HB0761

CoreRISCV_AXI4 v2.0 Handbook

04 2017

Microsemi.
Power Matters.™

Microsemi makes no warranty, representation, or guarantee regarding the information contained herein or the suitability of its products and services for any particular purpose, nor does Microsemi assume any liability whatsoever arising out of the application or use of any product or circuit. The products sold hereunder and any other products sold by Microsemi have been subject to limited testing and should not be used in conjunction with mission-critical equipment or applications. Any performance specifications are believed to be reliable but are not verified, and Buyer must conduct and complete all performance and other testing of the products, alone and together with, or installed in, any end-products. Buyer shall not rely on any data and performance specifications or parameters provided by Microsemi. It is the Buyer's responsibility to independently determine suitability of any products and to test and verify the same. The information provided by Microsemi hereunder is provided "as is, where is" and with all faults, and the entire risk associated with such information is entirely with the Buyer. Microsemi does not grant, explicitly or implicitly, to any party any patent rights, licenses, or any other IP rights, whether with regard to such information itself or anything described by such information. Information provided in this document is proprietary to Microsemi, and Microsemi reserves the right to make any changes to the information in this document or to any products and services at any time without notice.

Microsemi Corporate Headquarters
One Enterprise, Aliso Viejo,
CA 92656 USA
Within the USA: +1 (800) 713-4113
Outside the USA: +1 (949) 380-6100
Sales: +1 (949) 380-6136
Fax: +1 (949) 215-4996
E-mail: sales.support@microsemi.com
www.microsemi.com

About Microsemi

Microsemi Corporation (Nasdaq: MSCC) offers a comprehensive portfolio of semiconductor and system solutions for aerospace & defense, communications, data center and industrial markets. Products include high-performance and radiation-hardened analog mixed-signal integrated circuits, FPGAs, SoCs and ASICs; power management products; timing and synchronization devices and precise time solutions, setting the world's standard for time; voice processing devices; RF solutions; discrete components; enterprise storage and communication solutions; security technologies and scalable anti-tamper products; Ethernet solutions; Power-over-Ethernet ICs and midspans; as well as custom design capabilities and services. Microsemi is headquartered in Aliso Viejo, Calif., and has approximately 4,800 employees globally. Learn more at www.microsemi.com.

1 Revision History

The revision history describes the changes that were implemented in the document. The changes are listed by revision, starting with the most current publication.

1.1 Revision 2.0

Updated changes related to CoreRISCV_AXI4 v2.0.

1.2 Revision 1.0

Revision 1.0 was released for CoreRISCV_AXI4 v1.0.

Contents

List of Figures

List of Tables

2 Introduction

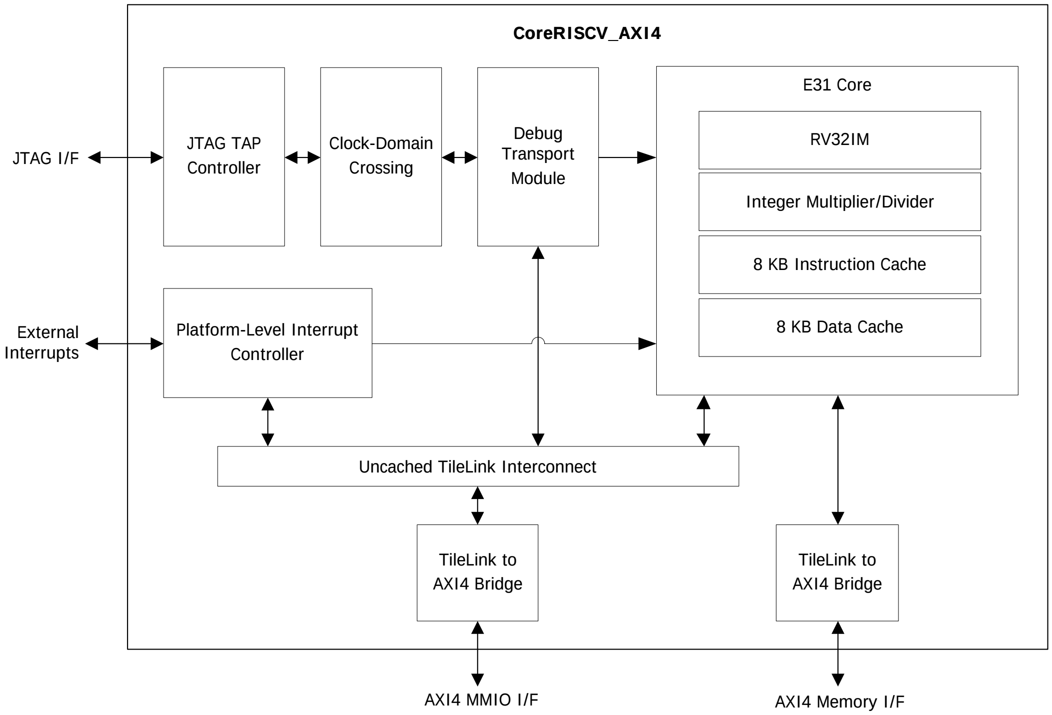
2.1 Overview

CoreRISCV_AXI4 is a softcore processor designed to implement the RISC-V instruction set for use in Microsemi FPGAs. The processor is based on the Coreplex E31 designed by SiFive, containing a high-performance single-issue, in-order execution pipeline E31 32-bit RISC-V core. The core includes an industry-standard JTAG interface to facilitate debug access, along with separate AXI4 bus interfaces for memory access and support for 31 dedicated interrupt ports.

Example Libero Designs and Firmware can be found through the following links:

- Libero Projects - https://github.com/RISCV-on-Microsemi-FPGA
- Firmware - https://github.com/RISCV-on-Microsemi-FPGA/riscv-junk-drawer/tree/master/examples

Figure 1 CoreRISCV_AXI4 Block Diagram



2.2 Features

CoreRISCV_AXI4 supports the following features:

- Designed for low power ASIC microcontroller and FPGA soft-core implementations.
- Integrated 8Kbytes instructions cache and 8 Kbytes data cache.
- A Platform-Level Interrupt Controller (PLIC) can support up to 31 programmable interrupt with a single priority level.
- Supports the RISCV standard RV32IM ISA.
- On-Chip debug unit with a JTAG interface.
- Two external AXI interfaces for IO and memory.

2.3 Core Version

This Handbook applies to CoreRISCV_AXI4 version 2.0.

Note: There are two accompanying manuals for this core:

- The RISCV Instruction Set Manual, Volume 1, User Level ISA, Version 2.1
- The RISCV Instruction Set Manual, Volume 2, Privileged Architecture, Version 1.9 (draft)

2.4 Supported Families

- PolarFire
- RTG4™
- IGLOO®2
- SmartFusion®2

2.5 Device Utilization and Performance

Utilization and performance data is listed in Table 1 for the supported device families. The data listed in this table is indicative only. The overall device utilization and performance of the core is system dependent.

Table 1 Device Utilization and Performance

Family	Sequential	Combinatorial	uSRAM	LSRAM	Math	Frequency (MHz)
SmartFusion2	5741	9608	46	8	2	99.01
IGLOO2	5741	9608	46	8	2	98.06
RTG4	5124	9282	46	8	2	68.67
PolarFire	4900	8614	67	10	2	122.40

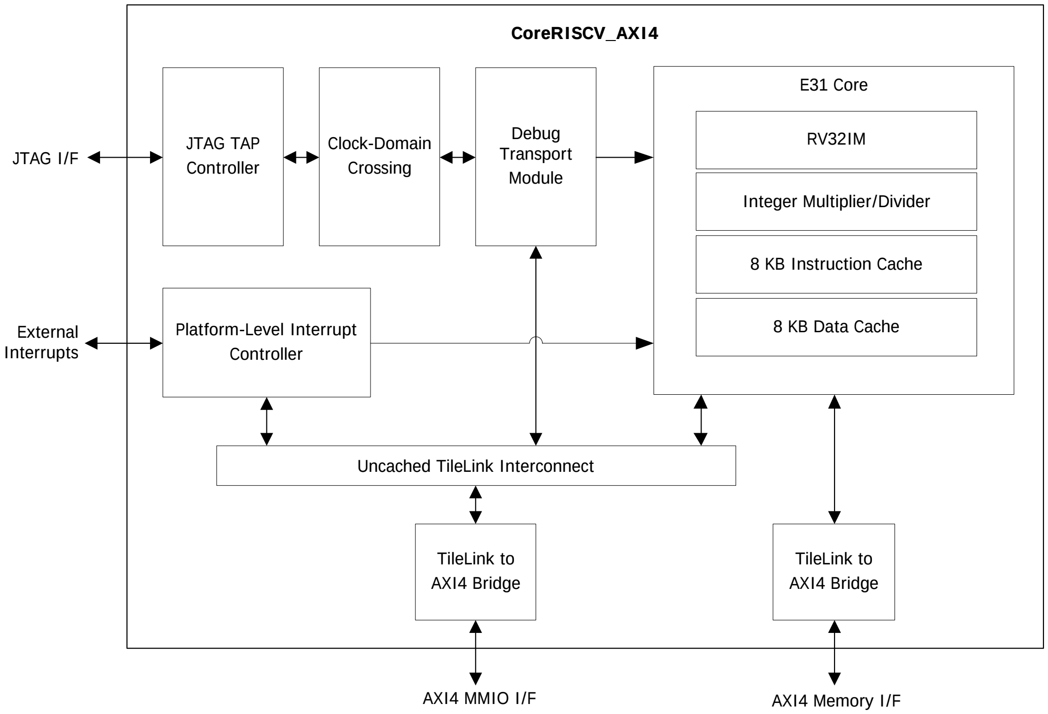
3 Functional Description

3.1 CoreRISCV_AXI4 Architecture

Table 2 CoreRISCV_AXI4 Architecture

Parameter	Value	Units	Notes
ISA Support	RV32IM		
Cores	1		
Harts/Cores	1		
Branch prediction	None		Static Not Taken
Multiplier occupancy	16	cycles	2-bit/cycles iterative multiply
I-cache size	8	KiB	
I-cache associativity	1	way	direct-mapped
I-cache line-size	64	bytes	
D-cache size	8	KiB	
D-cache associativity	1	way	direct-mapped
D-cache line-size	64	bytes	
Reset Vector	configurable		
External interrupts	31		
PLIC Interrupt priorities	1		Fixed priorities
External memory bus	AXI		
External I/O bus	AXI		
JTAG debug transport address width	5	bits	
Hardware breakpoints	2		

Figure 2 CoreRISCV_AXI4 Block Diagram



3.2 CoreRISCV_AXI4 Processor Core

CoreRISCV_AXI4 is based on the E31 Coreplex Core by SiFive. The core provides a single hardware thread (or hart) supporting the RISCV standard RV32IM ISA and machine-mode privileged architecture.

CoreRISCV_AXI4 provides a high-performance single-issue in-order 32-bit execution pipeline, with a peak sustainable execution rate of one instruction per clock cycle. The RISCV ISA standard M extensions add hardware multiply and divide instructions. CoreRISCV_AXI4 has a range of performance option including a fully pipelined multiply unit.

3.3 Memory System

CoreRISCV_AXI4 memory system supports configurable split first-level instruction and data caches with full support for hardware cache flushing, as well as uncached memory accesses. External connections are provided for both cached and uncached TileLink fabrics.

3.4 Platform-Level Interrupt Controller

CoreRISCV_AXI4 includes a RISC-V standard platform-level interrupt controller (PLIC) configured to support up 31 inputs with a single priority level.

3.5 Debug support via JTAG

CoreRISCV_AXI4 includes full external debugger support over an industry-standard JTAG port, supporting two hardware breakpoints.

3.6 External AXI interfaces

CoreRISCV_AXI4 includes two external AXI interfaces, bridged from the internal TileLink interfaces. The AXI memory interface is used by the cache controller to refill the instruction and data caches. The AXI I/O interface is used for uncached accesses to I/O peripherals.

4 Feature Description

4.1 CoreRISCV_AXI4 Processor Core

The CoreRISCV_AXI4 processor core comprises of an instruction fetch unit, an execution pipeline, and a data memory system.

4.1.1 CoreRISCV_AXI4 Instruction Fetch Unit

The CoreRISCV_AXI4 instruction fetch unit consists of an instruction memory system and no branch predictor.

The instruction memory system includes an instruction cache. The instruction cache is 8 KiB, direct-mapped, with a 64 bytes line size. The access latency is one clock cycle.

The instruction memory system is not coherent with the data memory system. Writes to memory may be synchronized with the instruction fetch stream with a FENCE.I instruction.

There are no dynamic branch prediction. All control-flow instructions are predicted as not taken, and incur a three-cycle penalty on all taken branches and jumps.

4.1.2 CoreRISCV_AXI4 Execution Pipeline

The CoreRISCV_AXI4 execution unit is a single-issue, in-order pipeline. The pipeline comprises five stages: instruction fetch, instruction decode and register fetch, execute, data memory access, and register writeback.

The pipeline has a peak execution rate of one instruction per clock cycle. It is fully bypassed, so most instructions have an apparent one-cycle result latency. There are several exceptions:

- LW has a two-cycle result latency, assuming a cache hit.
- LH, LHU, LB, and LBU have a three-cycle result latency, assuming a cache hit.
- MUL, MULH, MULHU, MULHSU, DIV, DIVU, REM, and REMU have between a 2-cycleand 34-cycle result latency, depending on the pipeline configuration and operand values.
- CSR reads have a three-cycle result latency.

The pipeline only interlocks on read-after-write and write-after-write hazards, so instructions may be scheduled to avoid stalls.

CoreRISCV_AXI4 includes an iterative multiplier with 16 cycles latency.

Branch and jump instructions transfer control from the memory access pipeline stage. Not taken branches and jumps incur no penalty, whereas taken branches and jumps incur a three-cycle penalty.

Most CSR writes result in a pipeline flush, a five-cycle penalty.

4.1.3 CoreRISCV_AXI4 Data Memory System

The CoreRISCV_AXI4 data memory system includes a data cache. The data cache size is 8 KiB, direct-mapped, with a line size of 64 bytes. The access latency is two clock cycles for full words and three clock cycles for smaller quantities. Misaligned accesses are not supported in hardware and result in a trap to support software emulation.

Stores are pipelined and commit on cycles where the data memory system is otherwise idle. Loads to addresses currently in the store pipeline result in a five-cycle penalty.

4.2 Platform-Level Interrupt Controller

This section describes the operation of the platform-level interrupt controller (PLIC). The CoreRISCV PLIC complies with the RISC-V Privileged Architecture specification, and supports 31 external interrupt sources.

4.2.1 Memory Map

The memory map for the CoreRISCV_AXI4 PLIC control registers is shown in Table 3. The PLIC memory map has been designed to only require naturally aligned 32-bit memory accesses.

4.2.2 Interrupt Sources

CoreRISCV_AXI4 contains both local interrupt sources wired directly to the hart contexts and global interrupt sources routed via the PLIC.

4.2.3 Interrupt Source Priorities

All external interrupt sources priority are hardwired at priority 1. Interrupts with the lowest ID have the highest effective priority.

4.2.4 Interrupt Pending Bits

The current status of the interrupt source pending bits in the PLIC core can be read from the pending array, organized as 32 words of 32 bits. The pending bit for interrupt ID N is stored in bit (N mod 32) of word (N=32). Bit 0 of word 0, which represents the non-existent interrupt source 0, is always hardwired to zero.

The pending bits are read-only. A pending bit in the PLIC core can be cleared by setting enable bits to only enable the desired interrupt, then performing a claim. A pending bit can be set by instructing the associated gateway to send an interrupt service request.

Table 3 PLIC Control Registers

Address	Description
0x4000_0000	Reserved
0x4000_0004	source 1 priority
0x4000_0008	source 2 priority
…	
0x4000_0FFC	source 1023 priority
0x4000_1000	Start of pending array
…	(read only)
0x4000_107C	End of pending array
0x4000_1800	
…	Reserved
0x4000_1FFF	
0x4000_2000	target 0 enables
0x4000_2080	target 1 enables
…	
0x401E_FF80	target 15871 enables
0x401F_0000	
	Reserved
0x401F_FFFC	
0x4020_0000	target 0 priority threshold
0x4020_0004	target 0 claim/complete
0x4020_1000	target 1 priority threshold
0x4020_1004	target 1 claim/complete
…	
0x43FF_F000	target 15871 priority threshold
0x43FF_F004	target 15871 claim/complete

4.2.5 Target Interrupt Enables

For each interrupt target, each device’s interrupt can be enabled by setting the corresponding bit in that target’s enables registers. The enables for a target are accessed as a contiguous array of 32_32-bit words, packed the same way as the pending bits. For each target, bit 0 of enable word 0 represents the non-existent interrupt ID 0 and is hardwired to 0. Unused interrupt IDs are also hardwired to zero. The enables arrays for different targets are packed contiguously in the address space.

Only 32-bit word accesses are supported by the enables array in RV32 systems.

Implementations can trap on accesses to enables for non-existent targets, but must allow access to the full enables array for any extant target, treating all non-existent interrupt source’s enables as hardwired to zero.

4.2.6 Target Priority Thresholds

The threshold for a pending interrupt priority that can interrupt each target can be set in the target’s threshold register. CoreRISCV_AXI4 has its threshold hardwired to zero. The target priority threshold is not relevant to CoreRISCV_AXI4because it includes only one hart.

4.2.7 Target Claim

Each target can perform a claim by reading the claim/complete register, which returns the ID of the highest priority pending interrupt or zero if there is no pending interrupt for the target. A successful claim will also atomically clear the corresponding pending bit on the interrupt source.

A target can perform a claim at any time, even if the EIP is not set.

4.2.8 Target Completion

A target signals it has completed running a handler by writing the interrupt ID it received from the claim to the claim/complete register. This is routed to the corresponding interrupt gateway, which can now send another interrupt request to the PLIC. The PLIC does not check whether the completion ID is the same as the last claim ID for that target. If the completion ID does not match an interrupt source that is currently enabled for the target, the completion is silently ignored.

4.2.9 Hart Contexts

CoreRISCV_AXI4 cores always support a machine-mode context for each hart. For machine-mode hart contexts, interrupts generated by the PLIC appear on meip in the mip register. Interrupt targets are mapped to hart contexts sequentially such that target X corresponds to hart ID X.

4.3 Power, Reset, Clock, Interrupt (PRCI)

PRCI is an umbrella term for memory-mapped control and status registers associated with physical hardware submodules that are only visible to machine-mode software. These include the registers controlling component power states, resets, clock selection, and low-level interrupts, hence the name, but other similar functions are also included.

4.3.1 PRCI Address Space Usage

Table 4 shows the memory map for PRCI on CoreRISCV_AXI4 systems.

Because PRCI is only visible to machine-mode software, the memory address space can be densely packed. To simplify interconnect implementation, PRCI interfaces are designed to only require 32-bit or larger accesses. Hardware modules might expose other memory-mapped interfaces suitable for use at lower privilege levels, but these should be mapped to the I/O memory region in a way that can be easily protected from each other using either physical or virtual memory protections.

4.3.2 MSIP Registers

Machine-mode software interrupts are generated by writing to a per-hart memory-mapped control register. The msip registers are 32-bit wide WARL registers, where the LSB is reflected in the msip bit of the associated hart's mip register. Other bits in the msip registers are hardwired to zero. The mapping supports up to 4095 machine-mode harts.

Table 4 Mapping Supports

Address	Description	
0x4400_0000	msip for hart 0	MSIP Registers (16 KiB)
0x4400_0004	msip for hart 1	
…		
0x4400_3FF8	msip for hart 4094	
0x4400_4000	mtimecmp for hart 0	Time Registers (32 KiB)
0x4400_4008	mtimecmp for hart 1	
…		
0x4400_BFF0	mtimecmp for hart 4094	
0x4400_BFF8	mtime	

4.3.3 Timer Registers

Machine-mode timer interrupts are generated by a real-time counter and a per-hart comparator. The mtime register is a 64-bit read-only register that contains the current value of the real-time counter. Each mtimecmp register holds its hart's time comparator. A timer interrupt is pending whenever the value in a hart's mtimecmp register is greater than or equal to mtime. The timer interrupt is reflected in the mtip bit of the associated hart's mip register.

4.4 JTAG Port

CoreRISCV_AXI4 microcontrollers use a single external industry-standard 1149.1 JTAG interface to test and debug the system. The JTAG interface can be directly connected off-chip in a single-chip microcontroller, or can be an embedded JTAG controller for a microcontroller complex designed to be included in a larger SoC.

4.4.1 JTAG Pinout

CoreRISCV_AXI4 uses the industry-standard JTAG interface which includes the four standard signals, TCK, TMS, TDI, and TDO, and optionally also the TRST connection.

On-chip JTAG connections must be driven (no pullups), with a normal two-state driver for TDO under the expectation that on-chip mux logic will be used to select between alternate on-chip

JTAG controllers' TDO outputs.

4.4.2 JTAG TAPC State Machine

The JTAG controller includes the standard TAPC state machine shown in Figure 3.

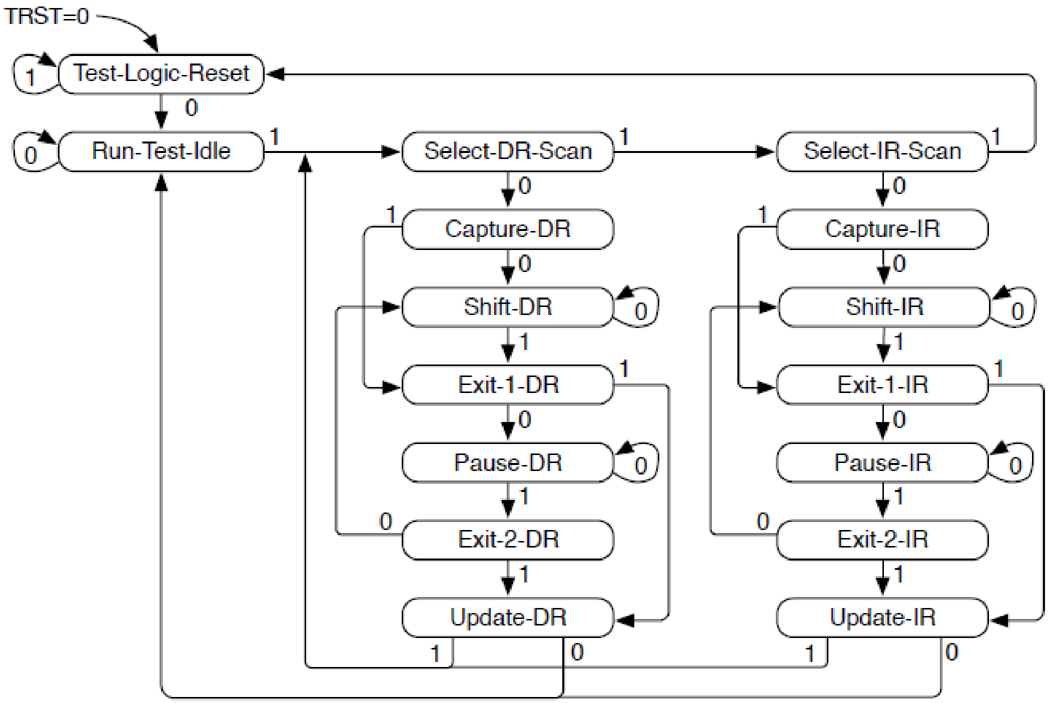
4.4.3 Resetting JTAG logic

The JTAG logic can be asynchrously reset by pulling TRST low, if TRST is available. The TRST signal should be deasserted cleanly while TMS is held high before the first active TCK edge. If TRST is not available, the JTAG logic can be reset by holding TMS high and providing five rising edges on TCK.

Table 5 Resetting JTAG Logic

Signal Name	Description	Direction	Off-Chip	On-Chip
TRST (optional)	Active-low Reset	Input	Must connect	Must connect
TCK	Test Clock	input	weak pull-up	Must connect
TMS	Test Mode Select	input	weak pull-up	Must connect
TDI	Test Data Input	input	weak pull-up	Must connect
TDO	Test Data Output	output	tri-state	Driven

Figure 3 JTAG TAPC State Machine



Only JTAG logic is reset by this action. The remaining system can be reset by using the JTAG interface to the debug module to write appropriate control registers.

4.4.4 JTAG Clocking

The JTAG logic always operates in its own clock domain clocked by TCK. The JTAG logic is fully static and has no minimum clock frequency. The maximum TCK frequency is part-specific.

4.4.5 JTAG Standard Instructions

BYPASS and IDCODE are provided. The CoreRISCV JTAG manufacturer's ID is 0x489.

4.4.6 JTAG Debug Commands

The JTAG DEBUG instruction gives access to the CoreRISCV_AXI4 debug module by connecting the debug scan register in between TDI and TDO. The debug scan register includes a 2-bit opcode field, a 5-bit debug module address field, and a 32-bit data field to allow various memory-mapped read/write operations to be specified with a single scan of the debug scan register. The Debug Module runs on a different clock than the JTAG logic, so the interface between the JTAG debug scan register and the Debug Module includes an asynchronous clock-domain crossing. Refer to the Design Constraints section of this document to determine the appropriate timing constraint that is required on this interface.

4.5 Debug

This section describes the operation of CoreRISCV_AXI4 trace and debug hardware, which follows the standard.

RISC-V debug spec. Currently only interactive debug and hardware breakpoints are supported.

4.5.1 Debug CSRs

This section describes the per-hart trace and debug registers (TDRs), which are mapped into the CSR space as follows:

Table 6 Debug Control and Status Registers (CSR)

CSR Number	Name	Description	Allowed Access Modes
0x7A0	tdrselect	Trace and debug register select	D,M
0x7A1	tdrdata1	First field of selected TDR	D,M
0x7A2	tdrdata2	Second field of selected TDR	D,M
0x7A3	tdrdata3	Third field of selected TDR	D,M
0x7B0	dcsr	Debug control and staus registers	D
0x7B1	dpc	Debug PC	D
0x7B2	dscratch	Debug scratch register	D

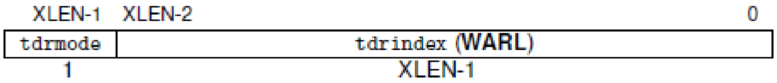
The dcsr, dpc, and dscratch registers are only accessible in debug mode, while the tdrselect and tdrdata1–3 registers are accessible from either debug mode or machine mode.

Trace and Debug Register Select (tdrselect)

To support a large and variable number of TDRs for tracing and breakpoints, they are accessed through one level of indirection where the tdrselect register selects which bank of three tdrdata1–3 registers are accessed via the other three addresses. The tdrselect register has the format shown below:

The MSB of tdrselect selects between debug mode (tdrmode=0) and machine mode (tdrmode=1) views of the registers, where only debug mode code can access the debug mode view of the TDRs. Any attempt to read/write the tdrdata1–3 registers in machine mode when tdrmode=0 raises an illegal instruction exception.

Figure 4 Instruction Exception



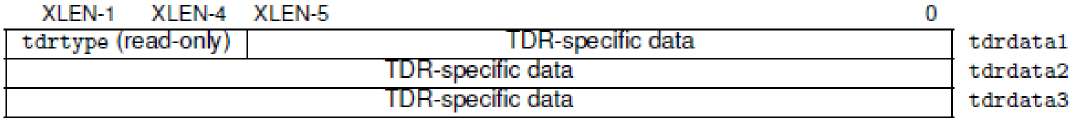
Note: The polarity of tdrmode was chosen such that debug mode needs only a single csrrwi instruction to write tdrselect in most cases.

The tdrindex field is a WARL field that will not hold indices of unimplemented TDRs. Even if tdrindex can hold a TDR index, it does not guarantee the TDR exists. The tdrtype field of tdrdata1 must be inspected to determine whether the TDR exists.

Test and Debug Data Registers (tdrdata1–3)

The tdrdata1–3 registers are XLEN-bit read/write registers selected from a larger underlying bank of TDR registers by the tdrselect register.

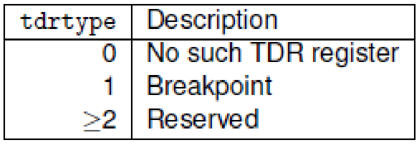
Figure 5 TDR registers



XLEN-1 XLEN-4	XLEN-5 0	
tdrtype (read-only)	TDR-specific data	tdrdata1
TDR-specific data		tdrdata2
TDR-specific data		tdrdata3

The high nibble of tdrdata1 contains a 4-bit tdrtype code that is used to identify the type of TDR selected by tdrselect. The currently defined tdrtypes are shown below:

Figure 6 Defined tdrtypes



tdrtype	Description
0	No such TDR register
1	Breakpoint
≥2	Reserved

Debug Control and Status Register dcsr

Debug PC dpc

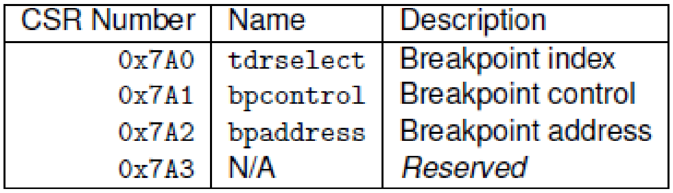
Debug Scratch dscratch

4.5.2 Breakpoints

Each implementation supports a number of hardware breakpoint registers, which can be flexibly shared between debug mode and machine mode.

When a breakpoint register is selected with tdrselect, the other CSRs access the following information for the selected breakpoint:

Figure 7 CSRs Access

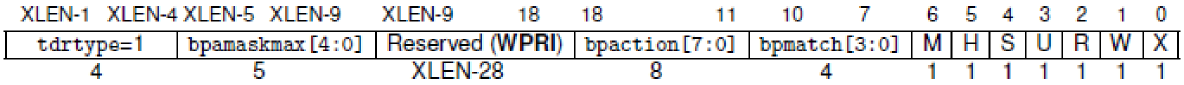


CSR Number	Name	Description
0x7A0	tdrselect	Breakpoint index
0x7A1	bpcontrol	Breakpoint control
0x7A2	bpaddress	Breakpoint address
0x7A3	N/A	*Reserved*

Breakpoint Control Register bpcontrol

Each breakpoint control register is a read/write register laid out as follows:

Figure 8 Register Laid Out



XLEN-1 XLEN-4	XLEN-5 XLEN-9	XLEN-9 18	18 11	10 7	6	5	4	3	2	1	0
tdrtype=1	bpamaskmax[4:0]	Reserved (WPRI)	bpaction[7:0]	bpmatch[3:0]	M	H	S	U	R	W	X
4	5	XLEN-28	8	4	1	1	1	1	1	1	1

The tdrtype field is a four-bit read-only field holding the value 1 to indicate this is a breakpoint containing address match logic.

The bpaction field is an eight-bit read-write WARL field that specifies the available actions when the address match is successful. Currently only the value 0 is defined, and this generates a breakpoint exception.

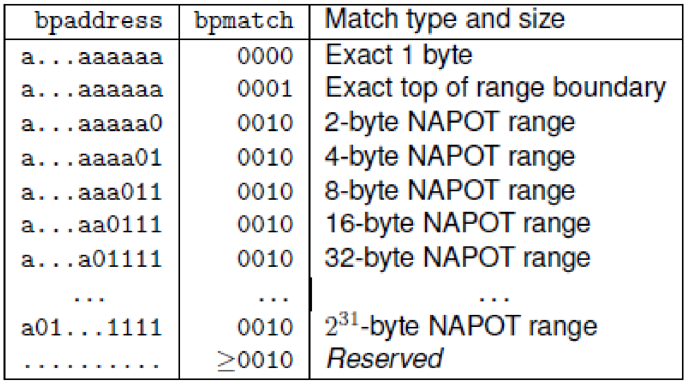
The R/W/X bits are individual WARL fields and if set, indicate an address match should only be successful for loads/stores/instruction fetches respectively, and all combinations of implemented bits must be supported.

The M/H/S/U bits are individual WARL fields and if set, indicate that an address match should only be successful in the machine/hypervisor/supervisor/user modes respectively, and all combinations of implemented bits must be supported.

The bpmatch field is a 4-bit read-write WARL field that encodes the type of address range for breakpoint address matching. Three different bpmatch settings are currently supported: exact, NAPOT, and arbitrary range. A single breakpoint register supports both exact address matches and matches with address ranges that are naturally aligned powers-of-two (NAPOT) in size. Breakpoint registers can be paired to specify arbitrary exact ranges, with the lower-numbered breakpoint register giving the byte address at the bottom of the range and the higher-numbered breakpoint register giving the address one byte above the breakpoint range.

NAPOT ranges make use of low-order bits of the associated breakpoint address register to encode the size of the range as follows:

Figure 9 Breakpoint Address Register



bpaddress	bpmatch	Match type and size
a...aaaaaa	0000	Exact 1 byte
a...aaaaaa	0001	Exact top of range boundary
a...aaaaa0	0010	2-byte NAPOT range
a...aaaa01	0010	4-byte NAPOT range
a...aaa011	0010	8-byte NAPOT range
a...aa0111	0010	16-byte NAPOT range
a...a01111	0010	32-byte NAPOT range
...	...	...
a01...1111	0010	2^{31}-byte NAPOT range
....	≥0010	*Reserved*

The bpamaskmax field is a 5-bit read-only field that specifies the largest supported NAPOT range. The value is the logarithm base 2 of the number of bytes in the largest supported NAPOT range.

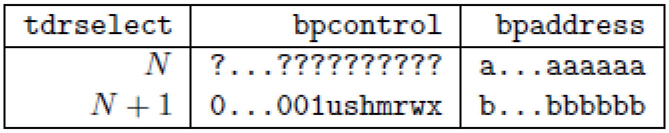
A value of 0 indicates that only exact address matches are supported (one byte range). A value of 31 corresponds to the maximum NAPOT range, which is 231 bytes in size. The largest range is encoded in bpaddr with the 30 least-significant bits set to 1, bit 30 set to 0, and bit 31 holding the only address bit considered in the address comparison.

Note: The unary encoding of NAPOT ranges was chosen to reduce the hardware cost of storing and generating the corresponding address mask value.

To provide breakpoints on an exact range, two neighboring breakpoints are combined as shown in Figure 9, with the lowest matching address in the lower-numbered breakpoint address and the address one byte above the last matching address in the higher-numbered breakpoint address. The bpmatch field in the upper bpcontrol register must be set to 01, after which the values in the upper bpcontrol register control the range match, and all values in the lower bpcontrol are ignored for the purposes of the range match.

The bpcontrol register for breakpoint 0 has the low bit of bpmatch hardwired to zero, so it cannot be accidentally made into the top of a range.

Figure 10 Creating a Range Breakpoint with a Match on Address a...aa _ address < b...bb.



tdrselect	bpcontrol	bpaddress
N	?...??????????	a...aaaaaa
$N+1$	0...001ushmrwx	b...bbbbbb

The value in the lower breakpoint's bpcontrol register for the purposes of the match generated by the upper breakpoint register. An independent breakpoint condition can be set in the lower bpcontrol using the same value in the lower bpaddress register.

Breakpoint Address Register (bpaddress)

Each breakpoint address register is an XLEN-bit read/write register used to hold significant address bits for address matching, and also the unary-encoded address masking information for NAPOT ranges.

Breakpoint Execution

Breakpoint traps are taken precisely. Implementations that emulate misaligned accesses in software will generate a breakpoint trap when either half of the emulated access falls within the address range. Implementations that support misaligned accesses in hardware must trap if any byte of an access falls within the matching range.

Debug-mode breakpoint traps jump to the debug trap vector without altering machine-mode registers.

Machine-mode breakpoint traps jump to the exception vector with "Breakpoint" set in the mcause register, and with badaddr holding the instruction or data address that cause the trap.

Sharing Breakpoints between Debug and machine Mode

When debug mode uses a breakpoint register, it is no longer visible to machine-mode (that is, the tdrtype will be 0). Usually, the debugger will grab the breakpoints it needs before entering machine mode, so machine mode will operate with the remaining breakpoint registers.

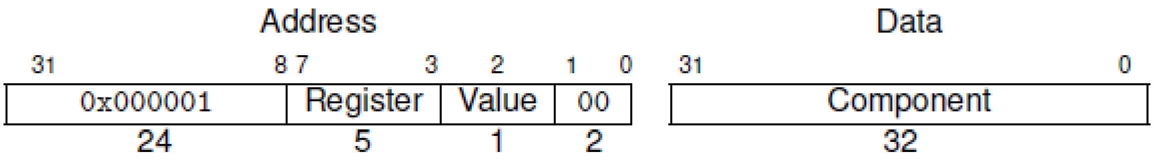
4.5.3 Debug Memory Map

This section describes the debug module's memory map when accessed through the regular system interconnect. The debug module is only accessible to debug code running in debug mode on a hart (or through a debug transport module).

Component Signal Registers (0x100–0x1FF)

The 8-bit address space from 0x100–0x1FF is used to access per-component single-bit registers. This region only supports 32-bit writes. On a 32-bit write to this region, the 32-bit data value selects a component, bits 7–3 of the address select one out of 32 per-component single-bit registers, and bit 2 is the value to be written to that single-bit register, as shown in Figure 11.

Figure 11 Single-bit Register



This addressing scheme was adopted so that RISC-V debug ROM routines can signal that a hart has stopped using a single store instruction to an absolute address (offset from register x0) and one free data register, which holds the hart ID.

The set of valid component identifiers is defined by each implementation.

There are only two per-component registers specified so far, the debug interrupt signal (register 0) and the halt notification register (register 1), resulting in the following four possible write actions.

Debug RAM (0x400–0x43f)

CoreRISCV systems provide at least 64 bytes of debug RAM.

Debug ROM (0x800–0xFFF)

This ROM region holds the debug routines on CoreRISCV systems. The actual total size may vary between implementations.

5 Interface

5.1 Configuration Parameters

5.1.1 CoreRISC_AXI4 Configurable Options

There are a number of configurable options that apply to CoreRISCV_AXI4 as shown in Table 7. If a configuration other than the default is required, use the configuration dialog box in SmartDesign to select appropriate values for the configurable options.

Table 7 CoreRISCV_AXI4 Configuration Options

Parameter	Valid Range	Default	Description
RESET_VECTOR_ADDR	0x8FFFFFFC >= RESET_VECTOR_ADDR >= 0x60000000	0x60000000	This is the address the processor will start executing from after a reset.

In order to facilitate instantiating this core on smaller Microsemi parts with limited RAM resources, all RAM inferences other than the Instruction and Data caches within CoreRISCV_AXI4 can be optionally implemented with fabric registers rather than allowing the synthesis tool to infer RAM. This requires modification of the coreriscv_axi4_defines.v file in the work/SmartDesign_name/CoreRISCV_AXI4_Instance_name/rtl/vlog/core folder of the Libero project as follows:

- To use regsiters for all RAM blocks other than the internal instruction and data caches, uncomment the USE_REGISTERS define.

Note: The contents of the coreriscv_axi4_defines.v file will need to be replaced every time that the SmartDesign sheet containing the CorRISCV_AXI4 instance is generated.

5.1.2 Signal Descriptions

Signal descriptions for CoreRISCV_AXI4 are defined in Table 8.

Table 8 CoreRISCV_AXI4 I/O Signals

Port Name	Width	Direction	Description
Global Signal Ports			
CLK	1	In	System clock. All other I/Os are synchronous to this clock.
RESET	1	In	Active-high reset signal. The source of this reset signal must be de-asserted synchronous to CLK via a reset synchronizer. Refer to the System Integration section for implementation details.
JTAG INTERFACE			
TDI	1	In	Test Data In (TDI). This signal is used by the JTAG device for downloading and debugging programs. Sampled on the rising edge of TCK.
TCK	1	In	Test Clock (TCK). This signal is used by the JTAG device for downloading and debugging programs.
TMS	1	In	Test Mode Select (TMS). This signal is used by the JTAG device when downloading and debugging programs. It is sampled on

			the rising edge of TCK to determine the next state.
TRST	1	In	Test Reset (TRST). This is an optional signal used to reset the TAP controllers state machine.
TDO	1	Out	Test Data Out (TDO). This signal is the data which is shifted out of the device during debugging. It is valid on FALLING/RISING edge of TCK.
DRV_TDO	1	Out	Drive Test Data Out (DRV_TDO). This signal denotes when the TDO output of this core is being driven.
External Interrupts			
IRQ	31	In	External interrupts from off-chip or peripheral sources. These are level-based interrupt signals.
AXI Slave Interface Ports			
AXI Memory Bus Master Interface			
AXI_MST_MEM_AWREADY	1	In	AXI4 Master Write Address Channel for performing Memory accesses.
AXI_MST_MEM_AWVALID	1	Out	
AXI_MST_MEM_AWADDR	32	Out	
AXI_MST_MEM_AWLEN	8	Out	
AXI_MST_MEM_AWSIZE	3	Out	
AXI_MST_MEM_AWBURST	2	Out	
AXI_MST_MEM_AWLOCK	1	Out	
AXI_MST_MEM_AWCACHE	4	Out	
AXI_MST_MEM_AWPROT	3	Out	
AXI_MST_MEM_AWQOS	4	Out	
AXI_MST_MEM_AWREGION	4	Out	
AXI_MST_MEM_AWID	5	Out	
AXI_MST_MEM_AWUSER	1	Out	
AXI_MST_MEM_WREADY	1	In	AXI4 Master Write Data Channel for performing Memory accesses.
AXI_MST_MEM_WVALID	1	Out	
AXI_MST_MEM_WDATA	64	Out	
AXI_MST_MEM_WLAST	1	Out	
AXI_MST_MEM_WSTRB	8	Out	
AXI_MST_MEM_WUSER	1	Out	
AXI_MST_MEM_BREADY	1	Out	AXI4 Master Write. Response channel for. Performing memory. Accesses.
AXI_MST_MEM_BVALID	1	In	
AXI_MST_MEM_BRESP	2	In	
AXI_MST_MEM_BID	5	In	
AXI_MST_MEM_BUSER	1	In	
AXI_MST_MEM_ARREADY	1	Out	AXI4 master Read Address. Channel for performing. Memory accesses. AXI4 master Read Address.
AXI_MST_MEM_ARVALID	1	In	
AXI_MST_MEM_ARCACHE	4	Out	
AXI_MST_MEM_ARPROT	3	Out	
AXI_MST_MEM_ARQOS	4	Out	

AXI_MST_MEM_ARREGION	4	Out	Channel for performing. Memory accesses.
AXI_MST_MEM_ARUSER	1	Out	
AXI_MST_MEM_ARID	5	Out	
AXI_MST_MEM_ARADDR	32	Out	
AXI_MST_MEM_ARLEN	8	Out	
AXI_MST_MEM_ARSIZE	3	Out	
AXI_MST_MEM_ARBURST	2	Out	
AXI_MST_MEM_ARLOCK	1	Out	
AXI_MST_MEM_RID	5	In	AXI4 Master Read Data. Channel for performing. Memory accesses.
AXI_MST_MEM_RDATA	64	In	
AXI_MST_MEM_RRESP	2	In	
AXI_MST_MEM_RLAST	1	In	
AXI_MST_MEM_RVALID	1	In	
AXI_MST_MEM_RREADY	1	Out	
AXI_MST_MEM_RUSER	1	In	
AXI_MST_MEM_WID	5	Out	This signal is included for compatibility with AXI3.
AXI Memory-Mapped I/O (MMIO) Bus Master Interface			
AXI_MST_MMIO_AWID	5	Out	AXI4 Master Write Address. Channel for performing. MMIO accesses.
AXI_MST_MMIO_AWADDR	32	Out	
AXI_MST_MMIO_AWLEN	8	Out	
AXI_MST_MMIO_AWSIZE	3	Out	
AXI_MST_MMIO_AWBURST	2	Out	
AXI_MST_MMIO_AWLOCK	1	Out	
AXI_MST_MMIO_AWVALID	1	Out	
AXI_MST_MMIO_AWREADY	1	In	
AXI_MST_MMIO_AWCACHE	4	Out	
AXI_MST_MMIO_AWPROT	3	Out	
AXI_MST_MMIO_AWUSER	1	Out	
AXI_MST_MMIO_AWREGION	4	Out	
AXI_MST_MMIO_AWQOS	4	Out	
AXI_MST_MMIO_WDATA	64	Out	AXI4 Master Write Data. Channel for performing. MMIO accesses.
AXI_MST_MMIO_WSTRB	8	Out	
AXI_MST_MMIO_WLAST	1	Out	
AXI_MST_MMIO_WVALID	1	Out	
AXI_MST_MMIO_WREADY	1	In	
AXI_MST_MMIO_WUSER	1	Out	
AXI_MST_MMIO_BID	5	In	AXI4 Master Write. Response channel for. MMIO accesses.
AXI_MST_MMIO_BRESP	2	In	
AXI_MST_MMIO_BVALID	1	In	
AXI_MST_MMIO_BREADY	1	Out	
AXI_MST_MMIO_BUSER	1	In	

AXI_MST_MMIO_ARCACHE	4	Out	AXI4 Master Read Address. Channel for performing. MMIO accesses.
AXI_MST_MMIO_ARPROT	3	Out	
AXI_MST_MMIO_ARQOS	4	Out	
AXI_MST_MMIO_ARREGION	4	Out	
AXI_MST_MMIO_ARUSER	1	Out	
AXI_MST_MMIO_ARID	5	Out	
AXI_MST_MMIO_ARADDR	32	Out	
AXI_MST_MMIO_ARLEN	8	Out	
AXI_MST_MMIO_ARSIZE	3	Out	
AXI_MST_MMIO_ARBURST	2	Out	
AXI_MST_MMIO_ARLOCK	1	Out	
AXI_MST_MMIO_ARVALID	1	In	
AXI_MST_MMIO_ARREADY	1	Out	
AXI_MST_MMIO_RID	5	In	AXI4 Master Read Data. Channel for performing. MMIO accesses.
AXI_MST_MMIO_RDATA	64	In	
AXI_MST_MMIO_RRESP	2	In	
AXI_MST_MMIO_RLAST	1	In	
AXI_MST_MMIO_RVALID	1	In	
AXI_MST_MMIO_RREADY	1	Out	
AXI_MST_MMIO_RUSER	1	In	
AXI_MST_MMIO_WID	5	Out	This signal is included for compatibility with AXI3.

6 Register Map and Descriptions

Table 9 Physical Memory Map (from E3 Coreplex Series)

Base	Top	Description	
0x0000_0000	0x0000_00FF	Reserved	Debug Area(4 KiB)
0x0000_0100		Clear debug interrupt to component	
0x0000_0104		Set debug interrupt to component	
0x0000_0108		clear halt notification from component	
0x0000_010C		set halt notification from component	
0x0000_0110	0x0000_03FF	Reserved	
0x0000_0400	0x0000_07FF	Debug RAM (≤1KiB)	
0x0000_0800	0x0000_0FFF	Debug ROM (≤1KiB)	
0x0000_1000		Reset	Small ROM Area (60 KiB)
0x0000_1004		NMI	
0x0000_1008		Reserved	
0x0000_100C		Configuration string address	
0x0000_1010	0x0000_XXXX	Trap vector table start	
0x0000_XXXX		Reset code	
		Interrupt handlers	
		Emulation routines	
		Register save/restore routines	
	0x0000_FFFF	User ROM	
0x0001_0000	0x3FFF_FFFF	Reserved	ROM/Misc./Reserved (≈1GiB)
0x4000_0000	0x43FF_FFFF	Platform-Level Interrupt Control (PLIC)	
0x4400_0000	0x47FF_FFFF	Power/Reset/Clock/Interrupt (PRCI)	
0x4800_0000	0x4800_0FFF	Device Bank 0:	On-Coreplex Devices (128 MiB)
…			
0x4800_F000	0x4800_FFFF	Device Bank 15:	
0x4801_0000	0x4FFF_FFFF	Reserved	
0x5000_0000	0x5FFF_FFFF	I/O	Off-Coreplex Devices (768 MiB)
0x6000_0000	0x7FFF_FFFF	AXI I/O Interface	
0x8000_0000	0x8FFF_FFFF	AXI Memory Interface	RAM Area (256 MiB)

7 Tool Flow

7.1 License

A license is not required to use this IP Core with Libero SoC.

7.1.1 RTL

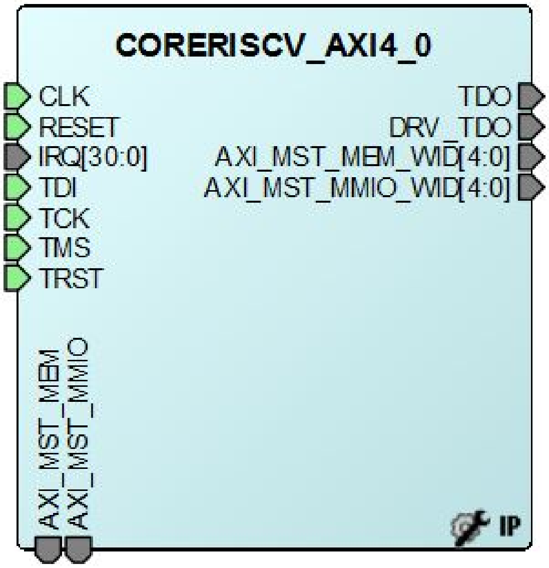
Complete RTL code is provided for the core, allowing the core to be instantiated with SmartDesign. Simulation, Synthesis, and Layout can be performed within Libero SoC.

7.2 SmartDesign

An example instantiated view of CoreRISCV_AXI4 is shown in Figure 12.

For more information on using SmartDesign to instantiate and generate cores, refer to the Using DirectCore in Libero® SoC User Guide.

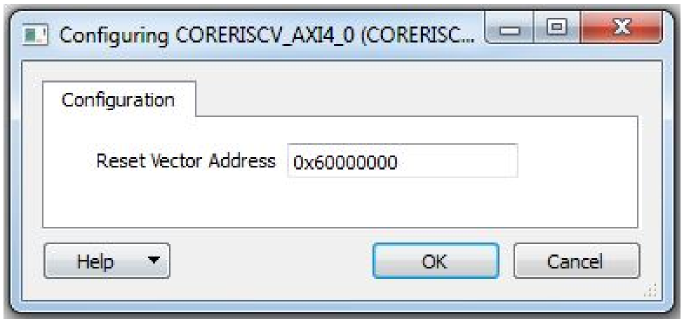
Figure 12 SmartDesign CoreRISCV_AXI4 Instance View



7.3 Configuring CoreRISCV_AXI4 in SmartDesign

The core can be configured using the configuration GUI within SmartDesign. An example of the GUI for the SmartFusion2 family is shown in Figure 13.

Figure 13 Configuring CoreRISCV_AXI4 in SmartDesign



Configuring CORERISCV_AXI4_0 (CORERISC...

Configuration

Reset Vector Address 0x60000000

7.4 Simulation Flows

No testbench is provided with CoreRISCV_AXI4.

The CoreRISCV_AXI4 RTL can be used to simulate the processor executing a program using a standard Libero generated HDL testbench.

7.5 Synthesis in Libero

Click the **Synthesis** icon in Libero SoC. The Synthesis window displays the Synplify Pro project. Set Synplify Pro to use the Verilog 2001 standard if Verilog is being used. To run **Synthesis**, select the **Run** icon.

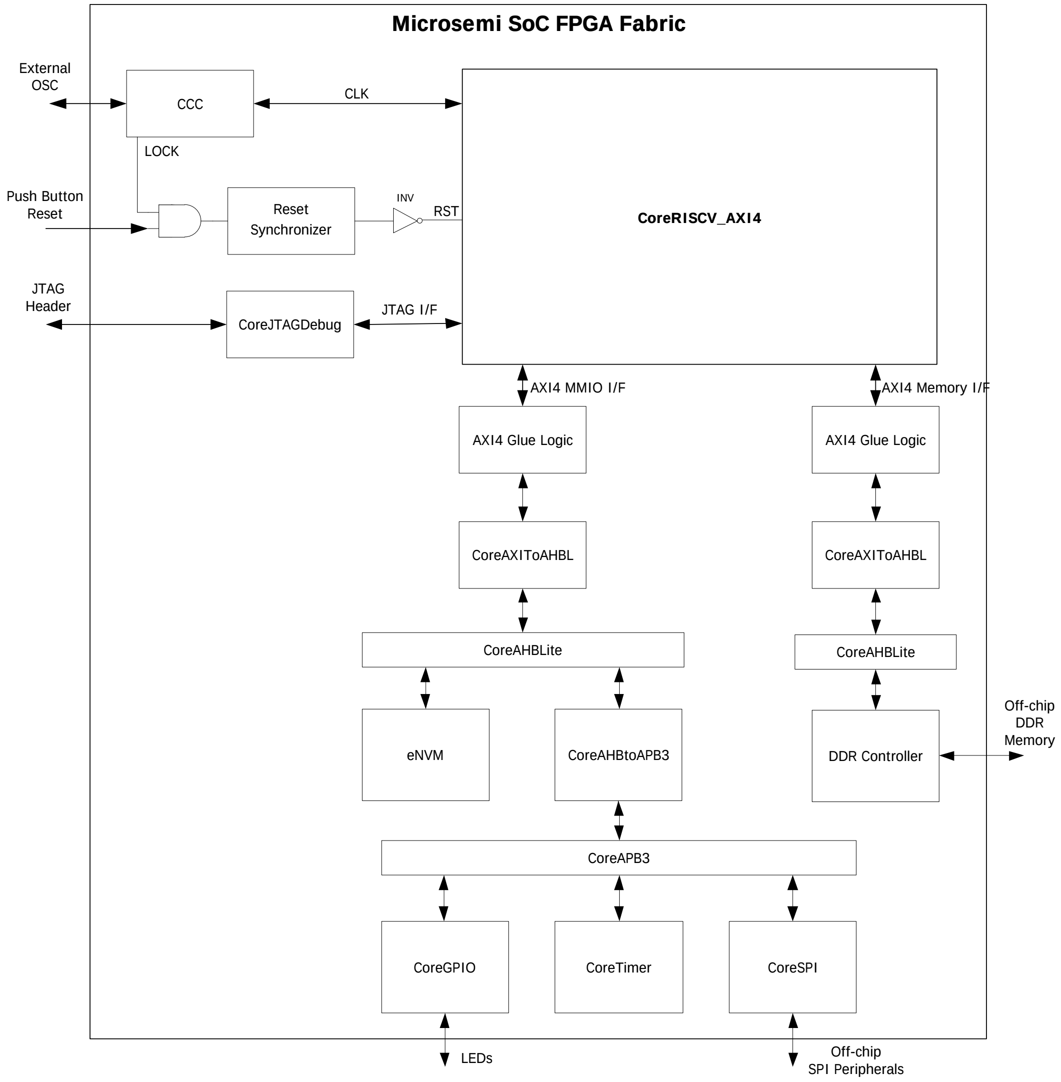
7.6 Place-and-Route in Libero

Click the **Layout** icon in the Libero SoC to invoke Designer. After synthesis has been completed the Place-and-Route tool can be run.

8 System Integration

8.1 Example System

Figure 14 CoreRISCV_AXI4 Example System

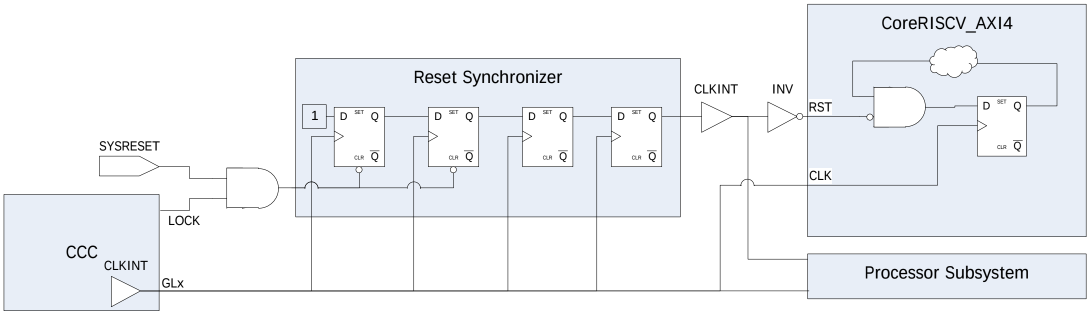


8.2 Reset Synchronization

8.2.1 RST

All sequential elements clocked by *CLK* within CoreRISCV_AXI4, which require a reset employ a synchronous reset topology. Since most designs source *CLK* from a CCC/PLL, it is common practice to AND the *LOCK* output of the CCC with the push button reset to generate the *RST* input for CoreRISCV_AXI4. However, this results in the reset being deasserted when the CLK comes up, hence the reset assertion is not clocked through the sequential reset elements and goes unnoticed most commonly leading to the processor locking-up. To guarantee that the RST assertion is seen by all sequential elements, a reset synchronizer is required on the *RST* input, as shown in Figure 15.

Figure 15 RST Reset Synchronization



The Verilog code snippet below implements the reset synchronizer block shown in Figure 15. The function of this block is to make the reset assertion and deassertion synchronous to CLK whilst guaranteeing that the reset will be seen asserted for one or more CLK cycles within CoreRISCV_AXI4 to ensure that it is registered by all sequential elements.

```
module reset_synchronizer (
    input  clock,
    input  reset,
    output reset_sync
);
reg [1:0]   sync_deasert_reg;
reg [1:0]   sync_asert_reg;

always @ (posedge clock or negedge reset)
    begin
        if (!reset)
            begin
                sync_deasert_reg[1:0] <= 2'b00;
            end
        else
            begin
                sync_deasert_reg[1:0] <= {sync_deasert_reg[0], 1'b1};
            end
    end


always @ (posedge clock)
    begin
        sync_asert_reg[1:0] <= {sync_asert_reg[0], sync_deasert_reg[1]};
    end
assign reset_sync = sync_asert_reg[1];

endmodule
```

To include this synchronizer in your Libero design, select Create HDL from the Design Flow tab in your Libero project. In the popup window, name the HDL file accordingly and select Verilog as the HDL type whilst unchecking the option to Initialize file with standard template. Copy and paste the Verilog code snippet above into this file and save the changes. From the Design Hierarchy tab drag and drop the file into the SmartDesign sheet containing the CoreRISCV_AXI4 instance and connect up the pins as shown above.

8.2.2 TRST

No reset synchronization is required on this reset input as all sequential elements in the debug logic within CoreRISCV_AXI4 use an asynchronous reset topology.

9 Design Constraints

Designs containing CoreRISCV_AXI4 require the application of the following constraints in the design flow to allow timing-driven placement and static timing analysis to be performed on CoreRISCV_AXI4. The procedure for adding the required constraints in the Enhanced Constraints flow in Libero v11.7 or later is as follows:

1. Double-click **Constraints** > **Manage Constraints** in the **Design Flow** window and click the **Timing** tab.

   Assuming that the system clock used to clock CoreRISCV_AXI4 is sourced from a PLL, select Derive to automatically create a constraints file containing the PLL constraints. Select **Yes** when prompted to allow the constraints to be automatically included for Synthesis, Place-and-Route, and Timing Verification stages.

   If changes are made to the PLL configuration in the design, update the contents of this file by clicking **Derive**. Select **Yes** when prompted to allow the constraints to be overwritten.

2. In the **Timing** tab of the **Constraint Manager** window, select **New** to create a new SDC file, and name it. Design constraints other than the system clock source derived constraints can be entered in this blank SDC file. Keeping derived and manually added constraints in separate SDC files allows the **Derive** stage to be reperformed if changes are made to the PLL configuration, without deleting all manually added constraints in the process.

3. Calculate the TCK period and half period. TCK is typically 6 MHz when debugging with FlashPro, with a maximum frequency of 30 MHz supported by FlashPro5. After completion, enter the following constraints in the blank SDC file:

```
create_clock -name { TCK } \
    -period TCK_PERIOD \
    -waveform { 0 TCK_HALF_PERIOD } \
    [ get_ports { TCK } ]
```

For example, the following constraints need to be applied for a design that uses a TCK frequency of 6 MHz:

```
create_clock -name { TCK } \
    -period 166.67 \
    -waveform { 0 83.33 } \
    [ get_ports { TCK } ]
```

4. Next constraints must be applied to paths crossing the clock domain crossing between the TCK and system clock clock domains. CoreRISCV_AXI4 implements two clock domain crossing FIFOs to handle the CDC and as such paths between the two clock domains may be declared as false paths to prevent min and max violations from being reported by SmartTime.

```
set_false_path -from [ get_clocks { TCK } ] \
               -to [ get_clocks { PLL_GEN_CLK } ]

set_false_path -from [ get_clocks { PLL_GEN_CLK } ] \
               -to [ get_clocks { TCK } ]
```

Where:

- PLL_GEN_CLK is the name applied to the create_generated_clock constraint derived in step 1 above.

5. Associate all constraints files with the Synthesis, Place-and-Route and Timing Verification stages in the **Constraint Manager > Timing** tab by selecting the related check boxes for the SDC files in which the constraints were entered in.