register. The select-IR-scan state structure is selected, and the code 1010 is serially loaded. When the update-IR state is entered, the instruction register is updated and the address register is selected.

**Figure 14-15 Loading the Instruction Register (Select-Address Register)**

The data register does not need to be accessed at this stage, as a read transaction is being performed.

The last stage of setting up the transaction-parameter registers involves writing to the transaction-command register with the required read transaction.

When all the transaction parameters are set up, the access is started by placing the TAP controller into the run-test/idle state.

To obtain information about the transaction, the JTAG status register is interrogated. Because this is a read-only register, the signal supplied on TDI is ignored when the register contents are shifted out through TDO. The appropriate bitfields are then checked to verify the read transaction. Refer to [“Accessing the Status Register”](#) on page 245 for a standard procedure to decode the JTAG Status register.

When the transaction has completed successfully, the data register is selected. The select-IR-scan state structure is selected and the code 1011 is serially loaded into the instruction register. The select-DR-scan state is then entered and data is shifted out from the selected device.

## 14.7 Actionpoints

The optional actionpoint system allows enhanced debugging of application software in real time. This section describes the architecture of the actionpoint system, and explains how it can be used by a host or native debugger, or to support ROM patching. See the *ARCV2 ISA Programmer's Reference* for details on the auxiliary registers used to configure the behavior of actionpoints.

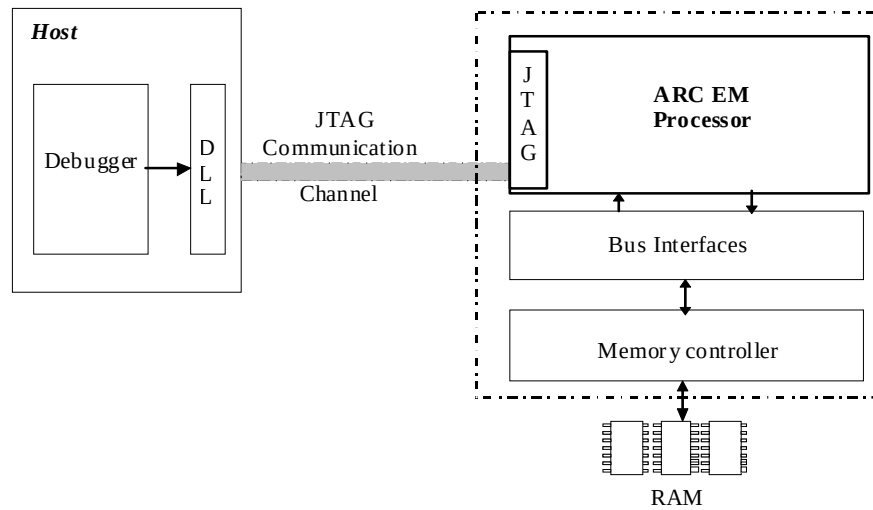
Actionpoints existed in certain previous ARC processors, but this document refers specifically to the actionpoints implementation for the ARC EM processor series.

The actionpoints system is highly configurable and supports the following:

- Monitoring the processor for the occurrence of a specified triggering event, or the coincidence of two or four related events
- Performing one of the following actions when a triggering event is detected:
  - Halting the processor
  - Raising a privilege-violation exception

The host debugger provides software support for the actionpoint mechanism.

[Figure 14-16](#) shows a typical debug system with ARC EM actionpoints.

**Figure 14-16 Debugger and an ARv2EM Processor**

### 14.7.1 Actionpoints Mechanism

The Actionpoint mechanism can be configured to watch for a particular event (or set of events) and then act upon that event when it occurs.

The following are the primary uses for actionpoints.

- Assist in debugging hardware by providing a means to detect the following:
  - Execution of specific instructions
  - Accesses to specific memory locations
  - Accesses to specific auxiliary registers
  - Presence of relevant bit patterns on external input ports
- Assist in patching ROM code by trapping accesses to programmed regions of memory when instructions are fetched from those areas

Actionpoints can be programmed to trigger on memory or auxiliary-register values, or on the address associated with those values, without requiring prior knowledge of the instructions that make the accesses.

The actionpoints mechanism supports breakpoints and watchpoints. A breakpoint is an actionpoint that triggers on a specific instruction, or range of instructions, being executed. A watchpoint is an actionpoint that detects specific address or data values being accessed by memory-referencing instructions, auxiliary-register read (LR) and write (SR) instructions, or the presence of specific values on the two optional external parameter ports.

Both breakpoints and watchpoints can be programmed to either halt the processor or raise an exception.

### 14.7.1.1 Actionpoint Targets

Actionpoints provide a flexible alternative to the traditional approach of a high number of breakpoints with limited functionality. An actionpoint can be set to monitor both data and address values from a number of sources (targets) within the ARC EM processor.

Two levels of functionality can be configured for actionpoints; these are referred to as *minimum* and *full* support.

A minimum configuration of actionpoints supports only those targets that are triggered by addresses. These targets include:

- Instruction addresses
- Memory read or write addresses
- Auxiliary-register addresses used in LR or SR instructions

A full configuration of actionpoints supports targets that are triggered by data values, in addition to all functionality provided by the minimum configuration. A full configuration includes the following targets:

- Instruction values
- Data read or written to memory
- Data read or written to auxiliary registers by LR or SR instructions
- External input-parameter ports

The complete set of possible actionpoint targets is detailed in [Table 14-11](#).

**Table 14-11 Actionpoint Monitoring Targets**

| Target                     | Configuration | Description                                                                                                                                               |
|----------------------------|---------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------|
| Instruction address        | Full, minimum | The address of the current instruction (PC)                                                                                                               |
| Memory address             | Full, minimum | The address of a load or store operation                                                                                                                  |
| Auxiliary-register address | Full, minimum | The address of an auxiliary-register access                                                                                                               |
| Instruction data           | Full          | The current instruction word <i>after it has been aligned</i> . For example, a 16-bit instruction is in the top 16 bits of the word.                      |
| Memory data                | Full          | Data written to the memory or data returned from memory.<br>NOTE: If both read and write are selected for this target, write data are implicitly selected |
| Auxiliary-register data    | Full          | Data written to an auxiliary register or data returned from an auxiliary register.                                                                        |
| External parameter 0       | Full          | An arbitrary 32-bit bus that can be connected to anything that might need to be monitored                                                                 |
| External parameter 1       | Full          | An arbitrary 32-bit bus that can be connected to anything that might need to be monitored                                                                 |



### 14.7.1.2 Actionpoints and Memory-Bus Masters

Actionpoints do not track memory transactions of external bus mastering devices such as DMA or other processors in a multiprocessor configuration.

### 14.7.1.3 Actionpoint Types

The ARC EM system can be configured with 2, 4, or 8 actionpoints, each of which has common functionality. Each actionpoint can be programmed to operate independently, or they can be programmed to operate in pairs or quads. This allows two or four arbitrary targets to be combined to create more complex triggering conditions.

When an actionpoint halts the processor, it also sets the additional Actionpoint Hit (AH) flag in the DEBUG register. This mechanism allows an external debug host to determine what caused the processor to halt. The AH flag is cleared when the halt flag (H) in the STATUS32 register is cleared, for example, by restarting or single-stepping the processor.

An actionpoint may be programmed to raise a privilege-violation exception when it triggers. This redirects execution through the EV\_PrivilegeV vector, with the Exception-Cause Register (ECR) set to 0x000702nn, where nn indicates the number of the actionpoint that triggered.

## 14.7.2 Overview of Actionpoint Auxiliary Registers

Actionpoints are programmed using a set of control and status registers mapped into the auxiliary-register space. [Table 14-12](#) provides an overview of the registers; see the *ARCV2 ISA Programmer's Reference* for complete details.

**Table 14-12 Auxiliary Registers for Actionpoints**

| Address | Register Type | Register Name | Description                              |
|---------|---------------|---------------|------------------------------------------|
| 0x05    | RG            | DEBUG         | Extensions to debug register             |
| 0x76    | R             | AP_BUILD      | Actionpoint build-configuration register |
| 0x220   | RW            | AP_AMV0       | Actionpoint 0 match value                |
| 0x221   | RW            | AP_AMM0       | Actionpoint 0 match mask                 |
| 0x222   | RW            | AP_AC0        | Actionpoint 0 control                    |
| 0x223   | RW            | AP_AMV1       | Actionpoint 1 match value                |
| 0x224   | RW            | AP_AMM1       | Actionpoint 1 match mask                 |
| 0x225   | RW            | AP_AC1        | Actionpoint 1 control                    |
| 0x226   | RW            | AP_AMV2       | Actionpoint 2 match value                |
| 0x227   | RW            | AP_AMM2       | Actionpoint 2 match mask                 |
| 0x228   | RW            | AP_AC2        | Actionpoint 2 control                    |
| 0x229   | RW            | AP_AMV3       | Actionpoint 3 match value                |
| 0x22A   | RW            | AP_AMM3       | Actionpoint 3 Match Mask                 |

**Table 14-12 Auxiliary Registers for Actionpoints**

| Address | Register Type | Register Name | Description                |
|---------|---------------|---------------|----------------------------|
| 0x22B   | RW            | AP_AC3        | Actionpoint 3 control      |
| 0x22C   | RW            | AP_AMV4       | Actionpoint 4 match value  |
| 0x22D   | RW            | AP_AMM4       | Actionpoint 4 match mask   |
| 0x22E   | RW            | AP_AC4        | Actionpoint 4 control      |
| 0x22F   | RW            | AP_AMV5       | Actionpoint 5 match value  |
| 0x230   | RW            | AP_AMM5       | Actionpoint 5 match mask   |
| 0x231   | RW            | AP_AC5        | Actionpoint 5 control      |
| 0x232   | RW            | AP_AMV6       | Actionpoint 6 match value  |
| 0x233   | RW            | AP_AMM6       | Actionpoint 6 match mask   |
| 0x234   | RW            | AP_AC6        | Actionpoint 6 control      |
| 0x235   | RW            | AP_AMV7       | Actionpoint 7 match value  |
| 0x236   | RW            | AP_AMM7       | Actionpoint 7 match mask   |
| 0x237   | RW            | AP_AC7        | Actionpoint 7 control      |
| 0x23F   | R             | AP_WP_PC      | Watchpoint program counter |

The access code definitions of these registers are explained in [Table 14-13](#). All bullet entries “•” indicate cases where permission for the requested operation (read or write) is granted in the given operating mode (user, kernel). The Debug Host column indicates the read and write accessibility available for the Debug Host for each level of access rights.

**Note**

Access to auxiliary registers through side-effect, such as altering flags in the STATUS32 register, is governed by the privilege requirements of any instruction that makes an implicit read or write to an auxiliary register. [Table 14-13](#) refers only to the following:

- Reads through the LR instruction
- Writes through the SR instruction
- Reads or writes from the Debug host interface

In Table 14-13, “P” indicates that any attempt to perform the requested operation raises a Privilege Violation exception; “I” indicates that the requested operation raises an Illegal Instruction exception; “-” indicates that the requested operation is ignored. In such cases, a read returns 0, and a write has no effect.

**Table 14-13 Key to Auxiliary Register Access Permissions for LR and SR Instructions**

| Access | User Mode |       | Kernel Mode |       | Debug Host |       |
|--------|-----------|-------|-------------|-------|------------|-------|
|        | Read      | Write | Read        | Write | Read       | Write |
| r      | •         |       | •           |       | •          | -     |
| R      | P         |       | •           |       | •          | -     |
| w      |           | •     |             | •     | -          | •     |
| rw     | •         | •     | •           | •     | •          | •     |
| W      |           | P     |             | •     | -          | •     |
| rW     | •         | P     | •           | •     | •          | •     |
| RW     | P         | P     | •           | •     | •          | •     |
| rG     | •         |       | •           |       | •          | •     |
| RG     | P         |       | •           |       | •          | •     |

#### 14.7.2.1 Access Requirements

All actionpoint registers are accessible only in kernel mode; accessing these registers in user mode raises a privilege-violation exception.

| Vector Name   | Vector Offset | Exception-Cause Register |
|---------------|---------------|--------------------------|
| EV_PrivilegeV | 0x1C          | 0x000702nn               |

If fewer than eight actionpoints are in the system, only the associated registers exist. Accessing non-existent actionpoint registers raises an instruction-error exception in both kernel and user modes.

| Vector Name       | Vector Offset | Exception Cause Register |
|-------------------|---------------|--------------------------|
| Instruction Error | 0x08          | 0x00020000               |

#### 14.7.2.2 Actionpoint Registers Host and Core Access

If both the processor core and host attempt to write to an actionpoint register at the same time, the host write always takes priority. To ensure that any write by the processor core in such situation is not lost, it is recommended to have a control-access procedure for the host and processor core.

When the core is running, reconfigure enabled actionpoints carefully, as reconfiguration can result in undefined behavior. The following is the best procedure:

1. The host halts the core.
2. The host checks the ASR status field and AH bit of the DEBUG register to see whether an actionpoint triggered during the halt.
3. The host takes appropriate action if necessary.

Following this, the host can reconfigure actionpoints, and then restart the processor.

### 14.7.2.3 Actionpoint Registers Reset State

The actionpoint registers are set to the values shown in [Table 14-14](#) when the processor is reset.

**Table 14-14 State of Auxiliary Registers on Reset**

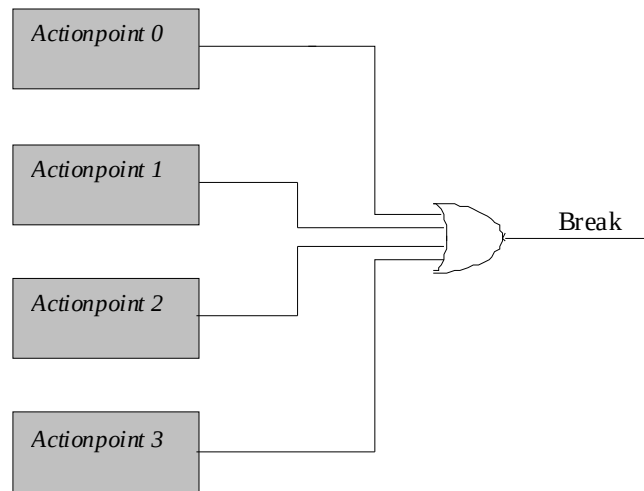
| Register Name      | Initial Reset Values                                                                                                                                                                                                                                                                |
|--------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| DEBUG              | 0x00000000 for 2, 4 or 8 actionpoints                                                                                                                                                                                                                                               |
| AP_AMV0 to AP_AMV7 | 0x00000000 for 2, 4 or 8 actionpoints                                                                                                                                                                                                                                               |
| AP_AMM0 to AP_AMM7 | 0x00000000 for 2, 4 or 8 actionpoints                                                                                                                                                                                                                                               |
| AP_AC0 to AP_AC7   | 0x00000000 for 2, 4 or 8 actionpoints                                                                                                                                                                                                                                               |
| AP_BUILD           | 0x00000004 = 2 actionpoints, full targets<br>0x00000104 = 4 actionpoints, full targets<br>0x00000204 = 8 actionpoints, full targets<br>0x00000404 = 2 actionpoints, minimum targets<br>0x00000504 = 4 actionpoints, minimum targets<br>0x00000604 = 8 actionpoints, minimum targets |

## 14.7.3 Programming Actionpoints

This section describes the process of programming actionpoints through the auxiliary-register mechanism. The enabling or disabling of actionpoints is supported by the debugger.

Actionpoints can be programmed by writing to their respective actionpoint control registers (AP\_AC0 to AP\_AC7), these registers are read/write registers mapped into the auxiliary-register space. The actionpoint registers are designed to be accessed by an external software debugger through the ARC EM processor's host interface.

A design can contain 2, 4, or 8 actionpoints. An example of a system with four actionpoints is shown in [Figure 14-17](#) on page 253.

**Figure 14-17 System with Four Actionpoints**

Actionpoints are managed by the debugger. All actionpoints are disabled upon reset.

The debugger can determine the number of actionpoints in the processor by reading the actionpoint build-configuration register.

#### 14.7.3.1 Basic Properties

Each actionpoint has a mask-and-compare operation in which the value of the target is compared with the value in the AP\_AMV $n$  register, and the AP\_AMM $n$  register masks some bits out of the comparison. This operation is shown in the following equation:

$$\text{actionpoint hit} = ((\text{target\_value} \mid \text{AP\_AMM}n) == (\text{AP\_AMV}n \mid \text{AP\_AMM}n))$$

By masking some bits out of the comparison, it is possible to specify a range of target values on which the actionpoint is triggered. This mechanism is used by the debugger, for example, to watch accesses to an array. You cannot precisely specify arbitrary ranges with a mask operation. However, the debugger uses the approximate range to reduce the number of times the processor is halted to check accesses to the array.

Each actionpoint can be combined in pairs or quads, where all the actionpoints that have been combined must meet their match conditions simultaneously. This feature can be used, for example, to more tightly specify a range or to trigger on writing a particular value to a particular auxiliary register.

When the mode bit of an actionpoint is set, the sense of the comparison is reversed. For example, the following values set actionpoint 0 to trigger whenever a read access to memory with an address outside the range 0x12345000 - 0x12345FFF occurs:

AP\_AMV0 = 0x12345678

AP\_AMM0 = 0x00000FFF

AP\_AC0 = 0x62

#### 14.7.3.2 Setting an Actionpoint

To set an actionpoint, follow these steps:

1. Determine how many actionpoints are available in the system by reading the actionpoint build configuration register, AP\_BUILD.
2. Write the data value for the comparison value to the actionpoint-match value register AP\_AMV $n$ .
3. Write the mask to be used for the comparison to the actionpoint-match mask register AP\_AMM $n$ .
4. Write the appropriate settings to the actionpoint control register AP\_AC $n$ .
5. Run the ARC EM processor normally until it halts or completes the test run.
6. Determine the reason for the processor halting by viewing the contents of the DEBUG register using the debugger
7. If a range was specified with the actionpoint-match mask register, read the contents of the actionpoint-match value register AP\_AMV $n$  to determine the value that caused the trigger.

Multiple actionpoints can be programmed to monitor various conditions. Repeat [steps 2, 3, and 4](#) for each additional actionpoint.

#### Example 14-1 Setting an Actionpoint to Break when PC = 0x00001234

| Step | Action                                               |
|------|------------------------------------------------------|
| 1    | Write 0x00001234 to actionpoint match value register |
| 2    | Write 0x00000000 to actionpoint match mask register  |
| 3    | Write 0x00000020 to actionpoint control register     |

#### 14.7.3.3 Detecting and Clearing an Actionpoint

If an actionpoint halts the processor, the actionpoint match value register, AP\_AMV $n$ , is updated with the corresponding target value that caused the halt. The DEBUG register is also updated to indicate that actionpoints halted the processor. The ASR field indicates which actionpoints caused the halt.

To reset an actionpoint, the debugger must disable the actionpoint by setting the Transaction Type (TT) field in the AP\_AC $n$  register to 0b00. If the conditions that triggered an actionpoint are still present when the processor is restarted after a halt, the actionpoint is triggered again immediately.

#### 14.7.3.4 Action on Triggering

When an actionpoint is triggered, it can do either of the following:

- Halt the processor (break), which is mainly used for host-debugger support.
- Trigger a privilege-violation exception, for implementing ROM patching or native debugging under the control of an operating system.

The AA bit controls the behavior of an actionpoint in the respective control register AP\_AC $n$ . The debugger should be aware that a latency is associated with the event occurring and the actionpoint being triggered.

#### 14.7.3.5 Actionpoints and Long-Immediate Data

If an actionpoint is targeting an instruction address and is set on a long immediate data value (limm), it is never triggered because the instruction address only points to valid instructions rather than limm fields.

### 14.7.3.6 Actionpoints and Instruction-Address Ranges

Actionpoints that trigger on instruction addresses compare the actionpoint-match value with the value in the PC before each instruction is executed. As the PC always points to the start address of an instruction, an exact match can only occur if the actionpoint-match value also points to the start address of an instruction. An imprecise match can occur if an actionpoint is set to match on a range of values, and that range includes the current PC.

### 14.7.3.7 Breakpoint Triggering Latency

A breakpoint actionpoint always triggers before the triggering instruction has executed. This is the case regardless of whether the associated action is to halt the processor or raise a privilege-violation exception.

If a breakpoint causes the processor to halt, then a host debugger sees the current PC pointing at the triggering instruction. It may then disable the actionpoint and restart the processor to continue execution.

If a breakpoint causes the processor to raise an exception, then the exception return address assigned to ERET is the address of the triggering instruction. Software may then take whatever action is required, in the knowledge that the triggering instruction did not execute.

### 14.7.3.8 Watchpoint Triggering Latency

A watchpoint actionpoint always triggers after the triggering instruction has executed. This is the case regardless of whether the associated action is to halt the processor or raise a privilege-violation exception. Due to the delay involved in detecting the triggering of watchpoints, a small number of additional instructions after the triggering instruction might also be executed before the watchpoint action is taken by the processor. In the ARC EM processor, the number of additional instructions executed between the triggering instruction and the triggering action may be one or two.

### 14.7.3.9 Adjacent Watchpoints

If two successive instructions both trigger a watchpoint, then the detection of the second watchpoint is not guaranteed. This is due to the inherent delay in detecting watchpoints, which means that both triggering instructions might complete before the first watchpoint takes effect.

The AP\_WP\_PC auxiliary register can be used in conjunction with the PC (AA = 0) or ERET (AA = 1) to determine the location of the triggering instruction in relation to the point at which the processor responded to the watchpoint.

For example, if all actionpoints are programmed to raise exceptions (AA = 1), and if the PC is positioned two instructions after AP\_WP\_PC on entering a watchpoint exception handler, then software can deduce that one additional instruction was committed after the triggering instruction. If an actionpoint would also have triggered on that instruction, then that second exception might not be detected on return from the first exception.

### 14.7.3.10 Coincident Breakpoints and Watchpoints

If two actionpoints are triggered by the same instruction, and the action of one is to cause a breakpoint and the other is to cause a watchpoint, the following sequence takes place:

- The processor first takes the action associated with the breakpoint, and the current PC remains pointing at the triggering instruction.

- When the host restarts the processor ( $AA = 0$ ), or the processor returns from its exception ( $AA = 1$ ), if the breakpoint halt has been disabled, the watchpoint is detected, and its exception takes effect. If the breakpoint has not been disabled it is triggered again.

#### 14.7.3.11 Coincident Halt and Exception Watchpoint Actions

Only one watchpoint is actioned when multiple watchpoints trigger on the same instruction, and in the case of pairs or quads it is the watchpoint with the lowest-numbered master.

If two watchpoints are triggered by the same instruction, then the lowest numbered watchpoint is taken, and the action is determined solely by that watchpoint. If the action is to halt the processor, then after a restart the processor does not respond to any other watchpoints that might have been triggered at the same time. Similarly, if the action is to raise an exception, then any other watchpoints that might have been triggered at the same time are not triggered on return from the exception.

#### 14.7.3.12 Simultaneous Independent Breakpoints and Watchpoints

It is possible for a watchpoint on instruction  $N$  to be triggered at the same time as a breakpoint on instruction  $N+M$ , where  $M$  is a small, implementation-dependent value defined by the watchpoint-triggering latency. When simultaneous independent actionpoints are triggered, the lowest-numbered actionpoint master is given priority. If that actionpoint is the breakpoint, then an actionpoint on instruction  $N+M$  is triggered before the watchpoint on instruction  $N$  – they happen in reverse order. Care is therefore needed when programming a mixture of breakpoints and watchpoints that are close to each other in the program.

For the ARC EM, the value of  $M$  is not greater than two.

### 14.7.4 Paired Actionpoints

The pair-mode option can be enabled ( $P = 1$ ) to perform detection of simultaneous events of two actionpoints paired together. The following two modes are available when using the pairing feature:

- *Pair mode* - Actionpoints are valid when both actionpoints are true at the same time. In pair mode, both paired actionpoints have to be true on the same cycle for the processor to be halted.
- *Independent mode* - In this mode, the actionpoint is set independently of the other actionpoints.

#### 14.7.4.1 Actionpoints in Pair Mode

In pair mode, the actionpoints can be set to operate so that references to a particular location are combined with the data contained in that location. This can be a write to memory, core or auxiliary registers. The pairing of actionpoints means that these events occur simultaneously.

For example, two actionpoints combined together in this manner require that the first actionpoint in the sequence be configured to monitor access to a specified address (memory, core, or auxiliary). The second actionpoint is treated as the data value at that location depending upon the setting in the Actionpoint Target (AT) field of the AP\_ACn register.

This allows the following scenarios to be programmed:

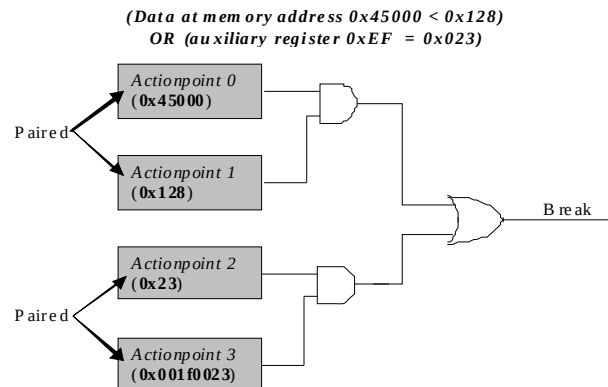
- Actionpoint 0 set to trigger on auxiliary-register address 0x22, with actionpoint 1 set to trigger on auxiliary-register data 0x5678.
- Actionpoint 0 set to trigger on memory-address 0x1000, with actionpoint 1 set to trigger on memory-data 0xFFFF.



Another example of actionpoints using pair mode is shown in [Figure 14-18](#) on page 257. The figure shows the following scenario:

- Actionpoint 0 triggers on memory address 0x45000, with actionpoint 1 triggering on memory data write of 0x00000128, OR
- Actionpoint 2 triggers on auxiliary-register address 0x23, with actionpoint 3 triggering on auxiliary-register data write 0x001F0023.

**Figure 14-18 Paired-Actionpoints Example**



This example uses actionpoints in pairs set to break when memory-address 0x45000 written with 0x00000128, or auxiliary-register 0x23 is written with 0x001F0023.

| Step | Action                                                                                                                                               |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.   | Write 0x00045000 to actionpoint-match value AP_AMV0                                                                                                  |
| 2.   | Write 0x00000000 to actionpoint-match mask AP_AMM0                                                                                                   |
| 3.   | Write 0x00000092 to actionpoint control AP_AC0 (set actionpoint 0 in pair mode with actionpoint 1; monitor writes on LD/ST address bus)              |
| 4.   | Write 0x00000128 to actionpoint-match value AP_AMV1                                                                                                  |
| 5.   | Write 0x00000000 to actionpoint-match mask AP_AMM1                                                                                                   |
| 6.   | Write 0x00000013 to actionpoint-control AP_AC1 (set actionpoint 1 to monitor writes on LD/ST data bus)                                               |
| 7.   | Write 0x00000023 to actionpoint-match value AP_AMV2                                                                                                  |
| 8.   | Write 0x00000000 to actionpoint-match mask AP_AMM2                                                                                                   |
| 9.   | Write 0x00000094 to actionpoint-control AP_AC2 (set actionpoint 2 in pair mode with actionpoint 3, monitor writes on auxiliary register address bus) |
| 10.  | Write 0x001F0023 to actionpoint-match value AP_AMV3                                                                                                  |
| 11.  | Write 0x00000000 to actionpoint-match mask AP_AMM3                                                                                                   |

| Step | Action                                                                                                                                                                                                                                |
|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 12.  | Write 0x00000015 to actionpoint-control AP_AC3<br>(set actionpoint 3 to monitor writes on auxiliary-register data bus)                                                                                                                |
| 13.  | Run the ARC EM processor normally until it halts or completes the test run.                                                                                                                                                           |
| 14.  | Determine the reason for the ARC EM processor halting, by viewing the contents of the DEBUG register using the debugger. This step provides information about the type of halt (breakpoint or self-halt), and the actionpoint number. |

**Note**

This mechanism should only be used for combining targets that are related, for example, data and address comparisons to a single destination (either auxiliary register or memory), as in a write to memory 0x10000 and data = 0x1234. Unrelated targets do not give meaningful results. The master actionpoint (the actionpoint that has the pair-mode bit set) is always chosen as the stopping mechanism for the processor. Follow the correct restart sequence for the actionpoint action-field bit AA as described in the *ARCV2 ISA Programmer's Reference* and “[Processor-Halt Actionpoints](#)” on page 260.

### 14.7.5 Independent-Mode Actionpoints

This mode treats each actionpoint as an independent entity.

The following are some example scenarios:

- Actionpoint 0 set to trigger on write to any auxiliary register, *or* actionpoint 1 set to trigger on PC = 0x400
- Actionpoint 0 set to trigger on PC value of 0x1000, *or* actionpoint 1 set to trigger on write to any auxiliary registers, *or* actionpoint 2 set to trigger on memory address 0x5000

An example of this mode of operation is provided in [Example 14-2](#).

#### Example 14-2 Setting Independent Actionpoints to Break on Write to an Auxiliary Register or if PC = 0x400

| Step | Action                                                                                                                                                                                                                        |
|------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.   | Write 0xFFFFFFFF to actionpoint-match mask AP_AMM0 (ignore all bits for comparison).                                                                                                                                          |
| 2.   | Write 0x00000014 to actionpoint-control AP_AC0 (trigger on writes to auxiliary registers).                                                                                                                                    |
| 3.   | Write 0x00000400 to actionpoint-match value AP_AMV1.                                                                                                                                                                          |
| 4.   | Write 0x00000000 to actionpoint-match mask AP_AMM1.                                                                                                                                                                           |
| 5.   | Write 0x00000020 to actionpoint-control register (set actionpoint 1).                                                                                                                                                         |
| 6.   | Run the processor normally until it halts or completes the test run.                                                                                                                                                          |
| 7.   | Determine the reason for the processor halting by viewing the contents of the DEBUG register using the debugger. This step provides information about the type of halt (breakpoint or self-halt), and the actionpoint number. |

## 14.7.6 Quad Actionpoints

You can activate quad mode by setting the Q bit to 1 on the first of four available actionpoints. In this mode, the actionpoint with the Q bit set and the next three actionpoints must all match for the actionpoint to trigger. The affected actionpoints are not triggered independently.

For example, you can use actionpoints to write to a memory address in the range 0x1000 - 0x123F.

This range can be divided into four ranges:

- Actionpoint 0 detects 0x1000 - 0x13FF in range.
- Actionpoint 1 detects 0x1300 - 0x13FF out of range.
- Actionpoint 2 detects 0x1280 - 0x12FF out of range.
- Actionpoint 3 detects 0x1240 - 0x127F out of range.

Actionpoint 0 is only triggered if the memory address is in the range 0x1000 - 0x13FF but outside the other three ranges.

[Example 14-3](#) sets quad actionpoints to break on a memory address in the range 0x1000 - 0x123F.

### Example 14-3 Setting Quad Actionpoints

| Step | Action                                                                                                                                                                 |
|------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 1.   | Write 0x00001000 to actionpoint-match value AP_AMV0.                                                                                                                   |
| 2.   | Write 0x000003FF to actionpoint match-mask AP_AMM0.                                                                                                                    |
| 3.   | Write 0x00000212 to actionpoint-control AP_AC0 (set actionpoint 0 in quad mode with actionpoint 1, actionpoint 2, actionpoint 3; monitor writes on LD/ST address bus). |
| 4.   | Write 0x00001300 to actionpoint-match value AP_AMV1.                                                                                                                   |
| 5.   | Write 0x000000FF to actionpoint-match mask AP_AMM1.                                                                                                                    |
| 6.   | Write 0x00000052 to actionpoint-control AP_AC1 (set actionpoint 1 to monitor writes on LD/ST address bus, match when out of range).                                    |
| 7.   | Write 0x00001280 to actionpoint-match value AP_AMV2.                                                                                                                   |
| 8.   | Write 0x0000007F to actionpoint-match mask AP_AMM2.                                                                                                                    |
| 9.   | Write 0x00000052 to actionpoint-control AP_AC2 (set actionpoint 2 to monitor writes on LD/ST address bus, match when out of range).                                    |
| 10.  | Write 0x00001240 to actionpoint-match value AP_AMV3.                                                                                                                   |
| 11.  | Write 0x0000003F to actionpoint match-mask AP_AMM3.                                                                                                                    |
| 12.  | Write 0x00000052 to actionpoint-control AP_AC3 (set actionpoint 3 to monitor writes on LD/ST address bus, match when out of range).                                    |
| 13.  | Run the ARC EM processor normally until it halts or completes the test run.                                                                                            |

**Example 14-3 Setting Quad Actionpoints**

| Step | Action                                                                                                                                                                                                                               |
|------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| 14.  | Determine the reason for the ARC EM processor halting, by viewing the contents of the DEBUG register using the debugger. This step provides information about the type of halt (breakpoint or self-halt), and the actionpoint number |
| 15.  | Read register AP_AMV0 to determine which address within the range caused the trigger.                                                                                                                                                |

**14.7.7 Processor-Halt Actionpoints**

You can activate this mode by setting the AA bit of the actionpoint-control register to 0 for the actionpoint.

When an actionpoint triggers, the following occurs:

- The processor is halted.
- The AH bit of the debug register is set to indicate that the halt is because of an actionpoint.
- The ASR field of the debug register is set to show which actionpoint is triggered to cause the halt.
- The AMV register of the triggering actionpoint is updated to hold the value that caused a hit.

For other targets, restart the processor core from where the processor halted, without invalidating instruction cache.

If the processor has halted, it can be restarted by clearing the H bit in STATUS32.



**Note** If the host or debugger restarts a processor that halted due to a breakpoint without disabling the breakpoint, the breakpoint immediately triggers again on restart.

**14.7.8 Software-Exception Actionpoints**

You can activate this mode by setting the AA bit of the actionpoint-control register to 1 for the actionpoint.

This mode is intended to be used for ROM patching and OS-assisted live debugging.

Actionpoints in this mode are not triggered when the AE bit of the STATUS32 register is set. This means that they do not cause a double exception leading to a machine check.

**14.7.8.1 ROM Patching through Software-Exception Actionpoints**

ROM patching is supported through the exception-actionpoints mechanism. In ROM patching, a breakpoint is set to trigger on the instruction address to be patched, causing an exception. The EV\_PrivilegeV vector points to the patch routine. The end of the routine is followed by either a standard interrupt return (to insert additional instructions) or a jump to a new resume location achieved by altering ERET (to skip erroneous instructions).

Steps to perform ROM patching:

1. Set EV\_PrivilegeV vector to jump to a privilege-violation exception handler.

2. Have the privilege-violation exception handler check that the cause of the exception is due to an actionpoint hit before jumping to the patch routine (as several other possible causes of a privilege violation also exist).
3. Program an actionpoint to cause an exception at the relevant instruction location.

When the PC value causes an actionpoint hit, an exception is raised, causing control to jump through the EV\_PrivilegeV exception vector to the handler and hence to the patch routine.

4. In the patch routine, execute whatever additional or replacement instructions are necessary. If you have multiple patches, examine the *nm* bits of the exception-cause register (ECR) to determine which actionpoint triggered and which patch code should be executed.

To return from the patch routine, do the following:

1. If the patch routine made any changes to ZNCV that must be preserved, transfer those changes to ERSTATUS, so that they are restored on exit from the exception.
2. Alter ERET to point to an instruction after the triggering instruction. This instruction might be the next instruction in sequence or a completely different instruction, for example, to skip a larger section of problematic code.
3. Issue an RTIE instruction



Breakpoints (not watchpoints) must always be used for ROM patching, as this ensures that the instruction on which the actionpoint is set will not be executed.

#### 14.7.8.2 Debugging an Operating System with Software-Exception Actionpoints

To debug an operating system, do the following:

1. Set the EV\_PrivilegeV vector to jump to the debugger exception handler.
2. Use the debugger to set an actionpoint on an instruction or action where you wish the debugger to take control.
3. Run or step the program. When the Architectural PC reaches the instruction, or the action conditions are met, a privilege-violation exception is raised causing control to jump to the debugger exception handler.
4. The debugger exception handler performs whatever actions are necessary (for example, reading registers), and then removes the actionpoint.
5. If the actionpoint is set to trigger on an instruction address or data, remove it before RTIE is executed; otherwise the processor continually retriggers the actionpoint and reenters its exception handler.
6. Execute an RTIE to return control to the original instruction, which now executes.

#### 14.7.9 Actionpoints on External Parameters

Two targets called external parameters are provided. These 32-bit buses can be connected to any arbitrary data source for monitoring, for example an external memory interface or a physical sensor.

In a standard build with actionpoints, these signals are tied to fixed values in the `misc.v` HDL file so that their connectivity can be tested.

The following table describes the signals:

**Table 14-15 Actionpoint External Parameter Signals**

| Signal                             | Default Value            | Description                                                                                                                                                                                                           |
|------------------------------------|--------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ap_param0[31:0]<br>ap_param1[31:0] | 0xDEADBEEF<br>0xC0FFEE23 | Bus to connect to parameter being monitored                                                                                                                                                                           |
| ap_param0_read<br>ap_param1_read   | 0<br>0                   | Indicates that this is a read operation. For some parameters, this signal has no real meaning and is just used as a qualifier for the monitored parameter and to determine which actionpoint mode would be triggered. |
| ap_param0_write<br>ap_param1_write | 0<br>0                   | Indicates that this is a write operation. As with ap_paramN_read, this signal might have no real meaning and is simply a qualifying signal.                                                                           |

## 14.8 SmaRT

Small real time trace (SmaRT) is an optional on-chip debug hardware component that captures instruction-trace history.

SmaRT stores the address of the most recent non-sequential instructions executed.

The MetaWare debugger enables SmaRT and displays the instruction trace history. The MetaWare debugger reads the saved instruction trace information via the JTAG port when the processor is halted.

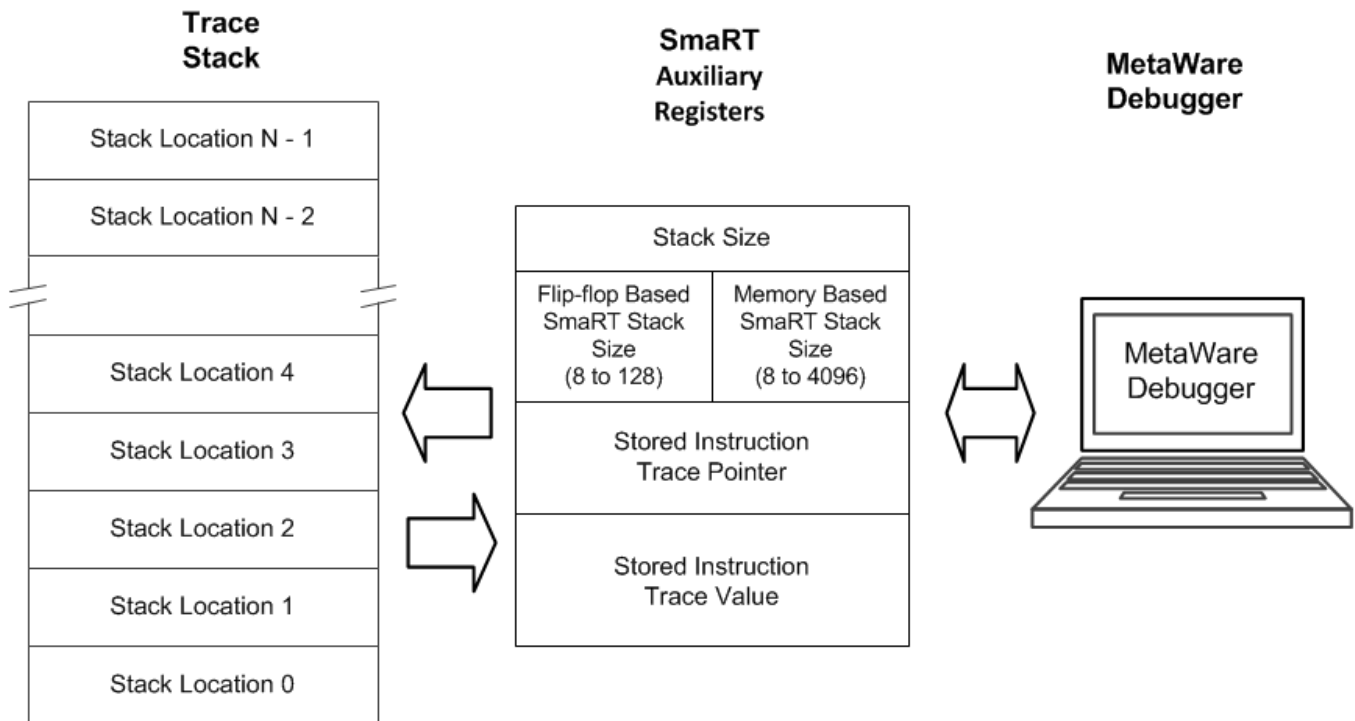
For more information about the programming model, see the *ARCo2 ISA Programmer's Reference*

### 14.8.1 Features

- Different stack sizes available — from 8 up to 4,096 entries — to record program-flow changes.
- Stores all non-sequential instructions, including jumps, branches, interrupts, exceptions, and loops. Multiple branches to the same location (inner loop) are stored as a single entry to maximize stack usage.
- Power-saving features include gating the system clock and zeroing all inputs when not in trace mode.
- The MetaWare debugger enables SmaRT and displays trace information, eliminating the need for a separate, dedicated trace port.

## 14.8.2 Overview of SmaRT Operation

Figure 14-19 SmaRT Operation



- Trace Stack - stores source and destination addresses whenever a change in program flow is detected by the core
- SmaRT Auxiliary Registers:
  - SMART\_BUILD – shows the configured size of the trace stack
  - SMART\_CONTROL – controls SmaRT operation and points to a location in the trace stack
  - SMART\_DATA – shows the value of a location in the trace stack (as defined in the SMART\_CONTROL register)
- MetaWare Debugger – enables SmaRT and reconstructs the instruction trace

## 14.8.3 Overview of SmaRT Auxiliary Registers

The SmaRT unit contains the following auxiliary registers:

| Register Address | Name          | Read/Write | Description                        |
|------------------|---------------|------------|------------------------------------|
| 0xFF             | SMART_BUILD   | R          | SmaRT build-configuration register |
| 0x700            | SMART_CONTROL | R/W        | SmaRT control register             |
| 0x701            | SMART_DATA    | R          | SmaRT data register                |

See the *ARCV2 ISA Programmer's Reference* for complete details.

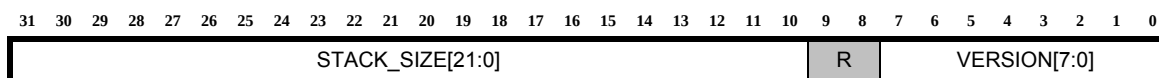
### SMART\_BUILD Configuration Register, SMART\_BUILD

Address: 0xFF

Access: R

The SMART\_BUILD read-only configuration register is at auxiliary address 0xFF and has the following contents:

**Figure 14-20 SMART\_BUILD Register**



The field descriptions are shown in the following table.

| Field            | Description                                                                                                                                                                                                                                                  |
|------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| VERSION[7:0]     | Version of SmaRT component<br>0x03 = Current version with triplets of registers in each location.                                                                                                                                                            |
| STACK_SIZE[21:0] | Trace stack size<br>Indicates the stack size and allows the debugger to identify the capabilities of the implementation for display purposes. Possible values are from 8 up to 4,096. Values above 128 are only available for the SRAM-based implementation. |
| R                | Reserved<br>Returns zero when read.                                                                                                                                                                                                                          |

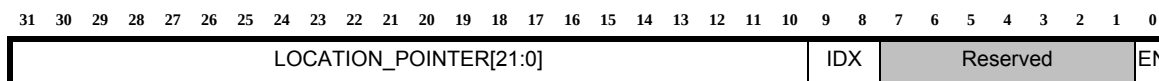
### SMART\_CONTROL Register

Address: 0x700

Access: RW

The SMART\_CONTROL register has the following contents:

**Figure 14-21 SMART\_CONTROL Register**





The field descriptions are shown in the following table.

| Field                  | Description                                                                                                                                                    |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LOCATION_POINTER[21:0] | Stack location pointer<br>Sets the value of the stored instruction stack pointer.                                                                              |
| IDX[1:0]               | Location index<br>Controls which value is returned in the SMART_DATA register.<br>0 = SRC_ADDR value<br>1 = DEST_ADDR value<br>2 = FLAGS value<br>3 = Reserved |
| EN                     | SmaRT enable<br>Enables the trace mechanism.<br>0x0 = Trace Disabled.<br>0x1 = Trace Enabled.                                                                  |
| Reserved               | Reserved<br>Returns zero when reading. Ignored when writing.                                                                                                   |

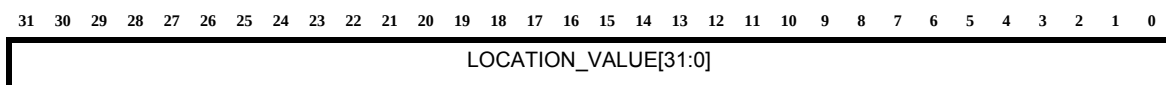
## SMART\_DATA Register

Address: 0x701

Access: R

The SMART\_DATA read-only register has the following contents:

### Figure 14-22 SMART\_DATA Register



The field descriptions are shown in the following table.

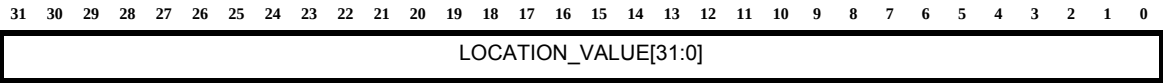
| Field                | Description                                                                                                                                                                          |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LOCATION_VALUE[31:0] | <p>Stored-instruction trace-stack value</p> <p>The value in the stack contained at the location given in the LOCATION_POINTER field and IDX field of the SMART_CONTROL register.</p> |

The SMART\_DATA register may be read safely only when the processor is in the halt state or another state in which SmaRT is not collecting data. Reading this register while SmaRT logic is active might return undefined data.

SRC\_ADDR Value, Index 0

The SRC\_ADDR value is read from the SMART\_DATA register when the IDX field in the SMART\_CONTROL register is set to 0.

Figure 14-23 SRC\_ADDR Value



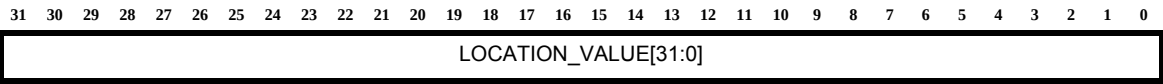
The field descriptions are shown in the following table.

| Field                | Description                                                                                                                                                                                |
|----------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LOCATION_VALUE[31:0] | Stored instruction trace stack location value<br>The value of the source address in the stack contained at the location given in the LOCATION_POINTER field of the SMART_CONTROL register. |

DEST\_ADDR Value, Index 1

The DEST\_ADDR value is read from the SMART\_DATA register when IDX field in the SMART\_CONTROL register is set to 1.

Figure 14-24 DEST\_ADDR Value Register



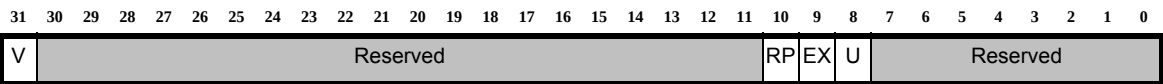
The field descriptions are shown in the following table.

| Field                | Description                                                                                                                                                                                     |
|----------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| LOCATION_VALUE[31:0] | Stored instruction-trace stack-location value<br>The value of the destination address in the stack contained at the location given in the LOCATION_POINTER field of the SMART_CONTROL register. |

FLAGS Value, Index 2

The FLAGS value is read from the SMART\_DATA register when the IDX field in the SMART\_CONTROL register is set to 2. The field descriptions are as follows:.

Figure 14-25 FLAGS Value Register



The field descriptions are shown in the following table.

| Field    | Description                                                                                   |
|----------|-----------------------------------------------------------------------------------------------|
| V        | Valid<br>Indicates that the value in the stack is valid.                                      |
| RP       | Repeat<br>Used to indicate that the same program-flow change occurred more than once.         |
| EX       | Exception<br>Indicates that the program-flow change is a result of an exception or interrupt. |
| U        | User mode<br>Set if the processor is in user mode after program-flow change.                  |
| Reserved | Reserved<br>Returns zero when read.                                                           |

**Reserved value, Index 3** Zero is read from the SMART\_DATA register when IDX field in the SMART\_CONTROL register is set to 3.

#### 14.8.4 Trace-Stack Operation

This section describes the way SmaRT handles the trace stack. The following examples are provided:

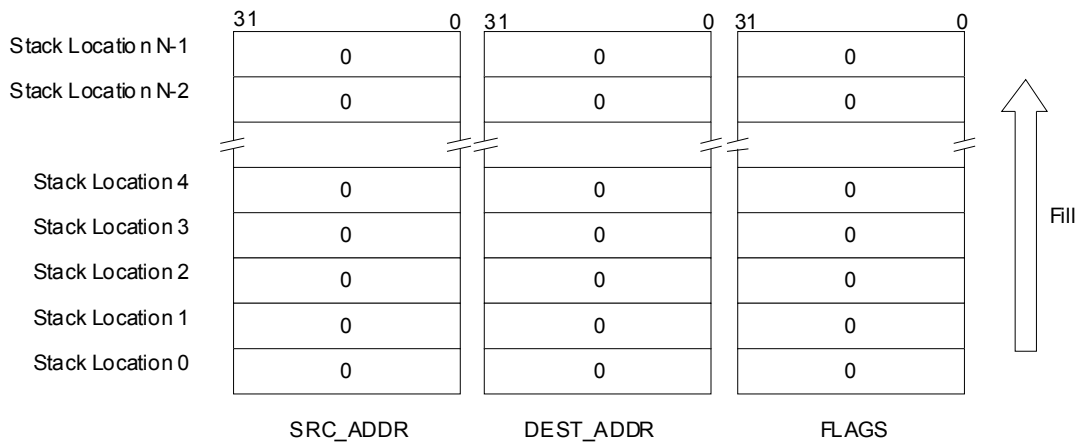
- [Stack Operation with Branch and Jump](#)
- [Stack Operation with Interrupt](#)
- [Stack Operation with Branch from Loop](#)

##### 14.8.4.1 Overview of Trace Stack Operation

The trace format is set up as a triplet of 32 bit registers, as shown in [Figure 14-26](#). Each triplet stores the following:

- The instruction address where a branch, jump, or interrupt happened (source address) in SRC\_ADDR
- The destination instruction address in DEST\_ADDR
- The control flags in FLAGS

The stack can have a total of eight or 32 stack locations. Each location in the stack has a triplet of 32-bit registers. The following examples use eight triplets.

**Figure 14-26 Stack Trace on Reset**

All stack locations are set to zero at reset. After SmaRT is enabled, stack location 0 contains the last instruction address jumped from (in SRC\_ADDR), and the last instruction address gone to (in DEST\_ADDR).

During trace capture the stack records non-contiguous instruction sequences. The stack is filled up from the bottom such that stack location 0 contains the last non-contiguous instruction sequence. The program counter is available in core register r63.

When a change in program flow is detected, the new pair of locations is pushed onto the bottom of the stack and the original entries are moved up. Entries at the top are lost; the processor is not stalled by overflow. Entries are never removed; they may only be read via the auxiliary-register interface. The SmaRT entries can be read when they are not actively being updated by the hardware; this happens when SmaRT tracing is disabled (SMART\_CONTROL.EN = 0) or when the core is halted, or both.

The SmaRT module also stores an entry in the stack when the target of a branch, jump, or conditional loop is the immediately following instruction.

When recording repeated jumps (for example, loops), instead of duplicating entries, the SmaRT module sets the repeat bit (RP) of FLAGS. This prevents loops filling up the stack and losing earlier information.

The SmaRT trace tracks an EI\_S instruction with two entries: the first entry records the address of the EI\_S itself as source address and the address of the target indexed instruction as destination address. The second entry records the address of the indexed instruction as source and the instruction after EI\_S as destination.

BRK, HALT, and SLEEP instructions are not recorded as they do not change the program flow.

#### 14.8.4.2 Stack Operation with Branch and Jump

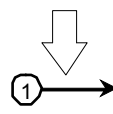
The following example walks through a program and shows when the branch and jump instruction addresses are stored in the stack.

SmaRT stores the address of the delay-slot instruction as the source instruction address, SRC\_ADDR.

## Example 14-4 SmaRT Operation with Branch and Jump

### Step 1

The program has started and reaches instruction address 0x478 (a branch instruction), as shown below.



```

468:  add    %r3,%r3,1
46c:  add    %r2,%r2,48
470:  cmp    %r2,0x1_1c40
478:  b       block_a
47c:  mov    %r2,0x1b90
484:  mov    %r3,1

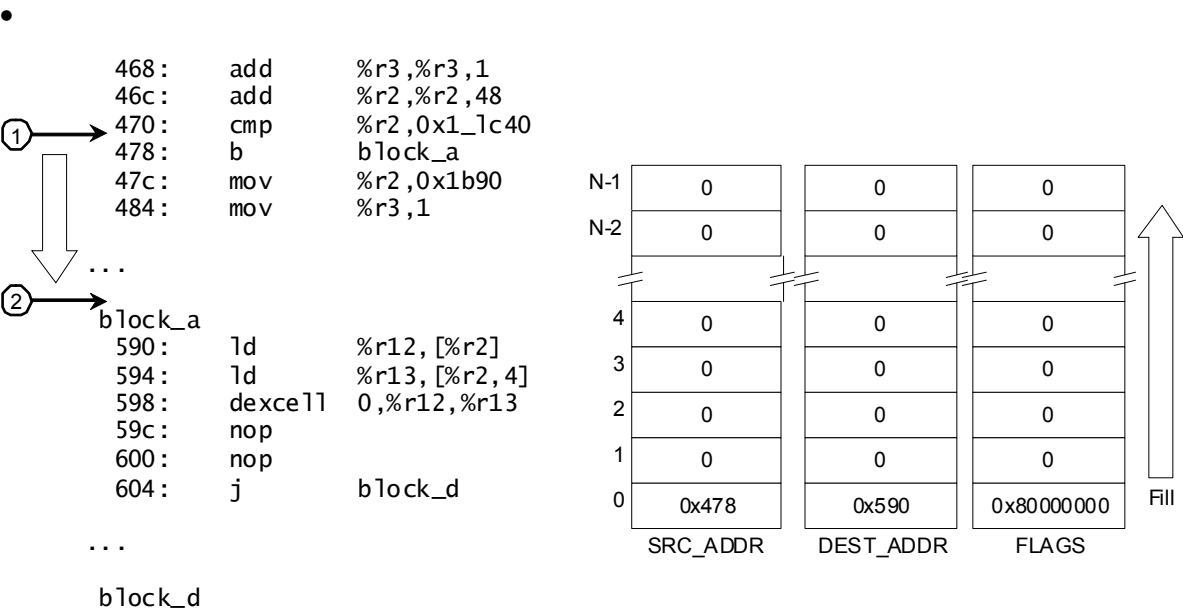
...

block_a
590:  ld      %r12,[%r2]
594:  ld      %r13,[%r2,4]
598:  dextrll 0,%r12,%r13
600:  nop
604:  j       block_d
  
```

### Step 2

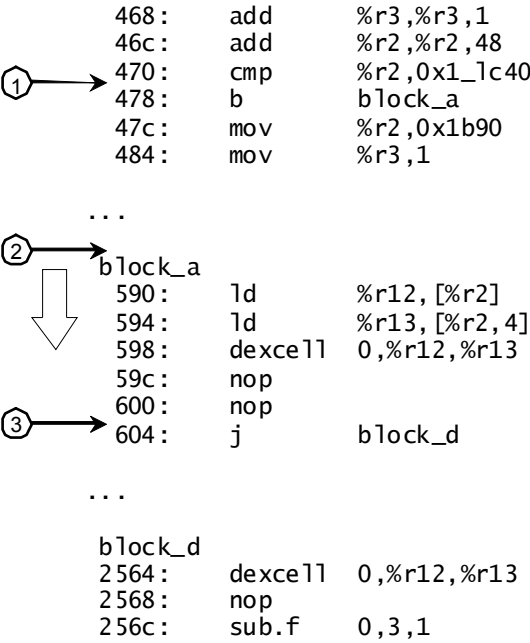
When the branch instruction is executed, the program goes to instruction address 0x590 (block\_a). Because a branch has occurred, the destination and source instruction addresses are saved on the stack.

- Stack location 0, DEST\_ADDR, is set to 0x590 (the destination instruction address).
- Stack location 0, SRC\_ADDR, is set to 0x478 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000000 (indicating a valid entry).



Step 3

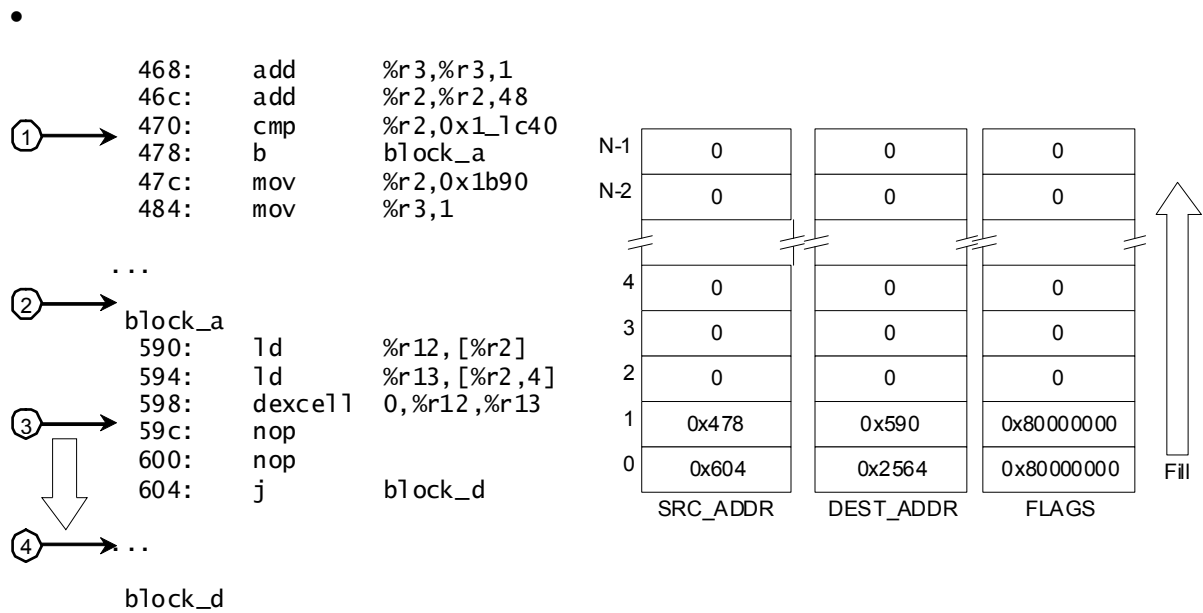
The program continues from address 0x590 to address 0x604 (a jump instruction).



Step 4

When the jump instruction is executed, the program goes to address 0x2564 (block\_d). Because a jump has occurred, the destination and source instruction addresses are saved on the stack, and the previous stack-location values are pushed up the stack.

- Stack location 0, DEST\_ADDR, is updated to 0x2564 (the destination instruction address).
- Stack location 0, SRC\_ADDR, is updated to 0x604 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000000 (indicating valid entry).
- Stack location 1, DEST\_ADDR, is set to 0x590 (the previous destination instruction address).
- Stack location 1, SRC\_ADDR, is set to 0x478 (the previous source instruction address).
- Stack location 1, FLAGS, is set to 0x80000000 (the previous valid entry).



### 14.8.4.3 Stack Operation with Interrupt

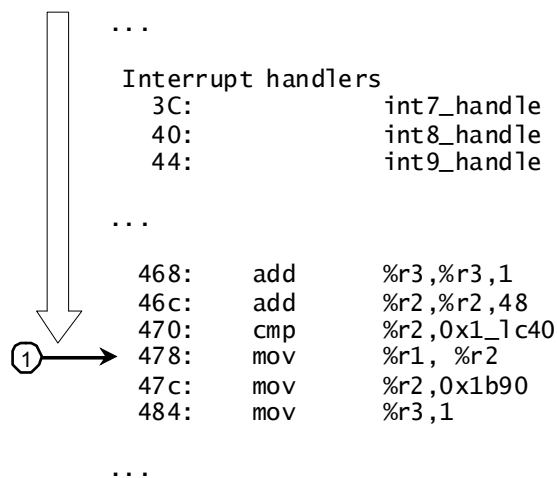
The following example walks through a program and shows how an interrupt is stored in the stack.

For an instruction error exception or SWI instruction, Smart stores the address of the current instruction as the source instruction address, SRC\_ADDR.

#### Example 14-5 Smart Operation with a Vectored Interrupt

##### Step 1

The program has started and reaches instruction address 0x478, as shown below. At this point an interrupt occurs.

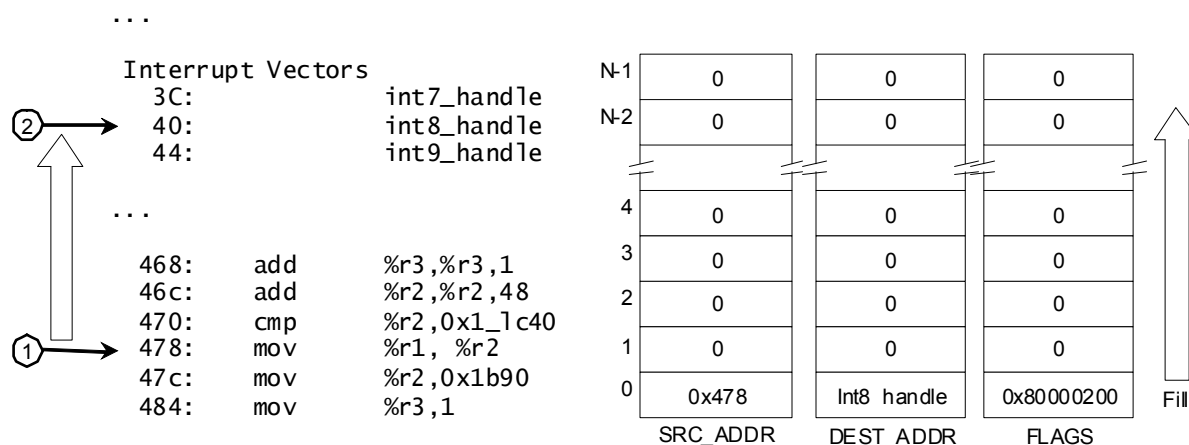


##### Step 2

After the interrupt occurs, the program goes to the interrupt vector table at entry-address 0x40. Because a interrupt has occurred, the destination interrupt handler start address, the source instruction address, and the exception bit (EX) are saved on the stack.

- Stack location 0, DEST\_ADDR, is set to -int8\_handle.
- Stack location 0, SRC\_ADDR, is set to 0x478 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000200 (indicating a valid entry and interrupt).





The interrupt handler redirects the program to int8\_handle.



#### Note

Vector table data is not recorded on the Smart stack. The vectoring operation is only recorded by setting bit 9 of the FLAGS register for this entry.

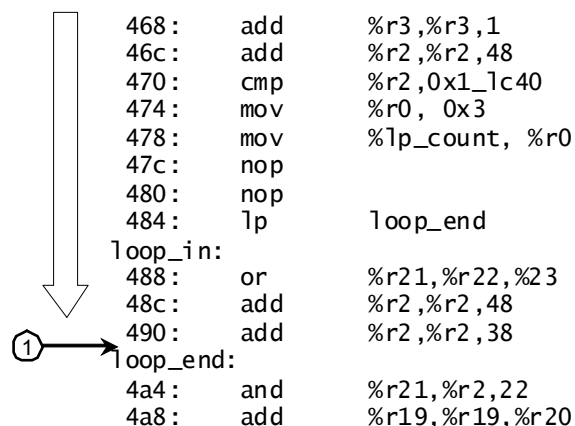
#### 14.8.4.4 Stack Operation with Loops

The following example program has a loop that is executed three times. This example shows when the important instruction addresses are stored in the stack.

#### Example 14-6 Smart Operation with Loops

##### Step1

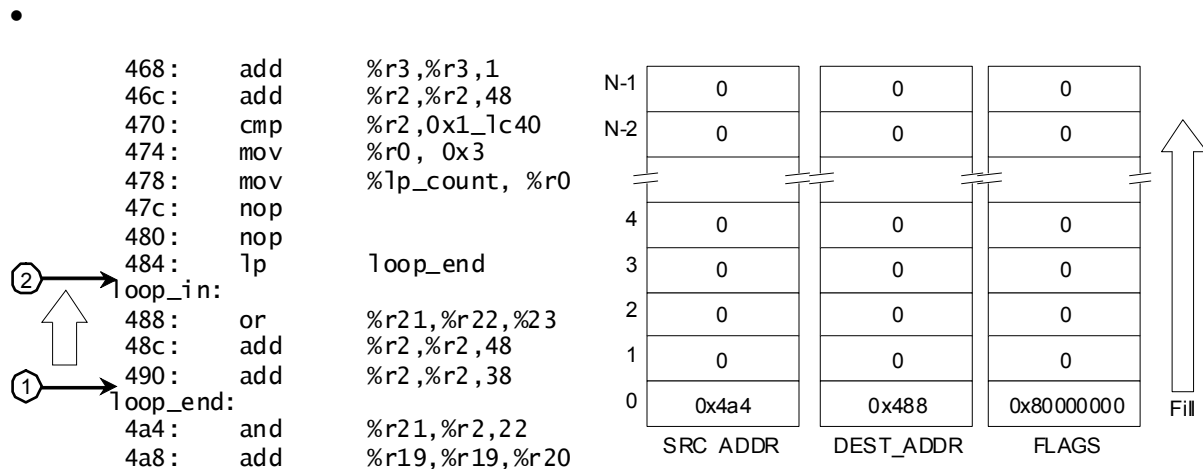
The program has started and reaches instruction address 0x4a4, and the end of the loop has been reached.



## Step 2

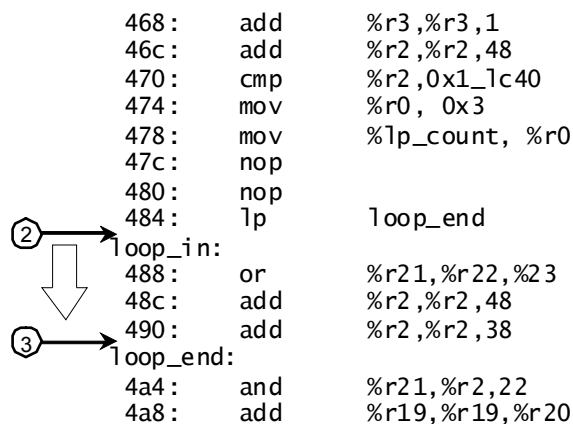
This is the first time through the loop, so the program goes back to instruction address 0x488 (loop\_in). Because a change in program flow has occurred, the destination and source instruction addresses are saved on the stack.

- Stack location 0, DEST\_ADDR, is set to 0x488 (the destination instruction address).
- Stack location 0, SRC\_ADDR, is set to 0x4a4 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000000 (indicating a valid entry).



## Step 3

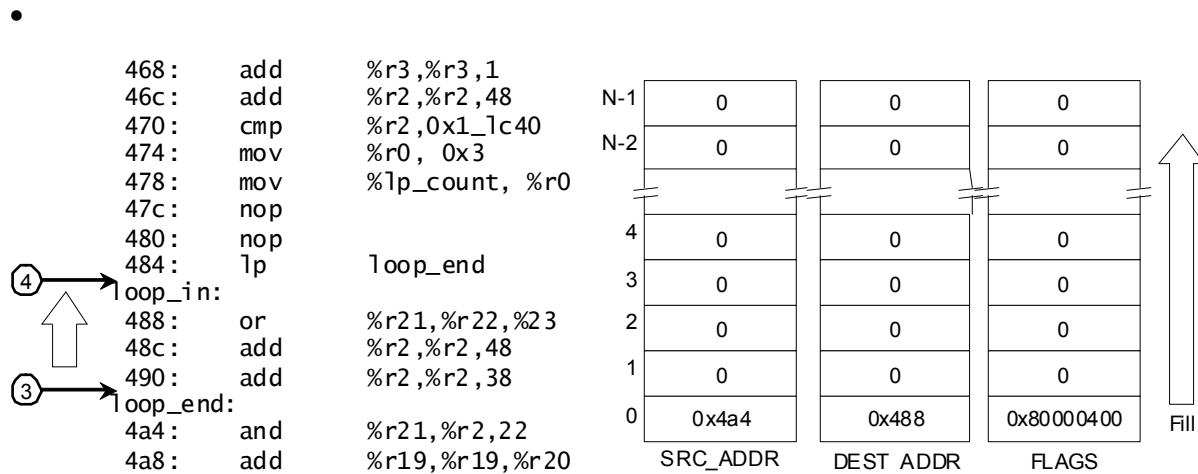
The program continues from 0x488 and reaches instruction address 0x4a4, and the end of the loop has been reached (for the second time).



This is the second time through the loop, so the program goes back to instruction address 0x488 (loop\_in).

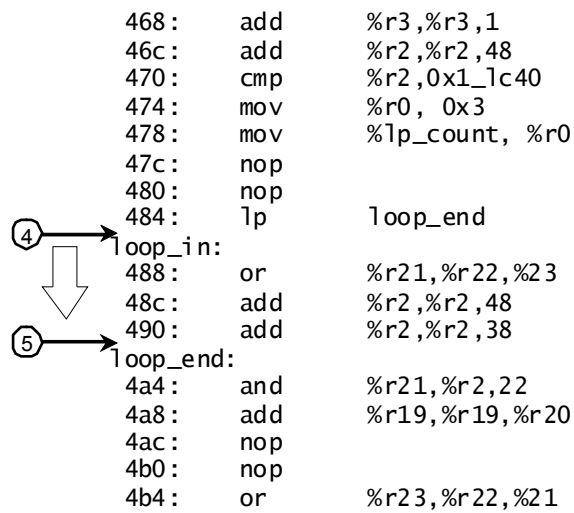
Because a change in program flow has occurred normally, both the destination and source instruction addresses should be saved on the stack. However, because the source and destination instruction addresses have already been saved, they do not need to be saved again. Instead, the repeat bit, RP, of FLAGS is set to 1 (normally this bit is set to 0).

- Stack location 0, DEST\_ADDR, is unchanged at 0x488 (the destination instruction address).
- Stack location 0, SRC\_ADDR, is unchanged at 0x4a4 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000400 (indicating valid entry and repeat).



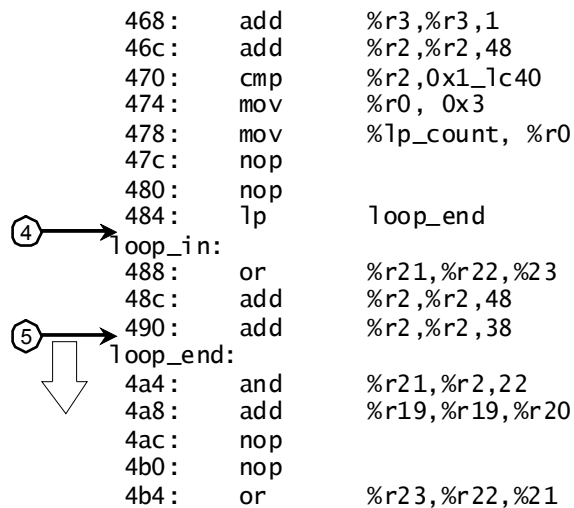
## Step 5

The program continues from 0x488 and reaches instruction address 0x4a4, and the end of the loop has been reached (for the third time).



## Step 6

This is the third time through the loop, so the program continues with the next instruction 0x4a8. The stack values are unchanged.



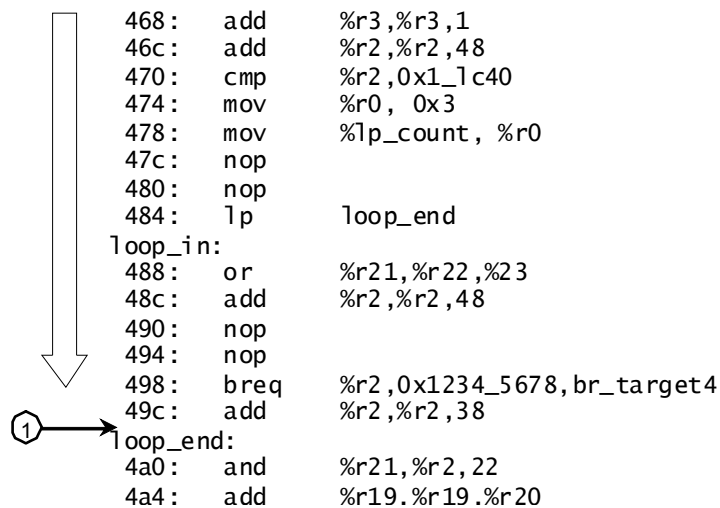
#### 14.8.4.5 Stack Operation with Branch from Loop

The following example walks through a program that has a loop (loop count is 3) and a branch within the loop. The example shows when the important instruction addresses are stored on the stack.

##### Example 14-7 Smart Operation with Branch from Loop

###### Step 1

The program has started and reaches instruction address 0x4a0. The end of the loop has been reached (the branch was not taken).

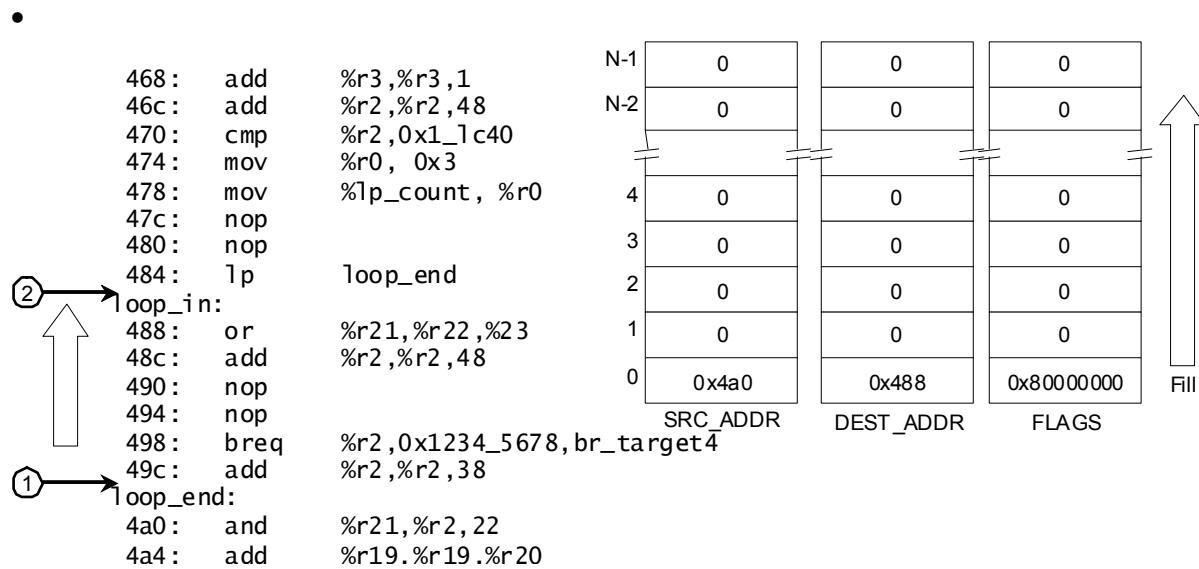


###### Step 2

This is the first time through the loop, so the program goes back to instruction address 0x488 (loop\_in).

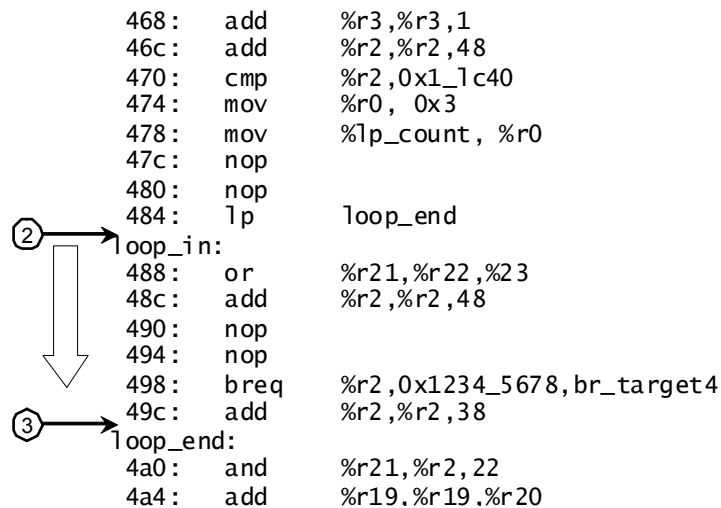
Because a change in program flow has occurred, the destination and source instruction addresses are saved on the stack.

- Stack location 0, DEST\_ADDR, is set to 0x488 (the destination instruction address).
- Stack location 0, SRC\_ADDR, is set to 0x4a0 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000000 (indicating a valid entry).



### Step 3

The program continues from 0x488 and reaches instruction address 0x4a0, and the end of the loop has been reached (for the second time).



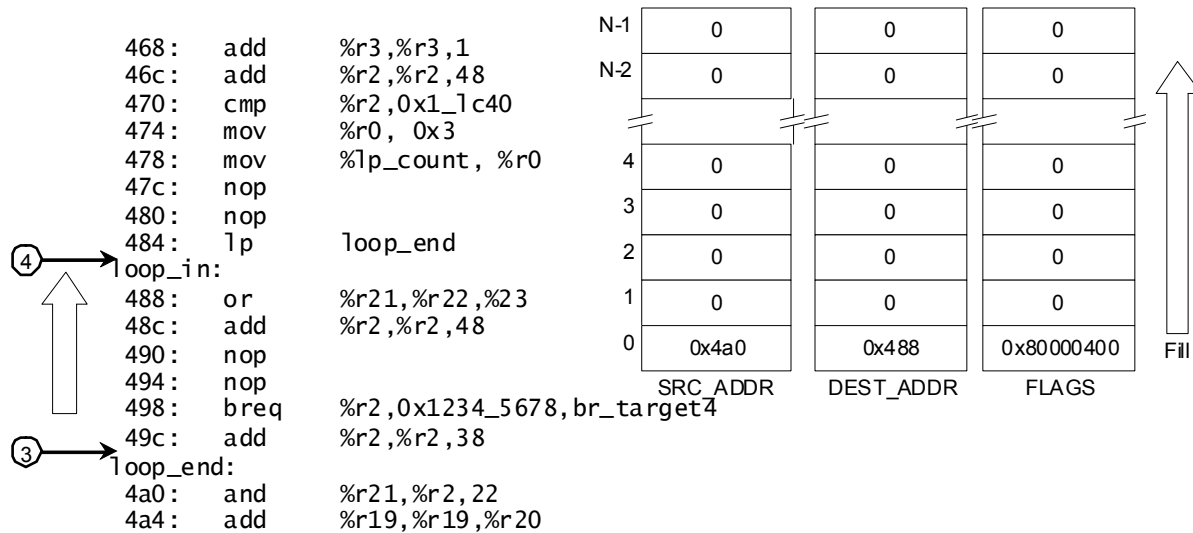
### Step 4

This is the second time through the loop, so the program goes back to instruction address 0x488 (loop\_in).

Because a change in program flow has occurred normally, both the destination and source instruction addresses should be saved on the stack. However, because the source and destination instruction addresses have already been saved, they do not need to be saved again. Instead, the repeat bit, RP, of FLAGS is set to 1 (normally this bit is set to 0).

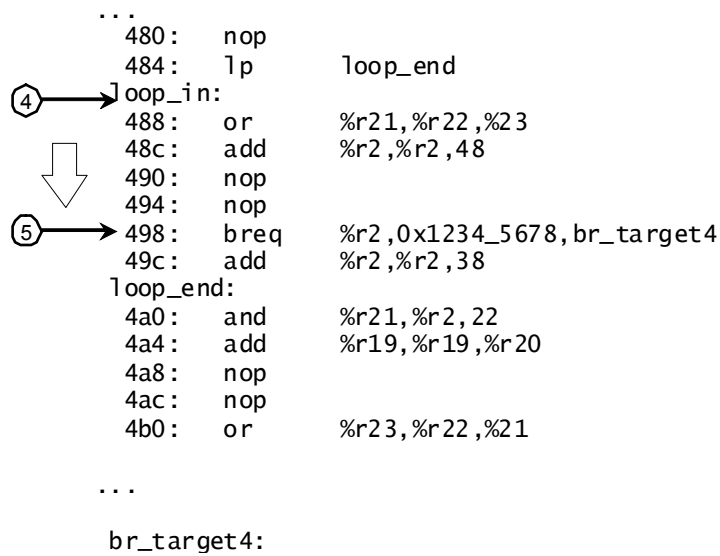
- Stack location 0, DEST\_ADDR, is unchanged at 0x488 (the destination instruction address).
- Stack location 0, SRC\_ADDR, is unchanged at 0x4a0 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000400 (indicating valid entry and repeat).

•



## Step 5

The program continues from instruction address 0x488 and reaches instruction address 0x498 (the branch instruction).

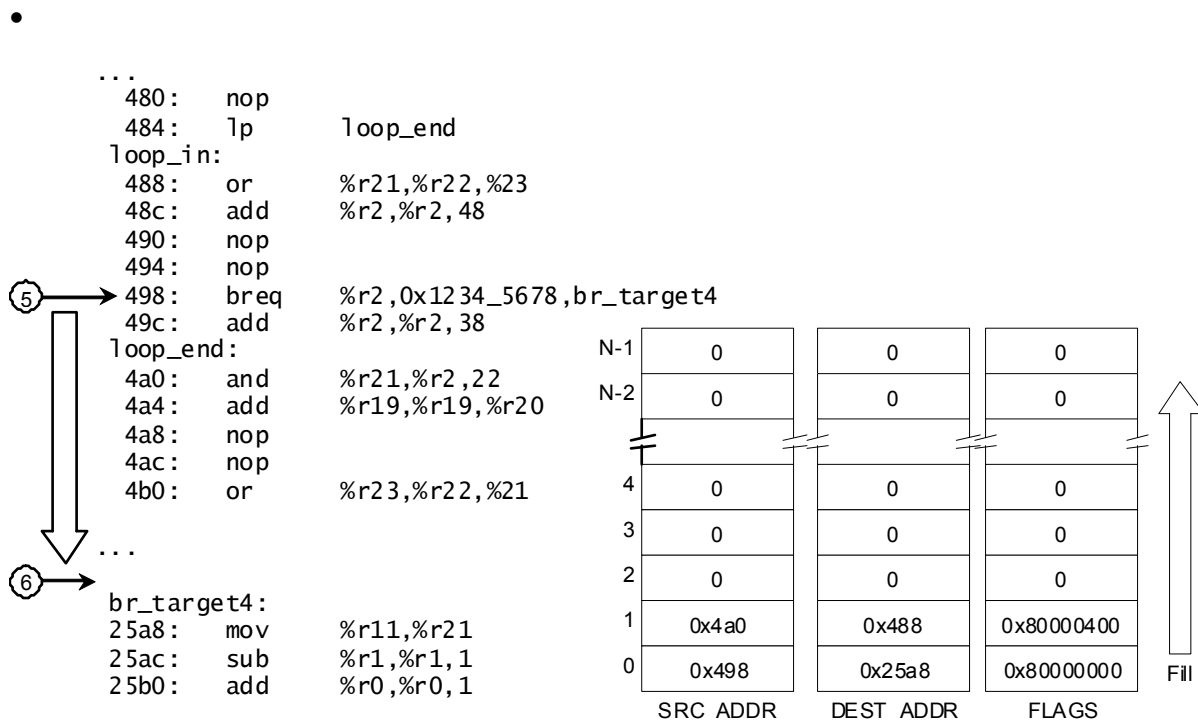


## Step 6

This time the branch instruction is taken and the program moves to 0x25a8 (br\_target4).

Because a branch has occurred within the loop, both the destination and source instruction addresses are saved on the stack, and the previous stack location values are pushed up the stack.

- Stack location 0, DEST\_ADDR, is updated to 0x25a8 (the destination instruction address).
- Stack location 0, SRC\_ADDR, is updated to 0x498 (the source instruction address).
- Stack location 0, FLAGS, is set to 0x80000000 (indicating valid entry).
- Stack location 1, DEST\_ADDR, is set to 0x488 (the previous destination instruction address).
- Stack location 1, SRC\_ADDR, is set to 0x4a0 (the previous source instruction address).
- Stack location 1, FLAGS, is set to 0x80000400 (the previous valid and repeat entry).





## 14.9 Real Time Trace Module Interface

This section provides a brief overview of the three interfaces from the ARC EM processor being traced to the Real Time Trace (RTT) module. These interfaces perform the following:

1. The first interface is used for programming the registers in the RTT module to configure the tracing.
2. The second interface provides the signaling required for the RTT module to generate the trace messages.
3. The third interface is used for optionally “freezing” the CPU due to buffers in RTT filling up.

For detailed information on RTT features, see *ARC EM RTT Databook*.

### 14.9.1 Overview

The programming interface is intended to be driven by three auxiliary registers added to the ARC EM Processor core, one each for address, data and command. The addresses allocated for these are 0x380, 0x381 and 0x382, respectively. For more details on these registers, see the *ARC EM Programmer's Reference*.

The RTT producer interface within the ARC EM processor drives a number of signals and buses that provide information and that allows the RTT module to determine whether to generate trace messages or not, which trace message to generate, and what the contents of that message should be. These configurable busses allow monitoring of CPU activity including program trace, memory transactions, auxiliary register transactions, and core register writes.

The interface to “freeze” the processor provides two signals from RTT, one is a pulse high to signal that the core should halt, the other is a pulse to signal that the core should start running again.

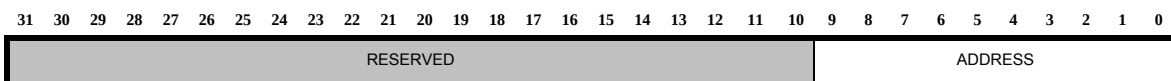
### 14.9.2 RTT Interface Registers

#### 14.9.2.1 ARC RTT Address Register, RTT\_ADDRESS

Address: 0x380

Access: RW

**Figure 14-27 RTT\_ADDRESS Register**



This register specifies the ARC RTT memory address to which you want to write to or read from. The address written to this register is presented on the ARC RTT interface address bus; the desired address must be setup prior to writing the RTT\_COMMAND register which initiates the ARC RTT read or write operation.

The operation that the processor performs on this ARC RTT memory address is specified by the RTT\_COMMAND register.

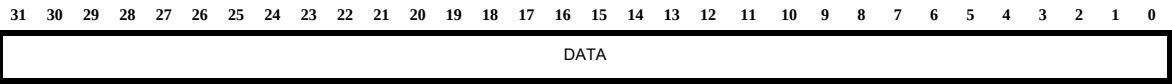
The RTT\_DATA register contains the data read from this ARC RTT memory address or the data you want to write to this memory address. For more information about this register, see the *ARC RTT Databook*.

14.9.2.2     **ARC RTT DATA Register, RTT\_DATA**

Address:    0x381

Access:     RW

**Figure 14-28 RTT\_DATA Register**



This register contains the data read from the ARC RTT or the data to be written to the ARC RTT.

The read or write operation is determined by the value of the RTT\_COMMAND register.

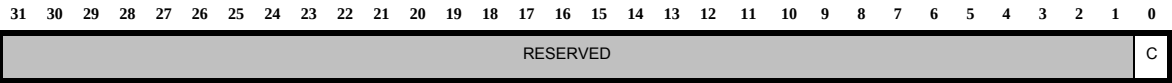
The RTT\_ADDRESS register specifies the ARC RTT memory address that is being accessed. For more information about this register, see the *ARC RTT Databook*.

14.9.2.3     **ARC RTT CMD Register, RTT\_COMMAND**

Address:    0x382

Access:     W

**Figure 14-29 RTT\_COMMAND Register**



This register specifies the actions that the processor performs on ARC RTT.

**Table 14-16 RTT\_COMMAND Bit Field Description**

| Field | Description                                                                                                                                 |
|-------|---------------------------------------------------------------------------------------------------------------------------------------------|
| C     | <b>Specifies the action performed on ARC RTT</b><br>0: indicates a read operation from ARC RTT<br>1: indicates a write operation to ARC RTT |

For more information about this register, see the *ARC RTT Databook*.

14.9.3     **Programming and Timing**

For both reads and writes, the actual action is triggered by writing to the command register. Writing to that register produces a one cycle pulse on the rtt\_read or rtt\_write ports.

### 14.9.3.1 Read Access

- Write the required address to the address register.
  - Address appears on `rtt_addr`.
- Write 0 to the command register.
  - This causes the value on `rtt_dataar` to be captured into the data register.
- Read the data register (LR).
  - Data value returned to the core register used in the LR instruction.

### 14.9.3.2 Write Access

- Write the required address to the address register.
  - Address appears on `rtt_addr`.
- Write the data to the data register (could be done prior to the address register).
  - Data value appears on `rtt_dataaw`.
- Write 1 to the command register.

For more information on the RTT interface signals, see the *Real Time Trace (RTT) Databook*.



# 15

## Performance Counters

This chapter describes the performance-monitoring features available in the ARC EM processor family, and how to use them.

### 15.1 Introduction

The ARC EM family processors includes a performance-monitoring system that enables the rapid optimization of the ARC-based systems.

This system helps to improve optimization techniques and can be used for the following:

- Hot-spot analysis — Locates the functions and instructions consuming the most CPU time.

Hardware-assisted statistical code profiling provides visibility of instruction-execution frequencies.

The system can also capture microarchitecture detail, revealing the average time taken for each instruction and allowing root causes of instruction stalls and pipeline flushes to be identified. The performance monitor collects data from unmodified code with minimal impact on system performance.

- Instruction flow metrics — Identify patterns of behavior common to the whole system.

An insight into the behavior of the system as a whole can be gained without requiring knowledge of any individual program block. System-level metrics are calculated from hardware counts of instruction and pipeline events. Example metrics include the following:

- IPC (instructions per clock)
- Cache-miss rates
- Branch-prediction accuracy
- Average load-use penalty
- Breakdown of stall causes
- Instruction mix (LD, ST, branch, and so on)
- Instruction size (16 or 32 bits)
- Average function size
- Average instructions between branches

- Time spent in interrupt or exception handlers

Analysis of metrics can highlight techniques for improvement in the program code, compiler configuration, memory-subsystem design, and CPU configuration.

- Timing measurements — Measure minimum, maximum, and average response times.

In systems with real-time requirements, it is important to measure time taken to perform critical tasks. The performance-monitoring system can capture minimum, maximum, and average times for key measures such as interrupt-response times, interrupt and exception handler execution times, interrupt-disabled time, sleep time, memory system response time, and so on. Time inside program code can also be measured — user-settable flags are provided for low-cost code instrumentation, allowing accurate measurement of time taken in critical functions or particular modes of operation.

- Real-time statistics display — View system parameters live as the system runs.

Metrics collected during a whole program run provide a picture of the system behavior. Graphs of metrics show change in real time as a response to workload or user interaction. Viewing plots of key metrics while the system runs lets you rapidly gain an impression of system behavior and response to workload, which can then be investigated further using more detailed analysis methods.

## 15.2 Components

This section describes the hardware and software components of the performance-monitoring system.

### 15.2.1 Hardware Components

The hardware for the performance-monitoring system consists of the following components:

- Countable conditions
  - A large number of status signals from the processor and related components is combined into a set of countable conditions. Special modules generate signals to allow the identification of pipeline stall causes, instruction types, memory accesses, and so on. Components within the same processor island share a common set of conditions.
  - Each condition is given a unique eight-character name in hardware, to allow software to discover which conditions are available for any given target.
  - User-settable flags are provided for instrumentation of code. Thirty-two flags are available, and can be used by programmers to provide A-to-B timing measurements, to show live statistics of execution time in different sections of the application, or to allow shared code to be profiled based on the calling function.
- Performance counters
  - This block provides a set of 48-bit counters, each of which can be triggered by one of the available countable conditions, or the inverse of that condition. Counting can take place on every cycle.
  - The number of counters can be configured upto 32.
  - The first eight counters can be configured as min/max mode.

- Counting can be restricted to a specified address range, or to take place only when a given countable condition is true (or false).

Table 15-1 lists the performance counters and their features.

**Table 15-1 Performance Counters and Features**

| Feature                               | Performance Counters                                   |
|---------------------------------------|--------------------------------------------------------|
| Sampling                              | Every cycle                                            |
| Counter storage                       | On-chip registers                                      |
| Programmable address range            | Yes                                                    |
| PC granularity                        | Yes                                                    |
| Number of countable conditions        | Up to 32 such as 8,16, 32 (depending on configuration) |
| Inversion of countable conditions     | Yes                                                    |
| Global enable by condition            | Yes                                                    |
| Live statistics output during runtime | Yes                                                    |

Use the performance counters for the following:

- Accurate timing measurements — average, minimum, and maximum times.
- System-level analysis of caches, memory system, and branch behavior. To inform both software architecture design and hardware-configuration choices.
- System-level analysis of code instruction mix and stall causes. To inform compiler-option selection, compiler development, and hardware-configuration choices.
- Counting conditions of interest, such as instruction-cache misses, interrupts taken, and so on.
- Live statistics display.

## 15.2.2 Software Components

Software support for the performance-monitoring system consists of the following main components:

- MetaWare debugger support
  - Program performance counters using the names of the conditions available in the hardware.
  - Display raw counts and metrics derived from counts such as instructions per clock, average instruction-cache miss time, number of stack operands per call, and so on.
  - View profiling-count data alongside disassembly or source.
  - Sort instructions or functions using profiling data.
  - Integrated ARCPlotter for graphical display of data captured from performance counters, whether real-time or after the event.

- ❑ Send live performance-counter data to the MetaWare debugger's ARCPLOTTER during run time.
- ❑ Control from GUI, command line, or automated script.

For further details, see the *MetaWare Debugger User's Guide*.

In addition to the software support described above, you can also program the blocks programmatically, using the register interfaces.

## 15.3 Usage Example

The following are the steps for programming and capturing performance count:

1. Program `cc_index` to zero.
2. Read `cc_name0` and `cc_name1` for the condition name in ASCII format.
3. Program `cc_index` plus one.
4. Repeat steps 2 and 3 until a null is read.
5. Disable PCT Global Enable and clear the counters in `PCT_CONTROL`.
6. Select a counter to be used in `PCT_INDEX`.
7. Choose a condition that needs to be counted.
8. Program `cc_index` for the condition chosen in `PCT_CONFIG`.
9. Repeat steps 6 through 8 for all the remaining conditions.
10. Enable Global Enable in `PCT_CONTROL`.
11. Run the test.
12. When the test is finished, read the counter values `PCT_COUNTL/H` by writing counter index to `PCT_INDEX`.

## 15.4 Countable Conditions

An array of countable-condition signals is generated by logic that can be optionally included in an ARC EM family processor design. On every cycle, each signal in the array indicates the status of some feature of the processor and/or related units. This set of signals is known as the wide condition array (WCA).

The countable conditions are designed to allow useful metrics to be computed, and to permit the counting and profiling of specific conditions.

The set of countable conditions available in any particular core configuration can vary. For this reason, registers are provided to allow a debugger or natively executing program to discover the supported conditions.

The countable conditions block contains no programmable elements. Its purpose is to collate the wide condition array into a form that can be sampled by other components.

The set of condition signals can be sampled by the performance counters hardware block.

The countable conditions are split into the following categories:



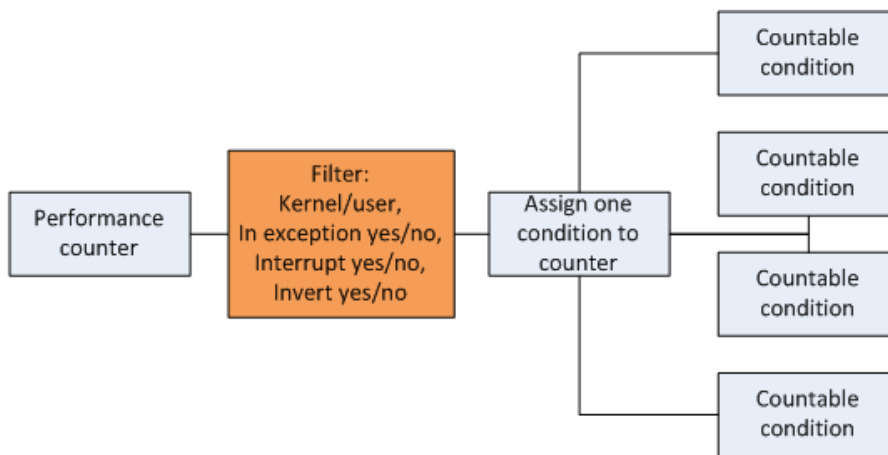
- ISA conditions — Instruction-execution conditions that are not timing-related, and for which the same answer is produced on any target executing the same code sequence.
- Other — Timing-related or microarchitecture-specific conditions.

### 15.4.1 Condition Filter

The conditions are filtered based on whether the execution is done by kernel, user, interrupt, or exception mode.

Figure 15-1 shows the condition filter mode.

**Figure 15-1 Condition Filter Mode**



## 15.5 Condition Registers

The registers shown in Table 15-2 are provided for interaction with the countable-conditions block.

**Table 15-2 Countable-Conditions Registers**

| Name     | Address | R/W | Description                                               |
|----------|---------|-----|-----------------------------------------------------------|
| CC_BUILD | 0xF6    | R   | Build-configuration register                              |
| CC_INDEX | 0x240   | R/W | Index to access name of condition                         |
| CC_NAME0 | 0x241   | R   | First four characters of condition name, set by CC_INDEX  |
| CC_NAME1 | 0x242   | R   | Second four characters of condition name, set by CC_INDEX |

For further details on the countable-conditions registers, see the *ARCV2 ISA Programmer's Reference Manual*.

## 15.6 Available Conditions

There are conditions available are:

- Logical conditions
- ISA conditions
- Microarchitecture-specific conditions

### 15.6.1 Logical Conditions

The conditions listed in this section are available on all ARC EM family processors for use in building metrics.

**Table 15-3 Logical Conditions**

| Name   | Short Description | Comments                                |
|--------|-------------------|-----------------------------------------|
| always | Always true       | True or 1 condition for use in metrics  |
| never  | Never true        | False or 0 condition for use in metrics |

### 15.6.2 ISA Conditions

The following is the minimum list of conditions required on all targets that implement the countable-conditions programming interface. Additional conditions can be added.

The prefixes have the following meanings in the condition names:

- i = counts a (committed) instruction
- q = counts an interrupt event
- e = counts an exception event

**Table 15-4 ISA Conditions**

| Name   | Short Description                    | Comments                                                                                                                                                                                                 |
|--------|--------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| iall   | Instructions                         | All committed instructions                                                                                                                                                                               |
| isleep | Instruction: SLEEP                   | SLEEP instruction committed                                                                                                                                                                              |
| ijmp   | Instruction: Jump/branch             | All Jump and Branch instructions<br>Includes: BBITn, B, Bcc, B_S, Bcc_S, BL, BLcc, BL_S, BRcc, BRNE_S, BREQ_S, J, J_S, Jcc, JEQ_S, JNE_S, JL, JL_S, JLcc<br>Excludes: RTIE, BRK, LP, LPcc, TRAP0, TRAP_S |
| ijmpc  | Instruction: Conditional Jump/branch | All jump instructions taken conditionally<br>Includes: Bcc, Bcc_S, BLcc, BRcc, BRNE_S, BREQ_S, Jcc, JEQ_S, JNE_S, JLcc<br>Excludes: LPcc                                                                 |

**Table 15-4 ISA Conditions**

| Name    | Short Description                        | Comments                                                                                                                                                                |
|---------|------------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ijmpu   | Instruction: Unconditional Jump/branch   | All jump instructions taken unconditionally.<br>Includes: B/BAL, B_S, BL/BLAL, BL_S, J/JAL, J_S, JL/JLAL, JL_S<br>Excludes: RTIE, BRK, LP, TRAP0, TRAP_S                |
| ijmpd   | Instruction: Jump/branch with delay slot | All jump instructions with delay slot.<br>Includes: BBITn.d, B.d, Bcc.d, BL.d, BLcc.d, BRcc.d, J.d, Jcc.d, JL.d, JLcc.d<br>Excludes: RTIE, BRK, LP, LPcc, TRAP0, TRAP_S |
| ijmptak | Instruction: Jump/branch with taken      | All jump/branch instructions taken                                                                                                                                      |
| icall   | Instruction: Call                        | All linked subroutine call instructions<br>Includes: BL, BLcc, BL_S, JL, JL_S, JLcc                                                                                     |
| imemrd  | Instruction: Mem read cached             | All memory read cached instruction committed<br>Includes: All variants of LD, POP and EX                                                                                |
| imemrdc | Instruction: Mem read                    | All memory read instruction committed<br>Includes: All variants of LD, POP and EX                                                                                       |
| imemwr  | Instruction: Mem write                   | Instruction is Memory Write<br>Includes: All variants of ST and PUSH<br>Excludes: EX                                                                                    |
| imemwrc | Instruction: Mem write cached            | All memory write cached Instruction committed<br>Includes: All variants of ST and PUSH<br>Excludes: EX                                                                  |
| ibi     | BI/BIH instruction committed             | BI/BIH instruction committed                                                                                                                                            |
| ijli    | JLI instruction committed                | JLI instruction committed                                                                                                                                               |
| ilr     | Instruction: LR                          | Instruction is Auxiliary Read<br>Includes: All variants of LR and AEX                                                                                                   |
| isr     | Instruction: SR                          | Instruction is Auxiliary Write<br>Includes: All variants of SR<br>Excludes: AEX                                                                                         |
| itrap   | Instruction: TRAP                        | Instruction is TRAP                                                                                                                                                     |
| ilp     | Instruction: LP                          | Loop setup instruction – LP, LPcc                                                                                                                                       |

**Table 15-4 ISA Conditions**

| Name    | Short Description                  | Comments                                                                                                                                                                                                                                                                     |
|---------|------------------------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| ilpend  | Instruction: LP loop-end condition | Instructions committed when loop-end condition is true. This is the condition that will cause the LP_COUNT register to decrement. Counting this condition gives total number of loop iterations (for more details, see the <i>ARCo2 ISA Programmer's Reference Manual</i> ). |
| ilpin   | Instruction: inside LP loop        | Instructions committed inside loop body.<br>LP_START <= PC < LPEND                                                                                                                                                                                                           |
| i2byte  | Instruction: 2-byte                | Committed 16-bit instructions                                                                                                                                                                                                                                                |
| i4lbyte | Instruction: 4-byte                | Committed 32-bit instructions                                                                                                                                                                                                                                                |
| i2lbyte | Instruction: 6-byte                | Committed 16-bit instruction with L IMM data                                                                                                                                                                                                                                 |
| i4lbyte | Instruction: 8-byte                | Committed 32-bit instruction with L IMM data                                                                                                                                                                                                                                 |
| etaken  | Exception Taken                    | When an instruction raises exception                                                                                                                                                                                                                                         |
| ieh     | Instrs, exception handler (AE)     | Instructions committed in exception handler                                                                                                                                                                                                                                  |
| qltaken | Interrupt taken                    | When an Interrupt is taken                                                                                                                                                                                                                                                   |
| icbmem  | Cache-Bypassed LD/ST               | Cache-Bypassed LD/ST instruction committed                                                                                                                                                                                                                                   |
| iuser   | Instrs, user mode                  | Instructions committed in user mode                                                                                                                                                                                                                                          |

### 15.6.3 Microarchitecture-Specific Conditions

The following conditions described in [Cycle-Time Conditions](#) section may be available on targets that implement the countable conditions programming interface.

#### 15.6.3.1 Cycle-Time Conditions

[Table 15-5](#) shows cycle-time conditions. Prefix C indicates cycles

**Table 15-5 Cycle-Time Conditions**

| Name   | Short Description | Comments                                                                                                                                                                                                                 |
|--------|-------------------|--------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| cycles | Cycles – Total    | Condition always true- every cycle                                                                                                                                                                                       |
| crun   | Cycles – Running  | True when ARC is running STATUS32[H]=0<br>Note: <i>running</i> means the processor is not in a halted state.<br>These cycles include time spent in sleep mode (DEBUG [ZZ] =1) and not in halted state (STATUS32 [H] =0). |

**Table 15-5 Cycle-Time Conditions**

| Name   | Short Description                        | Comments                                                                                                                                                                               |
|--------|------------------------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| csleep | Cycles – Sleeping                        | True when ARC is running and in sleep mode STATUS32[H]=0, DEBUG[ZZ]=1<br>Note: This condition excludes case the scenario that the processor is in halt and the sleep-mode flag is set. |
| cruni  | Cycles – Running with interrupt disabled | Condition is true when ARC is running with interrupt disabled (interrupts are disabled when in an interrupt handler or exception handler).                                             |

**15.6.3.2 Bubbles/Stall Conditions****Table 15-6 Bubbles/Stall Conditions**

| Name      | Short Description          | Comments                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
|-----------|----------------------------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| bstall    | Bubbles: All stalls        | Pipeline bubbles caused by any type of stall<br>Includes: Stalls at any stages from fetch to commit                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                   |
| bflush    | Bubbles: All flushes       | Pipeline bubbles caused by any type of flush<br>Includes: Flushes – IRQ/Exception, DMP-Retry, serializing instruction, backstop, BPU mispredict                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                       |
| bfetch    | Instruction-fetch stall    | Pipeline bubbles caused by there being no instruction fetch.<br>This condition is set true when no instruction can be clocked into the fetch stage, and the processor is running and not sleeping (STATUS32[H]=0, DEBUG[ZZ]=0), no debug transaction is underway, and the bubble is not caused by an instruction-cache miss or line fill stall.<br>Causes of this bubble include: <ul style="list-style-type: none"> <li>Initial delay on startup during clearing of BPU and MMU RAMs.</li> <li>Stall after fragged instruction – if the front end detects a <i>fragged</i> instruction, an instruction that is constructed from two non-contiguous 32-bit instruction words, then no further instructions are fetched until a pipeline flush takes place.</li> </ul> |
| bdebug    | Debug transaction          | Bubble caused by the insertion of a debug instruction into the pipeline, or related flush. Debug instructions are instructions that are inserted into processor pipeline to provide debugger (JTAG) access to processor.                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                              |
| bfbmp     | Branch mispredict stall    | Branch mispredict stall                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                               |
| bdslot    | dSlot fetch stall          | Delay slot fetch stall                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                |
| bnofetch  | No instruction-fetch stall | No instruction memory fetch stall                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
| buopstal  | Micro-op stall             | Micro-op dependency stall                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                             |
| baccstall | Accumulator stall          | Accumulator dependancy stall                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                          |

**Table 15-6 Bubbles/Stall Conditions**

| Name     | Short Description                                   | Comments                                            |
|----------|-----------------------------------------------------|-----------------------------------------------------|
| bauxflsh | Auxiliary Flush                                     | Auxiliary register serialization flush              |
| bfirgex  | Cycles - Interrupt or Exception taken               | Stall due to Interrupt or Exception                 |
| blimm    | Bubbles: All stalls caused by fetching a limm       | Bubbles: All stalls caused by fetching a limm       |
| bmisal   | Cycles - Bubble due to non-word aligned jump target | Cycles - Bubble due to non-word aligned jump target |
| bslpfl   | Cycles - SLEEP-related flush                        | Cycles - SLEEP-related flush                        |
| bdccmstl | Cycles - DCCM stall                                 | Cycles - DCCM stall                                 |
| bperb    | Cycles - Peripheral bus stall                       | Cycles - Peripheral bus stall                       |
| bmem     | Cycles - Memory stall in DC2 stage                  | Cycles - Memory stall in DC2 stage                  |
| bfhold   | Cycles - When flush is held by fetch stage          | Cycles - When flush is held by fetch stage          |
| bdins    | Cycles - Waiting for delay-slot instruction         | Cycles - Waiting for delay-slot instruction         |
| bindstl  | Cycles - Stall due to any input value dependency    | Cycles - Stall due to any input value dependency    |
| bopds    | Cycles - Stall due to register RAW dependency       | Cycles - Stall due to register RAW dependency       |
| bjli     | Cycles - Stall due to JLI instruction               | Cycles - Stall due to JLI instruction               |

**15.6.3.3 Instruction Cache-Related Conditions****Table 15-7 Instruction Cache-Related Conditions**

| Name  | Short Description                 | Description                              |
|-------|-----------------------------------|------------------------------------------|
| icm   | Instruction cache miss            | Event: Instruction-cache miss            |
| icoff | Instruction Cache disabled        | Event: Instruction-cache disabled        |
| ivic  | Instruction Cache Invalidate      | Event: Instruction-cache invalidate      |
| ivil  | Instruction Cache Invalidate Line | Event: Instruction-cache invalidate line |

### 15.6.3.4 Data Cache-Related Conditions

**Table 15-8 Data Cache-Related Conditions**

| Name  | Short Description          | Description                   |
|-------|----------------------------|-------------------------------|
| dcm   | Data Cache Miss            | Event: dcache miss            |
| dc lm | Data Cache Load Miss       | Event: dcache Load miss       |
| dcs m | Data Cache Store Miss      | Event: dcache Store miss      |
| dcoff | Data Cache Disabled        | Event: dcache disabled flush  |
| fldc  | Data Cache Flush           | Event: dcache flush           |
| fldl  | Data Cache Flush Line      | Event: dcache flush line      |
| ivdc  | Data Cache invalidate      | Event: dcache invalidate      |
| ivdl  | Data Cache Invalidate Line | Event: dcache invalidate line |

### 15.6.3.5 User Flag Conditions

These conditions are available when the performance-counter extension is present. Thirty two (32) flags are provided and can be individually controlled from kernel-mode code.

This allows the programmers to instrument their code, for example to time multiple different sections of code using the performance counters, or to make some specific conditions for performance counters.

**Table 15-9 User Flag Conditions**

| Name     | Short Description | Comments         |
|----------|-------------------|------------------|
| uflag_0  | User Flag 0       | User flag 0 set  |
| uflag_1  | User Flag 1       | User flag 1 set  |
| ...      | ...               | ...              |
| uflag_31 | User Flag 31      | User flag 31 set |

### 15.6.3.6 APEX Conditions

**Table 15-10 APEX Conditions**

| Name     | Short Description                        | Comments                                                                             |
|----------|------------------------------------------|--------------------------------------------------------------------------------------|
| iext     | Instruction, extension                   | The committed extension instruction                                                  |
| cexthold | Cycles-when flush is held by grad buffer | Cycles-when flush is held by the pending extension instructions in graduation buffer |
| bxflgs   | Bubble-xflag stall                       | Xflag stall because of SR or pending extension instructions                          |

**Table 15-10 APEX Conditions**

| Name   | Short Description               | Comments                                                          |
|--------|---------------------------------|-------------------------------------------------------------------|
| bcr1   | Bubble-extension core reg stall | Extension core register explicit/implicit stall                   |
| bcr2   | Bubble-extension core reg stall | Extension core register implicit/implicit stall                   |
| bar1   | Bubble-extension aux reg stall  | Extension aux register explicit/implicit stall                    |
| bar2   | Bubble-extension aux reg stall  | Extension aux register implicit/implicit stall                    |
| bbusy  | Bubble-user logic _busy stall   | Extension instruction stall                                       |
| bgrad  | Bubble-graduation buffer stall  | Graduation buffer cannot accept more extension instructions stall |
| btimed | Bubble-timed ext instruction    | Extension timed instruction commit conflict stall                 |

## 15.7 Performance Counters

You can add a performance-counter module to a design when using **an** the ARC EM family of processors. Performance counting is triggered by signals from the WCA. Performance counters require at least one unit generating condition signals to be present in the design (such as a pipeline-monitoring module).

### 15.7.1 Adding Counters

To add counters, specify: `-pct_counters` and the number of counters (8, 16, or 32).

Example: `-pct_counters 16`.

### 15.7.2 Characteristics of Counters

Each counter is forty-eight bits in size to reduce the possibility of overflows for high-frequency counts. Three or four registers are associated with each counter:

- The 48-bit count value
- A 48-bit snapshot register
- A 32-bit configuration register
- A MAX/MIN Count register (only first eight counters).

The snapshot feature allows simultaneous capture of all count values without halting counting.

One trigger condition per counter is selected from the WCA. The related counter increments on every cycle that the trigger condition is true. A condition-inversion bit can be used to increment the counter only when the condition is false.



A global-enable condition is selected from the WCA, and is used to enable or disable increments on all counters simultaneously. This feature can be set to any condition, but always-on and always-off conditions are available for simple enable/disable operation.

Address-range registers can be used to enable counting only when the program counter is within a specific address range.

A min/max mode allows capture of the shortest or longest time that a condition is continuously true.

A build-configuration register is provided to indicate the following:

- Version number of performance-counter module
- Number of counters
- Bit width of counters

### 15.7.3 Performance Counter Registers

Table 15-11 lists the name and description of the performance-counter registers provided for interacting with the performance-counter block.

**Table 15-11 Performance Counter Registers**

| Name        | Address | R/W | Description                      | Comments                                 |
|-------------|---------|-----|----------------------------------|------------------------------------------|
| PCT_BUILD   | 0xF5    | R   | Build-configuration register     | -                                        |
| PCT_COUNTL  | 0x250   | RW  | Lower-order current-count value  | One per counter                          |
| PCT_COUNTH  | 0x251   | RW  | Higher-order current-count value | One per counter                          |
| PCT_SNAPL   | 0x252   | R   | Lower-order snapshot value       | One per counter                          |
| PCT_SNAPH   | 0x253   | R   | Higher-order snapshot value      | One per counter                          |
| PCT_CONFIG  | 0x254   | RW  | Counter configuration            | One per counter                          |
| PCT_CONTROL | 0x255   | RW  | Control register                 | -                                        |
| PCT_INDEX   | 0x256   | RW  | Index-select register            | -                                        |
| PCT_MINMAXL | 0x257   | RW  | MIN/MAX lower order value        | One per Counter for first eight counters |
| PCT_MINMAXH | 0x258   | RW  | MIN/MAX high order value         | One per Counter for first eight counters |
| PCT_RANGEL  | 0x259   | RW  | Address range low                | -                                        |
| PCT_RANGEH  | 0x25A   | RW  | Address range high               | -                                        |
| PCT_UFLAGS  | 0x25B   | RW  | User flags control register      | -                                        |

**Note**

- All the counters can be programmed with an initial value. The counter increments from the start value that is programmed.
- An exception is not generated for any invalid values in the auxiliary registers.

For further details on the performance-counter registers, see the *ARCo2 ISA Programmer's Reference Manual*.

# 16

## UAUX Interface

### 16.1 Overview

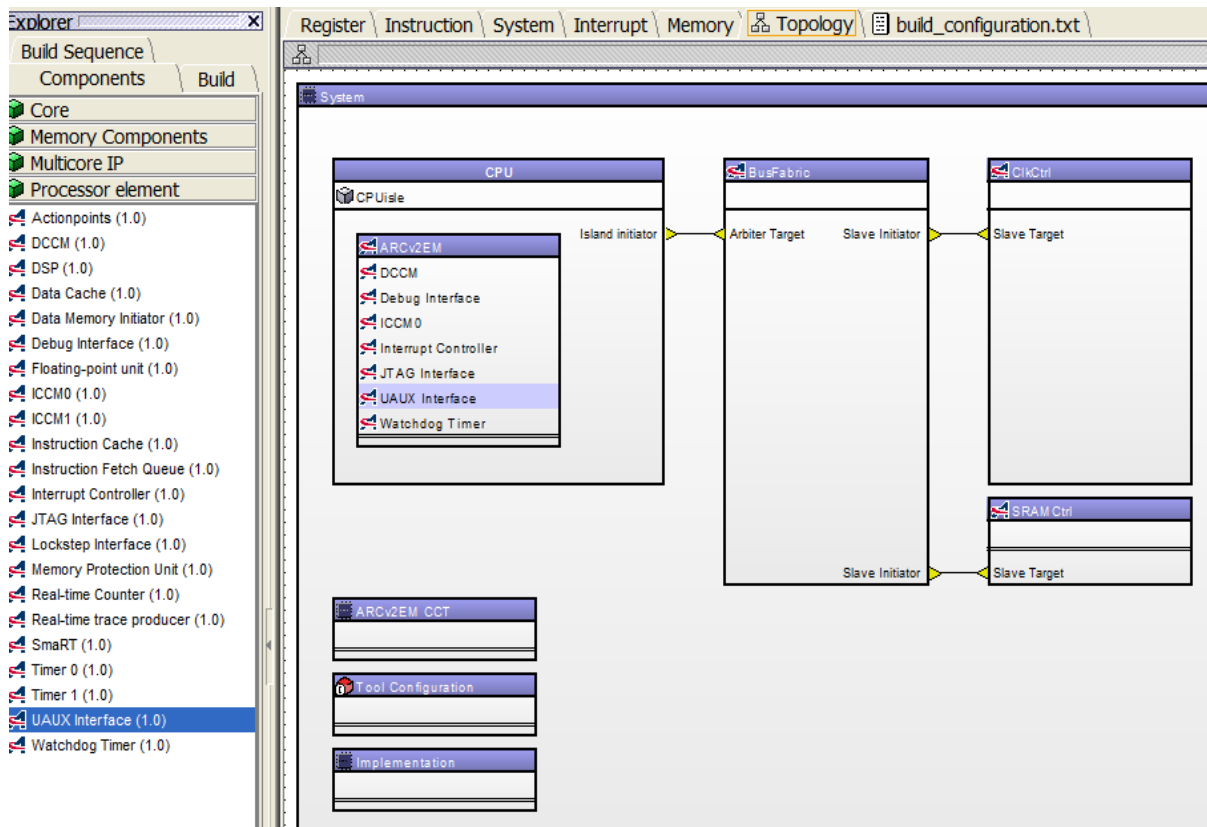
User-Auxiliary Register Interface (UAUX) is optional feature for ARC EM. This optional interface allows external user modules containing auxiliary registers to be connected to processor core. User code can access those external auxiliary registers using the same instructions used to access core and APEX auxiliary registers. The UAUX auxiliary registers address range is predefined to 0x8000\_0000 to 0xffff\_ffff. All auxiliary register access in this range will make request to the user module. APEX may also define auxiliary registers in this range. User must avoid any address conflict when assigning its auxiliary register addresses. This interface provides an easy mechanism to extend auxiliary registers without reconfiguring the ARC EM core.

UAUX component can be configured in two ways:

#### Using Graphical User Interface (GUI)

Add UAUX component from processor elements in the Components Tab by dragging it into your design.

**Figure 16-1 Adding UAUX using ARCHitect GUI**



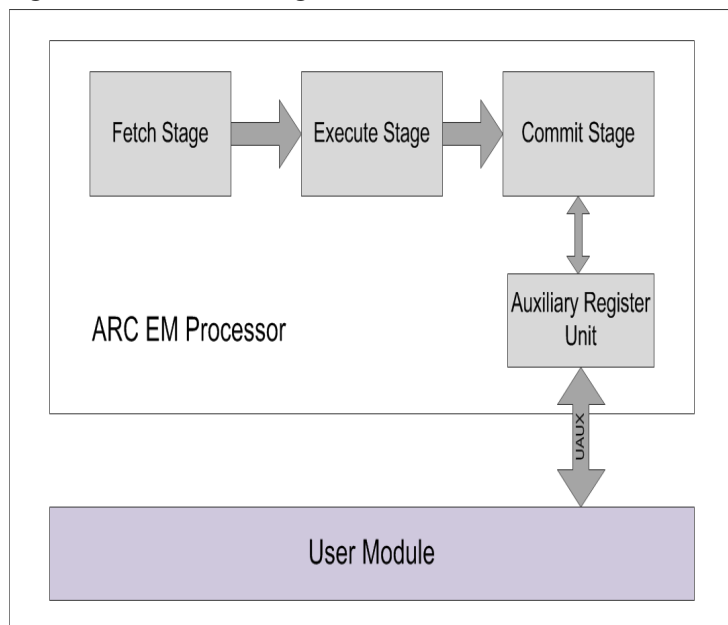
## Using Command Line

UAUX component can also be configured through a build configuration file by adding the following command:

```
-create com.arc.hardware.UAUX_Interface.1_0 "System.CPUislet.ARCv2EM.UAUX Interface".
```

## 16.2 Block Diagram

UAUX is an external interface to the ARCV2EM processor. It provides a channel to connect the external user module and the core auxiliary register unit. The block diagram is shown in Figure 16-2. In a multi-core design, each processor core will have its separate UAUX interface.

**Figure 16-2 Block Diagram of the UAUX External Bus Interface**

### 16.3 Interface Signals

Table 16-1 below describes the UAUX interface signal which can be categorized into three groups shown in Table 16-1.

**Table 16-1 UAUX Bus Interface Signals**

| Signal         | Width | Direction Core | Description                                                   | Group    |
|----------------|-------|----------------|---------------------------------------------------------------|----------|
| uaux_en_r      | 1     | Output         | UAUX Enable                                                   | Request  |
| uaux_addr_r    | 31    | Output         | UAUX Address                                                  | Request  |
| uaux_read      | 1     | Output         | UAUX Read (LR) operation                                      | Request  |
| uaux_write     | 1     | Output         | UAUX Write (SR) operation                                     | Request  |
| uaux_wdata_r   | 32    | Output         | Write Data                                                    | Request  |
| uaux_busy      | 1     | Input          | UAUX Read busy; Response not valid                            | Response |
| uaux_rdata     | 32    | Input          | UAUX Register Read Data                                       | Response |
| uaux_illegal   | 1     | Input          | Illegal access                                                | Response |
| uaux_k_rd      | 1     | Input          | Register Kernel read only                                     | Response |
| uaux_k_wr      | 1     | Input          | Register Kernel write only                                    | Response |
| uaux_unimpl    | 1     | Input          | Register unimplemented (Address hole)                         | Response |
| uaux_serial_sr | 1     | Input          | Processor will serialize if this is the last SR in a sequence | Response |

**Table 16-1 UAUX Bus Interface Signals**

| Signal         | Width | Direction Core | Description                            | Group    |
|----------------|-------|----------------|----------------------------------------|----------|
| uaux_strict_sr | 1     | Input          | Processor should serialize immediately | Response |
| uaux_cmt_phase | 1     | Output         | UAUX LR/SR is in Commit Phase          | Commit   |
| uaux_cmt_valid | 1     | Output         | UAUX committed (if not aborted)        | Commit   |

### 16.3.1 Violations

External user modules can control the exception behavior of UAUX access instructions by signaling related response signals. The details can be found in [Table 16-1](#). If the instruction violates any restrictions, the transaction will be aborted and the instruction will cause ARC EM core to execute corresponding exception handler.

**Table 16-2 UAUX Violation Mapping**

| Signal       | Violation           | Description                                                                             |
|--------------|---------------------|-----------------------------------------------------------------------------------------|
| uaux_illegal | Illegal Instruction | Write to read only or read from write only UAUX register                                |
| uaux_k_rd    | Privilege Violation | UAUX read transaction require Kernel Privilege, set to 0 when transaction is not read.  |
| uaux_k_wr    | Privilege Violation | UAUX read transaction require Kernel Privilege, set to 0 when transaction is not write. |
| uaux_unimpl  | Illegal Instruction | No such UAUX register.                                                                  |

### 16.3.2 Serialization

The processor core will execute the UAUX transaction in program order. If a UAUX write instruction has side effect to its following instruction, this may not take effect due to the pipelined nature of the processor core. In order to ensure such side effect, user module can assert `uaux_serial_sr` in response. And the core will serialize this UAUX transaction by flushing its pipeline. The processor core has the ability to fold back to back serializing AUX writes, and only the last serializing write will trigger the pipeline flush. If user module needs to disable this behavior and flush pipeline immediately after UAUX transaction is committed, it must assert `uaux_strict_sr` signal instead of `uaux_serial_sr` as response. User logic must not assert `uaux_serial_sr` when `uaux_strict_sr` is asserted.

## 16.4 Transactions

A UAUX transaction consists of three phases:

**Request Phase:** The ARC EM core starts a UAUX transaction by asserting `uaux_en_r` and setting other request group signals.

**Response Phase:** User modules can provide responses by setting response group signals. It can also insert wait cycles by asserting `uaux_busy`, and thus delay the commit phase.

**Commit Phase:** The ARC EM core completes the UAUX transaction by asserting the `uaux_cmt_phase` signal. `uaux_cmt_valid` indicates a transaction is successfully committed, otherwise this transaction is aborted and no effect should be taken on the user module side.

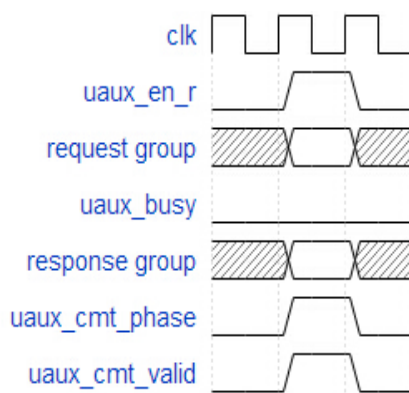
Points to be noted:

- All the request signals are valid only when `uaux_en_r` is asserted. In other words, when `uaux_en_r` is de-asserted all the request signals are invalid.
- The timing for the three phases can occur concurrently for the same transaction.
- The UAUX response phase is speculative for a given instruction. The commit phase makes the instruction non-speculative.
- Any UAUX response phase can be killed any time by de-asserting `uaux_en_r` and asserting the commit phase with commit valid low.
- The UAUX protocol only allows one outstanding transaction `uaux_en_r` will remain asserted during the whole transaction.
- If `uaux_busy` is 0 and the `uaux_unimpl` is 1 in response phase, all other response group signal must remain 0
- If the `uaux_read` and `uaux_write` signals are asserted. It is required that the user module implement an exchange behavior by returning the read value without the effect of the write.
- For EM implementation, commit group signals are combinational outputs of the `uaux_busy` and violation related response group input signals. The user module must avoid creating any timing loop when designing those signal logic.
- All request group signals have early timing, and commit group signals have late timing.

### 16.4.1 Basic Transaction

Figure 16-3 shows a basic UAUX transaction successfully committed.

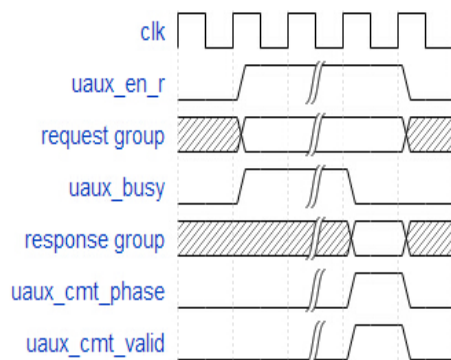
**Figure 16-3 Basic UAUX Transaction**



## 16.4.2 Wait Transaction

Figure 16-4 shows a UAUX transaction with wait cycles inserted by the user module and successfully committed.

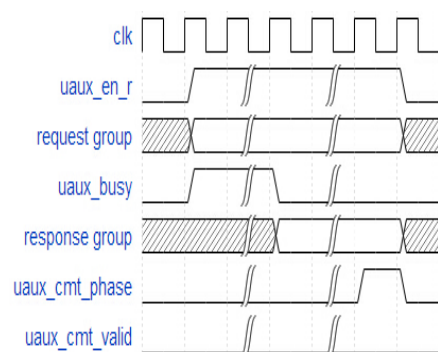
**Figure 16-4 UAUX Transaction with Waited Cycles**



## 16.4.3 Transaction Aborted

Figure 16-4 shows an example of UAUX transaction aborted by the processor core.

**Figure 16-5 UAUX Transaction Aborted**





# 17

## Power-Saving Features

This chapter describes the power-saving features of the ARC EM processor series. Information in this section can be used by SoC (System-on-a-Chip) designers and system integrators.

### 17.1 Overview

The ARC EM processor is suitable for deeply embedded systems with ultra-low power requirements. The power consumed by the processor core is just a portion of the total power of the embedded system. Memory, peripherals, and analog circuits also contribute to the total power consumption.

The ARC EM processor helps reduce the overall power consumption in several ways:

- The ARC EM processor is implemented as a fully static design with many low-power design techniques to reduce the operating-power consumption.
  - More than 95% of the flip-flops in the design are gated by clock gates inserted during synthesis.
  - Architectural clock gating further reduces standby power by gating the processor core clock early in the clock tree when the CPU is halted or in sleep state.
  - The design avoids unnecessary signal toggling in combinatorial logic, reducing dynamic power.
  - Always-on blocks related to wake-up and timers are isolated to allow clock gating or voltage reduction of remaining core modules during sleep modes.
  - Dynamic clock gating has been introduced to further reduce dynamic power using new gated clock domains.
- Architectural features of the ARC EM processor also contribute significant power savings in operating and standby modes.
  - The high code density of the ARC EM processor reduces the size of the memory (such as flash) required in the system, saving power and reducing system cost.
  - The design features a highly efficient pipeline architecture delivering a 10k gate base configuration (excluding memory macros), minimizing both static and dynamic power consumption.
  - The high cycle efficiency of the ARC EM processor allows it to be clocked at lower frequencies, reducing the total dynamic power.
- The sleep instruction enables multiple sleep modes and system level low-power states and supports prioritized interrupt wake-up sources.

- In interrupt-driven systems, the ARC EM processor can remain in sleep (or standby) mode most of the time waiting for an interrupt to wake it up.

## 17.2 Power-Saving Modes

The ARCV2 instruction-set architecture supports two main low-power states: the sleep state and the halt state. The sleep state further supports different CPU sleep modes and user-defined system level low-power states. In the sleep state and the halt state, the main CPU core clock is gated off at the root of the clock tree reducing dynamic power dissipation to a minimum.

While most CPU core and external interface clocks are off in sleep or halt modes, the clock control unit activates necessary clocks to respond to external interfaces. For example, the host can always perform transactions through the debug port even when the CPU is in a sleep or halt state. The target interface for external access to CPU closely-coupled memories can also be used when the CPU is in a sleep state or halt state. After a transaction during sleep is complete, the clocks are switched off again until the CPU exits the sleep or halt state.

The processor enters the halt state when you execute a BRK or FLAG instruction or by an external halt request pin. The processor is restarted from halt state by an external run request pin or by a debug host.

### 17.2.1 Sleep State

The ARC EM processor enters this state upon execution of the SLEEP instruction. The sleep instruction operand also allows for enabling interrupts and setting the minimum priority level that an interrupt must meet to wake the processor. For further information, see the *ARCV2 ISA Programmer's Reference*.

The processor remains in the sleep state until a qualified interrupt or a restart occurs. A pulse-sensitive interrupt cannot wake the processor; only qualified level-sensitive interrupts wake the processor.

The ARC EM processor indicates that it is in the sleep state by asserting output pin `sys_sleep_r`.

When the ARC EM processor is asleep, the core clock is turned off to reduce dynamic power consumption in the clock tree. Under these circumstances, the interrupt unit does not respond to pulse-sensitive interrupts. Users should therefore ensure that all wake-up interrupts are programmed to be level-sensitive.

If a level-sensitive interrupt is asserted when the processor is asleep, and its programmed priority is sufficient to interrupt the sleeping processor, the core clock is turned on to allow the processor to wake up and respond to the interrupt. When the core clock is re-enabled by a level-sensitive interrupt, any pulse-sensitive interrupt line is also enabled. Therefore, any higher-priority pulse-sensitive interrupt that is asserted at that time is taken in preference to the interrupt that turned on the clock.

If a debug operation is initiated when the processor is asleep, it turns on the core clock to enable the debug transaction to access the internal state of the core. At this time, pulse-sensitive interrupts are enabled briefly, and may wake up the processor if one is asserted and is of sufficient priority to wake up the processor.

Therefore, to ensure predictable wake-up behavior it is advised that only level-sensitive interrupts be used as wake-up interrupts.

#### 17.2.1.1 Sleep Modes

Further control over internal clocks is provided using the SLEEP instruction operand to enter a specific sleep mode that controls whether clocks for the internal timers and real-time clock are turned off during sleep state, and controls core outputs, `sys_sleep_mode_r[2:0]`. The sleep mode is also reflected in the Sleep Mode (SM) field of the DEBUG auxiliary register.

For each sleep mode, [Table 17-1](#) specifies the state of the various internal clocks during sleep state.

**Table 17-1 Internal Clocks During Sleep State**

| Sleep Mode (DEBUG.SM) | Core     | Timers   | Real-time Clock | IRQ Resync Clock         |
|-----------------------|----------|----------|-----------------|--------------------------|
| 0                     | Disabled | Enabled  | Enabled         | Disabled unless required |
| 1                     | Disabled | Disabled | Enabled         | Disabled unless required |
| 2                     | Disabled | Enabled  | Disabled        | Disabled unless required |
| 3                     | Disabled | Disabled | Disabled        | Disabled unless required |
| 4                     | Disabled | Disabled | Disabled        | Disabled unless required |
| 5                     | Disabled | Disabled | Disabled        | Disabled unless required |
| 6                     | Disabled | Disabled | Disabled        | Disabled unless required |
| 7                     | Disabled | Disabled | Disabled        | Disabled unless required |

### 17.2.1.2 System Level Low Power States

To facilitate different system level low power modes, the ARC EM sleep instruction operand has three bits that define the sleep mode and control core output pins, `sys_sleep_mode_r[2:0]`. These outputs can be used to control or provide status to system level power management logic, specifying the specific low-power state being entered. This, for example, can be used to determine how the CPU is to be awakened or whether a PLL is switched off or whether voltage to various modules should be reduced or switched off.

The use of these two outputs is up to the system designer. They are updated when the CPU enters the sleep mode indicated by assertion of `sys_sleep_r`; they do not return to 0 when sleep mode is exited, but rather retain their value until the next entry into sleep mode.

### 17.2.2 Halt State

The ARC EM can enter the halt state in any of the following ways:

- The FLAG instruction sets the halt bit (see the *ARCV2 ISA Programmer's Reference*).
- A BRK instruction is executed (see the *ARCV2 ISA Programmer's Reference*).
- The host (or debugger) sets the force-halt bit in the DEBUG auxiliary register.
- The processor encounters a watchpoint match (see [“Actionpoints”](#) on page 246).
- The external halt interface is used (see [“External Halt and Run Control Signals”](#) on page 168).

The processor remains in this state until the host (or debugger) clears the halt bit in the STATUS32 register. Another way of taking the processor out of the halt state is through the external run interface ([“External Halt and Run Control Signals”](#) on page 168). In either case, the processor resumes execution from the point it entered the halt state.

In this state, the processor doesn't respond to interrupt requests. The debugger has full access to the processor while the processor is in the halt state. The main CPU clock is off in halt state and the debug

interface generates a clock request to the clock control module when a debug transaction is requested by the external host.

The ARC EM processor indicates it is in the halt state asserting the output pin `sys_halt_r`.

### 17.2.3 Modules Always on in Sleep or Halted States

To facilitate wake-up from sleep, unhalt, and debug operations during sleep or halted states, certain modules receive an ungated (always-on) clock. The modules that receive an always-on clock include the following:

- A subset of the `irq_unit`, including timers
- `reset_ctrl`
- `clock_ctrl`
- `core_resync_in`

If the design contains an AHB-Lite DMI target port, then `biu_target_ahb` also uses the ungated clock. This clock is necessary because the AHB Lite protocol does not allow a target device to extend the address phase of a transaction; a capability needed to accommodate the clock turn on delay when a transaction is received while in sleep or halt mode.

### 17.2.4 External Interfaces Alive in Sleep or Halted States

The main or core clock might be temporarily switched on during sleep or halt state to service requests from external interfaces; external requests to the debug port or the DMI target port cause the clock to be enabled while the request is being serviced. In addition, external halt and run requests temporarily enable the clock to facilitate a transition between halt and sleep states. The CPU can be halted while in sleep mode; when the CPU is in the halted state, interrupts do not wake it. The processor can be restarted from the halted state by asserting the run pin or by a restart initiated by a debug host (unhalt).

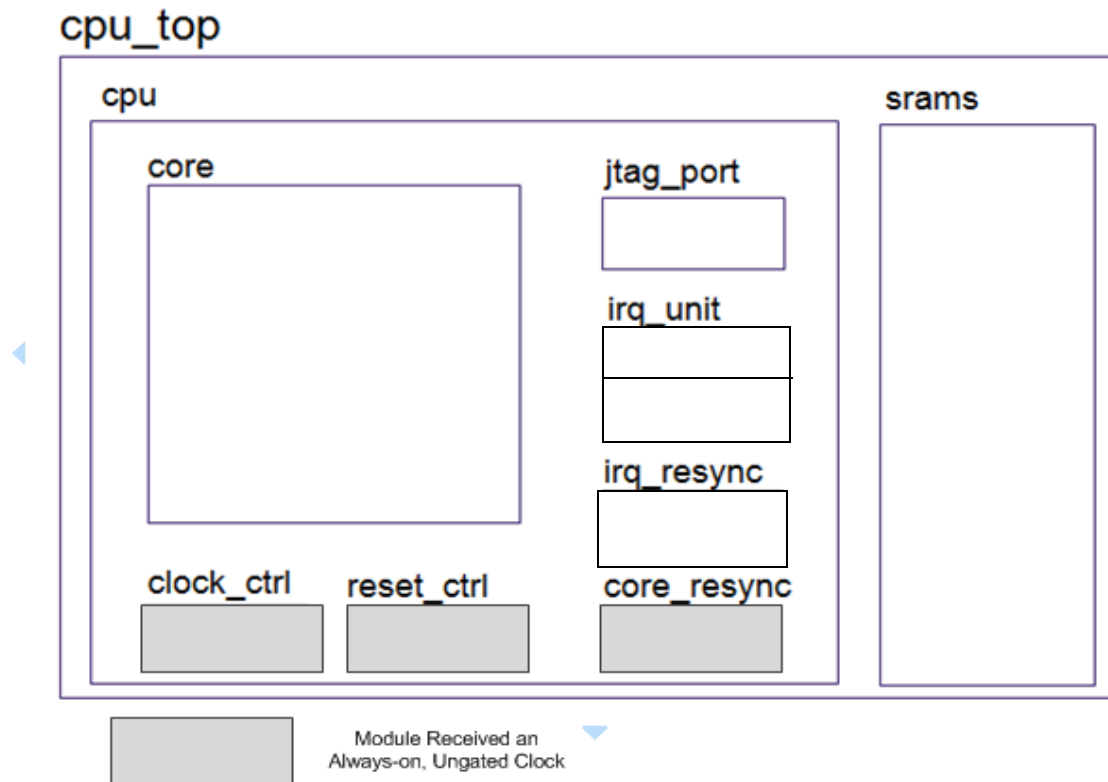
## 17.3 Architectural Clock Gating

In addition to supporting automatic clock-gate insertion during synthesis for the vast majority of flip-flops in the ARC EM design, the core supports clock gating at the macro level referred to as *architectural clock gating*. With this feature, the clock is gated near the root of the processor clock tree to minimize dynamic power dissipated in the clock tree when the processor is in sleep or halt mode. In these low power states, clock to all major processor blocks, including the core pipeline, is gated off.

### 17.3.1 Configuring Architectural Clock Gating

Architectural clock gating is a baseline feature and does not require explicit user selection. When the clock is gated (switched off), it is held in the low state. It is implemented in the `clkgate` module (equivalent to a typical clock-gating cell) within the `clock_ctrl` module located at hierarchy level `cpu_top.cpu.clock_ctrl`.

Figure 17-1 shows a block diagram of ARC EM partition with resynchronization modules.

**Figure 17-1 ARC EM Partition with Resynchronization Modules**

### 17.3.2 Actionpoints

Actionpoints support architectural clock gating. The debug enable bit (ED) at bit position 20 in the DEBUG register controls the gated clock driving the modules associated with action points.

### 17.3.3 Performance Counters and Hardware Profiler

Performance counters and the hardware profiler support architectural clock gating. The DEBUG register includes an enable performance monitoring (EP) bit at position 25. This enable bit controls the gated clock driving the modules associated with the performance counters.

## 17.4 Reset Control

The ARC EM core uses an asynchronous reset, which is first passed through two reset synchronizing flip-flops to ensure that de-assertion of the internal reset is synchronous.

The reset-control block is responsible for the synchronous de-assertion of the reset signal, such that all flip-flops come out of reset at the same clock edge.

## 17.5 Unit Level Clock Gating

In addition to architectural clock gating, the ARC EM processor supports unit level clock gating, which automatically disables the clock to major functional units when they are not in use. The units include

optional components that implement unit level clock gating and therefore use a separate gated clock synchronous with the main core clock. These units are:

- 32x32 integer multiplier
- Divide
- DSP Unit
- Floating Point Unit (part of APEX clock domain)

The module clocks are in a off state if either the architectural clock gate or the unit level clock gate is in the off state.

# 18

## Integration and Verification

After the core is configured for the application (see [Chapter 4, “Configurability”](#)), you can configure a simulation environment for initial verification. The IP library includes several support libraries for that purpose:

- The ARCV2MSS library contains testbench and memory models.
- The CCT library contains a small set of self-checking tests to verify any new configuration.

This chapter covers the configurability options for these libraries and additional simulation and verification options that you can use during a typical build.

### 18.1 IP Directory Structure

ARChitect maintains the design and verification RTL and all the associated scripts and support data in a fixed directory structure for each project.

You can specify the root directory for each project in the **New Project** dialog box. Projects created in batch mode can specify this path with the command-line option `-projectpath project_root_directory`, where *project\_root\_directory* is the desired root directory.

All directory paths specified in this chapter are located under *project\_root\_directory/build*, unless otherwise specified.

#### 18.1.1 Verilog Files

ARChitect generates Verilog code for several standard simulators:

- VCS®
- ncsim
- ModelSim Verilog

ARChitect also supports an optional VTOC (Verilog to C) target simulator. This option is part of the DesignWare ARC xCAM product.

The `verilog` directory contains testbench files and other files at the top level. The `verilog/RTL` directory contains Verilog files to make the ARC EM core.

## 18.1.2 Documents

The `docs/project.html` file contains a printable summary of the built project, including register maps, memory maps, instruction lists, and configuration information.

## 18.1.3 Build Logs

The `build` directory contains build-specific logs and other configuration data.

## 18.1.4 Argument Files

The `build` directory contains the options given to ARChitect. You can use it to recreate the design with the same configuration.

# 18.2 Support Libraries

This section describes the additional libraries available for the ARC EM processor family.

- [ARC Memory Subsystem \(MSSv2\)](#)
- [Confidence Tests: CCT Library](#)
- [Other Libraries](#)

## 18.2.1 ARC Memory Subsystem (MSSv2)

The ARC family of processors is a range of highly configurable microprocessors deployed in a soft form. This soft deployment provides for maximum flexibility in customizing and porting the processors across multiple technologies and libraries.

The ARC Memory Subsystem (MSSv2) library provides a framework for verifying ARC family processors in a simulation context as well as an FPGA context. The ARC MSSv2 library is an environment for ARC customers to:

- Evaluate the ARC processor functionality
- Assess the ARC processor performance
- Validate database integrity using Customer Confidence Tests (CCT)

The ARC MSSv2 library includes multiple components:

- BusFabric: An on-chip bus interconnect.
- ClockCtrl: A module for controlling the different clock domains.
- SRAMCtrl: An embedded SRAM controller, including a memory model supporting back-door access.
- Profiler: A performance monitor and controller
- DummyMaster: A dummy module to provide up to 4 bus masters
- DummySlave: A dummy module to provide up to 5 bus slaves



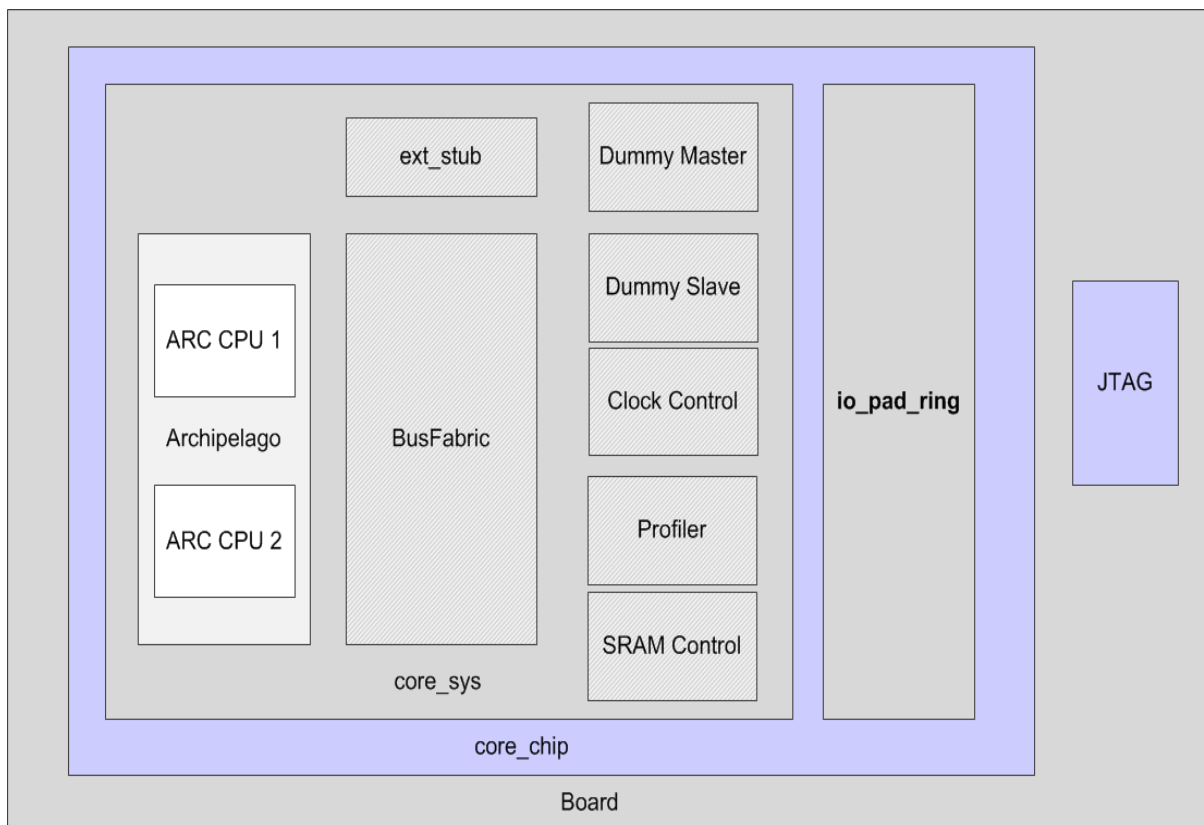
MSSv2 components can be used in the ARChitect GUI mode as well as the ARChitect command line mode. For more information on how to get started with MSSv2, see [Chapter 3, “Getting Started”](#).

Sections [“Design Hierarchy”](#) on page 313 to [“MSSv2 Components”](#) on page 314 define the detailed options for each of the MSSv2 components and list the software interfaces of the components.

### 18.2.1.1 Design Hierarchy

[Figures 18-1](#) shows the RTL design hierarchy as generated by the ARChitect. The design includes one or multiple ARC processors. If more than one processor is configured, the system hierarchy includes a processor archipelago level. If not, the processor is instantiated directly into the `cpu_sys` level. The archipelago level is used to assign unique names to CPU signals and to chain the JTAG interfaces.

**Figure 18-1 System Design Hierarchy**



The MSSv2 components (diagonal patterned) reside at the `core_sys` level, together with the CPU.

The `core_chip` level includes the I/O pads and is the hierarchy level at which FPGA synthesis runs.

At the board level, JTAG gets connected to a simulation PLI model and a miscellaneous module generates the xtal clock and the system reset.

### 18.2.1.2 Directory Structure

The MSSv2 components are implemented in Verilog. The components reside in the `build/verilog/RTL` directory created by the ARChitect. For simulation and synthesis purposes the directory includes a compile

and synthesis manifest per component. These manifest files are simple file lists, which can be passed to a simulation or synthesis tool.

To assist test software development, MSSv2 writes out a series of header files in the `build/User` directory. These files define the following:

- The system memory map
- The per-slave register map

For more information on these header files, see the [“MSSv2 Components”](#) on page 314.

### 18.2.1.3 MSSv2 Components

This section defines the options and behavior of the various MSSv2 components. All of the components support RTL simulation, FPGA mapping, and FPGA netlist simulation. The components are not supported for SoC implementation, that is, should not be put on silicon. The components are not supported beyond the ARChitect context.

For running simulations including MSSv2, see the section [“Verification Assistance”](#) on page 335. For running FPGA synthesis, see the *ARC RDF and Development Boards Bookshelf* available where you downloaded this *Databook*.

#### 18.2.1.3.1 BusFabric Function

The BusFabric component is responsible for interconnecting the master and slave modules in an MSSv2 based system. Masters are actively able to initiate bus transactions, while slaves are able to respond to bus transactions passively.

The BusFabric component supports the following features:

- Any number of masters and slaves
- Address-based transaction routing, master-side support for (a mix of) AHB, AHB-Lite, and BVC
- Slave-side support for (a mix of) AHB, AHB-Lite, APB3 and BVC
- Master and slave data width may be a mix of 32 bit or 64 bit
- Master and slaves may run at different clock ratios
- Support for exclusive and locked transactions
- Support for error responses
- Default slave support
- Configurable system memory map
- Programmable additional latency per master
- Performance monitoring per master
- Self-configuring, based on master/slave attributes

### 18.2.1.3.2 BusFabric Block Diagram

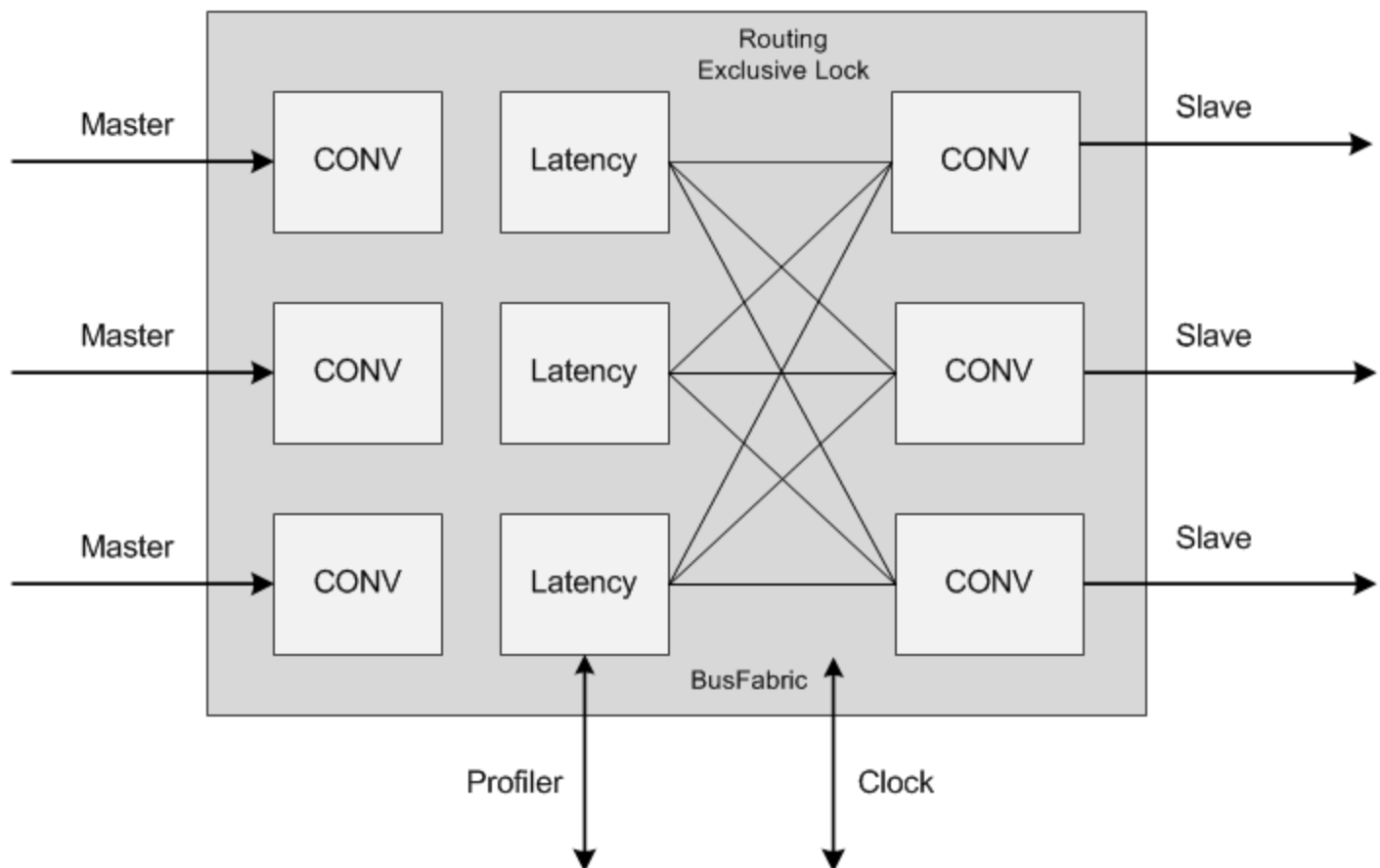
Figure 18-2 shows the BusFabric block diagram. The BusFabric has the following interfaces:

- Masters: Provide connections to bus master components, like the memory initiator port of a processor.
- Slaves: Provide connections to bus slave components, like the SRAM controller.
- Clock: Clock and clock enable inputs from the clock controller.
- Profiler: Input for latency control and outputs for bus profiling.

The BusFabric has the following main functions:

- Converter: Data width and clock-domain adaptation.
- Latency: Programmable additive latency and performance measurement.
- Routing: Transaction routing, locking and exclusive access handling.

**Figure 18-2 BusFabric Block Diagram**



### 18.2.1.3.3 BusFabric Timing Latency

Normally each MSSv2 master/slave converter module includes a register slice in the forward and return path adding four cycles to the total CPU memory access latency.

The `+define+PERF_TRANSPARENT` switch disables these register slices in the BusFabric, making them transparent. As a consequence, the BusFabric latency is reduced by four cycles. Also, in this transparent mode clocks cannot be switched dynamically.

The timing latency cost in the programmable latency module depends on the setting (refer to Profiler function about performance monitoring for BusFabric bus masters).

There is no timing latency cost in the combinational routing logic

#### 18.2.1.3.4 BusFabric System Memory Map

The BusFabric builds a system memory map based on the parameters from the slave components. For that purpose each slave defines a base address and an aperture parameter, although the aperture parameter might not be exposed to the user because it is fixed in some components.

Based on the system memory map, the BusFabric routes transaction from its master ports to its slave ports.

An ARC processor core includes one or multiple Closely Coupled Memories (CCMs). Such a processor includes one or multiple slave ports providing a Direct Memory Interface (DMI) such that external master components (for example, a second processor) can read/write into the processor CCMs. The BusFabric defines a single CCM base address parameter defining the base address for the CCMs in the system memory map. Note that this base address may be different from the processor internal CCM base address, which is typically programmable.

If multiple CCMs have been configured with a DMI port, each CCM DMI port will be allocated a position in the memory map at offsets of 1 MB.

The MSSv2 BusFabric enumerates all processor CCMs having associated DMI ports, starting with CPU 0, CPU 1, and so on and enumerating all the CCMs: ICCM0/1, DCCM. Then, the CCMs are allocated in the system address map from the base address as configured in the BusFabric.

For example, for the following two processors:

- Processor 0 having an ICCM0 and ICCM1 with a DMI interface
- Processor 1 having just a DCCM with a DMI interface

Suppose all CCMs are 4KB and the CCM base address is 0x40000000 (parameter value is 0x40000, because the parameter does not specify the lower 12 bits). The MSSv2 constructs the following address map for the CCM DMI ports:

- 0x40000000-0x400003ff: Processor0, ICCM0
- 0x40100000-0x401003ff: Processor0, ICCM1
- 0x40200000-0x402003ff: Processor1, DCCM

The BusFabric produces a C header file `build/User/alb_mss_fab_system_memmap.h` defining the system memory-map base addresses and aperture sizes as a series of `#define` macros. The header file may be included from C programs to abstract from actual addresses.

The code below shows an example header file:

```
// System memory map
#define MSS_CLKCTRL_0_BASE 0xc0000000
#define MSS_CLKCTRL_0_SIZE 0x1000
#define MSS_MEM_0_BASE 0x0000
```

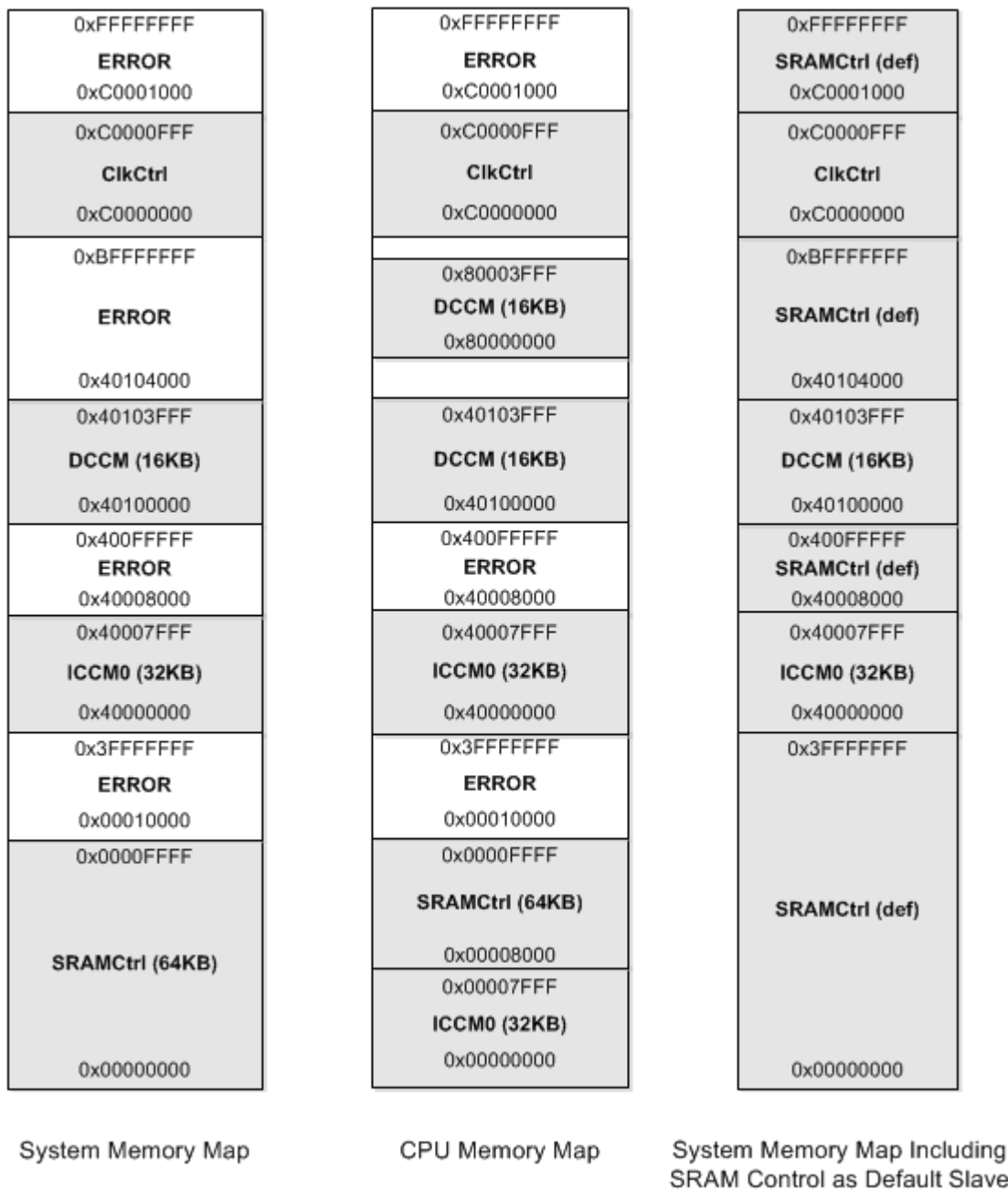
```
#define MSS_MEM_0_SIZE 0x10000  
#define HAS_DEFAULT_SLAVE 1
```

In this example:

- The clock controller slave has its base address defined at 0xc0000000; its aperture size is 4 KB, that is, the last address in its aperture is 0xc0000fff.
- The SRAM controller is at address 0, supporting 64 KB of memory.
- The system has a default slave, that is, all the bus transactions outside the listed apertures are routed to the default slave.

In Figure 18-3, the left and middle stacks show the system memory map as constructed by the BusFabric and the processor memory map as seen by the software of the processor. Basically the processor superimposes its internal CCM apertures on top of the system memory map. In this example the processor can access its ICCM0 memory through its internal port at address aperture 0x00000000 – 0x00007FFF and it can access the ICCM through the BusFabric at address aperture 0x40000000 – 0x40007FFF. Other components can only access the ICCM at the latter address range.

**Figure 18-3 System Memory Map, CPU Memory Map, and System Memory Map with Default Slave**



In [Figure 18-3](#), the right stack shows the same address map with the SRAM controller configured as the default slave, in which case all transactions get forwarded to the SRAM controller unless they specifically hit an aperture of another slave.

### 18.2.1.3.5 BusFabric Configuration Parameters

**Table 18-1 BusFabric Configuration Parameters**

| GUI Option             | Command-Line Option    | Type    | Values                     | Default Value | Description                                                                         |
|------------------------|------------------------|---------|----------------------------|---------------|-------------------------------------------------------------------------------------|
| Maximum latency        | -alb_mss_fab_lat       | Option  | 0, 8, 16, 32, 64, 128, 256 | 0             | Defines the maximum latency that may be programmed in the BusFabric latency modules |
| Default latency        | -alb_mss_fab_def_lat   | Integer | 0 - maximum latency        | 0             | The latency setting after reset                                                     |
| Default clock ratio    | -alb_mss_fab_def_ratio | Integer | 1-16                       | 1             | The clock ratio after reset                                                         |
| CCM base address [4KB] | -alb_mss_ccm_base      | Integer | 0 - maximum address        | 0x40000       | The base address for the CCMs in the System Address Map in units of 4KB pages       |

### 18.2.1.3.6 BusFabric Self-Configuration

The BusFabric configures itself based on the parameters of the masters and slaves. The most relevant parameters are the slave base addresses and the slave aperture sizes (\*base\_addr and \*size).

### 18.2.1.3.7 SRAM Controller Function

The SRAM controller component is an embedded memory controller, including a configurable memory size. The memory latency may be reprogrammed through the profiler component. Throughput can be measured through the profiler module. The controller fully pipelines transactions, even at non-zero-additive latency.

The SRAM Controller component supports the following features:

- Configurable SRAM size
- Optional system default slave
- AXI 64-bit slave interface
- Programmable additional latency
- Command throughput: One command per two cycles
- Data throughput: One data per cycle
- Support for RASCAL back-door memory access in MetaWare debugger
- Support for FPGA memory inference

For FPGA synthesis the SRAM controller needs to be compiled with the `FPGA_INFER_MEMORIES` define switch.

### 18.2.1.3.8 SRAM Controller Block Diagram

The SRAM controller has the following interfaces:

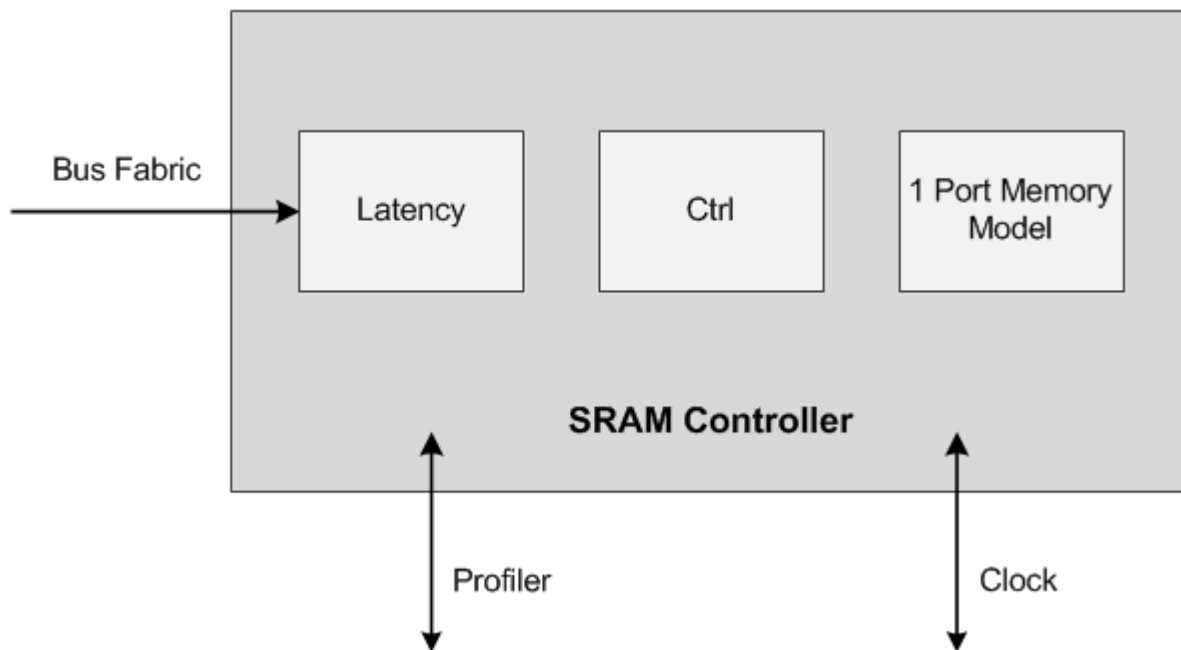
- BusFabric: the SRAM controller is a slave on the BusFabric, that is, the controller responds to transactions forwarded by the BusFabric.
- Clock: Clock and clock-enable inputs from the clock controller.
- Profiler: Input for latency control and outputs for performance profiling.

The Controller has the following main functions:

- Latency: programmable additive latency and performance measurement.
- Control: The controller converts the AXI protocol to a regular single port SRAM protocol.
- Single-port memory model: Behavioral model for synchronous single-port memory, including support for FPGA memory inference and RASCAL back-door access.

Figures 18-4 shows the block diagram of the SRAM Controller.

**Figure 18-4 SRAM Controller Block Diagram**



### 18.2.1.3.9 SRAM Controller Access Latency

The default memory access latency for SRAMCtrl module is four cycles. For performance monitoring considerations, the extra latency cost in SRAMCtrl can be configured by programming the Profiler of MSSv2 or setting the maximum latency of SRAMCtrl.



### 18.2.1.3.10 SRAM Controller Configuration Parameters

**Table 18-2 SRAM Controller Configuration Parameters**

| GUI Option                 | Command Line Option            | Type    | Values                     | Default Value | Description                                                                    |
|----------------------------|--------------------------------|---------|----------------------------|---------------|--------------------------------------------------------------------------------|
| Maximum latency            | '-alb_mss_mem_lat              | Option  | 0, 8, 16, 32, 64, 128, 256 | 0             | The maximum latency that may be programmed in the SRAM control latency modules |
| Default latency            | '-alb_mss_mem_def_lat          | Integer | 0 - maximum latency        | 0             | The latency setting after reset                                                |
| Default clock ratio        | '-alb_mss_mem_def_ratio        | Integer | 1-16                       | 1             | The clock ratio after reset                                                    |
| Memory base address (4 KB) | '-alb_mss_mem_base_addr        | Integer | 0 - maximum address        | 0x00000       | The base address for the CCMs in the System Address Map in units of 4KB pages  |
| Memory size                | '-alb_mss_mem_size             | Option  | 4 KB...4 GB                | 64 KB         | The size of the memory from 4 KB up to 4 GB, power of two                      |
| Default slave              | '-alb_mss_mem_is_default_slave | Boolean | No/yes                     | No            | If true then controller is the default slave in system                         |

### 18.2.1.3.11 Clock Controller Function

The clock controller generates all clocks in the system, deriving them from the single `core_chip` clock input. All clocks are generated synchronously, that is, the phases of the output clocks is aligned. The clock ratios and the clock-enable ratios can be reprogrammed on-the-fly through the APB3 slave interface of the clock controller.

The clock controller component supports the following features:

- Clock-ratio reprogramming
- Clock-enable reprogramming
- APB3 slave interface, with fixed 4 KB aperture
- Self-configuration

### 18.2.1.3.12 Clock Controller Block Diagram

The clock controller has the following interfaces:

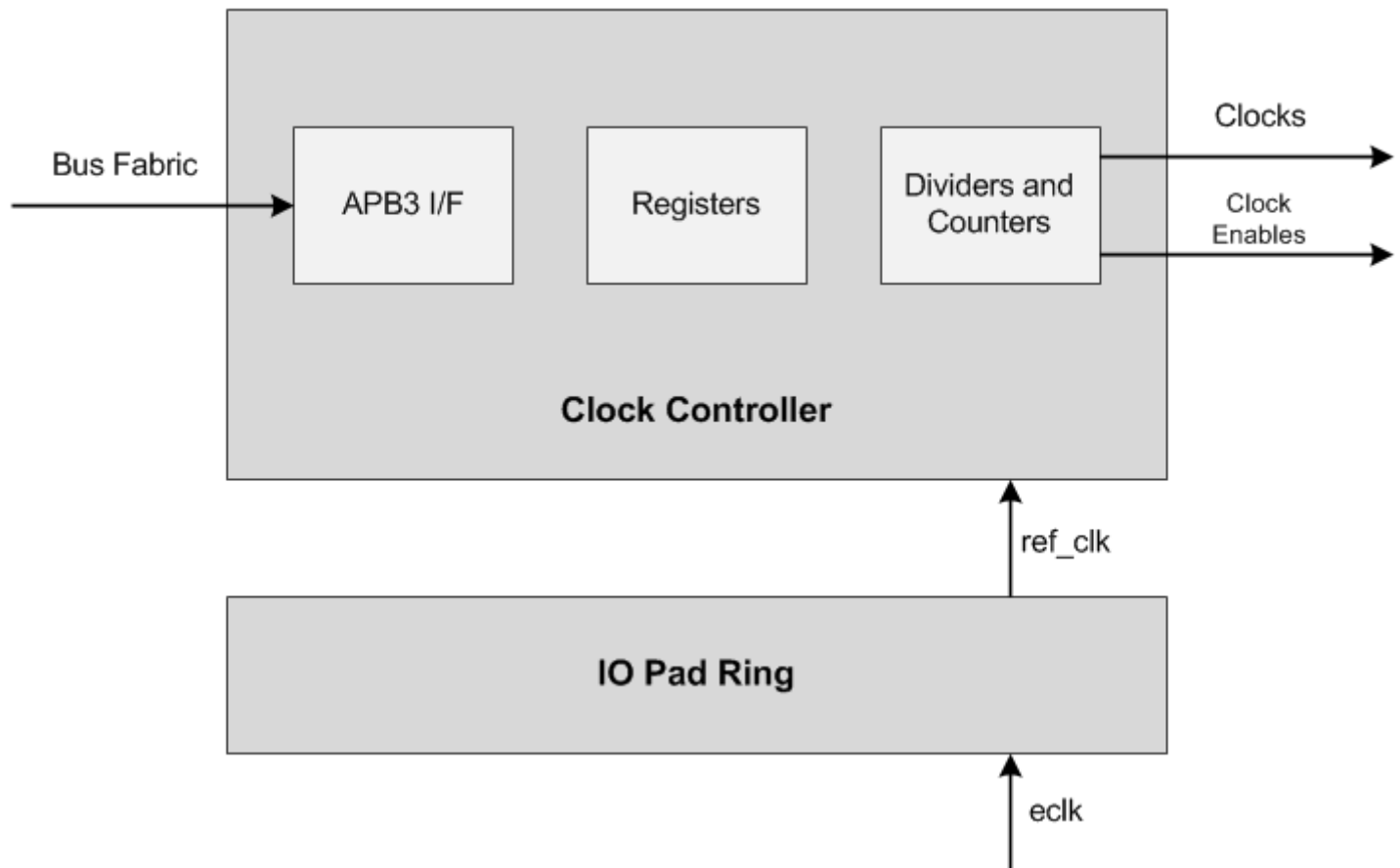
- `ref_clk`: Input reference clock from the I/O pad ring.
- `BusFabric`: APB3 slave interface to access the control and status registers.
- `Clocks`: Output clocks to the components in the system.
- `Clock enables`: Output clock enables to the components in the system.

The controller has the following main functions:

- APB3 interface: Module handling the APB protocol.
- Registers: The module control and status registers that can be accessed over the APB3 interface.
- Dividers and Counters: Module dividing down the reference clock and generating the clock enable outputs.

Figure 18-5 shows the block diagram of the clock controller.

**Figure 18-5 Clock Controller Block Diagram**



### 18.2.1.3.13 Clock Controller Configuration Parameters

**Table 18-3 Clock Controller Configuration Parameters**

| GUI Option          | Command Line Option         | Type    | Values              | Default Value | Description                     |
|---------------------|-----------------------------|---------|---------------------|---------------|---------------------------------|
| Default clock ratio | '-alb_mss_clkctrl_def_ratio | Integer | 1-16                | 1             | Clock ratio setting after reset |
| Base address [4KB]  | '-alb_mss_clkctrl_base_addr | Integer | 0 - maximum address | 0xc0000       | The latency setting after reset |

### 18.2.1.3.14 Clock Controller Self-Configuration

The clock controller configures itself based on the parameters of the other components in the system. The most relevant parameter is the component default clock ratio (`*def_ratio`), which specifies the clock ratio after reset.

### 18.2.1.3.15 Clock Controller Software Register Interface

In [Table 18-4](#), the 32-bit registers are located at 4 byte address offsets corresponding to the clock-controller base address as defined in the system memory map.

- The parameter *N* represents the total number of clock enables in the system, multiplied by 4.
- The parameter *M* represents the number of different peripheral clocks in the system, multiplied by 4.
- The parameter *P* represents the number of processor clocks, multiplied by 4.

**Table 18-4 Clock Controller Software Register View**

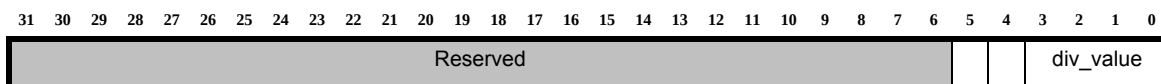
| System Memory Address Offset | Register Name                       | Access Type | Description                                              | Initial RESET Value                              |
|------------------------------|-------------------------------------|-------------|----------------------------------------------------------|--------------------------------------------------|
| 0                            | MSS_CLKCTRL_REG_CFG_ADDR            | R/W         | Clock and clock enable config register                   | 0                                                |
| 1..N                         | MSS_CLKCTRL_REG_<name>_RATIO_ADDR   | R/W         | Clock enable ratio for clock enable <i>name</i>          | From component <code>_def_ratio</code> parameter |
| N+1..N+M                     | MSS_CLKCTRL_REG_<name>_DIV_ADDR     | R/W         | Clock divider ratio for peripheral clock <i>name</i>     | 0                                                |
| N+M+1..N+M+P                 | MSS_CLKCTRL_REG_<name>_CLK_DIV_ADDR | R/W         | Clock divider ratio for CPU with unique name <i>name</i> | 0                                                |

For C programming purposes the clock controller module includes a C header file `<build>/User/alb_mss_clkctrl_regmap.h` defining all register addresses as C macros.

The MSS\_CLKCTRL\_REG\_CFG register is implemented to make the clock and clock enable settings to be activated at the same time. Valid value is 1'b1.

All of the clock controller registers define four-bit divider values as defined in the [Figure 18-6](#).

**Figure 18-6 Clock Controller Register**



The ratios are programmed in -1 coding, that is, a register value 0 corresponds to a 1:1 ratio; a register value 1 corresponds to a 1:2 ratio.

Both clock dividers and clock enabled dividers can be programmed at any value.

### 18.2.1.3.16 Profiler Function

The profiler provides performance measuring capabilities of all bus masters in the BusFabric, and the SRAM Controller bus. The profiler contains and controls for each of these buses performance counters to measure throughputs and latencies. For measuring latency impact on applications, the profiler also provides the ability to program additive latencies to each of the buses.

The profiler supports the following features:

- APB3 slave interface, with fixed 4 KB aperture.
- The following set of 64-bit performance counters for each of the BusFabric bus masters, and the SRAM Controller bus:
  - Number of write transactions (a transaction consists of one or more data beats).
  - Number of write data beats.
  - Total number of latency cycles from arrival of a write transaction until acceptance write transaction.
  - Total number of latency cycles from acceptance of a write transaction until arrival of first write data beat.
  - Total number of latency cycles from acceptance of a write transaction until arrival of last write data beat.
  - Total number of latency cycles from acceptance of a write transaction until arrival of write response.
  - Number of read transactions.
  - Number of read data beats.
  - Total number of latency cycles from arrival of a read transaction till acceptance of a read transaction.
  - Total number of latency cycles from acceptance of a read transaction until arrival of first read data beat.
  - Total number of latency cycles from acceptance of a read transaction until arrival of last read data beat.
- 64 bit clock tick counter to count the number of clock cycles during which the performance counters were enabled.
- Programmable additive latency for each of the BusFabric bus masters, and the SRAM controller bus.

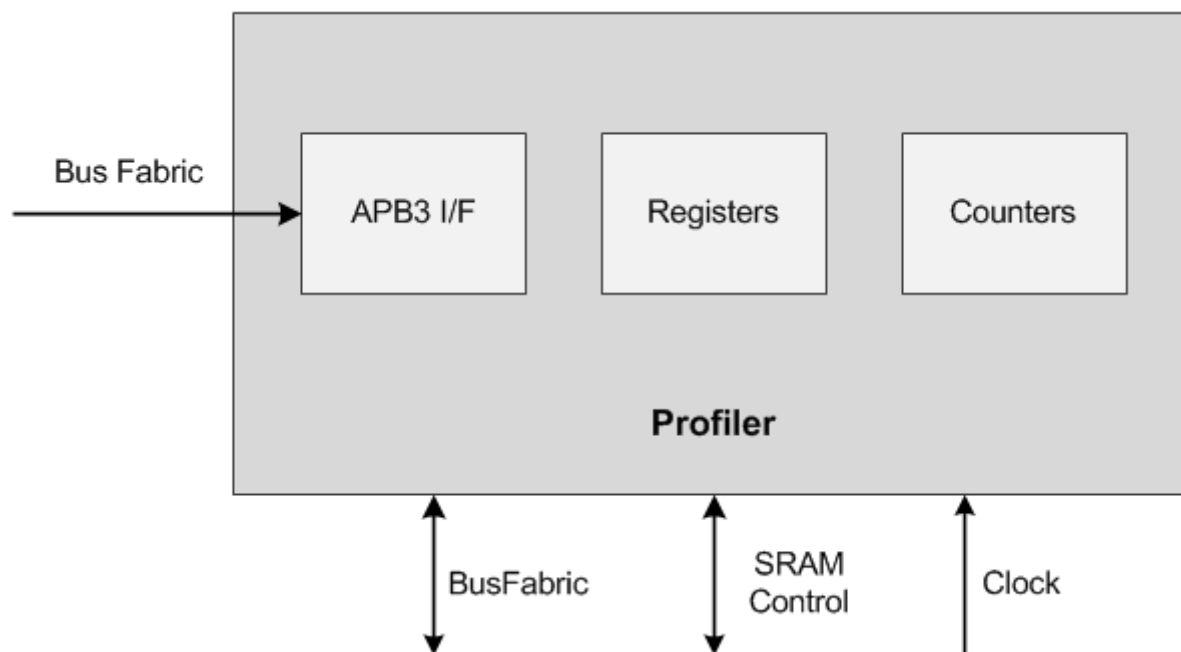
### 18.2.1.3.17 Profiler Block Diagram

Figure 18-7 shows the Profiler block diagram. The profiler has the following interfaces:

- BusFabric: the Profiler controller is a slave on the BusFabric, that is, the controller responds to transactions forwarded by the BusFabric.
- BusFabric: Output for latency control and inputs for performance profiling.
- SRAM Ctrl: Output for latency control and inputs for performance profiling.
- Clock: Clock and clock enable inputs from the clock controller.

The profiler has the following main functions:

- APB3 interface: Module for handling the APB protocol.
- Registers: The module control and status registers that can be accessed over the APB3 interface.
- Counters: Module that counts bus profile events.

**Figure 18-7 Profiler Block Diagram**

#### 18.2.1.3.18 Profiler Configuration Parameters

Table 18-5 lists the configuration parameters of the profiler.

**Table 18-5 Profiler Configuration Parameters**

| GUI Option                | Command Line Option         | Type    | Values               | Default Value | Description                                                                        |
|---------------------------|-----------------------------|---------|----------------------|---------------|------------------------------------------------------------------------------------|
| Default clock ratio       | -alb_mss_perfctrl_def_ratio | Integer | 1-16                 | 1             | The clock ratio setting after reset                                                |
| Memory base address [4KB] | -alb_mss_perfctrl_base_addr | Integer | 0 to maximum address | 0xc0001       | The base address for the profiler in the system address map in units of 4 KB pages |

#### 18.2.1.3.19 Profiler Self-Configuration

The Profiler configures itself based on the parameters of the bus masters and the SRAM controller.

### 18.2.1.3.20 Profiler Software Register Interface

[Figure 18-6](#) lists all registers of the profiler. Each of the 32-bit registers has multiples of four-byte address offsets regarding the Profiler base address. For more details, see section [“BusFabric System Memory Map”](#) on page [316](#).

For each of the BusFabric bus masters and the SRAM controller, if present, the profiler provides a set of registers to control the latency added to bus responses, and to read out current performance measurements of that bus. A maximum of 30 sets of registers is available for profiling BusFabric bus masters (register sets with  $0 \leq i < 30$  in Figure 18-6). When there are more bus masters, only the first 30 are under control of the Profiler. Profiling set  $i=30$  is reserved for the SRAM controller bus.

The maximum latency value that can be programmed in the LAT\_CTRL\_i registers is determined by the -alb\_mss\_fab\_lat option for the BusFabric bus masters, and -alb\_mss\_mem\_lat option for the SRAM control bus.

The performance counters are 64 bits wide. For each performance counter two read-only registers are present, that is, one for the lower 32 bits of the counter value and one for the upper 32 bits. The same applies to the global clock tick counter.

**Table 18-6 Profiler Software Register View**

| System Memory Address Offset | Register Name: MSS_PERFCTRL_REG . . . | Access Type | Description                                                                                                    | Initial Reset Value    |
|------------------------------|---------------------------------------|-------------|----------------------------------------------------------------------------------------------------------------|------------------------|
| 0x000 + (i*0x80)             | . . . _LAT_CTRL_i                     | R/W         | Additive latency for bus i                                                                                     | *_def_lat (see Note 1) |
| 0x008 + (i*0x80)             | . . . _AW_COUNT__LO_i                 | R           | Total number of write transactions for bus i (lower 32 bits)                                                   | 0                      |
| 0x00C + (i*0x80)             | . . . _AW_COUNT_HI_i                  | R           | Total number of write transactions for bus i (upper 32 bits)                                                   | 0                      |
| 0x010 + (i*0x80)             | . . . _W_COUNT__LO_i                  | R           | Total number of write data beats for bus i (lower 32 bits)                                                     | 0                      |
| 0x014 + (i*0x80)             | . . . _W_COUNT_HI_i                   | R           | Total number of write data beats for bus i (upper 32 bits)                                                     | 0                      |
| 0x018 + (i*0x80)             | . . . _AW_LAT_LO_i                    | R           | Total number of cycles from arriving of write transaction until acceptance for bus i (lower 32 bits)           | 0                      |
| 0x01C + (i*0x80)             | . . . _AW_LAT_HI_i                    | R           | Total number of cycles from arriving of write transaction until acceptance for bus i (upper 32 bits)           | 0                      |
| 0x020 + (i*0x80)             | . . . _WF_LAT_LO_i                    | R           | Total number of cycles from acceptance write transaction until first write data beat for bus i (lower 32 bits) | 0                      |
| 0x024 + (i*0x80)             | . . . _WF_LAT_HI_i                    | R           | Total number of cycles from acceptance write transaction until first write data beat for bus i (upper 32 bits) | 0                      |
| 0x028 + (i*0x80)             | . . . _WL_LAT_LO_i                    | R           | Total number of cycles from acceptance write transaction until last write data beat for bus i (lower 32 bits)  | 0                      |

**Table 18-6 Profiler Software Register View**

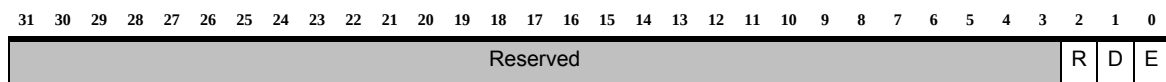
|                     |                    |   |                                                                                                               |      |
|---------------------|--------------------|---|---------------------------------------------------------------------------------------------------------------|------|
| 0x02C +<br>(i*0x80) | ..._WL_LAT_HI_i    | R | Total number of cycles from acceptance write transaction until last write data beat for bus i (upper 32 bits) | 0    |
| 0x030 +<br>(i*0x80) | ..._B_LAT_LO_i     | R | Total number of cycles from acceptance write transaction until write response for bus i (lower 32 bits)       | 0    |
| 0x034 +<br>(i*0x80) | ..._B_LAT_HI_i     | R | Total number of cycles from acceptance write transaction until write response beat for bus i (upper 32 bits)  | 0    |
| 0x038 +<br>(i*0x80) | ..._AR_COUNT__LO_i | R | Total number of read transactions for bus i (lower 32 bits)                                                   | 0    |
| 0x03C +<br>(i*0x80) | ..._AR_COUNT_HI_i  | R | Total number of read transactions for bus i (upper 32 bits)                                                   | 0    |
| 0x040 +<br>(i*0x80) | ..._R_COUNT__LO_i  | R | Total number of read data beats for bus i (lower 32 bits)                                                     | 0    |
| 0x044 +<br>(i*0x80) | ..._R_COUNT_HI_i   | R | Total number of read data beats for bus i (upper 32 bits)                                                     | 0    |
| 0x048 +<br>(i*0x80) | ..._AR_LAT_LO_i    | R | Total number of cycles from arriving of read transaction until acceptance for bus i (lower 32 bits)           | 0    |
| 0x04C +<br>(i*0x80) | ..._AR_LAT_HI_i    | R | Total number of cycles from arriving of read transaction until acceptance for bus i (upper 32 bits)           | 0    |
| 0x050 +<br>(i*0x80) | ..._RF_LAT_LO_i    | R | Total number of cycles from acceptance read transaction until first read data beat for bus i (lower 32 bits)  | 0    |
| 0x054 +<br>(i*0x80) | ..._RF_LAT_HI_i    | R | Total number of cycles from acceptance read transaction until first read data beat for bus i (upper 32 bits)  | 0    |
| 0x058 +<br>(i*0x80) | ..._RL_LAT_LO_i    | R | Total number of cycles from acceptance read transaction until last read data beat for bus i (lower 32 bits)   | 0    |
| 0x05C +<br>(i*0x80) | ..._RL_LAT_HI_i    | R | Total number of cycles from acceptance read transaction until last read data beat for bus i (upper 32 bit)    | 0    |
| 0xF80               | ..._CTRL           | W | Control the performance profiling                                                                             | N.A. |
| 0xF88               | ..._TICK_COUNT_LO  | R | Total number of clock cycles the profiling has been active (lower 32 bit)                                     | 0    |



**Table 18-6 Profiler Software Register View**

|                                                                                                                                                                                                                                                               |                   |   |                                                                           |            |
|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|-------------------|---|---------------------------------------------------------------------------|------------|
| 0xF8C                                                                                                                                                                                                                                                         | ..._TICK_COUNT_HI | R | Total number of clock cycles the profiling has been active (upper 32 bit) | 0          |
| 0xFB8                                                                                                                                                                                                                                                         | ..._COMP_VERSION  | R | Component Version                                                         | 0x10000    |
| 0xFBC                                                                                                                                                                                                                                                         | ..._COMP_TYPE     | R | Component Type                                                            | 0x41443F3F |
| <b>Note:</b><br>1. The initial reset value of the LAT_CTRL_i registers is defined by the alb_mss_fab_def_lat configuration option for BusFabric bus masters ( $0 \leq i < 30$ ) and the alb_mss_mem_def_lat configuration option for the SRAM controller bus. |                   |   |                                                                           |            |

One write-only register, MSS\_PERFCTRL\_REG\_CTRL, is present to control the performance profiling, that is, to enable, disable and/or reset the profiling. When writing a 1 into a bit, the corresponding function ((E)nable, (D)isable, (R)eset) is activated. Writing a 0 into a bit has no impact on that function.

**Figure 18-8 MSS\_PERFCTRL\_REG\_CTRL Fields Register**

For C programming purposes, the Profiler module includes a C header file `build/User/alb_mss_perfctrl_regmap.h` defining all register addresses as C macros.

### 18.2.1.3.21 Dummy Master Function

The Dummy Master provides dummy bus masters to allow customer's peripheral devices to be integrated into the MSSv2-based system and reuse the BusFabric for evaluation and fast prototyping.

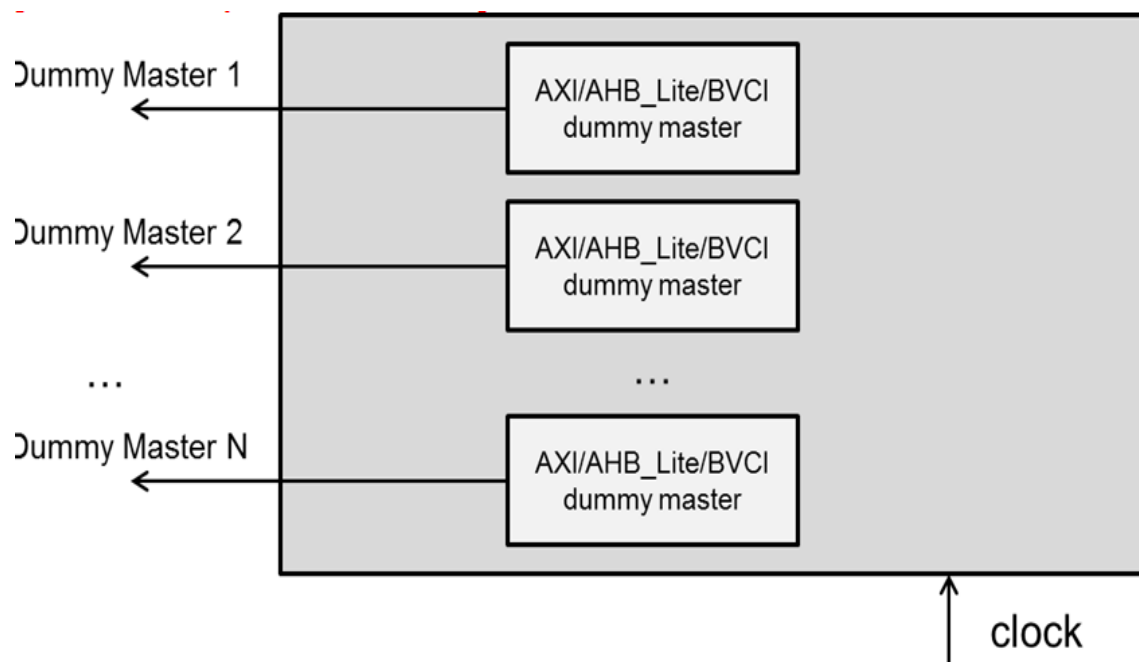
The Dummy Master component supports the following features:

- Configurable number of dummy masters (up to 4)
- Configurable bus protocols of each dummy master (AXI3/AHB\_Lite/BVCI)
- Configurable data width of each dummy master (32/64)

### 18.2.1.3.22 Dummy Master Block Diagram

The Dummy Master has the following interfaces:

- Clock: clock and clock enable inputs from the Clock Controller
- BusFabric: Dummy Master provides several master ports which can be routed in BusFabric for transaction arbitration and communication
- Standard bus interfaces: Provides AXI3/AHB\_Lite/BVCI bus protocol master interface connection to BusFabric.

**Figure 18-9 Dummy Master Block Diagram****18.2.1.3.23 Dummy Master Configuration Parameters****Table 18-7 Dummy Master Configuration Parameters**

| GUI Option                               | Command Line Option             | Type    | Values              | Default Value | Description                         |
|------------------------------------------|---------------------------------|---------|---------------------|---------------|-------------------------------------|
| Default Clock Ratio                      | -alb_mss_dummy_master_def_ratio | Integer | 1-16                | 1             | The clock ratio after reset         |
| Number of dummy masters                  | -alb_mss_dummy_master_num       | Integer | 1-4                 | 4             | Number of dummy masters             |
| Prefix of dummy master M (M from 1 to 4) | -alb_mss_dummy_master<M>_pref   | String  |                     | dmst<M><br>—  | The unique prefix of dummy master M |
| Bus protocol of dummy master M           | -alb_mss_dummy_master<M>_prot   | String  | AXI, AHB_Lite, BVCI | AXI           | Bus protocol of dummy master M      |
| Data width of dummy master M             | -alb_mss_dummy_master<M>_dw     | Integer | 32,64               | 32            | Data width of dummy master M        |

**18.2.1.3.25 Dummy Slave Function**

The Dummy Slave provides dummy bus slaves to allow customer's peripheral devices to be integrated into the MSSv2-based system and reuse the BusFabric for evaluation and fast prototyping.

The Dummy Slave component supports the following features:

- Configurable number of dummy slaves (up to 5)
- Configurable bus protocols of each dummy slave (APB/AXI3/AHB\_Lite/BVCI)
- Configurable data width of each dummy master (32/64)

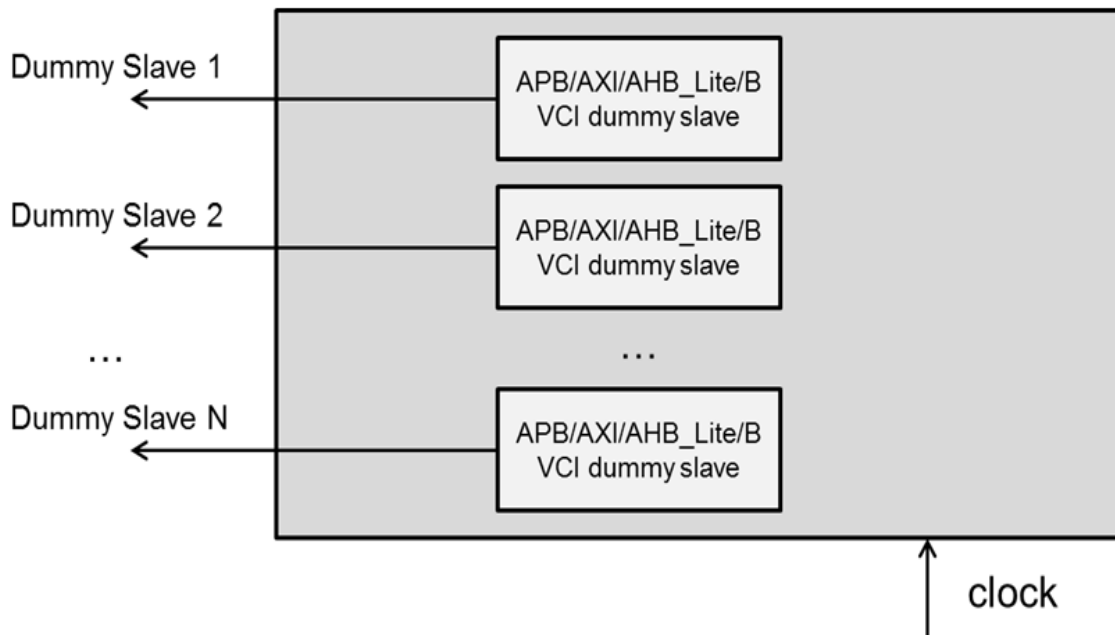
#### 18.2.1.3.26 Dummy Master Block Diagram

The Dummy Master has the following interfaces:

- Clock: clock and clock enable inputs from the Clock Controller
- BusFabric: Dummy Master provides several master ports which can be routed in BusFabric for transaction arbitration and communication

Standard bus interfaces: Provides APB/AXI3/AHB\_Lite/BVCI bus protocol master interface connection to BusFabric.

**Figure 18-10 Dummy Slave Block Diagram**



#### 18.2.1.3.27 Dummy Master Configuration Parameters

**Table 18-8 Dummy Master Configuration Parameters**

| GUI Option             | Command Line Option             | Type    | Values | Default Value | Description                 |
|------------------------|---------------------------------|---------|--------|---------------|-----------------------------|
| Default Clock Ratio    | -alb_mss_dummy_master_def_ratio | Integer | 1-16   | 1             | The clock ratio after reset |
| Number of dummy slaves | -alb_mss_dummy_slave_num        | Integer | 1-5    | 5             | Number of dummy slaves      |

**Table 18-8 Dummy Master Configuration Parameters (Continued)**

| GUI Option                              | Command Line Option            | Type    | Values                  | Default Value | Description                                                      |
|-----------------------------------------|--------------------------------|---------|-------------------------|---------------|------------------------------------------------------------------|
| Base address of dummy slave             | -alb_mss_dummy_slave_base_addr | Integer |                         | 0xc0006       | Base address of dummy slave 1 in the memory map, divided by 4KB. |
| Prefix of dummy slave S (S from 1 to 5) | -alb_mss_dummy_slave<S>_pref   | String  |                         | dslv<S>_      | The unique prefix of dummy slave S                               |
| Bus protocol of dummy slave S           | -alb_mss_dummy_slave<S>_prot   | String  | APB,AXI, AHB_Lite, BVCi | AXI           | Bus protocol of dummy slave S                                    |
| Data width of dummy slave S             | -alb_mss_dummy_slave<S>_dw     | Integer | 32,64                   | 32            | Data width of dummy slave S                                      |

### 18.2.2 Confidence Tests: CCT Library

The CCT IP library is bundled with the ARC EM package. This support library is supplied with the ARC EM for basic verification purposes only. It contains a small set of assembly and C-code tests and debugger command files to execute on a new build.

The CCT library is configured as a component, and must be included in the build to generate the tests for that build. Each build has specific requirements for memory and other resources. ARChitect is programmed to construct a subset of tests to execute correctly for the target configuration.

After creating a build with CCT tests, you can use the ARCTest tool to run these tests (see “[ARCTest](#)” on page 335). Note that the `tests` directory is not erased if the configuration is changed and re-compiled, so you might find unused tests during a long session of trying different build configurations under the same project (same `build` directory). These unused tests are not invoked by ARCTest.

Including the CCT library in the project does *not* mean that tests are automatically compiled and run. To compile the tests, you must check the following options in the **Explorer** pane, **Build Sequence** tab in ARChitect:

- `-build_test_code`: copies tests to `build/tests` directory and constructs appropriate makefiles. Also lists all valid tests in `build/test.xref` for ARCTest.
- `-compile_test_code`: assembles or compiles tests and generates ELF (\*.out) executables for loading into the simulation code memory. In the process, the compiler also generates .hex, .sym and .dmp files to assist with debugging the code.

### 18.2.3 Other Libraries

ARChitect includes support for back-end flow, including synthesis and power estimation.

Specific DesignWare and vendor libraries must be loaded to enable these features.

Refer to [Chapter C, “Synthesis and Physical Implementation”](#) and the *ARC® Synopsys Design Flow IP Library Bookshelf* for more details.

### 18.2.3.1 Build Options

ARChitect includes several options to control the content of a build. It is possible to produce an RTL simulation model only or direct ARChitect to produce additional data, such as makefiles for back-end flow. See [“Build-Sequence Options”](#) on page 333.

### 18.2.3.2 Simulator Options

- **-simulator**: select between **vcs**, **nc**, **modelsim**, **none**.

RTL for the core and testbench is always Verilog, regardless of the simulator selected. This option mainly impacts makefiles associated with compilation and simulation control for a specific simulator. Specifying **none** is used for generating xCAM simulations, as explained below.

- **-vcs64**: the default for executable models and PLI object code is 32-bit, even when running them on a 64-bit machine. In some cases, a larger design might require the executable to use the larger address space of a 64-bit machine.

This option might require additional tweaks to the build. For more information, see *ARC EM Integration Guide*.

- **-verilog\_2001**: ARChitect can produce code with earlier versions of Verilog that do not support the file I/O functions of the Verilog-2001 standard, but it is best for all new designs to utilize the Verilog-2001 file I/O by checking this box in the ARChitect GUI **Options** pane (setting this value to true).
- **-build\_xcam**: this option is in the **Explorer** pane, **Build Sequence** tab in ARChitect.

This option generates a compiled C version of the simulation model by means of a VTOC tool (Verilog to C compiler) included in DesignWare ARC xCAM. This option is only valid when the build includes the VTOC component, available as a separate IP library specifically for the DesignWare ARC processor cores. The VTOC component requires a separate license.

When a design is compiled for xCAM simulation, ARChitect creates a **build/vtoc** directory with the necessary files and data to compile and run the xCAM simulation.

## 18.3 Build-Sequence Options

This section provides a brief description of the options in the **Explorer** pane, **Build Sequence** tab in ARChitect, and other related options.



#### Note

View the ARChitect tool tips by hovering over the options, or use command-line option **-d** for arguments, defaults, notes, and important cautions.

- **-build\_html\_docs**: creates HTML documentation in the **build/docs** directory.
- **-build\_scripts**: creates scripts and makefiles.

Some of these files are essential for correct functioning of the tools and therefore might be created even if this option is not checked.

- `-build_hdl`: generates RTL files in the `build/verilog` directory.
- `-generate_structural_hdl`: generates necessary structural RTL modules. Keep this option checked (true) as a default.
- `-compile_hdl_for_simulation`: compiles the RTL to executable (binary) model for simulations.
- `-build_xcam`: translates Verilog to a two-state C model and compiles an executable (binary) model for simulations. This option requires the xCAM VTOC IP library and an appropriate license.
- `-build_test_code`: copies appropriate assembly and C-code tests in the `build/tests` directory and prepares all support information for running these tests.
- `-compile_test_code`: assembles and links the tests to an ARC EM executable binary.
- `-generate_ipxact`: generates an `ipxact.xml` file, describing the ARC EM core, for use with coreAssembler.

This option is listed in the **Implementation** component in the ARChitect **Options** pane.

- `-ipxact_relative_path_names`: selects between relative or absolute Verilog path names in `ipxact.xml`. This option is listed in the **Implementation** component in the ARChitect **Options** pane.

The following options are associated with the back-end reference design flow. Check the specific RDF documents for details.

- `-run_arcsyn`: uses ARCSyn to run the synthesis flow  
You can build your synthesis scripts by checking **Create Synthesis Scripts** in the ARChitect **Build Sequence** tab (if you have included the RDF components in your design).
- `-run_seif`: runs Synopsys EMBED-IT! integrator flow to generate configured memory instances. You must also enable `-run_arcrams` to run SEIF, and the SEIF component must be included in your design.
- `-run_arcrams`: inserts vendor memory models into the simulation and changes synthesis scripts to use the specific memories.
- `-run_arcformal`: generates and runs formal verification scripts.

## 18.4 Building the Design

After you configure the core and testbench and select all build options, the project is ready for generating RTL for simulations.

### Check Dependencies

The first step is to test for violations of library rules by invoking a dependency check. You can do so by selecting **Project > Check Dependencies...** or clicking the following button: 

You must resolve all reported errors; otherwise ARChitect does not proceed with the build process.

ARChitect also reports warnings. Warnings do not stop the build process, but it is important to review and understand them before proceeding with the build.

## Other Checks

Before you commit a design to the build process, check the following:

- IP libraries invoked with the project; see the GUI Project menu (or use the F12 key)
- Options in the **Build Sequence** pane of the **Explorer**.

## Build

The next step is to build the design. You can do so by selecting **Project > Build** or clicking the following button: 

## 18.5 Verification Assistance

### 18.5.1 ARCTest

After creating a build with CCT tests ("[Confidence Tests: CCT Library](#)" on page 332), you can use the ARCTest tool to run the tests. Confidence tests (CCT) are written to certain specifications, so they can be loaded and run by the MetaWare debugger with little or no user intervention. ARCTest is documented in the *ARChitect Tools Bookshelf*, and the MetaWare debugger is documented in the *MetaWare Development Toolkit Bookshelf*.

Tests are self-checking and communicate the final pass/fail result to the debugger by following simple conventions. Only valid tests are created and compiled for a given configuration, so they are expected to pass.

You can invoke ARCTest to run all valid tests. You can list all valid tests by issuing the following command in the `build` directory:

```
ARCTest -info
```

To run a specific test, check its number and issue the following command, where *N* is the test number:

```
ARCTest -sc_mode=batch -sc_arg=-on=exit_on_error -test=N
```



The debugger toggle `exit_on_error` is useful for preventing scripts from hanging; it instructs the debugger to exit instead of looping if a test fails.

You can find additional information about ARCTest options using the `-help` option:

```
ARCTest -help
```

#### 18.5.1.1 Files used by ARCTest and the MetaWare Debugger

The information about the target simulator and list of valid tests is contained in files generated by ARChitect in the `build` directory. The files are readable text, so you can check if a particular test is valid for the current build or if limited resources in the build do not allow correct execution of some tests.

- `rascal.env` contains information for the MetaWare debugger, including the target simulator.

Note that `build/rascal.env` is not compatible with the xCAM simulator. Replace it with `build/vtoc/rascal.env` before running xCAM simulations.

- `test.xref` contains a list of tests and their locations. This file might also include conditions for running the tests, but that feature is not applicable for ARC EM.

### 18.5.1.2 What If Tests Fail?

If tests fail or hang, you can debug them. In most cases, failures are related to environment issues rather than configuration or RTL errors.

Try the following steps:

1. Check the `PATH` and `LD_LIBRARY_PATH` environment variables are set as indicated in the *ARChitect User's Guide*.
2. The ARChitect tool suite relies heavily on JavaScript and `make` facilities. Make sure that these binaries are in your `PATH`.
3. Verify that the RTL build is correct. In the case of a compiled build, verify the existence of executable files. For example, if the target simulator is VCS, then a file called `board.simv` should be in the `build` directory.
4. Verify test compilation. Look in the test directory for a `testname.out` file. If it does not exist, then try to determine if there is a compilation error or path problems and fix them as needed.
5. If all the above seems right, then you may need to trace the simulation for additional information. Check [“Waveform Generation”](#) and [“Pipeline Monitor”](#) on page 337 for information on generating a waveform dump or a trace file.

You might have to compare the assembly code to the trace report. Look in the test directory for all the files that contain the test name in their filename. The most useful file for tracing the simulation is the `.dmp` file, which contains actual instruction addresses and a symbol table.

How to trace and debug a specific test is beyond the scope of this document. The simple items mentioned above should provide a good initial checklist.

## 18.5.2 Waveform Generation

ARChitect has an embedded option to simplify generation of a `.vpd` waveform trace when the target simulator is VCS. Configure and compile a design and then type the following command in the `build` directory:

```
make waves
```

Then run `ARCtest`, and the simulation produces a `vcdplus.vpd` file in the `build` directory. You can open this file with a suitable waveform viewer, such as DVE.

To disable the waveform generation, recompile with the following command:

```
make
```

To generate a waveform file for other simulators, edit `build/verilog/board.v` and insert the specific waveform command for the target simulator and waveform format.



### 18.5.3 Pipeline Monitor

The ARC HS pipeline monitor traces committed instructions. This trace provides a simulation timestamp (*not* the same as the clock), instruction address and mnemonics, and actual results.

You can choose to display the trace on the screen or direct it to the `trace.txt` file in the `build` directory.

Invoke the pipeline monitor at the configuration stage with the ARChitect option `-execution_trace_level`. This option has four possible values: **off**, **stats**, **full**, **full-file**. It is better to build with **stats** (the default), so that the pipe monitor module is included in the build.

The equivalent GUI command is under the **Implementation** component in the ARChitect **Options** pane.

See also the PIPEMON section in the *ARChitect Tools User's Guide* in the *ARChitect Tools Bookshelf* (PDISP is not applicable to ARC EM).

### 18.5.4 MetaWare Debugger

The MetaWare debugger examines the ARC EM build configuration registers (BCRs). If the version fields are valid, the debugger enables display of the appropriate ARC EM internal registers to debug your processor design on hardware or simulator.

By default, the nSIM simulator is loaded for ARC EM processors. See the *MetaWare Debugger User's Guide* and *nSIM User's Guide* for more information.

Note that the debugger retrieves and displays data only when the processor is stopped.

#### 18.5.4.1 MetaWare Debugger Graphical User Interface

To run the debugger in GUI mode enter the following command:

```
mdb main.out
```

Where:

`main.out` Executable output file created by the linker

In the **Debug a target or targets** dialog, click **Debugger Options** and select one of the following under **Target Selection**:

- **nSIM**
- **Hardware**

The options you can select in the **Debugger options** dialog change depending on which type of target you select.

### 18.5.4.2 MetaWare Debugger Command-Line Interface

To debug code that supports the ARC EM, use the `-arcv2hs` command-line option, for example:

```
mdb -cl -arcv2hs main.out
```

Where:

|                       |                                                       |
|-----------------------|-------------------------------------------------------|
| <code>-arcv2hs</code> | Specifies debugging on an ARC EM processor using nSIM |
| <code>main.out</code> | Executable output file created by the linker          |

You can use various debug commands to debug on an ARC EM processor. Here is an abridged list of commands.

|                       |                                                     |
|-----------------------|-----------------------------------------------------|
| <code>run</code>      | Start or continue execution of the current process  |
| <code>register</code> | Display core registers.                             |
| <code>aux</code>      | Display ARC auxiliary registers                     |
| <code>ssi</code>      | Step through source-code statements, into functions |
| <code>ss0</code>      | Step through source-code statements, over functions |

To run a program with no debug output enter the following command:

```
mdb -cl -arcv2hs -run main.out
```

# A

## Converting from ARC 600

The ARC EM processor series is based on the revised ARCV2 instruction-set architecture (ISA) and has a new, highly efficient microarchitecture based on a three-stage pipeline. Users familiar with ARC 600 can expect to see differences in both hardware and software when migrating to the ARC EM.

### A.1 Hardware Considerations

- ARC 600 has internal circuitry to ensure that reset is asynchronously applied and synchronously removed. The reset signal in ARC EM is synchronously de-asserted within the synthesizable EM module called `reset_ctrl.v`.
- The architecture of the bus bridges has changed, and in most cases this has the benefit of reducing latencies to memory. It is advisable to re-simulate to measure and verify the performance of ARC EM cores in your system.
- Interrupt pins are active low on ARC 600, but active high on ARC EM.
- Other important differences:
  - Memory map
  - AHB and BVIC
  - Peripheral-port option
  - SmaRT not in a separate library
  - CCT in a separate library
  - Directory structure
  - Memory-Protection Unit (optional); see [Table A-1](#)

**Table A-1 Differences between ARC 600 and ARC EM Memory-Protection Units**

| Feature                                   | ARC 600 | ARC EM |
|-------------------------------------------|---------|--------|
| Default region permissions                | No      | Yes    |
| Separate kernel and user mode permissions | No      | Yes    |
| Protected access to MPU aux registers     | No      | Yes    |

**Table A-1 Differences between ARC 600 and ARC EM Memory-Protection Units**

| Feature                                                                     | ARC 600                | ARC EM             |
|-----------------------------------------------------------------------------|------------------------|--------------------|
| Action on protection violation                                              | Memory-error interrupt | EV_ProtV exception |
| Precise reporting of violations                                             | No                     | Yes                |
| MPUFA register                                                              | Yes                    | Uses EFA register  |
| MPU_BUILD Version field                                                     | 0x01                   | 0x02               |
| Minimum region size                                                         | 8 KB                   | 2 KB               |
| MPUIC register                                                              | Yes                    | Renamed MPU_ECR    |
| MPUIC or MPU_ECR write violation code                                       | 01                     | 10                 |
| MPUIC or MPU_ECR exception code                                             | 0x0027                 | 0x0006             |
| Programmable cacheability flag per region                                   | Yes                    | No                 |
| Correct handling of MPU protection violations inside all interrupt handlers | No                     | Yes                |

## A.2 Software Considerations

The ARCV2 instruction-set architecture supports a superset of the capabilities of the ARCompact ISA, but is *not* backwards-binary-compatible. A range of new instructions is provided in the ARCV2 instruction-set architecture, allowing greater code density and higher performance. Some implications for software are as follows:

- C/C++ or assembly code must be recompiled or reassembled with the `-arcv2em` command-line option.
- The optional 32-bit multiply instructions supported by the ARCV2 instruction-set architecture are similar to those available to the ARC 700. The ARC 600 `mul64` and `mulu64` instructions, and their associated auxiliary registers, are not supported by the ARCV2 instruction-set architecture. [Chapter 8, “Cycle-Timing Characteristics”](#) explains how software performance can be optimized for these instructions when coding in assembly language.
- The interrupt architecture of the ARC EM is different from that of the ARC 600, as it is based on the more flexible interrupt system of the ARC 700. Take particular care when porting applications that use interrupts.



### Note

It is important to note the change of the interrupt entry table from instruction-based (usually a jump to handler) with 8 bytes per entry to a vector table, which only contains the target address of the handler (only 4 bytes per entry) for the specific interrupt or exception.

- The exception-handling architecture of the ARCV2 ISA is based on that of the ARC 700, rather than the ARC 600. In addition, new exception types have been added to handle division by zero and misaligned data accesses. Refer to the *ARCV2 ISA Programmer's Reference* when programming exception handlers or re-assembling legacy code containing exception handlers.

- Some ARC 600 or ARC 700 assembly code instructions might no longer be available. Be aware of this if you have existing ARC 600 or ARC 700 assembly code. Review the differences listed in the *ARCV2 ISA Programmer's Reference* as well as the instruction-set details documented there.



# B

## Key IP Library Command-Line Options

This appendix contains an excerpt from the listing obtained using the `ARCHitect2 -d` to show options for the ARC EM IP library. For generic ARCHitect command-line options, use `ARCHitect2 -h`.

### Example of Use

For example, `-num_actionpoints` sets the number of action points:

```
-num_actionpoints 4
```

In another example, `-byte_order` sets the endian orientation.

```
-byte_order big
```

Most IP library options can also be specified in the ARCHitect GUI, by making selections in the **Options** pane.

### B.1 Key Options



#### Caution

This listing presents highlights of the most important options. View the options using `-d` for complete information.

#### B.1.1 HTML Documentation and Software

```
-build_html_docs - Creates custom HTML documentation in the 'docs' directory.
Units:
Default value:   true
Possible values: true, false
-build_software - Creates software under the Software directory.
Units:
Default value:   false
Possible values: true, false
```

#### B.1.2 Scripts and HDL

```
-build_scripts - Creates synthesis scripts and configuration files,
Units:
Default value:   true
```

Possible values: true, false

-build\_hdl – Creates the behavioral and synthesizable HDL source code.  
Units:  
Default value: true  
Possible values: true, false

-generate\_structural\_hdl – Generate the necessary structural HDL.  
Units:  
Default value: true  
Possible values: true, false

-compile\_hdl\_for\_simulation – Compile the HDL ready for simulation.  
Units:  
Default value: true  
Possible values: true, false

-build\_xcam – Builds the XCAM cycle accurate model from HDL.  
When true, build the XCAM cycle accurate model from HDL.  
This happens only when the VTOC component (in the xCAM library)  
has been added to the design.

### B.1.3 Test Code

-build\_test\_code – Creates test source code under the 'tests' directory.  
Units:  
Default value: true  
Possible values: true, false

-compile\_test\_code – Compiles and assembles the test code.  
Units:  
Default value: true  
Possible values: true, false

### B.1.4 IP-XACT

-generate\_ipxact – Generate ipxact.xml file describing CPUisle or archipelago frontier.

-ipxact\_relative\_path\_names – Use relative path names for Verilog files in the ipxact.  
Otherwise, absolute path names are used.

### B.1.5 Verilog and Simulator

-verilog\_2001 – Enable Verilog 2001 file-io syntax.  
Units:  
Default value: true  
Possible values: true, false

-simulator – Specify the name of the simulator you wish to use.  
Units: no units  
Default value: vcs  
Possible values: none, modelsim, vcs, nc

-vcs64 – VCS simulator build uses 64-bit VCS.  
Be sure you have 64-bit-capable VCS installed.  
The setting of this option affects the content of the generated  
makefile\_interface\_vcs\_verilog.  
Units:  
Default value: false  
Possible values: true, false



## B.1.6 RASCAL and Trace

- fast\_rascal – Replace debug-port RTL with a PLI/FLI interface to the software debugger.  
Units:  
Default value: false  
Possible values: true, false
- execution\_trace\_level – Trace committed instructions and gather stats for debugger.  
At the "stats" level only the statistics are gathered and no trace is printed.

## B.1.7 ARChitect Tools

- clock\_skew – Setup uncertainty to be used during synthesis and layout steps.  
Units: ns  
Default value 0.2  
Possible values : any value
- clock\_speed – Target clock speed of the system.  
Units : MHz  
Default 200  
Possible values : any reasonable value that fits the chosen technology option
- run\_arcformal – Formal Verification using ARCformal.  
Units:  
Default value: false  
Possible values: true, false
- run\_arcrams – Run ARCrans on the current build.  
Units:  
Default value: false  
Possible values: true, false
- run\_arsyn – Synthesize design using ARCSyn (and make physical implementation).  
Units:  
Default value: false  
Possible values: true, false
- run\_seif – Run Synopsys Embedit Integrator Flow to generate memory instances.  
Units:  
Default value: false  
Possible values: true, false
- scan\_chains – Number of scan chains to be used when stitching the scan registers.  
Units: no units  
Default value: 4  
Minimum value: 0  
Maximum value: 10
- gatesim – Install sim test code and scripts to run ARCgatesim.  
Units:  
Default value: true  
Possible values: true, false
- floorplan – Floorplan definition to be used to lay out the RAMs of the design  
Units:  
Default value: none  
Possible values: none, User, create, ARCV2EM4, ARCV2EM4\_min, ARCV2EM4\_full, ARCV2EM6, ARCV2EM6\_min, ARCV2EM6\_full

-hold\_margin – Margin for hold-time checks.  
Units: ns  
Default value: 0.05  
Minimum value: 0.0  
Maximum value: 0.2

### B.1.8 Processor

-arcv2em – Specifies an ARC EM processor.  
Units:  
Default value: true  
Possible values: true, false  
-arc\_num – The processor number from the ARCNUM field of the IDENTITY register.  
Units:  
Default value: 0  
Minimum value: 0  
Maximum value: 255  
-unique\_name – Add prefix to Verilog modules to make them unique

### B.1.9 Synthesis

-infer\_alu\_adder – Synth: ALU Adder  
Units:  
Default value: instantiate  
Possible values: instantiate, infer  
-infer\_mpy\_wtree – Synth: Multiplier  
Units:  
Default value: instantiate  
Possible values: instantiate, infer

### B.1.10 Endian Orientation and Counter Sizes

-addr\_size – Specifies the address bus width (in bits).  
Units: bits  
Default value: 16  
Possible values: 16, 20, 24, 32  
-byte\_order – Specifies the endianness of the core.  
Units:  
Default value: little  
Possible values: little, big  
-lpc\_size – Specifies the size of the loop counter (in bits).  
Units: bits  
Default value: 8  
Possible values: 8, 16, 32  
-pc\_size – Specifies the size of the program counter (in bits).  
Units: bits  
Default value: 16  
Possible values: 16, 24, 32

### B.1.11 Interrupts and Reset

-halt\_on\_reset – Specifies whether the core is halted initially on reset.  
Units:

Default value: false  
 Possible values: true, false

-number\_of\_interrupts – Specifies total number of interrupts  
 Units: integer  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 240

-external\_interrupts – number of external interrupt pins  
 Units: integer  
 Default value: 0  
 Minimum value: 0  
 Maximum value: 240 (no timers configured), 239 (one timer is configured), 238 (two timers configured)

-intvbase\_preset – This option specifies the value of the interrupt vector base configuration register, VECBASE\_AC\_BUILD. The value is the address divided by 1024 (the number of 1KB block). On reset, this value is loaded into the interrupt vector base address register, INT\_VECTOR\_BASE. The interrupt vector base address can be set to any 1Kbyte-aligned address.  
 Units: Kbytes  
 Default value: 0  
 Minimum value: 0  
 Maximum value:  $((1 < (PC\_SIZE - 10)) - 1)$

-firq\_option – This option has fast IRQ enabled  
 Units: boolean  
 Default value: false  
 Possible values: true, false

-number\_of\_levels – Number of interrupt levels  
 Units: integer  
 Default value: 2  
 Possible values: 0 to 15

### B.1.12 Register File

-rgf\_impl – Specifies whether register file uses flip-flops or a hard macro.  
 Units:  
 Default value: flip\_flops  
 Possible values: flip\_flops, macro

-rgf\_num\_regs – Specifies the size (in 32b register) of the processor register file.  
 Units: integer  
 Default value: 16  
 Possible values: 16, 32

-rgf\_wr\_ports – This option defines the whilewrite ports on the register file.  
 Units: integer  
 Default value: 1  
 Possible values: 1, 2

-rgf\_num\_banks – Number of register banks.  
 Units: Integer  
 Default value: 1  
 Possible values: 1, 2

-rgf\_banked\_regs – Number of banked registers  
 Units: integer  
 Default value: 4  
 Possible values: 4, 8, 16, 32

### B.1.13 Memory

```
-off_chip_ram – True, if you have the memory arbitrator.
  Units:
  Default value:  yes
  Possible values: yes
-has_dmp_peripheral – Has DMP peripheral.
  Units:
  Default value:  false
  Possible values: true, false
-mem_bus_option – Protocol: Memory Bus.
  Units:
  Default value:  BVCI
  Possible values: BVCI, AHB, AHB-Lite-single, AHB-Lite-dual
-per_bus_option – Protocol: Peripheral Bus.
  Units:
  Default value:  BVCI
  Possible values: BVCI, AHB-Lite
-mem_bus_reg_interface – Specifies whether memory bus interface is registered or not.
  Units:          boolean
  Default value:  false
  Possible values: true, false
```

### B.1.14 Closely Coupled Memory

```
-dccm_dmi – indicates that DCCM has a DMI port
  Units: boolean
  Default value: false
  Possible values:  true or false
-dccm_size – Specifies the size of the Data Closely Coupled Memory (DCCM) in bytes.
  Units:          bytes
  Default value:  512
  Possible values: 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072,
  262144, 524288, 1048576
-iccm0_size – Specifies the size of the Instruction Closely Coupled Memory in bytes.
  Units:          bytes
  Default value:  512
  Possible values: 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144,
  524288, 1048576
-iccm0_base – Sets the initial memory region assignment for ICCM0.
  Units:          integer
  Possible value: 0, 1, 2, 3, 4, 5, 6, 7
-iccm0_dmi – indicates that ICCM0 has a DMI port
  Units:          boolean
  Default value:  false
  Possible values:  true or false
-iccm1_size – Specifies the size of the ICCM1 in bytes.
  Units:          bytes
  Default value:  512
  Possible values: 512, 1024, 2048, 4096, 8192, 16384, 32768, 65536, 131072, 262144,
  524288, 1048576
-iccm1_base – Sets the initial memory region assignment for ICCM1
  Units:          integer
```

Possible values: 0, 1, 2, 3, 4, 5, 6, 7

-iccm1\_dmi - indicates that ICCM1 has a DMI port

Units: boolean

Default value: false

Possible values: true or false

-iccm1\_banking - indicates that the ICCM1 is split into two parts that are interlaced with respect to accesses

Units: boolean

Default value: false

Possible values: true or false

## B.1.15 Instruction Optimizations

-bitscan\_option - Adds instructions for efficient search of bits.

Units:

Default value: false

Possible values: true, false

-code\_density\_option - Adds instructions that condense commonly used patterns.

Units:

Default value: false

Possible values: true, false

-div\_rem\_option - Adds non-blocking multi-cycle integer divide/remainder functions.

Units:

Default value: none

Possible values: none, radix-2, enhanced radix-4

-mpy\_option - Allows selection between several multiplier configurations.

Units:

Default value: wlh5

Possible values: none, w, wlh1, wlh2, wlh3, wlh4, wlh5

|   | MPY_OPTION | 16/L32/H32 | Instructions Enabled                |
|---|------------|------------|-------------------------------------|
| 0 | none       | -/-/-      | None                                |
| 1 | w          | 1/-/-      | MPYW, MPYWU                         |
| 2 | wlh1       | 1/1/1      | MPYW, MPYWU, MPY, MPYU, MPYH, MPYHU |
| 3 | wlh2       | 2/2/2      | MPYW, MPYWU, MPY, MPYU, MPYH, MPYHU |
| 4 | wlh3       | 2/3/3      | MPYW, MPYWU, MPY, MPYU, MPYH, MPYHU |
| 5 | wlh4       | 2/4/5      | MPYW, MPYWU, MPY, MPYU, MPYH, MPYHU |
| 6 | wlh5       | 5/9/9      | MPYW, MPYWU, MPY, MPYU, MPYH, MPYHU |

| Cycle Count | Meaning                                            |
|-------------|----------------------------------------------------|
| 16          | Timing for MPYW/MPYWU or MPY/MPYU on 16-bit values |
| L32         | Timing for MPY/MPYU when flag-enable bit is 0      |



Units:  
 Default value: 64bits  
 Possible values: 32bits, 64bits

### B.1.18 Clock Frequency

-frequency\_1\_divisor – Sets the division value for the 1st additional clock frequency.  
 Units:  
 Default value: 2  
 Minimum value: 1  
 Maximum value: 4

### B.1.19 JTAG Clock, Latency Unit, and HDL Library

-jtag\_tclk – Selects the frequency of the JTAG clock Tck (in MHz).  
 The maximum value permitted is 1/2 of the CPU clock\_speed and cannot exceed 600 MHz.  
 Units: MHz  
 Default value: 10  
 Minimum value: 1  
 Maximum value: 600  
 -Latency\_fifo\_depth – Specifies the depth of the Latency Unit FIFO.  
 Units:  
 Default value: 64  
 Possible values: 64, 128, 256  
 -latency\_unit – Adds Latency unit to profiler.  
 Units:  
 Default value: true  
 Possible values: true, false  
 -library\_name – Specifies the name for your HDL library.  
 Units: no units  
 Default value: user  
 Minimum value: 0  
 Maximum value: 10

### B.1.20 Actionpoints

-num\_actionpoints – Specifies number of actionpoints.  
 Units: integer  
 Default value: 2  
 Possible values: 2, 4, 8  
 -aps\_feature – Selects Actionpoint feature set.  
 Units:  
 Default value: min  
 Possible values: min, full

### B.1.21 SmaRT Stack Size

-smart\_stack\_entries – SmaRT Stack Size.  
 Units: integer  
 Default value: 8  
 Possible values: 8, 16, 32, 64, 128, 256, 512, 1024, 2048, 4096  
 -smart\_implementation – Selects between flip-flop or memory-based implementation. The memory option is generally more efficient for 64 entries or more.

Units: string  
 Default value: flip-flop  
 Possible values: flip-flop, memory

### B.1.22 Interrupt-Timer Levels

-timer\_0\_int\_level – Sets the interrupt level (and implicitly the priority) of timer 0.  
 Units: integer  
 Default value: 1  
 Possible values: 0 to (Number of Interrupt Levels-1)

-timer\_1\_int\_level – Sets the interrupt level (and implicitly the priority) of timer 1.  
 Units: integer  
 Default value: 0  
 Possible values: 0 to (Number of Interrupt Levels-1)

-timer\_0\_irq - Timer\_0 IRQ  
 Units: integer  
 Default value: 16  
 Possible values: 16

-timer\_1\_irq - Timer\_1\_IRQ  
 Units: integer  
 Default value: 17  
 Possible values: 17

### B.1.23 UART

-uart – Add the UART component to the design.  
 Units:  
 Default value: true  
 Possible values: true

-num\_of\_uart – Selects the whileUART instantiations.  
 Units:  
 Default value: 1  
 Minimum value: 1  
 Maximum value: 3

-uart\_baud\_rate – Specifies the default Baud Rate For the UART.  
 Units:  
 Default value: 115200  
 Minimum value: 1200  
 Maximum value: 115200

### B.1.24 Instruction Fetch Queue

-has\_ifqueue – Adds Instruction Fetch Queue to the build.  
 0= no IFQ  
 1= IFQ present

-ifqueue\_size – Specifies the number of entries in the instruction fetch queue.  
 Units: 32-bit words  
 Default value: 1  
 Possible values: 1, 2, 4, 8, 16

-ifqueue\_burst\_size - Burst size for bus data transfers(32-bit words)  
 Units: 32-bit words  
 Default value: 1  
 Possible values: 1, 2, 4, 8



## B.1.25 Instruction Cache and Instruction Closely Coupled Memory

```
-ic_size - Specifies the total size of the instruction cache in bytes.
  Units:          bytes
  Default value:   2048
  Possible values: 2048, 4096, 8192, 16384, 32768
-ic_bsize - Defines the cache line length in bytes.
  Units:          bytes
  Default value:   16
  Possible values: 16, 32, 64, 128
-ic_disable_on_reset - Disable instruction cache on Reset.
  Units:
  Default value:   false
  Possible values: true, false
-ic_feature_level - Specifies the feature level of the cache.
  Units:          integer
  Default value:   0
  Possible values: 0, 1, 2
-ic_pwr_opt_level - Dynamic Power Optimization.
  Units:          integer
  Default value:   0
  Possible values: 0, 1
-ic_size - Specifies the total size of the instruction cache in bytes.
  Units:          bytes
  Default value:   2048
  Possible values: 2048, 4096, 8192, 16384, 32768
-ic_ways - Specifies the whilecache ways
  Units:          bytes
  Default value:   1
  Possible values: 1, 2, 4
```

## B.1.26 Data Cache

```
-dc_size - Specifies the total size of the Data Cache in bytes.
  Units:          bytes
  Default value:   2048
  Possible values: 2048, 4096, 8192, 16384, 32768
-dc_bsize - Specifies the cache line length in bytes.
  Units:          integer
  Default value:   16
  Possible values: 16, 32, 64, 128
-dc_feature_level - Indicates locking and debug feature level.
  Units:          bytes
  Default value:   0
  Possible values: 0, 1, 2
-dc_uncached_region - Enable an uncached region defined by auxiliary register.
  Units:
  Default value:   false
  Possible values: true, false
-dc_ways - Specifies the whilecache ways.
  Units:          bytes
  Default value:   1
  Possible values: 1, 2, 4
```

## B.1.27 Memory Protection Unit

-mpu\_num\_regions -Number of regions  
 Units: integer  
 Default value: 8  
 Possible values: 1, 2, 4, 8, 16

## B.1.28 com.arc.hardware.EM.Floating\_point\_unit

Option: -fpu\_dp\_assist  
 Description: This enables double-precision acceleration support floating point instructions.  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -fpu\_div\_option  
 Description: This enables divide & square-root instructions  
 Units:  
 Default value: false  
 Possible values: true, false

Option: -fpu\_div\_cycles  
 Description: This enables divide & square-root instruction cycles  
 Units:  
 Default value: 14  
 Possible values: 1,17

Option: -fpu\_fma\_option  
 Description: This enables the multiply & accumulate instructions.  
 Units:  
 Default value: false  
 Possible values: true, false

## B.1.29 com.arc.hardware.EM.Real\_time\_trace\_producer

Option: -rtt\_feature\_level  
 Description: 'small' means that program trace only is available. 'medium' adds data trace. 'full' adds core and aux register trace.  
 Units:  
 Default value: small  
 Possible values: small, medium, full

Option: -rtt\_programming\_if  
 Description: One processor only can program the RTT system component. Select this option to choose this processor.  
 Units:  
 Default value: true  
 Possible values: true, false

## B.1.30 com.arc.hardware.ARCv2MSS.BusFabric

Option: -alb\_mss\_fab\_lat

Description: This specifies the maximum latency in the master latency units.  
Units: bytes  
Default value: 0  
Possible values: 0, 8, 16, 32, 64, 128, 256

Option: -alb\_mss\_fab\_def\_lat  
Description: This specifies the latency after reset for the master latency units.  
Units: bytes  
Default value: 0

Option: -alb\_mss\_fab\_def\_ratio  
Description: This specifies the clock ratio at reset.  
Units: bytes  
Default value: 1

Option: -alb\_mss\_ccm\_base  
Description: This specifies the base address at which the ICCMs and DCCMs will be placed in the memory map. The address should be divided by 4KB i.e. do not specify the lower 12 bits of the address.  
Units: bytes  
Default value: 262144

**B.1.31 com.arc.hardware.ARCv2MSS.ClkCtrl**

Option: -alb\_mss\_clkctrl\_base\_addr  
 Description: This specifies the clock controller base address in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address.  
 Units: bytes  
 Default value: 786432

Option: -alb\_mss\_clkctrl\_def\_ratio  
 Description: This specifies the clock ratio at reset.  
 Units: bytes  
 Default value: 1

**B.1.32 com.arc.hardware.ARCv2MSS.Profiler**

Option: -alb\_mss\_perfctrl\_base\_addr  
 Description: This specifies the profiler base address in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address.  
 Units: bytes  
 Default value: 786433

Option: -alb\_mss\_perfctrl\_def\_ratio  
 Description: This specifies the clock ratio at reset.  
 Units: bytes  
 Default value: 1

**B.1.33 com.arc.hardware.ARCv2MSS.SRAMCtrl**

Option: -alb\_mss\_mem\_base\_addr  
 Description: This specifies the memory controller base address in the memory map, divided by 4KB i.e. do not specify the lower 12 bits of the address.  
 Units: bytes  
 Default value: 0

Option: -alb\_mss\_mem\_lat  
 Description: This specifies the maximum latency in the memory latency unit.  
 Units: bytes  
 Default value: 0  
 Possible values: 0, 8, 16, 32, 64, 128, 256

Option: -alb\_mss\_mem\_def\_lat  
 Description: This specifies the latency after reset for the memory latency unit.  
 Units: bytes  
 Default value: 0

Option: -alb\_mss\_mem\_size  
 Description: This specifies size of the SRAM.

Units:  
Default value: 64KB  
Possible values: 4KB, 8KB, 16KB, 32KB, 64KB, 128KB, 256KB, 512KB, 1MB, 2MB, 4MB, 8MB, 16MB, 32MB, 64MB, 128MB, 256MB, 512MB, 1GB, 2GB, 4GB

Option: -alb\_mss\_mem\_def\_ratio  
Description: This specifies the clock ratio at reset.  
Units: bytes  
Default value: 1

Option: -alb\_mss\_mem\_is\_default\_slave  
Description: If true then all transactions without destination will be routed here.  
Units:  
Default value: false  
Possible values: true, false

### B.1.34 com.arc.hardware.UAUX\_Interface.1\_0

Option: -create com.arc.hardware.UAUX\_Interface.1\_0  
"System.CPUisle.ARCv2EM.UAUX Interface"  
Description: Configure UAUX interface component using command line.



## C

# Synthesis and Physical Implementation

The ARC® Synopsys Reference Design Flow is provided as an extension IP library for customers who require it. It includes a complete set of documentation in Adobe® PDF format. The starting place for viewing the documentation is the *ARC® Synopsys Design Flow IP Library Bookshelf* (SynopsysRDF\_Bookshelf.pdf).

## C.1 Register-File Implementation

The register file is configurable as RAM or flops. When configured as memory, it is handled in the same way as the rest of the memory integration.

## C.2 Integration: Memory Insertion

DesignWare ARC processor systems built by default contain reference RAM models that are used as placeholders in the RTL until the design has been targeted at a specific technology implementation. These models are suitable for simulation only. Before mapping the RTL to gates, you must replace the simulation models by instantiations of real memories built with a memory/register-file compiler. Such memories also have a behavioral view (.v) in addition to a timing model (.lib/.db) and a physical model (.plef/FRAM). Logic synthesis requires a timing model at the minimum. For topological synthesis, a FRAM view is desirable and is required for an accurate result. For the physical synthesis step in ICC, a FRAM view is essential.

You can replace the reference RAM models using any of the following methods:

- Synopsys SEIF RDF component (easiest way)
- ARCrans tool (documented in the *ARChitect Tools User's Guide*)
- Manually

The Synopsys SEIF component of the RDF supports the possibility of generating the required instances from Synopsys memory compilers and selecting and configuring the instances to a set of parameterized RTL wrappers. Compilers from other vendors can also be used, but in that case you must generate your own harness to match the supplied fake-RAM model's behavior. Run the Customer Confidence Tests (CCTs) supplied with the ARC EM IP library to verify the correct implementation of your own memory wrappers.

A required viewset of the preferred instances must be organized according to the directory structure indicated in the *ARChitect Tools User's Guide* included in the *ARChitect Tools Bookshelf*.

The ARCrams tool is provided for replacing the fake ram placeholder verilog with the configured ram wrappers; see the *ARChitect Tools User's Guide*.

## C.3 System Integration

The ARChitect configuration is the tool for configuring the ARC EM core. You can use it to add components and configure a complete CPU system. ARChitect produces a core, whose RTL is in the `build/verilog/RTL` directory, and a testbench. You can configure and run the complete reference design flow in the ARChitect GUI or using makefiles.

- For an introduction to ARChitect, see [“ARChitect Configuration Tool”](#) on page 51.
- For more on integration, see [Chapter 18, “Integration and Verification”](#).

## C.4 DFT

Compiling an ARC core using a test-ready strategy yields a design containing scan registers with a preliminary chain order. This arbitrary chaining can be reconfigured during layout implementation to better suit the actual physical placement. The scan chains are inserted as part of the synthesis flow using DFT-Compiler within Design Compiler.

The memories as generated by the Synopsys-supplied SEIF flow anticipate the use of SMS BIST at the top level of the end-user design. The physical instances that are generated are therefore configured with the integrated test option so that the test and functional data muxing is taken care of inside the hard core memory. This removes the need for user to add test muxes in the RTL.

## C.5 Interfacing ARC EM Memory

### C.5.1 Memory Functions

The following are the different memory functions that you can configure in an ARC EM processor:

- `iccm_ram` (synchronous single-port for read and write)
- `iccm1_ram` (synchronous single-port for read and write)
- `dccm_ram` (synchronous single-port for read and write)
- `ic_data_ram` (synchronous single-port for read and write)
- `ic_tag_ram` (synchronous single-port for read and write)
- `dc_data_ram` (synchronous single-port for read and write)
- `dc_tag_ram` (synchronous single-port for read and write)
- `smart_ram` (synchronous single-port for read and write)
- `regfile_ram` (2R/1W three-port register file to be built from two instances of two-port compilers)



## C.5.2 Memory Implementation Specification Using ram\_sizes.txt

When you build an ARC EM processor, ARChitect generates a `ram_sizes.txt` file in the `build` directory. This file contains a specification of the memory sizes (number of words times number of bits) and number of instances needed to implement each memory function. This file also specifies the memory types that are supported, for example, synchronous single read/write port memory, two-port (one read, one write), and so on.

All the configured memory functions are instantiated in the Verilog RTL hierarchy in the `srams` module defined at `build/verilog/RTL/srams.v`.

The following is an example of a `ram_sizes.txt` file for a configuration that includes all the memory functions listed in [“Memory Functions”](#) on page 360:

### # ICCM RAM

```
iccm_ram : Synchronous 1RW, 1 instance (s) x 16384 words x 32 bits; WriteMask 8 bit(s)
```

### # ICCM RAM

```
iccm1_ram : Synchronous 1RW, 16 instance (s) x 16384 words x 32 bits; WriteMask 8 bit(s)
```

### # DCCM RAM

```
dccm_ram : Synchronous 1RW, 1 instance (s) x 16384 words x 32 bits; WriteMask 8 bit(s)
```

### # I-Cache tag RAM

```
ic_tag_ram 2-way : Synchronous 1RW, 2 instance (s) x 256 words x 21 bits; WriteMask  
0 bit(s)
```

### # I-Cache data RAM

```
ic_data_ram 2-way : Synchronous 1RW, 2 instance (s) x 2048 words x 32 bits; WriteMask  
0 bit(s)
```

### # D-Cache tag RAM

```
dc_tag_ram 2-way : Synchronous 1RW, 2 instance (s) x 256 words x 22 bits; WriteMask  
22 bit(s)
```

### # D-Cache data RAM

```
dc_data_ram 2-way : Synchronous 1RW, 2 instance (s) x 2048 words x 32 bits; WriteMask  
8 bit(s)
```

### # SMART RAM

```
smart_ram : Synchronous 1RW, 1 instance (s) x 4096 words x 33 bits; WriteMask 0 bit(s)
```

### # Registerfile RAM (3-port)

```
regfile_ram : Synchronous 1R1W, 2 instance (s) x 32 words x 32 bits; WriteMask 0 bit(s)
```

**Note**

The register file mapping to macro is only supported for non-write-through memories.

**Notation of ram\_sizes.txt**

The notation in `ram_sizes.txt` also indicates whether you must implement a bit-write or a byte-write on the memory. For example, `WriteMask 8 bit(s)` specified on a 32-bit wide memory implies byte-write enabling.

When a memory instance requires more words than the sizes that the corresponding memory compiler can actually generate (for example, most Synopsys memory compilers are limited to a 16 K address size per instance), the memory has to be split into smaller chunks.

For ARC EM processor builds, the maximum number of addresses is never be specified as more than 16 K.

For example, when a 1 Mbyte ICCM is configured, the `ram_sizes.txt` file specifies 16 instances instead of one. The notation in the `rams_sizes.txt` file for ICCM is as follows:

```
iccm_ram : Synchronous 1RW, 16 instance (s) x 16384 words x 32 bits; WriteMask 8 bit(s)
```

**C.5.3 Memory-Port Specifications**

The memory behavioral models in the RTL determine the interface that you must use when replacing the behavioral reference with an actual macro instantiation. These interfaces are described in [Table C-1](#) to [Table C-10](#):

**Table C-1 iccm\_ram**

| Port | Direction from/to Core | Width              | Description                      |
|------|------------------------|--------------------|----------------------------------|
| clk  | Input                  | 1                  | Clock input                      |
| din  | Input                  | [`ICCM_DRAM_RANGE] | Write data input                 |
| addr | Input                  | [`ICCM_ADR_RANGE]  | Address for read or write        |
| cs   | Input                  | 1                  | RAM chip select, active high     |
| we   | Input                  | 1                  | Memory write enable, active high |
| wem  | Input                  | [`ICCM_BE_RANGE]   | Byte write enables, active high  |
| dout | Output                 | [`ICCM_DRAM_RANGE] | Read data output                 |

**Table C-2 iccm1\_ram (iccm1\_banking = 0)**

| Port | Direction from/to Core | Width               | Description      |
|------|------------------------|---------------------|------------------|
| clk  | Input                  | 1                   | Clock input      |
| din  | Input                  | [`ICCM1_DRAM_RANGE] | Write data input |

| Port | Direction from/to Core | Width               | Description                      |
|------|------------------------|---------------------|----------------------------------|
| addr | Input                  | [`ICCM1_ADR_RANGE]  | Address for read or write        |
| cs   | Input                  | 1                   | RAM chip select, active high     |
| we   | Input                  | 1                   | Memory write enable, active high |
| wem  | Input                  | [`ICCM1_BE_RANGE]   | Byte write enables, active high  |
| dout | Output                 | [`ICCM1_DRAM_RANGE] | Read data output                 |

**Table C-3** iccm1\_ram (iccm1\_banking = 1)

| Port | Direction from/to Core | Width                | Description                      |
|------|------------------------|----------------------|----------------------------------|
| clk  | Input                  | 1                    | Clock input                      |
| din  | Input                  | [`ICCM1_DRAM_RANGE]  | Write data input                 |
| addr | Input                  | [`ICCM1_B_ADR_RANGE] | Address for read or write        |
| cs   | Input                  | 1                    | RAM chip select, active high     |
| we   | Input                  | 1                    | Memory write enable, active high |
| wem  | Input                  | [`ICCM1_BE_RANGE]    | Byte write enables, active high  |
| dout | Output                 | [`ICCM1_DRAM_RANGE]  | Read data output                 |

**Table C-4** dccm\_ram

| Port | Direction from/to Core | Width              | Description                      |
|------|------------------------|--------------------|----------------------------------|
| clk  | Input                  | 1                  | Clock input                      |
| din  | Input                  | [`DCCM_DRAM_RANGE] | Write data input                 |
| addr | Input                  | [`DCCM_ADR_RANGE]  | Address for read or write        |
| cs   | Input                  | 1                  | RAM chip select, active high     |
| we   | Input                  | 1                  | Memory write enable, active high |
| wem  | Input                  | [`DCCM_BE_RANGE]   | Byte write enables, active high  |
| dout | Output                 | [`DCCM_DRAM_RANGE] | Read data output                 |

**Table C-5** dc\_data\_ram

| Port | Direction from/to Core | Width            | Description                      |
|------|------------------------|------------------|----------------------------------|
| clk  | Input                  | 1                | Clock input                      |
| din  | Input                  | [`DC_DRAM_RANGE] | Write data input                 |
| addr | Input                  | [`DC_ADR_RANGE]  | Address for read or write        |
| cs   | Input                  | 1                | RAM chip select, active high     |
| we   | Input                  | 1                | Memory write enable, active high |
| wem  | Input                  | [`DC_BE_RANGE]   | Byte write enables, active high  |
| dout | Output                 | [`DC_DRAM_RANGE] | Read data output                 |

Table C-6 dc\_tag\_ram

| Port | Direction from/to Core | Width            | Description                      |
|------|------------------------|------------------|----------------------------------|
| clk  | Input                  | 1                | Clock input                      |
| din  | Input                  | [`DC_TRAM_RANGE] | Write data input                 |
| addr | Input                  | [`DC_IDX_RANGE]  | Address for read or write        |
| cs   | Input                  | 1                | RAM chip select, active high     |
| we   | Input                  | 1                | Memory write enable, active high |
| wem  | Input                  | [`DC_TRAM_RANGE] | Byte write enables, active high  |
| dout | Output                 | [`DC_TRAM_RANGE] | Read data output                 |

Table C-7 ic\_data\_ram

| Port | Direction from/to Core | Width            | Description                      |
|------|------------------------|------------------|----------------------------------|
| clk  | Input                  | 1                | Clock input                      |
| din  | Input                  | [`IC_DRAM_RANGE] | Write data input                 |
| addr | Input                  | [`IC_ADR_RANGE]  | Address for read or write        |
| cs   | Input                  | 1                | RAM chip select, active high     |
| we   | Input                  | 1                | Memory write enable, active high |
| dout | Output                 | [`IC_DRAM_RANGE] | Read data output                 |

Table C-8 ic\_tag\_ram

| Port | Direction from/to Core | Width            | Description                      |
|------|------------------------|------------------|----------------------------------|
| clk  | Input                  | 1                | Clock input                      |
| din  | Input                  | [`IC_TRAM_RANGE] | Write data input                 |
| addr | Input                  | [`IC_IDX_RANGE]  | Address for read or write        |
| cs   | Input                  | 1                | RAM chip select, active high     |
| we   | Input                  | 1                | Memory write enable, active high |
| dout | Output                 | [`IC_TRAM_RANGE] | Read data output                 |

**Table C-9** smart\_ram

| Port | Direction from/to Core | Width            | Description                      |
|------|------------------------|------------------|----------------------------------|
| clk  | Input                  | 1                | Clock input                      |
| din  | Input                  | [`SM_DATA_MSB:0] | Write data input                 |
| addr | Input                  | [`SM_ADR_MSB:0]  | Address for read or write        |
| cs   | Input                  | 1                | RAM chip select, active high     |
| we   | Input                  | 1                | Memory write enable, active high |
| dout | Output                 | [`SM_DATA_MSB:0] | Read data output                 |

**Table C-10** regfile\_ram

Note: Two instances are instantiated in `srams.v` to realize a three-port interface. `reg.file_ram` is used only when the register file is implemented as a RAM macro.

| Port    | Direction from/to Core | Width             | Description                             |
|---------|------------------------|-------------------|-----------------------------------------|
| clk     | Input                  | 1                 | Clock input                             |
| wr_data | Input                  | [`DATA_RANGE:0]   | Register-file write data input          |
| wr_addr | Input                  | [`MRF_ADDR_RANGE] | Register-file write address             |
| wr_en   | Input                  | 1                 | Register-file write enable, active high |
| rd_en   | Input                  | 1                 | Register-file read enable, active high  |
| rd_data | Output                 | [`DATA_RANGE:0]   | Register-file read data output          |
| rd_addr | Input                  | [`MRF_ADDR_RANGE] | Register-file read address              |

## C.5.4 Memory-Wrapper Reference Implementation

If you are using Synopsys EmbedIt (SEIF) memory compilers, configure the Synopsys RDF SEIF component to the system being built in ARChitect. Synopsys EmbedIt memory compilers use the `ram_sizes.txt` file to generate the memories. The memory macros are automatically re-configured into the RTL using Verilog wrappers, and all the required aspects of the RDF flow are primed to use the CAE views of these macros in the rest of the implementation flow.

If you do not have access to Synopsys EmbedIt memory compilers, an additional tar archive is provided with ARC EM IP library to demonstrate how such Verilog wrappers can be made around the basic memory macros. This tar archive contains an example SEIF output. You can use this tar archive to understand the ARCrans directory structure and how to fit vendor rams into Verilog wrappers. You can also implement your own wrappers using the sample wrapper.

The following examples explain the various files that are generated in the SEIF and ARCrans flow.

### Example C-1 Example Build Configuration Specification of Various Memory Functions

```
##### SmaRT --- com.arc.hardware.SmaRT.1_0 #####
# Create SmaRT
-create com.arc.hardware.SmaRT.1_0 System.CPUisle.ARCv2EM.SmaRT
# smart_stack_entries --- Stack Size
-smart_stack_entries 4096
# smart_implementation --- Implementation: memory or flip-flop
-smart_implementation memory
##### Data Cache --- com.arc.hardware.Data_Cache.1_0 #####
# Create Data Cache
-create com.arc.hardware.Data_Cache.1_0 "System.CPUisle.ARCv2EM.Data Cache"
# dc_size --- This defines the total size of the Data Cache in bytes.
-dc_size 16384
# dc_ways --- This defines the number of cache ways.
-dc_ways 2
# dc_bsize --- This defines the cache line length in bytes.
-dc_bsize 32
# dc_feature_level --- Feature Level, indicates locking and debug feature level 00 =
Basic cache, with no locking or debug features 01 = Lock and flush features supported
10 = Lock, flush and advanced debug features supported 11 = Reserved
-dc_feature_level 2
# dc_uncached_region --- Enable an uncached region defined by aux reg
-dc_uncached_region false
##### Instruction Cache --- com.arc.hardware.Instruction_Cache.1_0 #####
# Create Instruction Cache
-create com.arc.hardware.Instruction_Cache.1_0 "System.CPUisle.ARCv2EM.Instruction
Cache"
# ic_size --- This defines the total size of the instruction cache in bytes.
-ic_size 16384
# ic_ways --- This defines the number of cache ways
-ic_ways 2
# ic_bsize --- This defines the cache line length in bytes.
-ic_bsize 32
# ic_disable_on_reset --- Disable on Reset
-ic_disable_on_reset false
# ic_feature_level --- This defines the feature level of the cache.
-ic_feature_level 2
```

```

# ic_pwr_opt_level --- Dynamic Power Optimization
-ic_pwr_opt_level 1
##### DCCM --- com.arc.hardware.DCCM.1_0 #####
# Create DCCM
-create com.arc.hardware.DCCM.1_0 System.CPUisle.ARCv2EM.DCCM
# dccm_size --- This defines the size of the Data Closely Coupled Memory (DCCM) in bytes
-dccm_size 16384
# dccm_dmi --- DCCM has DMI port
-dccm_dmi false
##### ICCM0 --- com.arc.hardware.ICCM0.1_0 #####
# Create ICCM0
-create com.arc.hardware.ICCM0.1_0 System.CPUisle.ARCv2EM.ICCM0
# iccm0_size --- This defines the size of ICCM0 in bytes.This ICCM has 0 wait states.
-iccm0_size 16384
# iccm0_base --- Sets the initial memory region assignment for ICCM0
-iccm0_base 0
# iccm0_dmi --- ICCM0 has DMI port
-iccm0_dmi false
##### ICCM1 --- com.arc.hardware.ICCM1.1_0 #####
# Create ICCM1
-create com.arc.hardware.ICCM1.1_0 System.CPUisle.ARCv2EM.ICCM1
# iccm1_size --- This defines the size of ICCM1 in bytes.This ICCM has 1 wait states.
-iccm1_size 1048576
# iccm1_base --- Sets the initial memory region assignment for ICCM1
-iccm1_base 0
# iccm1_banking --- With dual memory banks, the ICCM has better cycle performance.
-iccm1_banking true
# iccm1_dmi --- ICCM1 has DMI port
-iccm1_dmi false
# rgf_impl --- This defines whether the register file is implemented using flip-flops,
or with a hard macro.
-rgf_impl macro
# rgf_num_regs --- This defines the size (in 32b register) of the processor register
file.
-rgf_num_regs 32
# rgf_wr_ports --- This defines the number of write ports on the register file.
-rgf_wr_ports 1
# rgf_num_banks --- 2 register banks are useful if Fast IRQ, but may be selected even if
not.
-rgf_num_banks 1
# rgf_banked_regs --- Number of banked registers
-rgf_banked_regs 32

```

## Example C-2 Sample ram\_sizes.txt file

```

# I-Cache tag RAM
ic_tag_ram 2-way : Synchronous 1RW, 2 instance (s) x 256 words x 13 bits; WriteMask
0 bit(s)
# I-Cache data RAM
ic_data_ram 2-way : Synchronous 1RW, 2 instance (s) x 2048 words x 32 bits; WriteMask
0 bit(s)

# D-Cache tag RAM

```

```

dc_tag_ram 2-way : Synchronous 1RW, 2 instance (s) x 256 words x 22 bits; WriteMask
22 bit(s)
# D-Cache data RAM
dc_data_ram 2-way : Synchronous 1RW, 2 instance (s) x 2048 words x 32 bits; WriteMask
8 bit(s)

# ICCM RAM
iccm_ram : Synchronous 1RW, 1 instance (s) x 4096 words x 32 bits; WriteMask 8 bit(s)
# ICCM RAM
iccm1_ram : Synchronous 1RW, 16 instance (s) x 16384 words x 32 bits; WriteMask 8 bit(s)
# DCCM RAM
dccm_ram : Synchronous 1RW, 1 instance (s) x 4096 words x 32 bits; WriteMask 8 bit(s)
# Registerfile RAM (3-port)
regfile_ram : Synchronous 1R1W, 2 instance (s) x 32 words x 32 bits; WriteMask 0 bit(s)

# SMART RAM
smart_ram : Synchronous 1RW, 1 instance (s) x 4096 words x 49 bits; WriteMask 0 bit(s)

```

### Example C-3 Sample map\_file.txt

RDF SEIF generates a mapping file, `arcrams_lib/map_file.txt`, that you must use as input to ARCCrams. The following are contents of a sample mapping file:

```

256x13_0 mem_ts65njkb1p11sassl512sa04_256x13_cm4_bk4_cdfalse_bwe0_byes_rno.lib wrapper
ic_tag_ram
2048x32_0 mem_ts65njkb1p11sassl512sa04_2048x32_cm8_bk2_cdfalse_bwe0_byes_rno.lib
wrapper ic_data_ram
256x22_22 mem_ts65njkb1p11sassl512sa04_256x22_cm4_bk4_cdfalse_bwe1_byes_rno.lib wrapper
dc_tag_ram
2048x32_8 mem_ts65njkb1p11sassl512sa04_2048x32_cm8_bk2_cdfalse_bwe1_byes_rno.lib
wrapper dc_data_ram
4096x32_8 mem_ts65njkb1p11sassl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_byes_rno.lib
wrapper iccm_ram
16384x32_8 mem_ts65njkb1p11sassl512sa04_16384x32_cm16_bk8_cdfalse_bwe1_byes_rno.lib
wrapper iccm1_ram
4096x32_8 mem_ts65njkb1p11sassl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_byes_rno.lib
wrapper dccm_ram
32x32_0 mem_ts65njkb2p11asdr132ksa07p1_32x32_cm1_bk4_cdfalse_bwe0_byes_rno.lib wrapper
regfile_ram
4096x49_0 mem_ts65njkb1p11sassl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_byes_rno.lib
wrapper smart_ram

```

### Example C-4 Sample Liberty Timing Model Folder

The following are Liberty timing models, which get converted to .db, that you must use as input to ARCCrams:

```

arcrams_lib/LIB/mem_ts65njkb1p11sassl512sa04_256x13_cm4_bk4_cdfalse_bwe0_byes_rno.lib
arcrams_lib/LIB/mem_ts65njkb1p11sassl512sa04_256x13_cm4_bk4_cdfalse_bwe0_byes_rno_typ.1
ib
arcrams_lib/LIB/mem_ts65njkb1p11sassl512sa04_256x13_cm4_bk4_cdfalse_bwe0_byes_rno_min.1
ib
arcrams_lib/LIB/mem_ts65njkb1p11sassl512sa04_2048x32_cm8_bk2_cdfalse_bwe0_byes_rno.lib

```



```

arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_2048x32_cm8_bk2_cdfalse_bwe0_byes_rno_typ.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_2048x32_cm8_bk2_cdfalse_bwe0_byes_rno_min.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_256x22_cm4_bk4_cdfalse_bwe1_byes_rno.lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_256x22_cm4_bk4_cdfalse_bwe1_byes_rno_typ.l
ib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_256x22_cm4_bk4_cdfalse_bwe1_byes_rno_min.l
ib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_2048x32_cm8_bk2_cdfalse_bwe1_byes_rno.lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_2048x32_cm8_bk2_cdfalse_bwe1_byes_rno_typ.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_2048x32_cm8_bk2_cdfalse_bwe1_byes_rno_min.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_byes_rno.lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_byes_rno_typ.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_byes_rno_min.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_16384x32_cm16_bk8_cdfalse_bwe1_byes_rno.li
b
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_16384x32_cm16_bk8_cdfalse_bwe1_byes_rno_ty
p.lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_16384x32_cm16_bk8_cdfalse_bwe1_byes_rno_mi
n.lib
arcrams_lib/LIB/mem_ts65njb2p11asdr132ksa07p1_32x32_cm1_bk4_cdfalse_bwe0_byes_rno.lib
arcrams_lib/LIB/mem_ts65njb2p11asdr132ksa07p1_32x32_cm1_bk4_cdfalse_bwe0_byes_rno_typ.
lib
arcrams_lib/LIB/mem_ts65njb2p11asdr132ksa07p1_32x32_cm1_bk4_cdfalse_bwe0_byes_rno_min.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_byes_rno.lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_byes_rno_typ.
lib
arcrams_lib/LIB/mem_ts65njb1p11sasl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_byes_rno_min.
lib

```

### Example C-5 Sample Physical Abstract view files

The following are physical abstract views, which gets converted to Milkyway FRAM view, that you must use as input to ARCCrams:

```

arcrams_lib/LEFs/mem_ts65njb1p11sasl512sa04_256x13_cm4_bk4_cdfalse_bwe0_byes_rno.plef
arcrams_lib/LEFs/mem_ts65njb1p11sasl512sa04_2048x32_cm8_bk2_cdfalse_bwe0_byes_rno.ple
f
arcrams_lib/LEFs/mem_ts65njb1p11sasl512sa04_256x22_cm4_bk4_cdfalse_bwe1_byes_rno.plef
arcrams_lib/LEFs/mem_ts65njb1p11sasl512sa04_2048x32_cm8_bk2_cdfalse_bwe1_byes_rno.ple
f
arcrams_lib/LEFs/mem_ts65njb1p11sasl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_byes_rno.ple
f
arcrams_lib/LEFs/mem_ts65njb1p11sasl512sa04_16384x32_cm16_bk8_cdfalse_bwe1_byes_rno.p
lef
arcrams_lib/LEFs/mem_ts65njb2p11asdr132ksa07p1_32x32_cm1_bk4_cdfalse_bwe0_byes_rno.ple
f

```

```
arcrams_lib/LEFs/mem_ts65njb1p11sassl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_byes_rno.ple
f
```

### Example C-6 Sample Memory Behavioral Simulation Models

```
arcrams_lib/SimModels_verilog/mem_ts65njb1p11sassl512sa04_256x13_cm4_bk4_cdfalse_bwe0_
byes_rno.v
arcrams_lib/SimModels_verilog/mem_ts65njb1p11sassl512sa04_2048x32_cm8_bk2_cdfalse_bwe0_
byes_rno.v
arcrams_lib/SimModels_verilog/mem_ts65njb1p11sassl512sa04_256x22_cm4_bk4_cdfalse_bwe1_
byes_rno.v
arcrams_lib/SimModels_verilog/mem_ts65njb1p11sassl512sa04_2048x32_cm8_bk2_cdfalse_bwe1_
byes_rno.v
arcrams_lib/SimModels_verilog/mem_ts65njb1p11sassl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_
byes_rno.v
arcrams_lib/SimModels_verilog/mem_ts65njb1p11sassl512sa04_16384x32_cm16_bk8_cdfalse_bw
el_byes_rno.v
arcrams_lib/SimModels_verilog/mem_ts65njb2p11asdr132ksa07p1_32x32_cm1_bk4_cdfalse_bwe0_
byes_rno.v
arcrams_lib/SimModels_verilog/mem_ts65njb1p11sassl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_
byes_rno.v
arcrams_lib/SimModels_vhdl/
```

### Example C-7 Sample Verilog Wrappers

The following are the Verilog wrappers issued by RDF SEIF flow that match the selected memory macros pin compatible to the original interface

```
arcrams_lib/WrapperFiles/
arcrams_lib/WrapperFiles/wrapper_ic_tag_ram.v
arcrams_lib/WrapperFiles/wrapper_ic_data_ram.v
arcrams_lib/WrapperFiles/wrapper_dc_tag_ram.v
arcrams_lib/WrapperFiles/wrapper_dc_data_ram.v
arcrams_lib/WrapperFiles/wrapper_iccm_ram.v
arcrams_lib/WrapperFiles/wrapper_iccm1_ram.v
arcrams_lib/WrapperFiles/wrapper_dccm_ram.v
arcrams_lib/WrapperFiles/wrapper_regfile_ram.v
arcrams_lib/WrapperFiles/wrapper_smart_ram.v
```

### Example C-8 Sample Memory Macro Datasheets

The following are the memory macros datasheets generated by SEIF

```
arcrams_lib/Text/
arcrams_lib/Text/mem_ts65njb1p11sassl512sa04_256x13_cm4_bk4_cdfalse_bwe0_byes_rno_allp
vt.ds
arcrams_lib/Text/mem_ts65njb1p11sassl512sa04_2048x32_cm8_bk2_cdfalse_bwe0_byes_rno_all
pvt.ds
arcrams_lib/Text/mem_ts65njb1p11sassl512sa04_256x22_cm4_bk4_cdfalse_bwe1_byes_rno_allp
vt.ds
arcrams_lib/Text/mem_ts65njb1p11sassl512sa04_2048x32_cm8_bk2_cdfalse_bwe1_byes_rno_all
pvt.ds
arcrams_lib/Text/mem_ts65njb1p11sassl512sa04_4096x32_cm8_bk2_cdfalse_bwe1_byes_rno_all
pvt.ds
```

```

arcrams_lib/Text/mem_ts65njb1p11sassl512sa04_16384x32_cm16_bk8_cdfalse_bwe1_byes_rno_allpvt.ds
arcrams_lib/Text/mem_ts65njb2p11asdrl32ksa07p1_32x32_cm1_bk4_cdfalse_bwe0_byes_rno_allpvt.ds
arcrams_lib/Text/mem_ts65njb1p11sassl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_byes_rno_allpvt.ds

```

### Example C-9 Sample Verilog Wrapper Definition Files

Definition files referenced by the wrapper and configured by SEIF are written to the verilog/RTL dir.

```

verilog/RTL/dccm_ramdef.v
verilog/RTL/dc_data_ramdef.v
verilog/RTL/dc_tag_ramdef.v
verilog/RTL/iccm1_ramdef.v
verilog/RTL/iccm_ramdef.v
verilog/RTL/ic_data_ramdef.v
verilog/RTL/icgcelldef.v
verilog/RTL/ic_tag_ramdef.v
verilog/RTL/regfile_ramdef.v
verilog/RTL/smart_ramdef.v
verilog/RTL/stdcelldef.v
verilog/RTL/synopsys_tsmc_port_mappings.v

```

### Example C-10 Sample Smart Ram Verilog Wrapper Definition

```

verilog/RTL/smart_ramdef.v:

`define smart_ram
`ifdef BldCfgSizeWordsExtended
    `undef BldCfgSizeWordsExtended
`endif
`ifdef SizeBitsExtended
    `undef SizeBitsExtended
`endif
`define smart_ram_ts65njb1p11sassl512s
`define Memsmart_ram mem_ts65njb1p11sassl512sa04_4096x49_cm8_bk2_cdfalse_bwe0_byes_rno
`define smart_ram0
`ifdef bistprts
    `undef bistprts
`endif

```

### Example C-11 Sample Smart Ram Wrapper File

*arcrams\_lib/WrapperFiles/wrapper\_smart\_ram.v:*

ARCrms copies this wrapper file to *Verilog/RTL/smart\_ram.v*. Original behavioral models are moved to *verilog/RTL/smart\_ram.old*

```

`include "smart_ramdef.v"
`include "defines.v"
`include "synopsys_tsmc_port_mappings.v"
module smart_ram (
    clk,
    din,

```

```

    dout,
    addr,
    cs,
    we
);
input                               clk;
input [`SM_DATA_MSB:0]   din;
input [`SM_ADR_MSB:0]    addr;
input                               cs;
input                               we;
output [`SM_DATA_MSB:0]  dout;
`ifdef SizeBitsExtended
    wire [`SM_DATA_MSB+`SizeBitsExtended:0] dout_extra;
    wire [`SM_DATA_MSB+`SizeBitsExtended:0] din_extra;
    assign dout = dout_extra[`SM_DATA_MSB:0];
    assign din_extra = {{`SizeBitsExtended{1'b0}},din} ;
`else
    wire [`SM_DATA_MSB:0] dout_extra;
    wire [`SM_DATA_MSB:0] din_extra;
    assign dout = dout_extra[`SM_DATA_MSB:0];
    assign din_extra = din ;
`endif
`ifdef BldCfgSizeWordsExtended
    wire [`SM_ADR_MSB+`BldCfgSizeWordsExtended:0] addr_extra;
    assign addr_extra = {{`BldCfgSizeWordsExtended{1'b0}},addr} ;
`else
    wire [`SM_ADR_MSB:0] addr_extra;
    assign addr_extra = addr ;
`endif
`Memsmart_ram
`RAM_ts65njb1p11sasl512s(smart_ram0,dout_extra,addr_extra,din_extra,we,cs,clk)
endmodule

```

**Example C-12 Sample RAM Port Mappings**

The following are the `RAM\_ts65njb1p11sassl512s port mappings in the Verilog/RTL/synopsys\_tsmc\_port\_mapping.v file that is generated by the SEIF flow:

```
`define RAM_ts65njb1p11sassl512s (instname,q,addr,data,we,me,clk) \
  instname ( \
    .Q(q), \
    .SO(), \
    .ADR(addr), \
    .D(data), \
    .WE(we), \
    .ME(me), \
    .CLK(clk), \
    .TADR(), \
    .TD(), \
    .TWE(), \
    .TME(), \
    .TCLK(clk), \
    .BISTE(), \
    .TEST1(), \
    .RME(), \
    .RM(), \
    .CD(), \
    .CAPT(), \
    .BISTMODE(), \
    .SWT(1'b0), \
    .SI(1'b0), \
    .SE(1'b0), \
    .DFTMODE(), \
    .AWT() \
  );
```



# D

## ARC UART

You can add the UART component from the DesignWare® ARC® Memory Subsystem (MSS) as part of the testbench for confidence tests. This appendix describes the ARC UART.

**Note**

This UART model and associated testbench logic are supported only for the BVCI memory interface. This code is not a released product. Customers are encouraged to use a Synopsys DesignWare AHB UART in their actual systems.

### D.1 UART Features

The ARC UART is a simple UART (Universal Asynchronous Receiver Transmitter) with an integrated baud-rate generator. The ARC UART supports asynchronous transmission with eight data bits, one stop bit, and no parity. A single interrupt is output with two separate interrupt-enable bits in the UART STATUS register.

A programmable baud-rate generator is provided. It generates a 4x oversampled bit clock and a 1x bit clock. The ARC UART has a four-deep receive FIFO. The UART registers are mapped into memory through the memory arbitrator. Up to three UARTs can be enabled, each having its own interrupt to an ARC processor.

#### D.1.1 Block Diagram

[Figure D-1](#) on page [376](#) shows the basic blocks for the ARC UART.

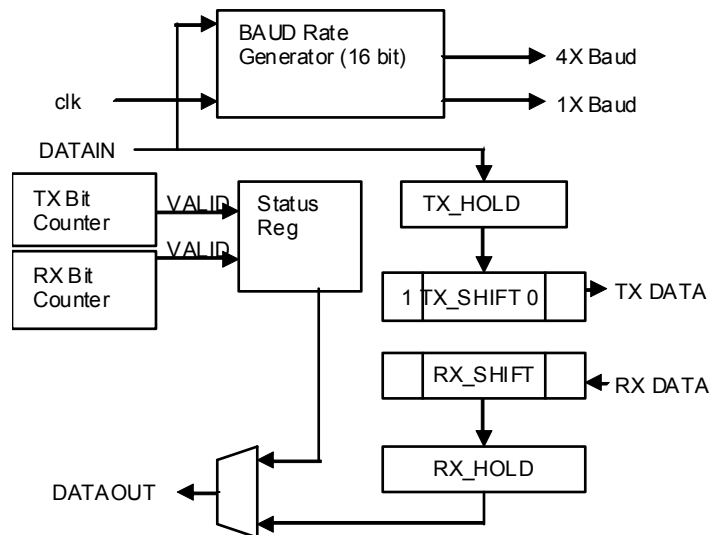
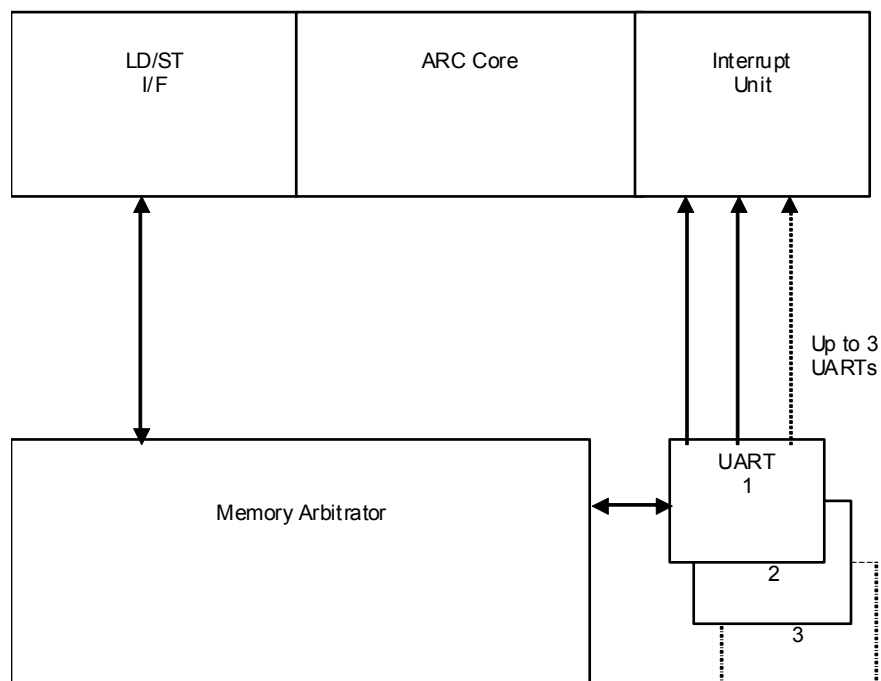
**Figure D-1 Basic ARC UART Block Diagram****D.1.2 System Block Diagram**

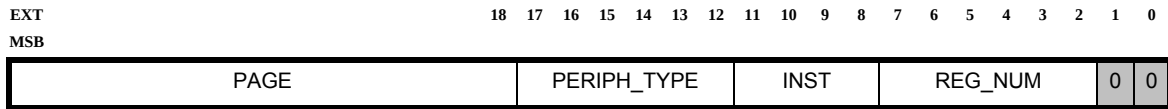
Figure D-2 shows the integration with an ARC processor.

**Figure D-2 ARC UART and ARC System Diagram**



### D.1.3 Register Base Address

Each ARC UART in a design has its register base address mapped in the following manner:



| Field Name  | Description                                                                                                                                                                                                                                                                                                                                          |
|-------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| REG_NUM     | register # - These give addresses along longword boundaries (e.g. 0,4,8,etc). These register addresses correspond to the register definition table below.                                                                                                                                                                                            |
| INST        | peripheral instance # - this can be 0-F although only 8 UARTs are supported currently. UART 0 is mapped to peripheral instance number 0, UART 1 is mapped to peripheral instance number 1, etc.                                                                                                                                                      |
| PERIPH_TYPE | peripheral type – 1 = ARC UART, 2 = ARC EMAC.                                                                                                                                                                                                                                                                                                        |
| PAGE        | Peripherals page start address (bit 18 to maximum bit available for memory addressing, EXTMSB). The default value of PAGE is set to 0x3F to give a peripheral page of 0x00FC0000 – 0x00FFFFFF. Note that the page start address is configurable at build time, and its current value can be read from the build configuration register, P_BASE_ADDR. |

This gives the following register base addresses for the ARC UART:

- 0x00FC1000: ARC UART 0
- 0x00FC1100: ARC UART 1
- 0x00FC1200: ARC UART 2

### D.1.4 Control Registers

This subsection describes the control registers for the ARC UART. The following table shows the memory map of the control registers.

**Table D-1 ARC UART Control Registers**

| Register Name | Register # | Address Offset |
|---------------|------------|----------------|
| ID0           | 0          | 0x00           |
| ID1           | 1          | 0x04           |
| ID2           | 2          | 0x08           |
| ID3           | 3          | 0x0C           |
| DataRx/DataTx | 4          | 0x10           |
| STATUS        | 5          | 0x14           |
| BAUDL         | 6          | 0x18           |

**Table D-1 ARC UART Control Registers (Continued)**

|       |   |      |
|-------|---|------|
| BAUDH | 7 | 0x1C |
|-------|---|------|

**D.1.4.1 ID Register (ID 0)**

Access: Read

Address: 0x00



The field description for this register is shown in the following table.

| Field Name | Description                                                                                         |
|------------|-----------------------------------------------------------------------------------------------------|
| ID[5:0]    | Configuration number. This number is set to 0x01 and indicates that the peripheral is the ARC UART. |

**D.1.4.2 ID Register (ID 1)**

Access: Read

Address: 0x04



The field description for this register is shown in the following table.

| Field Name | Description                         |
|------------|-------------------------------------|
| NID[5:0]   | Ones complement version of ID[5:0]. |

**D.1.4.3 ID Register (ID 2)**

Access: Read

Address: 0x08

|          |   |   |   |   |   |   |   |
|----------|---|---|---|---|---|---|---|
| 7        | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
| Revision |   |   |   |   |   |   |   |

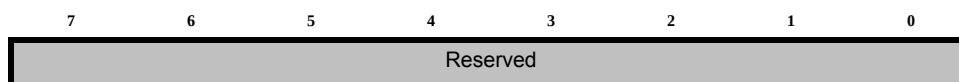
The field description for this register is shown in the following table.

| Field Name    | Description                     |
|---------------|---------------------------------|
| REVISION[7:0] | Revision number of the ARC UART |

#### D.1.4.4 ID Register (ID 3)

Access: Read

Address: 0x0c



The field description for this register is shown in the following table.

| Field Name | Description              |
|------------|--------------------------|
| RESERVED   | Unused; defaults to 0x00 |

#### D.1.4.5 Data Receive Register (DataRX)

Access: Read

Address: 0x10



The field description for this register is shown in the following table.

| Field Name | Description      |
|------------|------------------|
| DATARX     | RX Data Register |



#### Note

Any read of the RxDATA register by the host debug system is intentionally non-destructive. Only load instructions by the ARC processor can remove a character from the receive data FIFO.

**D.1.4.6 Data Transmit Register (DataTX)**

Access: Write

Address: 0x10



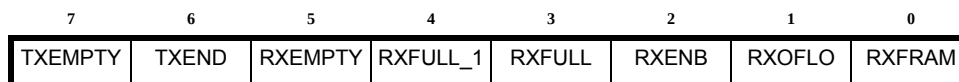
The field description for this register is shown in the following table.

| Field Name | Description      |
|------------|------------------|
| DATATX     | TX Data Register |

**D.1.4.7 Status Register (STATUS)**

Access: Read/Write

Address: 0x14



The field descriptions for this register are shown in the following table.

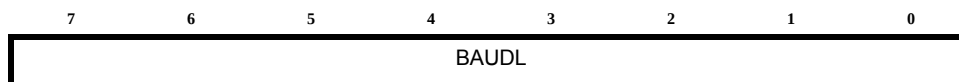
**Table D-2 Field Descriptions for Status Register**

| Field Name | Description                                                                                                                           |
|------------|---------------------------------------------------------------------------------------------------------------------------------------|
| TXEMPTY    | TX holding register empty. (read only) Equals 1 if DataTX is empty and can be written. Equals 0 if DataTx is full, do not write data. |
| TXENB      | TX Interrupt Enable. If 1 then TXEMPTY=1 will interrupt. If 0, masked.                                                                |
| RXEMPTY    | RxFIFO Empty. (read only) Equals 1 if RxFIFO is empty. Automatically cleared when data is read from RxDATA.                           |
| RXFULL_1   | RxFIFO Full minus 1. Equals 1 if RxFIFO is full minus 1.                                                                              |
| RXFULL     | RxFIFO Full. Equals 1 if RxFIFO is full.                                                                                              |
| RXENB      | Rx Interrupt Enable. If 1 then RXEMPTY=0 or RXOFLO or RXFRAM=1 will interrupt. If 0, masked.                                          |
| RXOFLO     | Rx Data Overflow. Set to 1 when a character has been received but RXFULL is still 1.                                                  |
| RXFRAM     | RX Framing Error. Set to 1 when a stop bit was not detected.                                                                          |

### D.1.4.8 Baud Rate Register Low Byte (BAUDL)

Access: Write

Address: 0x18



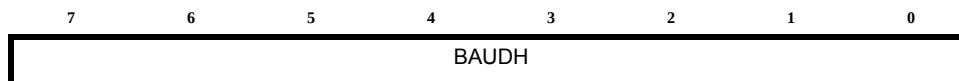
The field description for this register is shown in the following table.

| Field Name | Description                                                                                                                                                                                                                                                                               |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BAUDL      | Baud rate register [7:0]. Baud register values are computed as:<br>$\text{Baud register value} = (\text{clock\_freq} / (\text{BAUD} * 4)) - 1 \% 256$ .<br>(where % is a modulus operation)<br>Example: If clock_freq=12 MHz and you want 19.2k baud, $(12e6 / (19.2e3 * 4)) - 1 = 155$ . |

### D.1.4.9 Baud Rate Register High Byte (BAUDH)

Access: Write

Address: 0x1c



The field description for this register is shown in the following table.

| Field Name | Description                                                                                                                                                                                                                                           |
|------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| BAUDH      | Baud rate register [15:8]. Baud register values are computed as:<br>$\text{Baud register value} = (\text{clock\_freq} / (\text{BAUD} * 4)) - 1 / 256$ .<br>Example: If clock_freq=12 MHz and you want 19.2k baud, $(12e6 / (19.2e3 * 4)) - 1 = 155$ . |

## D.2 UART Functions

This section describes the functions of the ARC UART.

### D.2.1 Interrupts

If the ARC UART is interrupt-driven, a single interrupt is output with two separate interrupt-enable bits in the STATUS register.

### D.2.2 Transmit Operation

The transmit logic waits for an ARC processor to load a byte into the TX\_HOLD register. This byte is transferred into the TX\_SHIFT register. The TX\_BIT\_CTR begins running, and the start bit is output on TX\_DATA. After all 10 bits (start, eight data bits, stop bits) have been shifted, the next byte is immediately loaded into the TX\_SHIFT register if another byte is available in the TX\_HOLD register.

If TXENB=1, then when TXEMPTY=1 the interrupt line is active. TXEMPTY is cleared by writing to the DataTX register.

### D.2.3 Receive Operation

The receive logic is more complex than transmit as it must synchronize to the incoming data stream. RX\_DATA is first synchronized twice to eliminate meta-stability problems. Then the RX\_BIT\_CTR begins running, and the current phase of the 4X clock is sampled. This allows sampling of each bit at the 1/2 to 3/4 point in the bit. After all 10 bits are shifted in, the start and stop bits are checked and the framing error is activated if they are not correct. The data are transferred into the RX\_FIFO register and another byte can be received while the ARC processor is reading the RX\_FIFO register.

If RXENB=1, then when a character is received and placed in the FIFO, RXEMPTY goes to zero and an interrupt is generated. RXOFLO and RXFRAM also set the interrupt line active. RXEMPTY is automatically set or cleared by RxFifo logic. RXEMPTY is set to one any time the RxFIFO is empty. RXEMPTY is cleared when a character is received and loaded into the FIFO. Reading the DataRX register sets RXEMPTY high again after all characters have been read out of the FIFO. RXOFLO and RXFRAM are set by hardware but must be written with zero by software to clear the bit. If the ARC processor has not read a byte from a UART full-receive FIFO (RxFifo) before another byte has been shifted in, then the RX\_OFLO bit is set. However, the data in the Rx FIFO are not overwritten.

The ARC UART has a receive FIFO with a depth of 4.



#### Note

Any read of the DataRX register by the host debug system is intentionally non-destructive. Only load instructions by the ARC processor can remove a character from the receive data FIFO.

### D.2.4 Baud Rate Generation

A programmable baud-rate generator is provided. It generates a 4X oversampled bit clock and a 1X bit clock. The baud-rate registers are write-only.

To set the baud rate, set the baud registers (BAUDL and BAUDH).



#### Note

Any changes to the baud rate (as set by BAUDL and BAUDH) take immediate effect. If the baud rate is changed during a transmission, erroneous characters are transmitted.

The 16-bit baud register value can be calculated with the following equation:

$$\text{baud\_register\_value} = (\text{Clock\_Frequency} / (4 * \text{baud\_rate})) - 1.$$

Therefore

$$\text{BAUDL} = \text{baud\_register\_value} \% 256$$

(where % is a modulus operation)

and

$$\text{BAUDH} = \text{baud\_register\_value} / 256$$



E

ARConnect Commands

This section describes the details of each ARConnect command, the related format of MCIP\_WDATA and MCIP\_READBACK registers, and the command usages from point view of processor core by using SR/LR instructions.

E.1 CMD\_CHECK\_CORE\_ID

The processor core connected to ARConnect is numbered with CORE ID from CORE 0 to CORE N-1 (depend on the cores number). Each core can use the CMD\_CHECK\_CORE\_ID command to find out its CORE ID.

The MCIP\_READBACK register returns a value of CORE ID.

E.1.1 Parameter Format

N/A

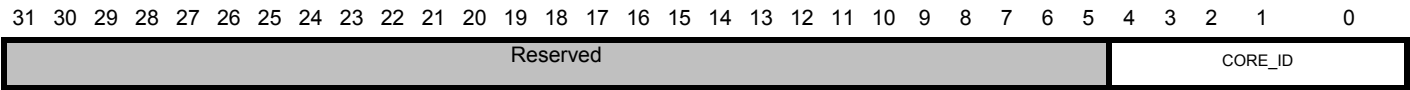
E.1.2 MCIP\_WDATA Register Format

N/A

E.1.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns an encoded value for CORE ID after the core issued CMD\_CHECK\_CORE\_ID command, as shown in [Figure E-1](#)

Figure E-1 MCIP\_READBACK Register Format for CMD\_CHECK\_CORE\_ID Command



The field descriptions are shown in [Table E-1](#)

**Table E-1 MCIP\_READBACK Register Field Descriptions for CMD\_CHECK\_CORE\_ID Command**

| Field  | Bits  | Description                                                                                         |
|--------|-------|-----------------------------------------------------------------------------------------------------|
| CoreID | [4:0] | Indicates CORE ID of this core issuing CMD_CHECK_CORE_ID command<br>0 = CORE0<br>1 = CORE1<br>..... |

### E.1.4 Command Usage: Check the CORE\_ID of the core issuing CMD\_CHECK\_CORE\_ID Command

If a core wishes to check its CORE\_ID, it can set the command CMD\_CHECK\_CORE\_ID to MCIP\_CMD register by a single SR instruction. Example as below,

Example 1, generate a level-sensitive interrupt to core1 in the system

```
SR r2, [MCIP_CMD]; // the value of r2 is 0x00000000, send CMD_CHECK_CORE_ID
                    // command to the ARConnect to check its CORE_ID

LR r4, [MCIP_READBACK]; // the CORE ID value is read back to the MCIP_READBACK register and
                        // stored into r4 register of core1. If the value is 0x00000001, it indicates
                        // this core is CORE 1.
```

## E.2 CMD\_INTRPT\_GENERATE\_IRQ

Each core can be an interrupt initiator core and can use the CMD\_INTRPT\_GENERATE\_IRQ command to generate an interrupt to another core or to external system in the system.

PARAMETER field accompany with this command is used to distinguish whether the interrupt is generated to the core or to the external system, and specify the CORE ID of the interrupt receiver core if the interrupt is generated to the core.

Each core is assigned an internal INTRPT\_STATUS register in the inter-core interrupt unit to record the interrupt status (acknowledged or not) from it to all other cores. The CMD\_INTRPT\_GENERATE\_IRQ command would set one bit in the interrupt initiator core's INTRPT\_STATUS register corresponding to the interrupt receiver core.

The interrupt line of the interrupt receiver core is asserted after the ARConnect manipulating this command and keeps high until the interrupt receiver core acknowledges all the inter-core interrupts it received

### E.2.1 Parameter Format

Parameter [8] defines whether the interrupt is generated to the core or to the external system.

- 1: generating a pulse-sensitive interrupt to external system, and any value in Parameter [7:0] is ignored
- 0: generating a level-sensitive interrupt to another core

Parameter [4:0] specifies the CORE ID of the receiver core. Set its value to 0, 1, 2, or 3.

### E.2.2 MCIP\_WDATA Register Format

N/A

### E.2.3 MCIP\_READBACK Register Format

N/A

### E.2.4 Command Usage: Generate Inter-core Interrupts

Each core can be an interrupt initiator core and can use the CMD\_INTRPT\_GENERATE\_IRQ command to generate an interrupt to another core or to external system.

Example 1, generate a level-sensitive interrupt to core1 in the system

```
SR 0x00000101, [MCIP_CMD]; // send CMD_INTRPT_GENERATE_IRQ command to ARConnect
                        // to generate a level-sensitive interrupt to core1
```

Example 2, generate a pulse-sensitive interrupt to external system.

```
SR r1, [MCIP_CMD]; // the value of r1 is 0x00010001, indicates sending CMD_INTRPT_GENERATE
                    // _IRQ command to ARConnect and generating a pulse-sensitive interrupt to
                    // external system.
```

#### NOTES:

- Indexing to the core itself is meaningless and is ignored. For example, if the CORE ID of current core is 3, setting the parameter field equals 3 (Parameter [7:0]=3) accompany with this command is meaningless and no inter-core interrupt is generated.
- For processor cores, only level-sensitive interrupt can wake up a sleeping core and generate acknowledgment to the interrupt initiator if the priority of this interrupt is higher than or equal to that of the sleep mode.

## E.3 CMD\_INTRPT\_GENERATE\_ACK

Each core can be an interrupt receiver core and can use the CMD\_INTRPT\_GENERATE\_ACK command to generate an interrupt acknowledgement to the interrupt-initiating core.

PARAMETER field is used to specify the CORE ID of the interrupt initiator core.

The CMD\_INTRPT\_GENERATE\_ACK command would clear one bit in the interrupt initiator core's INTRPT\_STATUS register corresponding to the interrupt receiver core. However, the interrupt line of this interrupt receiver core is only de-asserted after all inter-core interrupts it received have been served.

### E.3.1 Parameter Format

Parameter [4:0] specifies the CORE ID of the interrupt initiator core. Set its value to 0, 1, 2, or 3.

### E.3.2 MCIP\_WDATA Register Format

N/A

### E.3.3 MCIP\_READBACK Register Format

N/A

### E.3.4 Command Usage: Generate Inter-core Interrupts Acknowledgment

For an interrupt receiver core to generate inter-core interrupt acknowledge to other interrupt initiator cores, it can set the command field value to 0x02 in the MCIP\_CMD register and combined with CORE ID in the parameter field by a single SR instruction.

Example as below, generate inter-core interrupt acknowledge to core2,

```
SR r5, [MCIP_CMD]; // the value of r5 is 0x00000202, send CMD_INTRPT_GENERATE_ACK
```

```
// command to the ARConnect and generate interrupt acknowledge to core2.
```

#### NOTES:

- Indexing to the core itself is meaningless and is ignored. For example, if the CORE ID of the core is 1, setting the parameter field equals 1 (Parameter[7:0]=1) accompany with this command is meaningless and no inter-core interrupt acknowledgement will be generated.
- Since level-sensitive interrupt is only de-asserted after been acknowledged, software is responsible to program the interrupt receiver core to acknowledge that interrupt in the service routine to clear down the interrupt line prior to exiting the service routine. For an interrupt receiver core to serve the inter-core interrupts and acknowledge the interrupts (by CMD\_INTRPT\_GENERATE\_ACK), it's recommended that the core should guarantee (by leaving enough time period or by using CMD\_INTRPT\_CHECK\_SOURCE) the inter-core interrupt lines have been de-asserted by ARConnect before exiting the interrupt handler to avoid the same interrupt being served more than one time.

## E.4 CMD\_INTRPT\_READ\_STATUS

As an interrupt initiator, each core can use the CMD\_INTRPT\_READ\_STATUS command to check the interrupt status generated from it to other cores.

PARAMETER field is used to specify the CORE ID of the interrupt receiver core.

The MCIP\_READBACK register returns the value of one bit in the core's internal INTRPT\_STATUS register corresponding to the interrupt receiver core specified by PARAMETER field.

### E.4.1 Parameter Format

N/A

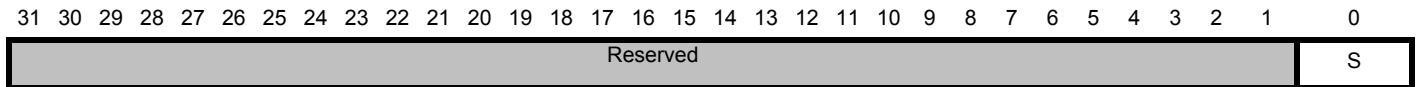
### E.4.2 MCIP\_WDATA Register Format

N/A

### E.4.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns one-bit valid value after the core issued CMD\_INTRPT\_READ\_STATUS command, as shown in [Figure E-2](#).

**Figure E-2 MCIP\_READBACK Register Format for CMD\_INTRPT\_READ\_STATUS Command**



The field descriptions are shown in [Table E-2](#)

**Table E-2 MCIP\_READBACK Register Field Descriptions for CMD\_INTERRUPT\_READ\_STATUS Command**

| Field | Bits | Description                                                                                                                                                                                                    |
|-------|------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| S     | [0]  | <p>Interrupt Status of the specified Inter-core Interrupt</p> <p>0 = there is no outstanding interrupt or a former interrupt has been serviced (acknowledged)</p> <p>1 = there is an outstanding interrupt</p> |

### E.4.4 Command Usage: Check the Status of Inter-core Interrupts

For an interrupt initiator core to check the interrupt status, it can set the command field value to 0x03 in the MCIP\_CMD register and combined with CORE ID in the parameter field by a single SR instruction, and use a LR instruction to get the value returns from the MCIP\_READBACK register.

Example as below, assume core1 has generated an interrupt to core3 and wants to check its service status.

```
SR r2, [MCIP_CMD];    // the value of r2 is 0x00000303, send CMD_INTRPT_READ_STATUS
```

```
    // command to the ARConnect to check the status of interrupt
```

```
    // from this core to core3.
```

```
LR r4, [MCIP_READBACK]; // the data is read back to the MCIP_READBACK register and stored into
```

```
    // r4 register of core1. If the value is 0x00000001, indicates that status of
```

```
    // interrupt from this core to core3 has not been served yet.
```

```
    // If the value is 0x00000000, indicates that either there is no
```

```
    // outstanding interrupt or the interrupt has been acknowledged.
```

#### NOTE:

- Indexing to the core itself is meaningless and always returns zero. For example, if the CORE ID of the core is 3, setting the parameter field equals 3 (Parameter[7:0]=3) accompany with this command is meaningless and the data read back from the MCIP\_READBACK register is always zero.
- It is recommended that an interrupt initiator core checks the status of a previous interrupt it sent (by using CMD\_INTRPT\_READ\_STATUS command) before generating another interrupt(s) to the same interrupt receiver core. This is to ensure that the previous interrupt is not overridden.

## E.5 CMD\_INTRPT\_CHECK\_SOURCE

Each core can be an interrupt receiver core. And all inter-core interrupts from other cores to one interrupt receiver core have been combined into single interrupt line.

For interrupt receivers, each core can use the CMD\_INTRPT\_CHECK\_SOURCE command to find out which core(s) has been generated inter-core interrupt and waiting to be served.

The MCIP\_READBACK register returns a vector value gathered from the internal INTRPT\_STATUS registers of all cores (one bit of each register, corresponding to the CORE ID of core).

### E.5.1 Parameter Format

N/A

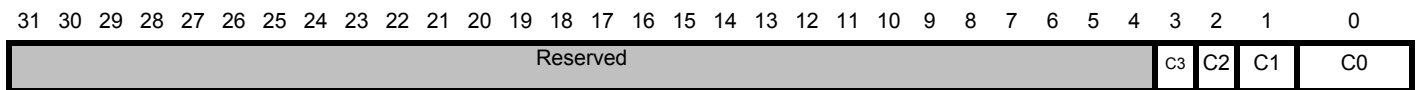
### E.5.2 MCIP\_WDATA Register Format

N/A

### E.5.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns a vector value after the core issued CMD\_INTRPT\_CHECK\_SOURCE command, as shown in [Figure E-3](#).

**Figure E-3 MCIP\_READBACK Register Format for CMD\_INTRPT\_CHECK\_SOURCE Command**



The field descriptions are shown in [Table E-3](#)

**Table E-3 MCIP\_READBACK Register Field Descriptions for CMD\_INTRPT\_CHECK\_SOURCE Command**

| Field | Bits | Description                                                                                                                                                                                                   |
|-------|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C0    | [0]  | Interrupt Status of CORE0.<br>Indicates whether CORE0 is generating interrupt to current core.<br>0 = CORE0 doesn't generate interrupt to current core.<br>1 = CORE0 is generating interrupt to current core. |
| C1    | [1]  | Interrupt Status of CORE1.<br>Indicates whether CORE1 is generating interrupt to current core.<br>0 = CORE1 doesn't generate interrupt to current core.<br>1 = CORE1 is generating interrupt to current core. |
| C2    | [2]  | Interrupt Status of CORE2.<br>Indicates whether CORE2 is generating interrupt to current core.<br>0 = CORE2 doesn't generate interrupt to current core.<br>1 = CORE2 is generating interrupt to current core. |

| Field | Bits | Description                                                                                                                                                                                                   |
|-------|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| C3    | [3]  | Interrupt Status of CORE3.<br>Indicates whether CORE3 is generating interrupt to current core.<br>0 = CORE3 doesn't generate interrupt to current core.<br>1 = CORE3 is generating interrupt to current core. |

### E.5.4 Command Usage: Check the Status of Inter-core Interrupts

If an interrupt receiver core wishes to check the source of interrupt generated by which core(s), it can set the command field value to 0x04 in the MCIP\_CMD register by a single SR instruction. If there are multiple interrupt initiator cores generate interrupt to this interrupt receiver core, the core can choose to serve them one by one or all of them together in the interrupt service routine for different application scenarios.

Example as below, assume core1 receives an inter-core interrupt and wants to check the source of this interrupt.

```
SR r2, [MCIP_CMD];    // the value of r2 is 0x00000004, send CMD_INTRPT_CHECK_SOURCE
                      // command to the ARConnect to check the source of interrupt generated by
                      // which initiator core(s)

LR r4, [MCIP_READBACK]; // the vector value is read back to the MCIP_READBACK register and
                      // stored into r4 register of core1. If the value is 0x00000006, it indicates
                      // that both core2 and core3 are generating interrupt to core1.
```

#### NOTE:

If the interrupt receiver core has acknowledged the interrupt initiator core by using CMD\_INTRPT\_GENERATE\_ACK command, the corresponding bit of INTRPT\_STATUS will be cleared.

## E.6 CMD\_SEMA\_CLAIM\_AND\_READ

Each core can use the CMD\_SEMA\_CLAIM\_AND\_READ command to claim a particular semaphore and verify its ownership.

PARAMETER field is used to specify the index of semaphore.

The MCIP\_READBACK register returns the claim result.

### E.6.1 Parameter Format

Parameter [4:0] defines the index of semaphore; Set its value from 0, 1, to N-1 (N is the number of semaphores configured in MCIP\_SEMA\_BUILD register during the system build-time).

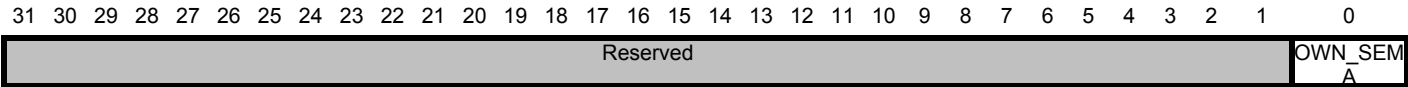
### E.6.2 MCIP\_WDATA Register Format

N/A

E.6.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns the status of a particular semaphore to check whether the core has obtained the semaphore it claimed after the CMD\_SEMA\_CLAIM\_AND\_READ command.

Figure E-4 MCIP\_READBACK Register Format for CMD\_SEMA\_CLAIM\_AND\_READ Command



The field descriptions are shown in

Table E-4 MCIP\_READBACK Register Field Descriptions for CMD\_SEMA\_CLAIM\_AND\_READ Command

| Field    | Bits | Description                                                                                                                                                         |
|----------|------|---------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| OWN_SEMA | [0]  | Semaphore ownership Status<br>0 = the semaphore is not available for the core, and the claim request failed<br>1 = the core has successfully obtained the semaphore |

E.6.4 Command Usage: Claim Semaphore and Check Semaphore ownership

If a core wants to claim the ownership of a semaphore before accessing shared resources, it can set the command field value to 0x11 (CMD\_SEMA\_CLAIM\_AND\_CHECK) and combined with semaphore index in the parameter field in the MCIP\_CMD register by a SR instruction, and then use a LR instruction to get the data returns from the MCIP\_READBACK register checking whether the core has obtained the ownership of particular semaphore or not.

Example as below

```
SR 0x00000811, [MCIP_CMD]; // 0x0008 indicates that the core wants to claim-and-check the
                               // semaphore with index 8 (SEMA8).
                               // 0x11 is command CMD_SEMA_CLAIM_AND_READ.
```

```
LR r5, [MCIP_READBACK]; // if the value of r5 is 1, indicates that core has obtained the
                           // ownership of SEMA8. Otherwise, value 0 indicates that this
                           // semaphore is not available for this claim request
```

NOTE: Mutual exclusion is supported in inter-core semaphore unit. That is, only the semaphore has been released can other.

E.7 CMD\_SEMA\_RELEASE

If the core has already obtained the ownership of a semaphore, to release the semaphore and allow it to be available for other cores, the core can use the CMD\_SEMA\_RELEASE command to release its ownership.

PARAMETER field specifies the index of semaphore.



### E.7.1 Parameter Format

Parameter [4:0] defines the index of semaphore; Set its value from 0, 1, 2 to N-1 (N is the number of semaphores configured in MCIP\_SEMA\_BUILD register during the system build-time).

### E.7.2 MCIP\_WDATA Register Format

N/A

### E.7.3 MCIP\_READBACK Register Format

N/A

### E.7.4 Command Usage: Release Semaphore Owned by Core

If a core want to release the ownership of one particular semaphore, it can set the command filed value to 0x12 (CMD\_SEMA\_RELEASE) and combined with semaphore index in the parameter field in the MCIP\_CMD register by one SR instruction.

Example as below, assume the core has the ownership of SEMA3 and wants to release it.

```
SR r4, [MCIP_CMD];          // the value of r4 is 0x00000312, send CMD_SEMA_RELEASE
                             // command to the ARConnect to release the ownership of SEMA3
```

## E.8 CMD\_SEMA\_FORCE\_RELEASE

If the core has already obtained the ownership of a semaphore, to release the semaphore and allow it to be available for other cores, the core can use the CMD\_SEMA\_RELEASE command to release its ownership.

PARAMETER field specifies the index of semaphore.

### E.8.1 Parameter Format

Parameter [4:0] defines the index of semaphore; Set its value from 0, 1, 2 to N-1 (N is the number of semaphores configured in MCIP\_SEMA\_BUILD register during the system build-time).

### E.8.2 MCIP\_WDATA Register Format

N/A

### E.8.3 MCIP\_READBACK Register Format

N/A

### E.8.4 Command Usage: Release Semaphore Owned by Core

In some special conditions or applications, if a core wants to compulsively release a particular semaphore that has been occupied by other cores in the system, it can set the command filed value to 0x13 (CMD\_SEMA\_FORCE\_RELEASE) and combined with semaphore index in the parameter field in the MCIP\_CMD register by one SR instruction.

Example as below, assume SEMA4 has been claimed by other cores.

```
SR 0x00000413, [MCIP_CMD]; // Send CMD_SEMA_FORCE_RELEASE command to the ARConnect
                             // to release SEMA4 to allow it available
```

## E.9 CMD\_MSG\_SRAM\_SET\_ADDR

Message Passing SRAM is a local memory of ARConnect. Each core can use CMD\_MSG\_SRAM\_SET\_ADDR command to set the address of message passing SRAM for subsequent read or write SRAM operations.

Each core is assigned an internal MSG\_ADDR register in the inter-core message passing unit to record the accessing address for the core.

The MCIP\_WDATA register contains the data to be written into the MSG\_ADDR register.

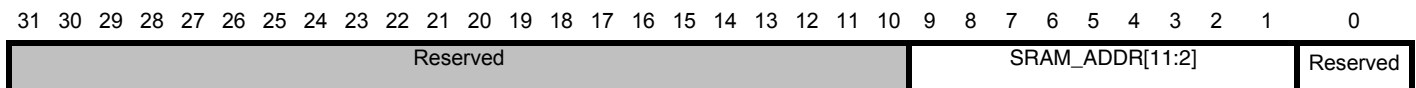
### E.9.1 Parameter Format

N/A

### E.9.2 MCIP\_WDATA Register Format

The data to be written into the MCIP\_WDATA register before the CMD\_MSG\_SRAM\_SET\_ADDR command is shown in [Figure E-5](#)

**Figure E-5 MCIP\_READBACK Register Format for CMD\_MSG\_SRAM\_SET\_ADDR Command**



**NOTE:** Read/write access to the message passing SRAM is word-aligned, so always set the two least-significant bits of SRAM\_ADDR to zero. And the address exceeds the size of SRAM is truncated automatically.

### E.9.3 Command Usage:

The usage of CMD\_MSG\_SRAM\_SET\_ADDR, see [Command Usage: Read SRAM Address Offset Value](#)

## E.10 CMD\_MSG\_SRAM\_READ\_ADDR

Each core can use the CMD\_MSG\_READ\_SRAM\_ADDR command to read the value of its internal MSG\_ADDR register from the inter-core message passing unit.

The MCIP\_READBACK register returns the value of internal MSG\_ADDR register belonged to the core.

### E.10.1 Parameter Format

N/A

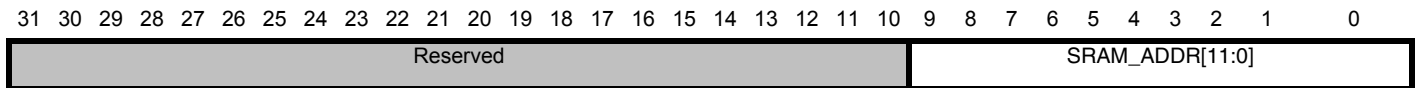
## E.10.2 MCIP\_WDATA Register Format

N/A

## E.10.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns the current address for the core to access the message passing SRAM, as shown in [Figure E-6](#).

**Figure E-6 MCIP\_READBACK Register Format for CMD\_MSG\_SRAM\_READ\_ADDR Command**



## E.10.4 Command Usage: Read internal SRAM Address

Example as below,

```
SR 0x00000022, [MCIP_CMD]; // Send CMD_MSG_SRAM_READ_ADDR command to the ARConnect
                             // to get the current value of core's internal MSG_ADDR register
LR r5, [MCIP_READBACK]; // The address value returns to the MCIP_READBACK register
                             // and stored to r5 of the core
```

## E.11 CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET

Each core can use CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET command to set the address offset value of message passing SRAM for subsequent read or write SRAM operations in incremental mode.

Each core is assigned an internal MSG\_ADDR\_OFFSET register in the inter-core message passing unit to record the address offset value for the core.

The value of MCIP\_WDATA register accompanies with CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET command is then stored into and update the MSG\_ADDR\_OFFSET register.

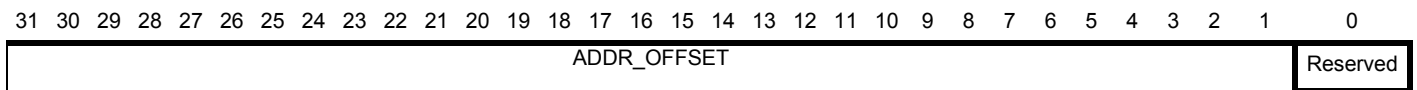
### E.11.1 Parameter Format

N/A

### E.11.2 MCIP\_WDATA Register Format

The data to be written into the MCIP\_WDATA register before the CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET command is shown in [Figure E-7](#).

The offset value is a signed value and encoded with 2's complement, which ranges from -64 to +60. This value is passed to the inter-core message-passing unit for subsequent incremental read/write SRAM operations.

**Figure E-7 MCIP\_READBACK Register Format for CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET Command****NOTE:**

The offset value must be word aligned, so always set the two least-significant bits of ADDR\_OFFSET to zero.

The hardware result is unpredictable if the valid offset value exceeds the allowed range.

**E.11.3 MCIP\_READBACK Register Format**

N/A

**E.11.4 Command Usage: Read internal SRAM Address**

The usage of CMD\_MSG\_SRAM\_SET\_ADDR, [Command Usage: Write data into message passing SRAM](#)

**E.12 CMD\_MSG\_SRAM\_READ\_ADDR\_OFFSET**

Each core can use the CMD\_MSG\_READ\_SRAM\_ADDR\_OFFSET command to read the address offset value for this core.

The MCIP\_READBACK register returns the value of internal MSG\_ADDR\_OFFSET register belonged to the core.

**E.12.1 Parameter Format**

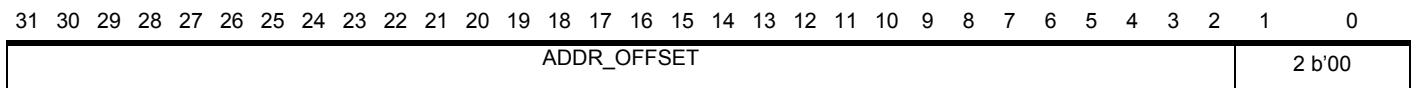
N/A

**E.12.2 MCIP\_WDATA Register Format**

N/A

**E.12.3 MCIP\_READBACK Register Format**

The MCIP\_READBACK register returns the SRAM-address offset value from the inter-core message passing unit (set by last CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET command). The offset value is a signed value and encoded with 2's complement, which ranges from -64 to +60, as shown in

**Figure E-8 MCIP\_READBACK Register Format for CMD\_MSG\_SRAM\_READ\_ADDR\_OFFSET Command****E.12.4 Command Usage: Read SRAM Address Offset Value**

The SRAM address offset value written by last CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET command can be read out for some debug purpose. This can be done via the CMD\_MSG\_SRAM\_READ\_ADDR\_OFFSET command. The MCIP\_READBACK register returns the sign extended address offset value.

Example as below,

```
SR r3, [MCIP_CMD]; // the value of r3 is 0x00000024. Send CMD_MSG_SRAM_READ_ADDR_OFFSET
                    // command to the ARConnect to get SRAM address offset value set by last
                    // CMD_MSG_SRAM_SET_ADDR_OFFSET command.
LR r4, [MCIP_READBACK]; // signed value of the SRAM address offset will return to the
                        // MCIP_READBACK register and store to r4 of the core.
```

**E.13 CMD\_MSG\_SRAM\_WRITE**

Each core can use CMD\_MSG\_SRAM\_WRITE command to trigger a write operation to the message passing SRAM.

The address of message passing SRAM is determined by the internal MSG\_ADDR register (which can be updated by the CMD\_MSG\_SRAM\_SET\_ADDR command, the CMD\_MSG\_SRAM\_WRITE\_INC command, or the CMD\_MSG\_SRAM\_READ\_INC command).

The MCIP\_WDATA register contains the data to be written into this specified address.

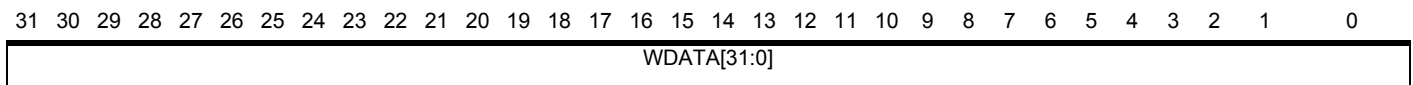
**E.13.1 Parameter Format**

N/A

**E.13.2 MCIP\_WDATA Register Format**

The MCIP\_WDATA register must contain the data to be written into the specified address of SRAM before write-data commands are sent to the message passing SRAM.

The layout of MCIP\_WDATA register is shown [Figure E-9](#)

**Figure E-9 MCIP\_READBACK Register Format for CMD\_MSG\_SRAM\_WRITE Command****E.13.3 MCIP\_READBACK Register Format**

N/A

### E.13.4 Command Usage: Write data into message passing SRAM

For a core wants to send messages to other cores through the message passing unit, it should first write the data into the MCIP\_WDATA register by a SR instruction, and then send the SRAM write command to the MCIP\_CMD register by another SR instruction.

The address of the SRAM can be set by the CMD\_MSG\_SRAM\_SET\_ADDR command. In this situation, two more SR instructions needed before the SRAM write operation.

Example as below,

```
SR r1, [MCIP_WDATA]; // the value of r1 is 0x014, indicates the address of SRAM
SR r2, [MCIP_CMD];   // the value of r2 is 0x00000021, send CMD_MSG_SRAM_SET_ADDR
                      // command to the ARConnect
SR r3, [MCIP_WDATA]; // r3 contains the data needed to be written into the SRAM for following
                      // SRAM write operation
SR r4, [MCIP_CMD];   // the value of r4 is 0x00000025, send CMD_MSG_SRAM_WRITE command to
                      // the ARConnect, and the data in the MCIP_WDATA register will be written
                      // into address 0x014
```

## E.14 CMD\_MSG\_SRAM\_WRITE\_INC

To facilitate streamlined write operations to consecutive or regular addresses in message passing SRAM, incremental write mode is supported in inter-core message passing unit.

After setting the address offset value and the start access address, each core can use CMD\_MSG\_SRAM\_WRITE\_INC command to continuously write data into the message passing SRAM without specifying the access address each time.

This command implicitly updates the internal MSG\_ADDR register by (current MSG\_ADDR value + MSG\_ADDR\_OFFSET value) after execution.

The MCIP\_WDATA register contains the data to be written into current address of message passing SRAM.

### E.14.1 Parameter Format

N/A

### E.14.2 MCIP\_WDATA Register Format

The MCIP\_WDATA register must contain the data to be written into the specified address of SRAM before write-data commands are sent to the message passing SRAM.

The layout of MCIP\_WDATA register can refer to [Figure E-9](#)

### E.14.3 MCIP\_READBACK Register Format

N/A

## E.14.4 Command Usage: Write data in incremental mode

Example as below,

```
SR r1, [MCIP_WDATA]; // the value of r1 is 0x000, indicates the base address of SRAM
SR r2, [MCIP_CMD];    // the value of r2 is 0x00000021, send CMD_MSG_SRAM_SET_ADDR
                      // command to the ARConnect
SR r3, [MCIP_WDATA]; // the value of r3 is 0x00000004, indicates the address offset value of SRAM
                      // for following incremental SRAM write operations is positive 4 (+4)
SR r4, [MCIP_CMD];    // the value of r4 is 0x00000023, send CMD_MSG_SRAM_SET_ADDR_OFFSET
                      // command to the ARConnect
SR r5, [MCIP_WDATA]; // r5 contains the data needed to be written into the SRAM for following
                      // SRAM write operation
SR r6, [MCIP_CMD];    // the value of r6 is 0x00000026, send CMD_MSG_SRAM_WRITE_INC
                      // command to the ARConnect, and the data in the MCIP_WDATA register
will
                      // be written into address 0x000, then the SRAM address will automatically
                      // incremented to 0x004 (current addr+offset value)
SR r7, [MCIP_WDATA]; // r7 contains the data needed to be written into the SRAM for following
                      // SRAM write operation
SR r6, [MCIP_CMD];    // send CMD_MSG_SRAM_WRITE_INC command to the ARConnect, and the
                      // data in the MCIP_WDATA register will be written into address 0x004, then
                      // the SRAM address will automatically incremented to 0x008 (current
                      // addr+offset value)
```

**NOTE:** The offset value set by CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET can be either positive or negative.

## E.15 CMD\_MSG\_SRAM\_WRITE\_IMM

Each core can use CMD\_MSG\_SRAM\_WRITE\_IMM command to write data into message passing SRAM with the address specified by an immediate.

Parameter field specifies this immediate address value.

**NOTE:** The CMD\_MSG\_SRAM\_WRITE\_IMM command doesn't update the value of core's MSG\_ADDR register.

### E.15.1 Parameter Format

Parameter[11:0] specifies the SRAM address by an immediate.

### E.15.2 MCIP\_WDATA Register Format

The MCIP\_WDATA register must contain the data to be written into the specified address of SRAM before write-data commands are sent to the message passing SRAM.

The layout of MCIP\_WDATA register can refer to [Figure E-9](#).

### E.15.3 MCIP\_READBACK Register Format

N/A

### E.15.4 Command Usage: Write data with immediate address

For a core wants to send messages to other cores through the inter-core message passing unit, it should first write the data into the MCIP\_WDATA register by a SR instruction, and then send the SRAM write command to the MCIP\_CMD register by another SR instruction.

The address of the SRAM can come from the immediate value accompany with CMD\_MSG\_SRAM\_WRITE\_IMM command.

Example as below,

```
SR r6, [MCIP_WDATA]; // r3 contains the data needed to be written into the SRAM for following
                        // SRAM write operation
SR 0x00000827, [MCIP_CMD]; // parameter[11:0] is 0x008, indicate the address of SRAM is 0x008
                        // for current SRAM operation. Send CMD_MSG_SRAM_WRITE_IMM
                        // command to the ARConnect, and the data in the MCIP_WDATA
                        // register will be written into address 0x008
```

**NOTE:** Read/write access to the message passing SRAM is word-aligned, so always set the Parameter[1:0] of MCIP\_CMD accompany with CMD\_MSG\_SRAM\_WRITE\_IMM command to zero. And the address exceeds the size of SRAM is truncated automatically.

## E.16 CMD\_MSG\_SRAM\_READ

Each core can use CMD\_MSG\_SRAM\_READ command to trigger a read operation to the message passing SRAM.

The address of message passing SRAM is determined by the internal MSG\_ADDR register (which can be updated by the CMD\_MSG\_SRAM\_SET\_ADDR command, the CMD\_MSG\_SRAM\_WRITE\_INC command, or the CMD\_MSG\_SRAM\_READ\_INC command).

The MCIP\_READBACK register contains the data read out from this specified address.

### E.16.1 Parameter Format

N/A

### E.16.2 MCIP\_WDATA Register Format

N/A

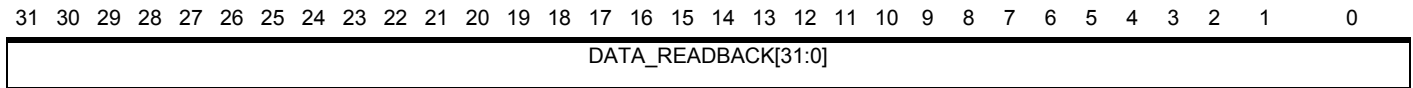


### E.16.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns the data read from specified address of SRAM for CMD\_MSG\_SRAM\_READ command.

The layout of the MCIP\_READBACK register is shown in [Figure E-10](#)

**Figure E-10 MCIP\_READBACK Register Format for CMD\_MSG\_SRAM\_READ Command**



### E.16.4 Command Usage: Write data with immediate address

For a core wants to receive messages from other cores through inter-core message passing unit, it should first send the SRAM read command to the MCIP\_CMD register by a SR instruction, and then use a LR instruction to read the data from the MCIP\_READBACK register.

The address of the SRAM can be set by the CMD\_MSG\_SRAM\_SET\_ADDR command. In this situation, two more SR instructions needed before the SRAM read operation.

Example as below,

```
SR r1, [MCIP_WDATA]; // the value of r1 is 0x014, indicates the address of SRAM
SR r2, [MCIP_CMD];   // the value of r2 is 0x00000021, send CMD_MSG_SRAM_SET_ADDR
                      // command to the ARConnect
SR r3, [MCIP_CMD];   // the value of r3 is 0x00000028, send CMD_MSG_SRAM_READ command
                      // to the ARConnect
LR r4, [MCIP_READBACK]; // the data from SRAM address 0x014 is returned to MCIP_READBACK
                      // register, and sent to r4 of the core
```

## E.17 CMD\_MSG\_SRAM\_READ\_INC

To facilitate streamlined write operations to consecutive or regular addresses in message passing SRAM, incremental write mode is supported in inter-core message passing unit.

After setting the address offset value and the start access address, each core can use CMD\_MSG\_SRAM\_READ\_INC command to continuously read data from the message passing SRAM without specifying the access address each time.

This command implicitly updates the internal MSG\_ADDR register by (current MSG\_ADDR value + MSG\_ADDR\_OFFSET value) after execution.

The MCIP\_READBACK register contains the data read out from current address of message passing SRAM.

### E.17.1 Parameter Format

N/A

### E.17.2 MCIP\_WDATA Register Format

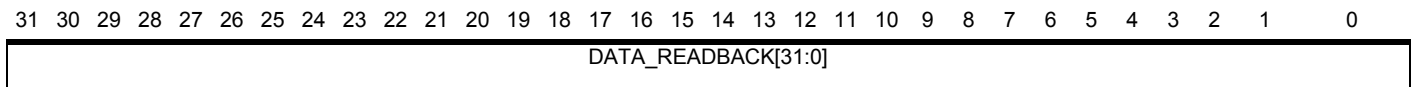
N/A

### E.17.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns the data read from specified address of SRAM for CMD\_MSG\_SRAM\_READ\_INC command.

The layout of the MCIP\_READBACK register, see [Figure E-11](#)

**Figure E-11 MCIP\_READBACK Register Format for CMD\_MSG\_SRAM\_READ\_INC Command**



### E.17.4 Command Usage: Read data in incremental mode

Example as below,

```
SR r1, [MCIP_WDATA];    // the value of r1 is 0x00c, indicates the base address of SRAM
SR r2, [MCIP_CMD];      // the value of r2 is 0x00000021, send CMD_MSG_SRAM_SET_ADDR
                        // command to the ARConnect
SR r3, [MCIP_WDATA];    // the value of r3 is 0xffffffc, indicates the address offset value of SRAM
                        // for following incremental SRAM write operations is negative 4 (-4)
SR r4, [MCIP_CMD];      // the value of r4 is 0x00000023, send CMD_MSG_SRAM_SET_ADDR_OFFSET
                        // command to the ARConnect
SR r5, [MCIP_CMD];      // the value of r5 is 0x00000029, send CMD_MSG_SRAM_READ_INC
                        // command to the ARConnect, and the data will be read out from SRAM
                        // address 0x00c, then the SRAM address will automatically updated to 0x008
                        // (current addr - |offset value|)
LR r10, [MCIP_READBACK]; // the data from SRAM address 0x00c is returned to the
                        // MCIP_READBACK register, and sent to r10 of the core
SR r5, [MCIP_CMD];      // the value of r5 is 0x00000029, send CMD_MSG_SRAM_READ_INC
                        // command to the ARConnect, and the data will be read out from SRAM
                        // address 0x008, then the SRAM address will automatically updated to 0x004
                        // (current addr - |offset value|)
```

```
LR r11, [MCIP_READBACK]; // the data from SRAM address 0x008 is returned to the
                        // MCIP_READBACK register, and sent to r11 of the core
```

**NOTE:** The offset value set by CMD\_MSG\_SRAM\_SET\_ADDR\_OFFSET can be either positive or negative by setting the corresponding parameter field of the command.

## E.18 CMD\_MSG\_SRAM\_READ\_IMM

Each core can use CMD\_MSG\_SRAM\_READ\_IMM command to read data from message passing SRAM with the address specified by an immediate.

PARAMTER field specifies this immediate address value.

**NOTE:** The CMD\_MSG\_SRAM\_READ\_IMM command doesn't update the value of core's MSG\_ADDR register.

### E.18.1 Parameter Format

Parameter[11:0] specifies the SRAM address by an immediate

### E.18.2 MCIP\_WDATA Register Format

N/A

### E.18.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns the data read from specified address of SRAM for CMD\_MSG\_SRAM\_READ\_IMM command.

The layout of the MCIP\_READBACK register, see [Figure E-11](#)

### E.18.4 Command Usage: Read data in immediate mode

For a core wants to receive from other cores through the inter-core message passing unit, it should first send the SRAM read command to the MCIP\_CMD register by a SR instruction, and then use a LR instruction to read the data from MCIP\_READBACK register.

The address of the SRAM can come from the immediate value accompany with CMD\_MSG\_SRAM\_READ\_IMM command.

Example as below,

```
SR 0x00000082A, [MCIP_CMD]; // parameter[11:0] is 0x008, indicate the address of SRAM is 0x008
                        // for current SRAM operation. Send CMD_MSG_SRAM_READ_IMM
                        // command to the ARConnect, and the data will be read out from
SRAM
                        // address 0x008.

LR r1, [MCIP_READBACK]; // the data from SRAM address 0x008 is returned to MCIP_READBACK
                        // register, and sent to r1 of the core.
```

## E.19 CMD\_DEBUG\_RESET

Each core can use the CMD\_DEBUG\_RESET command to generate a reset request to ARConnect or to generate reset request to core(s) in the system. If the command is to generate reset to the core, an internal MCD\_CORE register in the multi-core debug unit is used to determine which core(s) in the system are selected to be reset.

PARAMETER field accompany with this command is used to distinguish whether to generate a reset request to ARConnect or to the core.

The MCIP\_WDATA register contains the data to be written into the internal MCD\_CORE register

### E.19.1 Parameter Format

N/A

### E.19.2 MCIP\_WDATA Register Format

The MCIP\_WDATA register must contain the data to update the internal MCD\_CORE register before the CMD\_DEBUG\_RESET command.

For a 4-core system, the layout of the MCIP\_WDATA register is shown [Figure E-12](#)

**Figure E-12 MCIP\_READBACK Register Format for CMD\_DEBUG\_RESET Command**

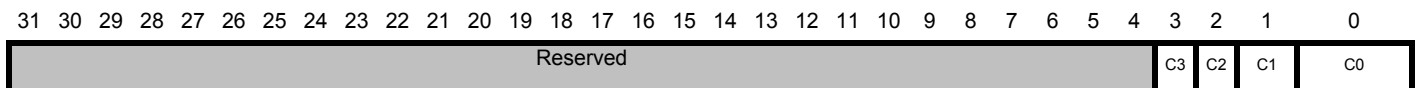


**Table E-5 MCIP\_READBACK Field Descriptions for CMD\_DEBUG\_RESET Command**

| Field | Bits | Description                                                                                                       |
|-------|------|-------------------------------------------------------------------------------------------------------------------|
| C0    | [0]  | CORE0 select.<br>Indicate CORE0 is selected for the current command<br>0 = Not selected<br>1 = CORE0 is selected. |
| C1    | [1]  | CORE1 select.<br>Indicate CORE1 is selected for the current command<br>0 = Not selected<br>1 = CORE1 is selected. |
| C2    | [2]  | CORE2 select.<br>Indicate CORE2 is selected for the current command<br>0 = Not selected<br>1 = CORE2 is selected. |

| Field | Bits | Description                                                                                                       |
|-------|------|-------------------------------------------------------------------------------------------------------------------|
| C3    | [3]  | CORE3 select.<br>Indicate CORE3 is selected for the current command<br>0 = Not selected<br>1 = CORE3 is selected. |

### E.19.3 MCIP\_READBACK Register Format

N/A

### E.19.4 Command Usage: Selective Reset Cores

Example as below, assume the CORE ID is 3,

```
SR r1, [MCIP_WDATA]; // r1 value is 0x00000003, indicates CORE3 want to reset CORE0 and CORE1
                        // for the following CMD_DEBUG_RESET command. This value will be stored
                        // into the internal MCD_CORE register in the multi-core debug sub-unit
SR 0x00000031, [MCIP_CMD]; // send CMD_DEBUG_RESET command to ARConnect
```

## E.20 CMD\_DEBUG\_RESET

Each core can use the CMD\_DEBUG\_HALT command to generate halt request to core(s) in the system. The internal MCD\_CORE register in the multi-core debug unit is used to determine which core(s) in the system are selected to be halted.

The MCIP\_WDATA register contains the data to be written into the internal MCD\_CORE register.

### E.20.1 Parameter Format

N/A

### E.20.2 MCIP\_WDATA Register Format

For a 4-core system, the layout of the MCIP\_WDATA register for CMD\_DEBUG\_HALT command can refer to [Figure E-12](#)

### E.20.3 MCIP\_READBACK Register Format

N/A

### E.20.4 Command Usage: Selective Halt Cores

Example as below, assume current CORE ID is CORE3,

```
SR r1, [MCIP_WDATA]; // r1 value is 0x00000007, indicates CORE3 want to halt CORE0~2 for the
                        // following CMD_DEBUG_HALT command. This value will be stored into the
```

```
// internal MCD_CORE register in the multi-core debug sub-unit.
SR 0x00000032, [MCIP_CMD]; // send CMD_DEBUG_HALT command to the ARConnect .
```

## E.21 CMD\_DEBUG\_RUN

Each core can use the CMD\_DEBUG\_RUN command to generate run request to core(s) in the system. The internal MCD\_CORE register in the multi-core debug unit is used to determine which core(s) in the system are selected to be halted.

The MCIP\_WDATA register contains the data to be written into the internal MCD\_CORE register.

### E.21.1 Parameter Format

N/A

### E.21.2 MCIP\_WDATA Register Format

For a 4-core system, the layout of the MCIP\_WDATA register for CMD\_DEBUG\_RUN command can refer to [Figure E-12](#)

### E.21.3 MCIP\_READBACK Register Format

N/A

### E.21.4 Command Usage: Selective Run Cores

Example as below, assume current CORE ID is CORE3,

```
SR r1, [MCIP_WDATA]; // r1 value is 0x00000009, indicates CORE3 want to restart CORE0 and CORE3
// for the following CMD_DEBUG_RUN command. This value will be stored into
// the internal MCD_CORE register in the multi-core debug unit.
SR 0x00000033, [MCIP_CMD]; // send CMD_DEBUG_RUN command to the ARConnect .
```

## E.22 CMD\_DEBUG\_SET\_MASK

Each core can use the CMD\_DEBUG\_SET\_MASK command to set the internal MASK register(s) in the multi-core debug unit and shield the halt events (e.g. actionpoint halt, self-halt, breakpoint halt) from each core to generate a global halt pulse.

PARAMTER field is used to specify the mask bits.

Each core is assigned with a MASK register. The internal MCD\_CORE register in the multi-core debug unit is used to determine which core(s) in the system are selected to be updated its MASK register.

The MCIP\_WDATA register contains the data to be written into the internal MCD\_CORE register

### E.22.1 Parameter Format

Parameter[3:0] specifies the mask bits:

- Parameter[0] defines whether a core halt status (whenever the STATUS32[H] is set) is masked or not
  - 0: masked, no global halt triggered when core is halted
  - 1: No masked, whenever the core is halted, a global halt is triggered
- Parameter[1] defines whether an actionpoint caused halt (if actionpoint option is enabled) is masked or not
  - 0: masked, no global halt triggered when core is halted by an actionpoint
  - 1: No masked, if an actionpoint caused halt occurs, a global halt is triggered
- Parameter[2] defines whether a breakpoint caused halt is masked or not
  - 0: masked, no global halt triggered when core is halted by a breakpoint
  - 1: No masked, if a breakpoint caused halt occurs, a global halt is triggered
- Parameter[3] defines whether a self-halt is masked or not
  - 0: masked, no global halt triggered when core is halted by a self-halt
  - 1: No masked, if a self-halt occurs, a global halt is triggered

### E.22.2 MCIP\_WDATA Register Format

For a 4-core system, the layout of the MCIP\_WDATA register for CMD\_DEBUG\_SET\_MASK command can refer to [Figure E-12](#)

### E.22.3 MCIP\_READBACK Register Format

N/A

### E.22.4 Command Usage:

The usage of CMD\_DEBUG\_SET\_MASK, see [E.23.3](#)

## E.23 CMD\_DEBUG\_READ\_MASK

Each core can use CMD\_DEBUG\_READ\_MASK command to read the internal MASK register belongs to it.

The internal MCD\_CORE register in the multi-core debug unit is used to determine which core in the system is selected read its MASK register.

The MCIP\_WDATA register contains the data to be written into the internal MCD\_CORE register.

The MCIP\_READBACK register returns the value of internal MASK register of the specified core.

### E.23.1 Parameter Format

N/A

### E.23.2 MCIP\_WDATA Register Format

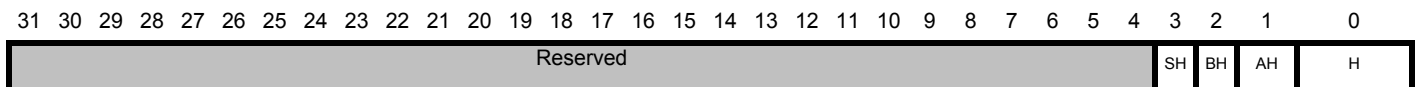
For a 4-core system, the layout of the MCIP\_WDATA register for CMD\_DEBUG\_SET\_MASK command is shown [Figure E-12](#).

**NOTE:** The CMD\_DEBUG\_READ\_MASK command support only select one single core to read its internal MASK register at one time. That is, only one bit in the MCIP\_WDATA accompany with this command can be valid.

### E.23.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns the content of internal MASK register of the specified core after CMD\_DEBUG\_READ\_MASK command, as shown in

**Figure E-13 MCIP\_READBACK Register Format for CMD\_DEBUG\_READ\_MASK Command**



Where each bit indicates core halted (H), actionpoint halt (AH), software breakpoint halt (BH), and self halt (SH) events will trigger a global halt.

**NOTE:** H bit is corresponding to the H field of the STATUS32 register of the core. And it's the complement of core's EN port.

### E.23.4 Command Usage: Read Mask Register of Specific Core

The internal MASK register belongs to each core in the multi-core debug unit can be read out by the core. This is done via CMD\_DEBUG\_READ\_MASK command used in conjunction with MCIP\_WDATA register. MCIP\_READBACK register returns the value of the MASK register.

Example as below,

```
SR r1, [MCIP_WDATA]; // the value of r1 is 0x00000001, indicates the CORE1 is selected for the
                      // following debug command
SR r2, [MCIP_CMD]; // the value of r2 is 0x00000035, send CMD_DEBUG_READ_MASK command
                      // to the ARConnect
LR r10, [MCIP_READBACK]; // the value of MASK register for CORE1 returns to MCIP_READBACK,
                          // and then store to r10
```

## E.24 CMD\_DEBUG\_SET\_SELECT

Each core can use the CMD\_DEBUG\_SET\_SELECT command to generate a halt request to the core(s) in the system when a global halt pulse in the multi-core debug unit is triggered.

The internal MCD\_CORE register in the multi-core debug unit is used to determine which core(s) in the system are selected to be generated the halt request(s).

If one core wants to halt any other core(s) in the system triggered by core's halt events, it's recommended that the CMD\_DEBUG\_SET\_SELECT command is used before the CMD\_DEBUG\_SET\_MASK command, so that no global halt pulse will be skipped.



The MCIP\_WDATA register contains the data to be written into the internal MCD\_CORE register.

### E.24.1 Parameter Format

N/A

### E.24.2 MCIP\_WDATA Register Format

For a 4-core system, the layout of the MCIP\_WDATA register for CMD\_DEBUG\_SET\_SELECT command is shown in [Figure E-12](#).

### E.24.3 MCIP\_READBACK Register Format

N/A

### E.24.4 Command Usage: Read Mask Register of Specific Core

Example as below, assume current CORE ID is CORE3,

```
SR 0x0000000f, [MCIP_WDATA]; // all cores in the 4-core system are selected for the following
                                // debug commands
SR 0x00000036, [MCIP_CMD]; // send CMD_DEBUG_SET_SELECT command to the ARConnect
                                // unit, all the cores selected by previous SR instruction will be halted
                                // by the global halt pulse.
SR r1, [MCIP_WDATA]; // the value of r1 is 0x00000007, indicates CORE0, CORE1, and CORE2 are
                                // selected for following debug commands
SR 0x00000f34, [MCIP_CMD]; // send CMD_DEBUG_SET_MASK command to the ARConnect,
                                // parameter[3:0] = 4'b1111 indicates any kind of halt events (H,AH,BH
                                // and SH) will generate a global halt pulse
```

## E.25 CMD\_DEBUG\_READ\_SELECT

Each core can use CMD\_DEBUG\_READ\_SELECT command to read the internal SELECT register in the multi-core debug unit.

The SELECT register is a common register for all cores, so there is no need to set any value in the MCIP\_WDATA register for core selection.

The MCIP\_READBACK register returns the value of internal SELECT register

### E.25.1 Parameter Format

N/A

### E.25.2 MCIP\_WDATA Register Format

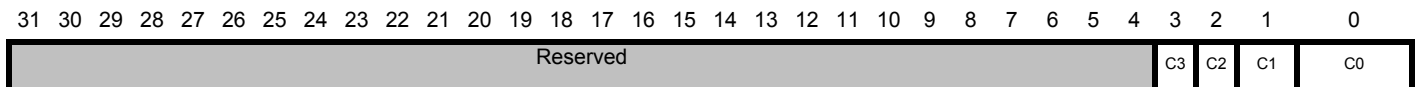
N/A

### E.25.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register returns the contents of internal SELECT register set by a previous CMD\_DEBUG\_SET\_SELECT command.

Take 4-core configuration as the example, the register layout is shown in [Figure E-14](#)

**Figure E-14 MCIP\_READBACK Register Format for CMD\_DEBUG\_READ\_SELECT Command**



### E.25.4 Command Usage: Read Select Register Value

Example as below,

```
SR r5, [MCIP_CMD]; // the value of r5 is 0x00000037, send CMD_DEBUG_READ_SELECT command
                    // to the ARConnect
LR r15, [MCIP_READBACK]; // the value of SELECT register returns to MCIP_READBACK, and
                          // then read out to r15
```

## E.26 CMD\_DEBUG\_READ\_EN

Each core can use the CMD\_DEBUG\_READ\_EN command to read the status (halt or run) of all cores in the system.

The MCIP\_READBACK register returns a vector data; each bit corresponding to the status of one core.

### E.26.1 Parameter Format

N/A

### E.26.2 MCIP\_WDATA Register Format

N/A

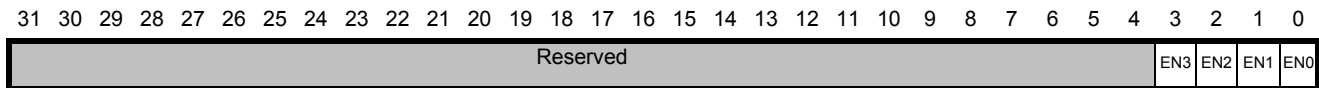
### E.26.3 MCIP\_READBACK Register Format

The CMD\_DEBUG\_READ\_EN command returns the current state of the EN bits (the complement of H bit in STATUS32 register) of all the cores in the system to the MCIP\_READBACK register.

For a 4-core configuration, the register layout is shown in [Figure E-15](#).

A bit value of 1 indicates that the corresponding core is running; a bit value of 0 indicates the corresponding core is halted.

**Figure E-15 MCIP\_READBACK Register for CMD\_DEBUG\_READ\_EN**



The field descriptions are shown in [Table E-6](#).

**Table E-6 MCIP\_READBACK Register Field Descriptions for CMD\_DEBUG\_READ\_EN Command**

| Field | Bits  | Description                                                                   |
|-------|-------|-------------------------------------------------------------------------------|
| EN[n] | [3:0] | 1: indicates that the core is running<br>0: indicates that the core is halted |

#### E.26.4 Command Usage: Read EN status of all cores

```
SR r5, [MCIP_CMD]; // the value of r5 is 0x00000038; send CMD_DEBUG_READ_EN command to
// ARConnect
```

```
LR r10, [MCIP_READBACK]; // the EN bits return to the MCIP_READBACK register and store to r10
```

### E.27 CMD\_DEBUG\_READ\_CMD

Each core can use the CMD\_DEBUG\_READ\_CMD command to check the last command sent to the multi-core debug unit.

An internal MCD\_CMD register in the multi-core debug unit records the last command and the parameter data (if needed) it received.

The MCIP\_READBACK register reads out the value of this internal MCD\_CMD register.

#### E.27.1 Parameter Format

N/A

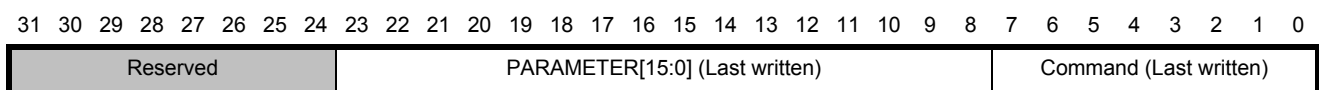
#### E.27.2 MCIP\_WDATA Register Format

N/A

#### E.27.3 MCIP\_READBACK Register Format

The CMD\_DEBUG\_READ\_CMD command returns the last-written contents of the internal MCD\_CMD register to the MCIP\_READBACK register, as shown in [Figure E-16](#).

**Figure E-16 MCIP\_READBACK Register for CMD\_DEBUG\_READ\_CMD**



### E.27.4 Command Usage: Read last command the multi-core debug unit received

```
SR r5, 0x00000039; // send CMD_DEBUG_READ_CMD command to ARConnect
LR r7, [MCIP_READBACK]; // last written command and its PARAMETER field
// returns to MCIP_READBACK and store to r7 of the core
```

## E.28 CMD\_DEBUG\_READ\_CORE

Each core or the debugger can use the CMD\_DEBUG\_READ\_CORE command to read the value of internal MCD\_CORE register in the multi-core debug unit.

The MCIP\_READBACK register returns the read out data.

### E.28.1 Parameter Format

N/A

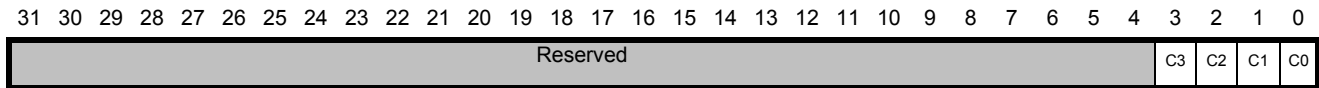
### E.28.2 MCIP\_WDATA Register Format

N/A

### E.28.3 MCIP\_READBACK Register Format

The CMD\_DEBUG\_READ\_CORE command returns the last-written contents of the MCD\_CORE register to the MCIP\_READBACK register. For a 4-core configuration, the layout of MCIP\_READBACK register is shown in Figure E-17.

**Figure E-17 MCIP\_READBACK Register for CMD\_DEBUG\_READ\_CORE**



The field descriptions are shown in Table E-7.

**Table E-7 MCIP\_READBACK Register Field Descriptions for CMD\_DEBUG\_READ\_CORE Command**

| Field | Bits | Description                                                                                                                 |
|-------|------|-----------------------------------------------------------------------------------------------------------------------------|
| C0    | [0]  | CORE0 select<br>Indicates that the CORE0 is selected for debug related command<br>0 = Not selected<br>1 = CORE0 is selected |
| C1    | [1]  | CORE1 select<br>Indicates that the CORE1 is selected for debug related command<br>0 = Not selected<br>1 = CORE1 is selected |
| C2    | [2]  | CORE2 select<br>Indicates that the CORE2 is selected for debug related command<br>0 = Not selected<br>1 = CORE2 is selected |
| C3    | [3]  | CORE3 select<br>Indicates that the CORE3 is selected for debug related command<br>0 = Not selected<br>1 = CORE3 is selected |

### E.28.4 Command Usage: Read MCD\_CORE register value in the multi-core debug unit

```
SR 0x0000003A, [MCIP_CMD]; // send CMD_DEBUG_READ_CORE command to the ARConnect
LR r9, [MCIP_READBACK]; // the value of MCD_CORE register at last written returns to
                        // MCIP_READBACK, and store to r9 of the core
```

## E.29 CMD\_GRTC\_CLEAR

The global 64-bit Real-Time Counter (RTC) is supported in ARConnect to enable the software to synchronize at a shared time under multiple independent clock domain system.

Each core can use the CMD\_GRTC\_CLEAR command to clear the global 64-bit RTC.

**E.29.1 Parameter Format**

N/A

**E.29.2 MCIP\_WDATA Register Format**

N/A

**E.29.3 MCIP\_READBACK Register Format**

N/A

**E.29.4 Command Usage: Clear the global RTC**

SR r13, [MCIP\_CMD]; // the value of r13 is 0x00000041, indicates the CMD\_CLEAR\_GRTC command  
 // set and will be sent to the ARConnect to clear the value of global RTC Timer.

**E.30 CMD\_GRTC\_READ\_LO**

Each core can use the CMD\_GRTC\_READ\_LO command to read the lower 32-bit of global RTC counter.

The MCIP\_READBACK register returns the lower 32-bit data.

**E.30.1 Parameter Format**

N/A

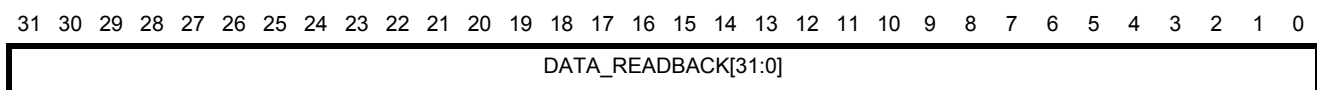
**E.30.2 MCIP\_WDATA Register Format**

N/A

**E.30.3 MCIP\_READBACK Register Format**

The MCIP\_READBACK register layout is shown in [Figure E-18](#).

**Figure E-18 MCIP\_READBACK Register for CMD\_GRTC\_READ\_LO**

**E.30.4 Command Usage:**

See [“Command Usage: Read the global RTC”](#) on page 415.

**E.31 CMD\_GRTC\_READ\_HI**

Each core can use the CMD\_GRTC\_READ\_HI command to read the higher 32-bit of global RTC counter.

The MCIP\_READBACK register returns the higher 32-bit data.

### E.31.1 Parameter Format

N/A

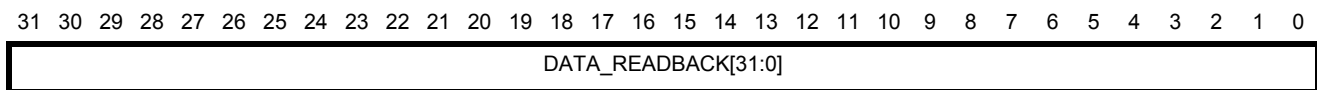
### E.31.2 MCIP\_WDATA Register Format

N/A

### E.31.3 MCIP\_READBACK Register Format

The MCIP\_READBACK register layout is shown in [Figure E-19](#).

**Figure E-19 MCIP\_READBACK Register for CMD\_GRTC\_READ\_HI**



### E.31.4 Command Usage: Read the global RTC

SR 0x00000042, [MCIP\_CMD]; // send CMD\_GRTC\_READ\_LO command to the ARConnect unit

LR r5, [MCIP\_READBACK]; // the lower 32-bit value of global RTC returns to MCIP\_READBACK and //store to r5 of the core

SR 0x00000043, [MCIP\_CMD]; // send CMD\_GRTC\_READ\_HI command to the ARConnect unit

LR r6, [MCIP\_READBACK]; // the higher 32-bit value of global RTC returns to MCIP\_READBACK and //store to r6 of the core

## E.32 CMD\_PMU\_SET\_PMODE

Each core can use the CMD\_PMU\_SET\_PMODE command to set the power mode of any combination of the cores in the system.

The PARAMTER field is used to specify the power mode.

The MCIP\_WDATA register contains the specific power mode.

### E.32.1 Parameter Format

Parameter [5:0] indicates the MP domain, RTT domain, or a core

- 6'd0 ~ 31: to core domain, and MCIP\_WDATA used to set PM mode. 6'd0 indicate Core 0 to 6'd31 indicate Core 31.
- 6'd32: to MP domain
- 6'd34: to RTT domain

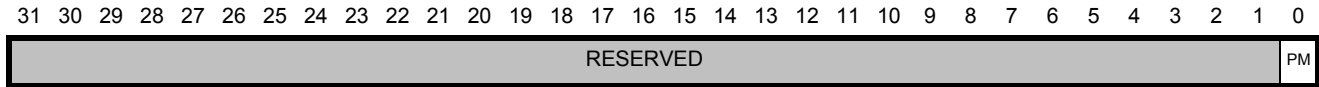
Indexing to the core itself is meaningless and is ignored. For example, if the CORE ID of current core is 3, setting the parameter field equals 3 (Parameter [5:0] =3) accompany with this command is meaningless and core 3 will not be powered down.

When either PM1 or PM2 mode is entered, no transition between those two modes is allowed. For example, if CORE1 has entered into PM1 mode, it cannot be programmed into PM2 mode before it is powered up.

### E.32.2 MCIP\_WDATA Register Format

The MCIP\_WDATA register is used to specify the power mode. The layout of the MCIP\_WDATA register for a four-core system is shown in [Figure E-20](#).

**Figure E-20 MCIP\_WDATA Register for CMD\_PMU\_SET\_PMODE**



The field descriptions are shown in [Table E-8](#).

**Table E-8 MCIP\_WDATA Register Field Descriptions for CMD\_PMU\_SET\_PMODE Command**

| Field | Bits | Description                                                                              |
|-------|------|------------------------------------------------------------------------------------------|
| PMode | [0]  | Indicate power mode<br>2'h0: Power on mode (default)<br>2'h1: PM1 mode<br>2'h2: PM2 mode |

### E.32.3 MCIP\_READBACK Register Format

N/A

### E.32.4 Command Usage: Set Power Mode of Core

The following code causes core1 to enter into the PM1 mode.

```
SR 0x00000001, [MCIP_WDATA]; // set core selection to MCIP_WDATA register, it indicates PM1.
```

```
SR r1, [MCIP_CMD]; // the value of r1 is 0x00000051, Core 1 enter into PM1 mode
```

Your software needs to check current power status of the cores before it issues the CMD\_PMU\_SET\_PMODE command. The PMU ignores the command to power down the core if it has already been in power down mode.

## E.33 CMD\_PMU\_READ\_PSTATUS

Each core can use the CMD\_PMU\_READ\_PSTATUS command to read the internal PSTATUS register of the specified core.

Each core is assigned a PSTATUS register to record its current power mode, previous power mode, power up cause, power down cause, etc...

The MCIP\_READBACK register returns the data read out from the PSTATUS register of the specified core.

### E.33.1 Parameter Format

Parameter [5:0] indicates: the MP domain, a core, or the RTT domain.

- 6'd0 ~ 31: to Core domain. 6'd0 indicate Core 1 ... 6'd31 indicate Core 32.
- 6'd32: to MP domain
- 6'd34: to RTT domain



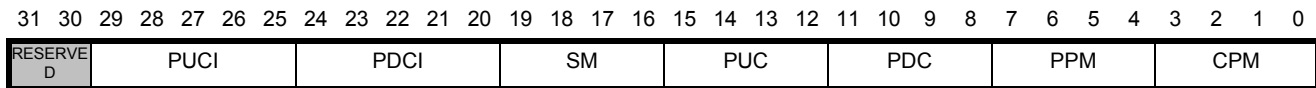
### E.33.2 MCIP\_WDATA Register Format

N/A

### E.33.3 MCIP\_READBACK Register Format

The MCIP\_READBACK returns power management related information of the specified core as shown in [Figure E-21](#).

**Figure E-21 MCIP\_READBACK Register for CMD\_PMU\_READ\_PSTATUS**



The field descriptions are shown in [Table E-9](#).

**Table E-9 MCIP\_READBACK Register Field Descriptions for CMD\_PMU\_READ\_PSTATUS Command**

| Field | Bits    | Description                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                                     |
|-------|---------|-----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| CPM   | [3:0]   | Current power mode of the core.<br>4'h0: Power on mode (default)<br>4'h1: PM1 mode<br>4'h2: PM2 mode                                                                                                                                                                                                                                                                                                                                                                                                                                                            |
| PPM   | [7:4]   | Previous power mode of the core.<br>4'h0: Power on mode (default)<br>4'h1: PM1 mode<br>4'h2: PM2 mode                                                                                                                                                                                                                                                                                                                                                                                                                                                           |
| PDC   | [11:8]  | This field is only valid for processor core.<br>Power down cause.<br>4'h0: SLEEP instruction (default)<br>4'h1: PMU status register set via core (by either core or debugger)                                                                                                                                                                                                                                                                                                                                                                                   |
| PUC   | [15:12] | This field is only valid for processor core.<br>Power up cause.<br>4'h0: External reset (default)<br>4'h1: PMU status register set via core (by either core or debugger)<br>4'h3: Timer 0 Interrupt<br>4'h4: Timer 1 Interrupt<br>4'h5: Other Interrupts (including inter-core interrupts, external interrupts)<br>4'h6: debugger set DEBUGP register<br>4'h7: Watchdog reset (from any timer, including timer2 watchdog)<br>4'h8: external run or selective run request<br>4'h9: selective reset generated by multi-core debug unit<br>4'ha: Timer 2 Interrupt |
| SM    | [19:16] | This field is only valid for processor core.<br>Sleep mode of core (sys_sleep_mode_r signal)                                                                                                                                                                                                                                                                                                                                                                                                                                                                    |

| Field | Bits    | Description                                                                                    |
|-------|---------|------------------------------------------------------------------------------------------------|
| PDCI  | [24:20] | This field is only valid for processor core.<br>CORE ID (from 0 to N-1) that causes power down |
| PUCI  | [29:25] | This field is only valid for processor core.<br>CORE ID (from 0 to N-1) that causes power up   |

### E.33.4 Command Usage: Read PMU status register of Specified core

Example shows as below, read PMU status register of the core1.

```
SR r5, [MCIP_CMD]; // the value of r5 is 0x00000052, it programs reading core 1 PMU status register
```

```
LR r6, [MCIP_READBACK]; // core1 PMU status register value will be loaded to r6 register
```

## E.34 CMD\_PMU\_SET\_PCNT

An internal power-up duration counter PCNT is implemented in the power management unit to control the applied reset cycles after each domain exit from power down mode. Each core can use the CMD\_PMU\_SET\_PCNT command to set the PCNT.

The reset value of PCNT is 1. And if the PCNT is set to N, reset will apply (N+1) cycles after each domain exit from power down mode.

The MCIP\_WDATA register is used to set the value of this power-up duration counter.

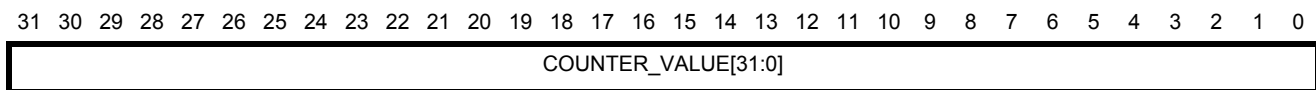
### E.34.1 Parameter Format

N/A

### E.34.2 MCIP\_WDATA Register Format

The MCIP\_WDATA layout is shown in [Figure E-22](#).

**Figure E-22 MCIP\_WDATA Register for CMD\_PMU\_SET\_PCNT**



### E.34.3 MCIP\_READBACK Register Format

N/A

### E.34.4 Command Usage: Set Power-up duration time

```
SR 0x00006053, [MCIP_WDATA]; // set the value of power-up duration time
```

```
SR r2, [MCIP_CMD]; // send the CMD_PMU_SET_PCNT command to ARConnect.
```

## E.35 CMD\_PMU\_READ\_PCNT

Each core can use the CMD\_PMU\_READ\_PCNT command to read the value of power-up duration counter in the PMU unit.

The MCIP\_READBACK register returns the content of power-up duration register.

### E.35.1 Parameter Format

N/A

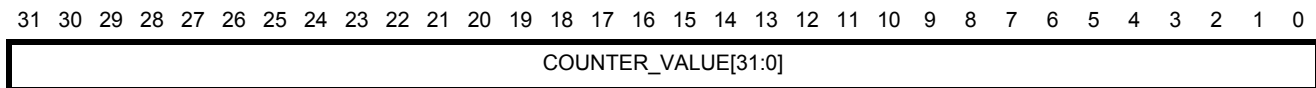
### E.35.2 MCIP\_WDATA Register Format

N/A

### E.35.3 MCIP\_READBACK Register Format

The MCIP\_READBACK layout is shown in [Figure E-23](#).

**Figure E-23 MCIP\_READBACK Register for CMD\_PMU\_READ\_PCNT**



### E.35.4 Command Usage: Read Power-up duration counter value

```
SR r2, [MCIP_CMD]; // the value of r2 is 0x00000054, it sends CMD_PMU_READ_PCNT command to read
                    // power-up duration counter value.
```

```
LR r4, [MCIP_READBACK]; // data returned to the MCIP_READBACK register is then stored to r4.
```

## E.36 CMD\_PMU\_READ\_DVFS

Each core can use the CMD\_PMU\_READ\_DVFS command to read the internal DVFS\_STATUS register

The MCIP\_READBACK register returns the data read back from DVFS\_STATUS register.

### E.36.1 Parameter Format

N/A

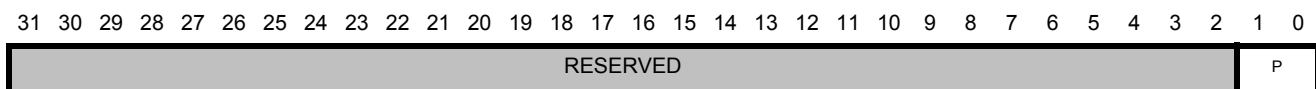
### E.36.2 MCIP\_WDATA Register Format

N/A

### E.36.3 MCIP\_READBACK Register Format

The MCIP\_READBACK layout is shown in [Figure E-24](#).

**Figure E-24 MCIP\_READBACK Register for CMD\_PMU\_READ\_DVFS**



The field descriptions are shown in [Table E-10](#).

**Table E-10 MCIP\_READBACK Register Field Descriptions for CMD\_PMU\_READ\_DVFS Command**

| Field | Bits  | Description                                                                                                                                                                                     |
|-------|-------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| P     | [1:0] | Performance parameter.<br>2'h0: Performance Level 0 (Highest performance).<br>2'h1: Performance Level 1.<br>2'h2: Performance Level 2.<br>2'h3: Performance Level 3 (Lowest power consumption). |

#### E.36.4 Command Usage: Read performance level for specified core

```
SR 0x00000001, [MCIP_WDATA]; // select core 1.
```

```
SR r1, [MCIP_CMD]; // the value of r1 is 0x00000056, it sends CMD_PMU_READ_DVFS command to read  
//performance level value.
```

```
LR r3, [MCIP_READBACK]; // performance level value of core 1 returned to MCIP_READBACK register and then  
//stored into r3.
```

# Glossary

|                        |                                                                                                                                                                                                                                                          |
|------------------------|----------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| active command queue   | Command queue from which a model is currently taking commands; see also command queue.                                                                                                                                                                   |
| application design     | Overall chip-level design into which a subsystem or subsystems are integrated.                                                                                                                                                                           |
| ARCompact              | Instruction-set architecture used in the ARCtangent-A5, ARC 600, and ARC 700 processor series.                                                                                                                                                           |
| BFM                    | Bus-Functional Model — A simulation model used for early hardware debug. A BFM simulates the bus cycles of a device and models device pins, as well as certain on-chip functions. See also Full-Functional Model.                                        |
| big-endian             | Data format in which most significant byte comes first; normal order of bytes in a word.                                                                                                                                                                 |
| blocked command stream | A command stream that is blocked due to a blocking command issued to that stream; see also command stream, blocking command, and non-blocking command.                                                                                                   |
| blocking command       | A command that prevents a testbench from advancing to next testbench statement until this command executes in model. Blocking commands typically return data to the testbench from the model.                                                            |
| bubble                 | Empty cycle in the instruction pipeline between the dispatch of two instructions during which no instruction commits.                                                                                                                                    |
| command channel        | Manages command streams. Models with multiple command channels execute command streams independently of each other to provide full-duplex mode function.                                                                                                 |
| command stream         | The communication channel between the testbench and the model.                                                                                                                                                                                           |
| component              | A generic term that can refer to any synthesizable IP or verification IP in the DesignWare Library. In the context of synthesizable IP, this is a configurable block that can be instantiated as a single entity (VHDL) or module (Verilog) in a design. |

|                      |                                                                                                                                                                                                                                                            |
|----------------------|------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| configuration        | The act of specifying parameters for a core prior to synthesis; can also be used in the context of VIP.                                                                                                                                                    |
| configuration intent | Range of values allowed for each parameter associated with a reusable core.                                                                                                                                                                                |
| cycle command        | A command that executes and causes HDL simulation time to advance.                                                                                                                                                                                         |
| decoder              | Software or hardware subsystem that translates from an “encoded” format back to standard format.                                                                                                                                                           |
| design context       | Aspects of a component or subsystem target environment that affect the synthesis of the component or subsystem.                                                                                                                                            |
| design creation      | The process of capturing a design as parameterized RTL.                                                                                                                                                                                                    |
| direct-write         | Method for allowing extension instructions to update registers internally, even in situations that would not normally be allowed.                                                                                                                          |
| dual role device     | Device having the capabilities of function and host (limited).                                                                                                                                                                                             |
| endian               | Ordering of bytes in a multi-byte word; see also little-endian and big-endian.                                                                                                                                                                             |
| Full-Functional Mode | A simulation model that describes the complete range of device behavior, including code execution. See also BFM.                                                                                                                                           |
| GPIO                 | General Purpose Input Output.                                                                                                                                                                                                                              |
| hard IP              | Non-synthesizable implementation IP.                                                                                                                                                                                                                       |
| HDL                  | Hardware Description Language – examples include Verilog and VHDL.                                                                                                                                                                                         |
| instantiate          | The act of placing a core or model into a design.                                                                                                                                                                                                          |
| interface            | Set of ports and parameters that defines a connection point to a component.                                                                                                                                                                                |
| IP                   | Intellectual property — A term that encompasses simulation models and synthesizable blocks of HDL code.                                                                                                                                                    |
| little-endian        | Data format in which the least-significant byte comes first.                                                                                                                                                                                               |
| master               | Device or model that initiates and controls another device or peripheral.                                                                                                                                                                                  |
| master actionpoint   | Actionpoint that has the pair-mode bit set when paired actionpoints are used.                                                                                                                                                                              |
| monitor              | A device or model that gathers performance statistics of a system.                                                                                                                                                                                         |
| non-blocking command | A testbench command that advances to the next testbench statement without waiting for the command to complete.                                                                                                                                             |
| peripheral           | Generally refers to a small core that has a bus connection, specifically an APB interface.                                                                                                                                                                 |
| RAW hazard           | Read-After-Write hazard. Danger of unexpected results due to timing and ordering of read/write instructions. For example, using a non-blocking register in the source field of an instruction following an instruction that direct-writes to the register. |

|                                        |                                                                                                                                                                                                       |
|----------------------------------------|-------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------------|
| RAZ/IOW                                | Read As Zero/Ignore On Write: The typical behavior of reserved register fields.                                                                                                                       |
| RTL                                    | Register Transfer Level. A higher level of abstraction that implies a certain gate-level structure. Synthesis of RTL code yields a gate-level design.                                                 |
| SDRAM                                  | Synchronous Dynamic Random Access Memory; high-speed DRAM adds a separate clock signal to control signals.                                                                                            |
| SDRAM controller                       | A memory controller with specific connections for SDRAMs.                                                                                                                                             |
| slave                                  | Device or model that is controlled by and responds to a master.                                                                                                                                       |
| SoC                                    | System on a chip.                                                                                                                                                                                     |
| soft IP                                | Any implementation IP that is configurable. Generally referred to as synthesizable IP.                                                                                                                |
| static controller                      | Memory controller with specific connections for Static memories such as asynchronous SRAMs, Flash memory, and ROMs.                                                                                   |
| synthesis intent                       | Attributes that a core developer applies to a top-level design, ports, and core.                                                                                                                      |
| synthesizable IP                       | A type of Implementation IP that can be mapped to a target technology through synthesis. Sometimes referred to as Soft IP.                                                                            |
| TAP                                    | Test Access Port: Another name for the JTAG port used for debugging and host interaction.                                                                                                             |
| technology-independent                 | Design that allows the technology (that is, the library that implements the gate and via widths for gates) to be specified later during synthesis.                                                    |
| Testsuite Regression Environment (TRE) | A collection of files for standalone verification of the configured component. The files, tests, and functionality vary from component to component.                                                  |
| VIP                                    | Verification Intellectual Property — A generic term for a simulation model in any form.                                                                                                               |
| WAW hazard                             | Write-After-Write hazard. Danger of unexpected results due to timing and ordering of write instructions. For example, one multi-cycle instruction following another might not perform the final write |
| wrap, wrapper                          | Code, usually VHDL or Verilog, that surrounds a design or model, allowing easier interfacing. Usually requires an extra, sometimes automated, step to create the wrapper.                             |
| zero-cycle command                     | A command that executes without HDL simulation time advancing.                                                                                                                                        |





# Index

## A

- actionpoints [69](#), [161](#), [246](#)
- addressable code space [60](#)
- AHB [47](#), [61](#), [149](#), [151](#)
- AHB-Lite [47](#), [62](#), [82](#), [149](#), [151](#), [153](#), [156](#)
- ALU instructions [135](#)
- APEX [177](#)
- ARC 600 [339](#)
- ARC 700 [340](#)
- ARC EM4 [45](#)
- ARC EM6 [45](#)
- ARChitect [51](#)
  - GUI [52](#)
- arc\_num [60](#)
- ARCompact [340](#)
- ARCtest [335](#)
- ARCV2EM [238](#)
- assembler, MetaWare [57](#)
- associativity
  - data cache [66](#)
  - instruction cache [65](#)
- auxiliary register set
  - smart build configuration [264](#)
  - smart control [264](#)
  - smart data register [265](#)

## B

- BCRs [337](#)
- block diagram [147](#)
- boot loader [81](#)
- breakpoints [247](#)
- breakpoints in ROM [69](#)
- build configuration register
  - smart [264](#)
- BVCI [47](#), [61](#), [62](#), [82](#), [149](#), [151](#), [156](#)

## C

- CCT [332](#)

- cJTAG [227](#)

- clock [165](#), [167](#)
- clock speed [61](#)
- closely coupled memory [65](#), [75](#)
- compiler, C/C++ [56](#)
- components
  - adding [60](#)
  - configuring [60](#)
- core version [49](#)
- coreAssembler [71](#), [334](#)
- critical path [61](#)
- customer support [31](#)
- cycle timing [131](#)

## D

- data cache [84](#)
- data memory initiator [66](#), [82](#)
- debug interface [68](#), [171](#), [219](#), [243](#)
- debug reset [169](#)
- degree of associativity
  - data cache [66](#)
  - instruction cache [65](#)

- deliverables [45](#)

- DFT-Compiler [360](#)

- documentation [30](#)

## E

- EMBED-IT! [334](#)
- endian orientation [60](#)
- exceptions, timing of [144](#)
- extensions [177](#)

## F

- features [40](#)
- FPX floating point [49](#)

## G

- gate count [42](#)

## H

- halt [159](#), [307](#)

external 168  
 hardware ID 49  
**I**  
 ICCM boot loader 81  
 instruction cache 66, 103  
 instruction fetch queue 66  
 instructions  
   sequencing 132  
   timing 133, 134, 135, 137, 138, 139, 142, 144  
 instruction-set architecture 42, 63  
 intermixed 16-/32-bit instructions 40  
 interrupt 61  
   pins 61, 158  
   response time 144  
   vector table 61  
 interrupts 61, 145  
 IP library file 51  
 IP library options 343  
**J**  
 JTAG 47, 66, 158, 219  
   two-wire 227  
**L**  
 latency 133  
 licensing 57  
 limms and actionpoints 254  
 load/store 136  
 loop-counter registers 60  
 loops, zero-overhead *see* zero-overhead loops  
**M**  
 memory map 73, 83  
 memory-mapped peripherals 82  
 MetaWare debugger 337  
 MetaWare Development Toolkit 56  
 multiply options 63  
 multiprocessor configurations 67, 71  
**P**  
 PDISP 337  
 performance density 40  
 peripheral data bus 82  
 pipeline 131  
 PIPEMON 337  
 power, saving 305  
**R**  
 RAM 359  
 Reference Design Flow 359

register file 62, 63  
 reset 61, 165, 167  
   debug or watchdog 169  
 ROM patching 69, 247, 254  
 run 159  
   external 168  
**S**  
 SEIF 334, 360  
 self-modifying code 81  
 signals, status 170  
 simulation tools 56  
 sleep 306  
 SMART\_BUILD 264  
 SMART\_BUILD configuration register 264  
 SMART\_CONTROL 264  
 SMART\_DATA 265  
 standards 47  
 support, technical 31  
 SWAP instruction 63  
**T**  
 technical support 31  
 templates 45, 53  
 timer 0 68  
 timer 1 68  
 timers  
   reset signal 169  
 timing  
   cycle 131  
   exceptions 144  
 turbo mode 138  
 two-wire JTAG 227  
**U**  
 UART 375  
**V**  
 Verilog 311, 333  
 VTOC 333  
**W**  
 watchdog reset 169  
 watchpoints 247  
 waveforms 336  
**X**  
 xCAM 333, 334  
**Z**  
 zero-overhead loops 60, 131, 144