CoreCortexM1 v3.0

HB0732 Handbook

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

© 2017 Microsemi Corporation. All rights reserved. Microsemi and the Microsemi logo are trademarks of Microsemi Corporation. All other trademarks and service marks are the property of their respective owners.

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.

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.

50200732-1/2.17

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 CoreCortexM1 v3.0.

1.2 Revision 1.0

Revision 1.0 was the first publication of this document. Created for CoreCortexM1 v2.0.

Contents

1 Revision History 3
1.1 Revision 2.0 3
1.2 Revision 1.0 3
2 Preface 8
2.1 Purpose 8
2.2 Intended Audience 8
3 Introduction 9
3.1 Overview 9
3.2 Features 10
3.3 Core Version 10
3.4 Supported Families 10
3.5 Device Utilization and Performance 10
4 Functional Description 11
4.1 External Interface 11
4.1.1 Write Buffer 11
4.1.2 Tightly Coupled Memory (TCM) Interface 12
4.2 Clocking and Resets 13
4.2.1 Clocks 13
4.2.2 Resets 13
4.2.3 Reset Control Logic 13
4.2.4 Debug 14
4.2.4.1 About Debug 14
4.3 JTAG Debug Interface 15
4.4 Programmer's Model 17
4.4.1 Processor Operating States 17
4.4.2 Processor Operating Modes 17
4.4.3 Main Stack and Process Stack Access 17
4.4.4 Data Types 17
4.4.5 Registers 17
4.4.6 General Purpose Registers 18
4.4.7 Special Purpose Program Status Registers (xPSR) 18
4.4.8 Special Purpose Priority Mask Register 19
4.4.9 Special Purpose Control Register 19
4.4.10 Memory Map 19
4.4.11 Subsystem Restrictions 20
4.4.12 Exceptions 20
4.4.12.1 Exception Types 21
4.4.12.2 Exception Priority 21
4.4.12.3 Servicing an Exception 22
4.4.13 Nested Vectored Interrupt Controller 23
5 Interface 25
5.1 Ports 25
5.2 Configuration Parameters 27
5.2.1 CoreCortexM1 Configurable Options 27
6 Tool Flow 28
6.1 License 28
6.1.1 RTL 28

6.2 SmartDesign.......... 28
6.2.1 Configuring CoreCortexM1 in SmartDesign.......... 29
6.3 Simulation Flows.......... 30
6.4 Synthesis in Libero.......... 30
6.5 Place-and-Route in Libero.......... 30

7 BFM Usage Flow.......... 31
7.1 Functionality.......... 32
7.1.1 CoreCortexM1 Pin Compatibility.......... 32
7.1.2 CoreCortexM1 Bus Cycle Accuracy.......... 32
7.1.3 Scripting.......... 32
7.1.4 Self-Checking.......... 32
7.1.5 Endianness.......... 32
7.1.6 Interrupt Support.......... 32
7.1.7 Log File Generation.......... 32
7.1.8 BFM Script Language.......... 32
7.1.8.1 Write.......... 32
7.1.8.2 Read.......... 33
7.1.8.3 Readcheck.......... 33
7.1.8.4 poll.......... 33
7.1.8.5 wait.......... 34

8 System Integration.......... 35
8.1 Integration of CoreCortexM1 with the example design.......... 35
8.1.1 Interface with Core Bootstrap.......... 35
8.1.2 Interface with PF_SRAM_AHBL_AXI.......... 36

9 Ordering Information.......... 38
9.1 Ordering Codes.......... 38

Figures

Figure 1 Block Diagram 9
Figure 2 Timing Diagram for Read without Wait States 11
Figure 3 ITCM Signal Timings 12
Figure 4 CoreCortex-M1 interfaces with PF_SRAM_AHBL_AXI 12
Figure 5 JTAG 10-Pin Connector Pinout 15
Figure 6 JTAG 20-Pin Connector Pinout 15
Figure 7 Cortex -M1 Register Set 18
Figure 8 CoreCortexM1 Memory Map 19
Figure 9 Stack Contents from an Exception 22
Figure 10 Exception Entry Without Wait States 23
Figure 11 CoreCortexM1 I/O Signals 25
Figure 12 SmartDesign CoreCortexM1 Instance View 29
Figure 13 SmartDesign CoreCortexM1 Configuration window 29
Figure 14 Simulation Environment for a CoreCortexM1 System Inside Libero IDE Board Level System 31
Figure 15 CoreCortexM1 Example Design 36

Tables

Table 1 CoreCortexM1 Device Utilization and Performance 10
Table 2 JTAG 10-Pin Connector Signals 15
Table 3 JTAG 20-Pin Connector Signals 16
Table 4 CoreCortexM1 Exceptions 21
Table 5 CoreCortexM1 NVIC Registers 23
Table 6 CoreCortexM1 Port Descriptions 25
Table 7 CoreCortexM1 Configuration Options 27
Table 8 SPIRIT IP-XACT Attributes 34
Table 9 Ordering Codes 37

2 Preface

2.1 Purpose

This handbook provides details about the architecture and implementation of the 32-bit ARM® Cortex®-M1 microprocessor developed by ARM specifically for use in FPGAs. CoreCortexM1 processor has a three-stage pipeline and runs the ARMv6-M instruction set; it is essentially a functional subset of the Cortex-M3. The streamlined CoreCortexM1 developed for use in embedded applications.

2.2 Intended Audience

FPGA designers using Libero® System-on-Chip (SoC).

3 Introduction

3.1 Overview

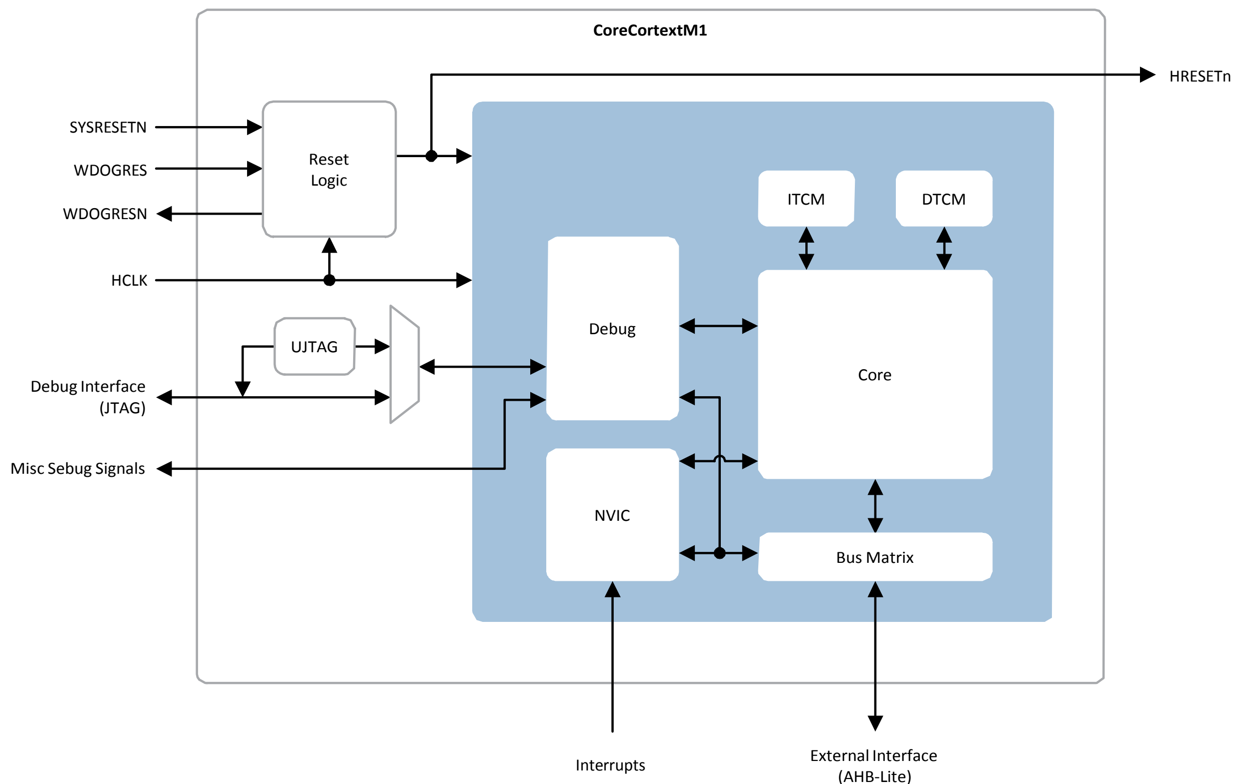
CoreCortexM1 processor is a general purpose 32-bit microprocessor that offers high performance and small size in FPGAs.

CoreCortexM1 processor runs a subset of the Thumb-2 instruction set (ARMv6-M) that includes all base 16-bit Thumb instructions and a few Thumb-2 32-bit instructions (BL, MRS, MSR, ISB, DSB, and DMB). This enables writing very tight and efficient processor code, which is ideal for the limited memory typically found in deeply-embedded applications.

Figure 1 shows a block diagram of the CoreCortexM1 processor available for use in Microsemi FPGAs. The components within the blue box in this diagram are preconfigured and fixed for each available configuration of the core. These components are contained as an encrypted netlist. At the top level, there is an RTL wrapper surrounding the netlist component and this contains reset synchronization logic and debug related logic.

The main blocks in CoreCortexM1 are as shown in Figure 1 and include the processor core, the Nested Vectored Interrupt Controller (NVIC), the AHB interface, and the debug unit. The processor has 13 general purpose 32-bit registers as well as a Link register (LR), a program counter (PC), a stack pointer (SP) and a Program Status register (xPSR). The core has been configured with the operating system extensions present, a second stack pointer is available for use. A dedicated memory interface is available for access to Instruction and Data Tightly Coupled Memories (ITCM and DTCM).

Figure 1 • **Block Diagram**



The NVIC is closely coupled to the CoreCortexM1 core to achieve low-latency interrupt processing. The processor state is automatically saved on interrupt entry and restored on interrupt exit, with no instruction overhead to simplify software development.

The 16-bit length of the CoreCortexM1 Thumb instruction allows it to approach twice the code density of the standard 32-bit ARM code, while retaining most of the ARM performance advantages over a traditional 16-bit processor that uses 16-bit registers. This is possible because Thumb code operates on the 32-bit register set in the processor. Thumb code is able to provide up to 65% of the code size of ARM, and 160% of the performance of an equivalent ARM processor connected to a 16-bit memory system.

3.2 Features

- 32-bit RISC architecture (ARMv6-M)
- 32-bit AHB-Lite bus interface
- Three-stage pipeline
- 32-bit ALU
- 4-GB memory addressing range (the upper 0.5 GB is reserved)
- Real-time debug
- JTAG interface

3.3 Core Version

This handbook is for CoreCortexM1 version 3.0.

3.4 Supported Families

- PolarFire

3.5 Device Utilization and Performance

Table 1 shows the utilization and performance data for the PolarFire device families.

Speed Grade – -1 , Core Voltage – 1.0 and Operating Condition- EXT .

***Table 1* • CoreCortexM1 Device Utilization and Performance**

FPGA Family	Device	Speed Grade	FPGA Resources			Utilization	Clock Rate (MHz)
			Combinatorial	Sequential	Total		
PolarFire	MPF300T	-1	10065	6840	16905	~5.64%	70 MHz

Note: Data in this table were achieved using synthesis and layout settings optimized for speed along.

4 Functional Description

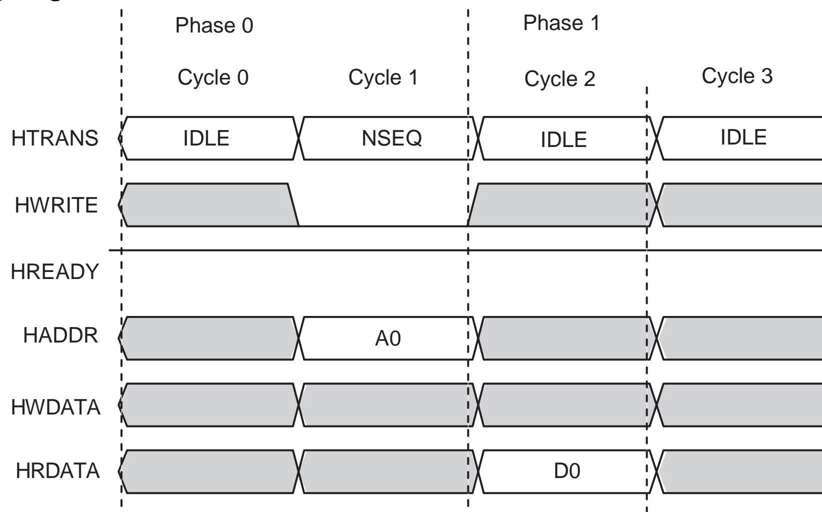
The CoreCortexM1 processor has an external AHB-Lite interface that can be used to connect to other AMBA components. Dedicated interfaces are available to provide fast access to the ITCM and DTCM.

4.1 External Interface

The external interface is an AMBA AHB-Lite bus interface. Descriptions of the bus signals are shown in Table 6 on page 25. The processor accesses and debug accesses to external AHB peripherals can occur over this bus interface. Because AHB fetches take two cycles longer than TCM fetches, instructions and data should ideally be contained in TCM where possible.

Figure 2 shows the timing of a read without wait states on the external interface. The Address A0 is presented on HADDR one cycle later than the internal address is generated, and the returned data D0 is registered again before use in the processor. This enables the AHB peripherals sufficient time to use the address generated.

Figure 2 • **Timing Diagram for Read without Wait States**



Processor accesses and debug accesses share the external interface. Debug accesses take priority over processor accesses. Giving the highest priority to debug means that debug cannot be locked out by a continuously executing stream of core instructions. Timing of processor accesses might be changed by the presence of debug accesses. Debug accesses tend to be infrequent, so debug accesses generally do not have a major impact on processor accesses.

Any vendor-specific components with an AHB interface can populate this bus. Unaligned accesses to this bus are not supported.

4.1.1 Write Buffer

To prevent bus wait cycles from stalling the processor during data stores, buffered stores to the external interfaces go through a one-entry write buffer. If the write buffer is full, subsequent accesses to the bus stall until the write buffer has drained. The write buffer is only used if the bus waits for the data phase of the buffered store; otherwise the transaction is immediately completed on the bus.

The DMB and DSB instructions wait for the write buffer to drain before completing. If an interrupt arrives while DMB/DSB is waiting for the write buffer to drain, the processor returns to the opcode after the DMB/DSB, on completion of the interrupt. This is because interrupt processing is a so-called memory

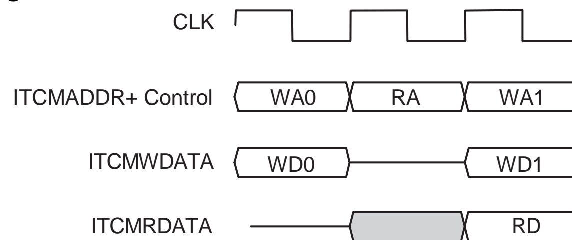
barrier operation. In other words, all reads and writes occur before the interrupt appear to happen before the interrupt, so the DMB and DSB instructions must appear to have completed before the interrupt.

4.1.2 Tightly Coupled Memory (TCM) Interface

Dedicated low latency memory interfaces are available for accessing ITCM and DTCM.

Because reads are speculatively fetched from TCMs, Device and Strongly-Ordered memory types such as FIFOs in TCM space are not supported. The processor does not support wait states for the memory interfaces. Figure 3 shows the signal timings for ITCM. The DTCM signal timings are the same as ITCM signal timings.

Figure 3 • **ITCM Signal Timings**



The write address, WA0, and write data, WD0, are presented in the same cycle. The Read Address (RA) is presented in one cycle, and the memory generates the Read Data (RD) in the next cycle. The sequence that Figure 3 shows is only possible on ITCM, where the RA read is an instruction fetch, and WA1 write is the product of a store instruction. The WA0-RA sequence is possible on DTCM.

In situations where there is only one large RAM available, but the user wants to run from RAM, the RAM should be mapped to location 0x6000000. The user should download the program (via debugger) to this location and start execution from this point. The user’s software should be written/linked assuming this location during the debug stage. This is the only SRAM area in the memory map that supports both instruction fetches and data accesses. When the debugging has finished, the software needs to be rewritten and linked to assume the starting point of location 0x00000000.

4.2 Clocking and Resets

4.2.1 Clocks

HCLK is the main clock input and clocks the majority of the logic in the processor. In debug logic the TCK input is used to clock logic in the debug access port. HCLK is also used for clocking some debug components when these are present.

4.2.2 Resets

Within the CoreCortexM1Top level of hierarchy (see Figure 1 on page 9) a reset synchronization block takes in a number of reset signals and produces several reset outputs. The synchronization block ensures that resets which may assert asynchronously are de-asserted synchronous to the clock domain to which they are relevant.

SYSRESETN is the main reset input.

WDOGRES and WDOGRESN ports are provided to facilitate support of a watchdog type component such as CoreWatchdog. WDOGRES is the "bark" signal from the watchdog and WDOGRESn is a reset output to the watchdog.

For the debug logic, the NTRST reset input is functional and is used to reset logic clocked by TCK within the debug port.

The following paragraphs describe the reset outputs from the reset synchronization block and detail the reset sources for each reset output.

4.2.3 Reset Control Logic

Reset control logic creates a system reset signal to the processor (which connects to its SYSRESETN input) that is synchronized to HCLK and which asserts when any of the following occur:

- Assertion of the top level SYSRESETN input.
- Assertion of the top level NRESET input
- De-assertion of the top level watchdog input (WDOGRES)

The NRESET input is effectively an "additional" reset input. It could be driven from any source.

When Reset control logic is included, a HRESETN output is driven by the same merged, synchronized signal that drives the system reset input of the processor. HRESETN can be used to reset other components in the system.

When Reset control logic is not included, the HRESETN, WDOGRESN outputs and the NRESET, WDOGRES inputs are not used. Reset to other components in the system should be from the external source (Example: Combination of power on reset and lock of the clock conditioning circuit).

4.2.4 Debug

The ARM Debug Architecture allow the debugger to talk via a JTAG port directly to the core.

4.2.4.1 About Debug

Debug facilitates the following:

- Core halt
- Core stepping
- Core register access
- Read/Write to TCMs
- Read/Write to AHB address space
- Breakpoints
- Watchpoints

The main debug components are as follows:

- Debug control registers to access and control debugging of the core
- Breakpoint Unit (BPU) to implement breakpoints
- Data Watchpoint (DW) unit to implement watchpoints and trigger resources
- Debug memory interfaces to access external ITCM and DTCM
- ROM table

All debug components exist on the internal Private Peripheral Bus (PPB), 0xE000ED30 to 0xE000EEFF. Access to the debug components, debug control, and configuration are available.

Debug control and data access occur through the Advanced High-Performance Bus-Access Port (AHB-AP).

Access includes the following:

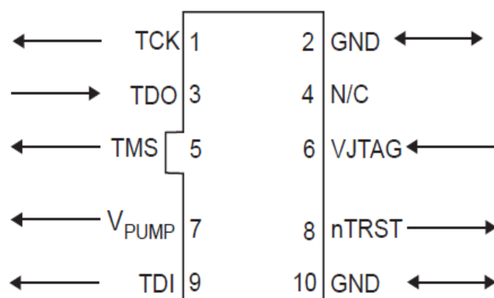
- The AHB-PPB. Through this bus, the debugger can access debug, including the following:
  - debug control
  - DW unit
  - BPU unit
  - The ROM Table
  - The AHB address space.

The AHB slaves in the debug system always expect 32-bit AHB transfers. If a byte or halfword access is created from the DAP, the transfer is extended to a 32-bit access and all 32 bits in the register are accessed. Figure 1 on page 9 shows an overview of how the debug system interacts with the rest of the processor.

4.3 JTAG Debug Interface

The Microsemi FlashPro4 programmer, which is used to program the FPGA and debug the CoreCortexM1 core using SoftConsole, uses a standard 10-pin JTAG interface, shown in Figure 5. Other debuggers, including those in the RealView and IAR tools, use a 20-pin, 2.54 mm pitch IDC connector (Figure 6 on page 15). The cable can be used to mate with a keyed box header on the target.

Figure 5 • JTAG 10-Pin Connector Pinout



The signals on the 10-pin JTAG interface are shown in Table 2.

Table 2 **• JTAG 10-Pin Connector Signals**

Signal	Description
VPUMP	3.3 V Programming voltage
GND	Signal reference
TCK	JTAG clock
TDI	JTAG data input to device
TDO	JTAG data output from device
TMS	JTAG mode select
nTRST	Programmable output pin may be set to Off, Toggle, Low, or High Level
VJTAG	Reference voltage from the target board
N/C	Programmer does not connect to this pin

Figure 6 • JTAG 20-Pin Connector Pinout

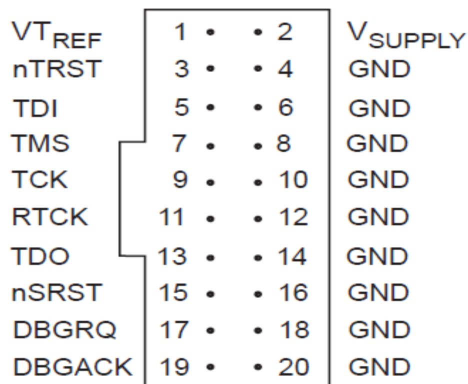


Table 3 • JTAG 20-Pin Connector Signals

Signal	I/O	Description
DBGACK	–	This pin is connected in the RealView ICE run control unit, but is not supported in the current release of the software. It is reserved for compatibility with other equipment to be used as a debug acknowledge signal from the target system. Microsemi recommends pulling this signal LOW on the target.
DBGRQ	–	This pin is connected in the RealView ICE run control unit, but is not supported in the current release of the software. It is reserved for compatibility with other equipment, to be used as a debug request signal to the target system. This signal is tied LOW. When applicable, RealView ICE uses the core's scanchain 2 to put the core in debug state. Microsemi recommends pulling this signal LOW on the target.
GND	–	Ground
nSRST	Input / output	Open collector output from RealView ICE to the target system reset. This is also an input to RealView ICE so that a reset initiated on the target can be reported to the debugger. This pin must be pulled HIGH on the target to avoid unintentional resets when there isno connection.
nTRST	Output	Open collector output from RealView ICE to the Reset signal on the target JTAG port. This pin must be pulled HIGH on the target to avoid unintentional resets when there is no connection.
RTCK	Input	Return Test Clock signal from the target JTAG port to RealView ICE. Some targets must synchronize the JTAG inputs to internal clocks. To assist in meeting this requirement, a returned, and retimed, TCK can be used to dynamically control the TCK rate. RealView ICE provides Adaptive Clock Timing, which waits for TCK changes to be echoed correctly before making further changes. Targets that do not have to process TCK can simply ground this pin.
TCK	Output	Test Clock signal from RealView ICE to the target JTAG port. Microsemi recommends pulling this pin LOW on the target.
TDI	Output	Test Data In signal from RealView ICE to the target JTAG port. Microsemi recommends pulling this pin HIGH on the target.
TDO	Input	Test Data Out from the target JTAG port to RealView ICE. Microsemi recommends pulling this pin HIGH on the target.
TMS	Output	Test Mode signal from RealView ICE to the target JTAG port. This pin must be pulled HIGH on the target to avoid adverse effects from any spurious TCKs when there is no connection.
VSUPPLY	Input	This pin is not connected in the RealView ICE run control unit. It is reserved for compatibility with other equipment to be used as a power feed from the target system.
VTREF	Input	This is the target reference voltage. It indicates that the target has power, and It must be at least 0.628 V. VTREF is normally fed from VDD on the target hardware and might have a series resistor (though this is not recommended). There is a 10 kB pull-down resistor on VTREF in RealView ICE.

4.4 Programmer's Model

The CoreCortexM1 processor implements a subset of the Thumb-2 (ARMv7) architecture called ARMv6-M. This includes all of the 16-bit Thumb-2 instructions and some of the 32-bit instructions. The processor does not support ARM instructions.

The Thumb-2 (ARMv7) instruction set architecture (ISA) was developed by ARM for the Cortex family of processors. Offering increased efficiency and performance, the Thumb-2 ISA differs from previous ARM architectures in that it includes both 16- and 32-bit instructions. The previous 16-bit Thumb instruction set and 32-bit ARM instruction set were separate and had to be executed from different modes within the processor. The Thumb-2 ISA gives users all the advantages of the reduced code size of the 16-bit Thumb instructions and the higher performance of the 32-bit ARM instructions. This is achieved in a single ISA that can be executed without requiring any context switching within the processor, increasing the efficiency of the code as it executes and improving the performance and throughput of the Cortex family of processors.

4.4.1 Processor Operating States

The CoreCortexM1 processor has two operating states:

- **Thumb state:** This is normal execution, running the set of 16-bit, halfword-aligned Thumb and Thumb-2 instructions; as well as the 32-bit BL, MRS, MSR, ISB, DSB, and DMB instructions.
- **Debug state:** This is the state when halting debug

4.4.2 Processor Operating Modes

The CoreCortexM1 processor supports two modes of operation:

- **Thread mode:** Entered on Reset, and can be re-entered as the result of an exception return
- **Handler mode:** Entered as the result of an exception

4.4.3 Main Stack and Process Stack Access

Out of reset, all code uses the main stack with the processor in Thread mode. A second (process) stack can be used in addition to the main stack. The stack pointer (R13) becomes a banked register.

4.4.4 Data Types

The processor supports the following data types:

- 32-bit words
- 16-bit halfwords
- 8-bit bytes

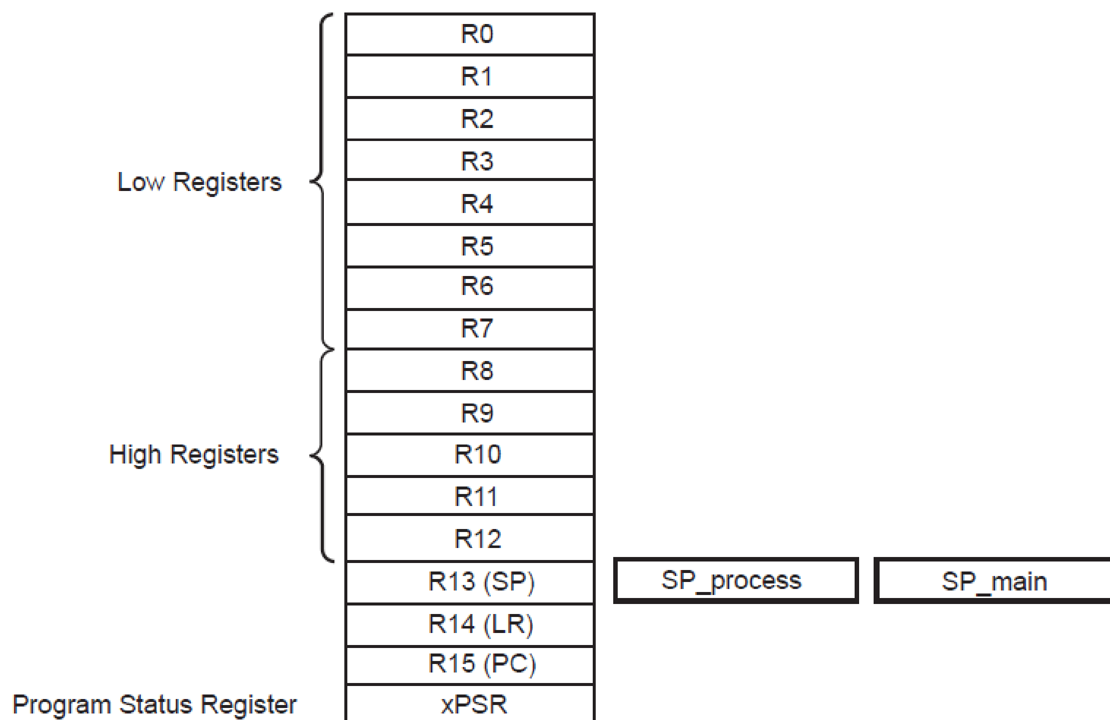
Note: Unless otherwise stated, the core can access all regions of the memory map, including the code region, with all data types. To support this, the system must support sub-word writes without corrupting neighboring bytes in that word (that is, individual byte enables for writes).

4.4.5 Registers

The processor has the following 32-bit registers (shown in Figure 7):

- 13 general purpose registers, R0–R12
- Stack Pointer (SP), R13
- Link Register (LR), R14
- Program Counter (PC), R15
- Program status registers, xPSR

Figure 7 • **Cortex -M1 Register Set**



4.4.6 General Purpose Registers

The general purpose registers, R0–R12, have no architecture-specific uses.

- **Low Registers:** Registers R0–R7 are accessible by all instructions that specify a general purpose register.
- **High Registers:** Registers R8–R12 are accessible by some, but not all 16-bit instructions. The R13, R14, and R15 registers have the following special functions:
  - **Stack Pointer:** Register R13 is used as the Stack Pointer (SP). Because the SP ignores writes to bits [1:0], it is auto-aligned to a word (four-byte) boundary. The stack pointer has banked aliases, SP_process and SP_main
  - **Link Register:** Register R14 is the subroutine Link Register (LR). The LR receives the return address from the Program Counter (PC) when a Branch and Link (BL) instruction is executed. The LR is also used for exception returns. At all other times, R14 can be treated as a general purpose register.
  - **Program Counter:** Register R15 is the Program Counter (PC). Bit [0] is always 0, so instructions are always aligned to 16-bit halfword (two-byte) boundaries.

4.4.7 Special Purpose Program Status Registers (xPSR)

Processor status at the system level is divided into three categories and can be accessed as individual registers, a combination of any two of the three, or a combination of all three using the MRS and MSR instructions.

- **Application PSR (APSR):** Contains the condition code flags. Before entering an exception, the processor saves the condition code flags on the stack. The APSR can be accessed with the MSR and MRS instructions.
- **Interrupt PSR (IPSR):** Contains the Interrupt Service Routine (ISR) number of the current exception
- **Execution PSR (EPSR):** Contains the Thumb state bit (T-bit). Unless the processor is in Debug state, the EPSR is not directly accessible and all fields read as zero using an MRS instruction. MSR instruction writes are ignored.

On entering an exception, the processor saves the combined information from the three status registers on the stack.

4.4.8 Special Purpose Priority Mask Register

Use the special purpose Priority Mask Register to boost execution priority. The special purpose Priority Mask Register can be accessed by using the MSR and MRS instructions. The CPS instruction to set or clear PRIMASK can also be accessed.

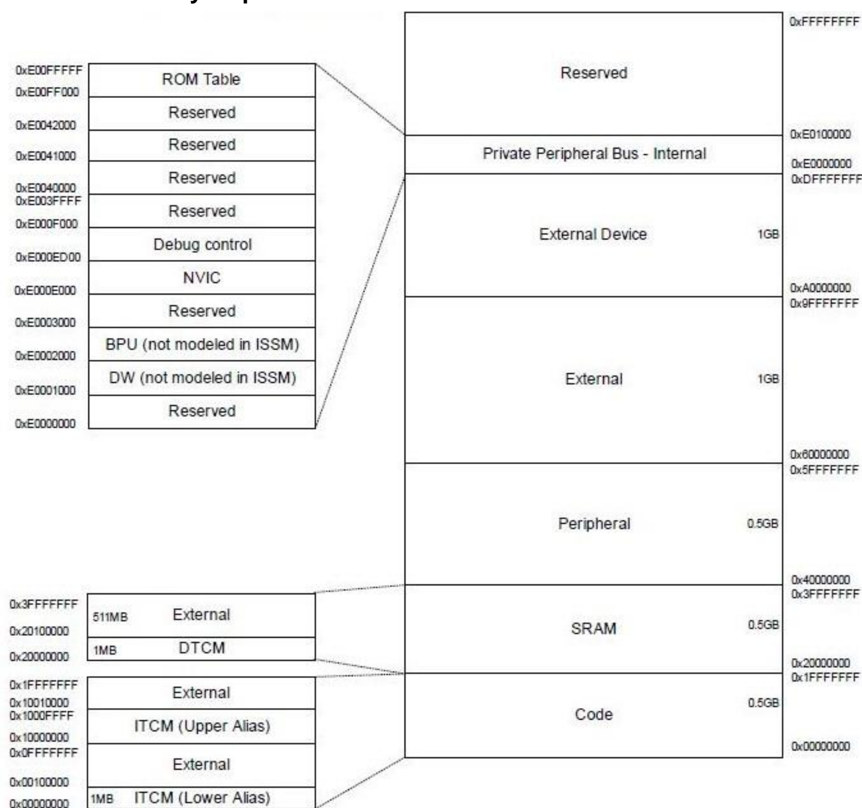
4.4.9 Special Purpose Control Register

The special purpose Control Register identifies the stack pointers usage.

4.4.10 Memory Map

CoreCortexM1 has a defined memory map with the various processor interfaces addressed by different memory map regions, as shown in Figure 8. The processor can access all regions within the memory map with the exception of the reserved regions. The reserved regions are Execute Never (XN) and instruction accesses are prevented by the processor hardware. The SRAM, Peripheral, External Device, and Private Peripheral Bus regions in the memory map are also XN. Instructions can be executed from the Code and External (not External Device) regions of the memory map.

Figure 8 • **CoreCortexM1 Memory Map**



The processor views memory as a linear collection of bytes numbered in ascending order from 0.

For example:

- Bytes 0–3 hold the first stored word
- Bytes 4–7 hold the second stored word

CoreCortexM1 always accesses code in little-endian format. Little-endian is the default memory format for ARM processors. The processor contains a configuration option that enables the user to select either the little-endian or big-endian format during implementation. Currently, only little-endian configurations of CoreCortexM1 are supported on Microsemi devices.

4.4.11 Subsystem Restrictions

The fixed memory map of the CoreCortexM1 places certain restrictions on the processor subsystem.

The Code region encompasses two CoreAHB/CoreAHBLite 256 MB slots (0 and 1). If the Data region of the memory map is being mapped to a device (RAM, for example), this must be mapped to AHB slot 2 or 3.

1. If the user has internal or external RAM mapped to the Data space, the external SRAM should be mapped up to the second SRAM space (0x60000000 and above).
2. Similarly, if the user wishes to run from internal or external SRAM, this SRAM should be mapped to 0x60000000 and above.
3. If the user wishes to run from NVM, NVM can be left at location 0x00000000. ITCM can be filled from NVM after reset and the ITCM then remapped to 0x00000000 using the Auxiliary Control register. From then on instruction fetches in the ITCM range would go to the ITCM, and instruction fetches above this space (but still within Code space) would go to NVM.

4.4.12 Exceptions

The processor and the Nested Vectored Interrupt Controller (NVIC) prioritize and handle all exceptions. All exceptions are handled in Handler mode. The processor state is automatically stored to the stack on an exception, and automatically restored from the stack at the end of the exception handler Interrupt Service Routine (ISR).

The following features enable efficient, low-latency exception handling:

- Automatic state saving and restoring. The processor pushes state registers on the stack before entering the ISR, and pops them after exiting the ISR with no instruction overhead.
- Automatic reading of the vector table entry that contains the ISR address in code memory or data SRAM
- Closely-coupled interface between the processor and the NVIC to enable early processing of interrupts and processing of late-arriving interrupts with higher priority
- One configurable interrupt
- Separate stacks for Handler and Thread modes if OS extensions are implemented
- ISR control transfer using the calling conventions of the C/C++ standard, Procedure Call Standard for the ARM Architecture (PCSAA)
- Priority masking to support critical regions

4.4.12.1 Exception Types

The types of exceptions supported in CoreCortexM1 are listed in Table 4. A fault is an exception that results from an error condition. Faults can be reported synchronously or asynchronously to the instruction that caused them. In general, faults are reported synchronously. Faults caused by writes over the bus are asynchronous faults. A synchronous fault is always reported with the instruction that caused the fault. An asynchronous fault may vary in how it is reported with respect to the instruction that caused the fault.

Table 4 • **CoreCortexM1 Exceptions**

Position	Exception Type	Priority	Description	Activated
–	–	–	Stack top is loaded from first entry of vector table on reset.	–
1	Reset	–3 (highest)	Invoked on power-up and warm reset. On first instruction, drops to lowest priority. Thread mode.	Asynchronous
2	Non- maskable Interrupt	–2	Cannot be marked, prevented by activation, by any other exception. Cannot be preempted by any other exception other than Reset.	Asynchronous
3	Hard Fault	–1	All classes of Fault.	Synchronous or asynchronous
4–10	–	–	Reserved.	–
11	SVCall	Configurable	System service call with SVC instruction.	Synchronous
12–13	–	–	Reserved.	–
14	PendSV	Configurable	Pendable request for system service. This is only pended by software.	Asynchronous
15	SysTick	Configurable	System tick timer has fired.	Asynchronous
16–48	External Interrupt	Configurable	Asserted from outside the processor, IRQ[2n-1:0], and fed through the NVIC (prioritized).	Asynchronous

4.4.12.2 Exception Priority

In the processor exception model, priority determines when and how the processor handles exceptions. Software priority levels can be assigned to interrupts.

The NVIC supports software-assigned priority levels. A priority level from 0 (highest) to 3 (lowest) can be assigned to an interrupt by writing to the two-bit IP_N field in an Interrupt Priority Register. Hardware priority ranges from –3 (highest), to 3 (lowest). By default, external interrupts have a hardware priority of 3, but hardware priority decreases with increasing interrupt number. The programmable priority level overrides the hardware priority. For example, IRQ(4) would have a default priority lower than IRQ(2), but if IRQ(4) is assigned a software priority of 1 and IRQ(2) is assigned 0, then IRQ(2) has priority over IRQ(4).

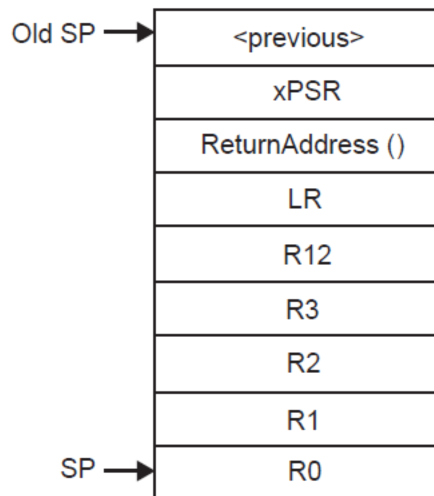
4.4.12.3 Servicing an Exception

When the processor invokes an exception, it automatically pushes the following eight registers in two stages to the stack in the following order:

1. Processor Status Register (xPSR)
2. ReturnAddress ()
3. Link Register (LR)
4. R12
5. R3
6. R2
7. R1
8. R0

The SP is decremented by eight words on the completion of the stack push. Figure 9 shows the contents of the stack after an exception preempts the current program flow.

Figure 9 • **Stack Contents from an Exception**

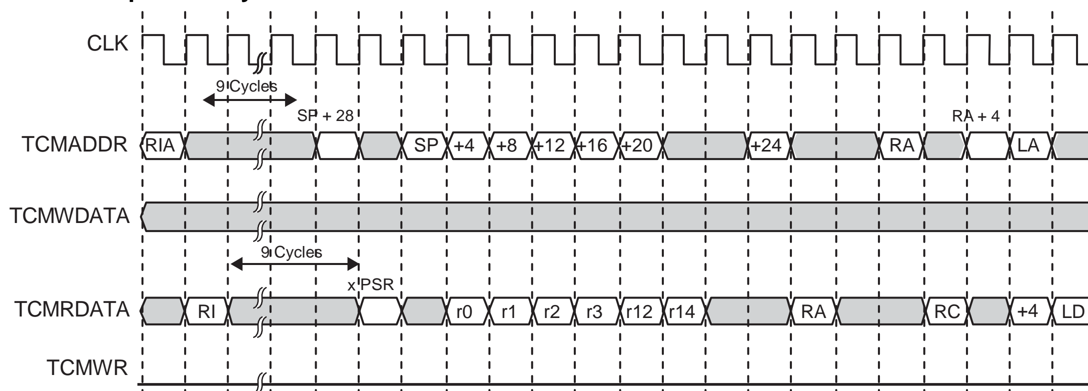


When the processor services an exception, it takes the following steps before it enters the exception service routine.

1. It pushes 8 registers: xPSR, ReturnAddress (), R0, R1, R2, R3, R12, and LR on the selected stack.
2. It reads the vector from the appropriate vector table entry, for example: (0x0) + (exception_number *4). This vector table read is done after all eight registers in the previous step are pushed onto the stack.
3. On Reset only, SP_main is updated from the first entry in the vector table. Other exceptions do not modify SP_main at this time and in this manner.
4. Updates PC with vector table read location. No other late-arriving exceptions can be processed until the first instruction of the exception starts to execute.
5. LR is set to EXC_RETURN to exit from the exception.

Figure 10 shows a timing example of an exception entry without wait states.

Figure 10 • Exception Entry Without Wait States



After returning from the exception, the processor automatically pops the eight registers from the stack. The interrupt return value, EXC_RETURN, passes as a data field in the LR, so exception functions can be normal C/C++ functions and do not require a veneer.

4.4.13 Nested Vectored Interrupt Controller

The NVIC facilitates low-latency exception and interrupt handling and implements System Control Registers. The NVIC supports reprioritizable interrupts. The NVIC and the processor core interface are closely coupled, which enables low-latency interrupt processing and efficient processing of late arriving interrupts. The NVIC registers are listed in Table 5 and can only be accessed using word transfers. Any attempt to write a halfword or byte individually causes corruption of the register bits. All NVIC registers and system debug registers are little-endian regardless of the endianness state of the processor.

Table 5 • CoreCortexM1 NVIC Registers

Name of Register	Type	Address	Reset Value
Irq 0 to 31 Set Enable Register	R/W	0xE000E100	0x00000000
Irq 0 to 31 Clear Enable Register	R/W	0xE000E180	0x00000000
Irq 0 to 31 Set Pending Register	R/W	0xE000E200	0x00000000
Irq 0 to 31 Clear Pending Register	R/W	0xE000E280	0x00000000
Priority 0 Register	R/W	0xE000E400	0x00000000
Priority 1 Register	R/W	0xE000E404	0x00000000
Priority 2 Register	R/W	0xE000E408	0x00000000
Priority 3 Register	R/W	0xE000E40C	0x00000000
Priority 4 Register	R/W	0xE000E410	0x00000000
Priority 5 Register	R/W	0xE000E414	0x00000000
Priority 6 Register	R/W	0xE000E418	0x00000000
Priority 7 Register	R/W	0xE000E41C	0x00000000

The processor supports both level and pulse interrupts. A level interrupt is held asserted until it is cleared by the ISR accessing the device. A pulse interrupt is a variant of an edge model. The edge must be sampled on the rising edge of the processor clock (HCLK) instead of being asynchronous.

For level interrupts, if the signal is not deasserted before the return from the interrupt routine, the interrupt remains pending and re-activates. This is particularly useful for FIFO and buffer-based devices because it ensures that they drain either by a single ISR or by repeated invocations, with no extra work. This means that the device continues to assert the signal until the device is empty.

A pulse interrupt must be asserted for at least one HCLK cycle to enable the NVIC to latch the pending bit.

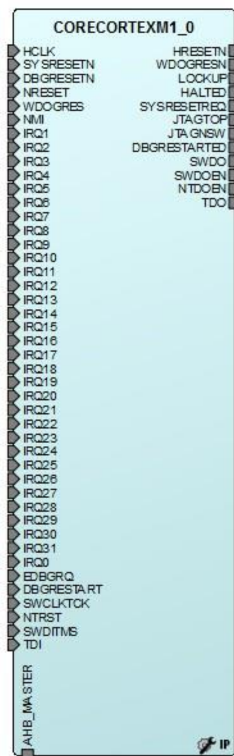
A pulse interrupt can be reasserted during the ISR so that the interrupt can be pending and active at the same time. If this occurs, the application design must ensure that a second pulse does not arrive before the first pulse is activated. The second pulse cannot set the pending bit and would have no effect, because the interrupt is already pending. However, if the interrupt is activated for at least one cycle, the NVIC latches the pending bit. After the ISR activates, the pending bit is cleared. After the bit is cleared if the interrupt is asserted again while it is activated, it can latch the pending bit again.

5 Interface

5.1 Ports

The section describes about the port signals for CoreCoretexM1.

Figure 11 • **CoreCortexM1 I/O Signals**



The port signals for CoreCortexM1 are described in Table 6 and shown in Figure 11.

Table 6 • **CoreCortexM1 Port Descriptions**

Name	Width	Type	Description
HCLK	1	Input	Main processor clock
SYSRESETN	1	Input	Active low system reset
HRESETn	1	Output	Reset output to other components in the system
WDOGRES	1	Input	"Bark" signal from watchdog
WDOGRESN	1	Output	Reset signal to watchdog
LOCKUP	1	Output	Status output which, when asserted, indicates that the processor is in the lock-up state.
HALTED	1	Output	Status output which, when asserted, indicates that the processor is in halting debug mode. This output is only functional when the core has been configured to include debug logic.
NMI	1	Input	Non-maskable interrupt

Table 6 • **CoreCortexM1 Port Descriptions**

IRQ0 TO irq31	32	Input	External interrupts 0 to 31
HADDR	32	Output	AHB-Lite address bus
HBURST	3	Output	AHB-Lite burst indication
HPROT	4	Output	AHB-Lite protection control signals
HRDATA	32	Input	AHB-lite read data bus
HREADY	1	Input	AHB-Lite "bus ready" signal
HRESP	2	Input	AHB-Lite response signal; indicates OKAY or ERROR status for each transfer on the bus.
HSIZE	3	Output	AHB-Lite size indication; byte, halfword, word, for example.
HTRANS	2	Output	AHB-Lite transfer type indication. Can be IDLE, BUSY, NONSEQUENTIAL OR SEQUENTIAL.
HWDATA	32	Output	AHB-Lite write data bus
HWRITE	1	Output	AHB-Lite transfer direction indication. High for a write transfer.
HMASTLOCK	1	Output	AHB-Lite signal that indicates if a transfer is part of a locked sequence.
EDBGRQ	1	Input	External debug request. This input is only functional when the core has been configured to include debug logic.
JTAGTOP	1	Output	Indicates state of JTAG controller. This output is only functional when the core has been configured to include debug.
JTAGNSW	1	Output	Indicates whether JTAG or Serial Wire (SW) based debug is in use. High = JTAG, low = SW. This output is only functional when the core has been configured to include debug.
SWDO	1	Output	Serial data output. This output is only functional when the core has been configured to include debug.
SWDOEN	1	Output	Active high serial data output enable. This output is only functional when the core has been configured to include debug.
NTDOEN	1	Output	Active low output enable for JTAG TDO signal. This output is only functional when the core has been configured to include debug.
SWCLKTCK	1	Input	JTAG clock input. This input is only functional when the core has been configured to include debug.
NTRST	1	Input	JTAG reset signal, active low. This input is only functional when the core has been configured to include debug.
SWDITMS	1	Input	JTAG test mode select. Also serves as serial data input when SW debug is in use. This input is only functional when the core has been configured to include debug.
TDI	1	Input	JTAG data input. This input is only functional when the core has been configured to include debug.
TDO	1	Output	JTAG data output. This output is only functional when the core has been configured to include debug.

5.2 Configuration Parameters

5.2.1 CoreCortexM1 Configurable Options

There are a number of configurable options that apply to CoreCortexM1 as shown in Table 7. If a configuration other than the default is required, select the configuration dialog box in SmartDesign to select appropriate values for the configurable options.

Table 7 • **CoreCortexM1 Configuration Options**

Name	Valid Range	Default	Description
Debug Interface	0 or 1	1	To select the type of JTAG Interface 0 = FlashPro 1 = ARM JTAG (e.g. J-Link/ U-Link)
Include reset control logic	0 or 1	1	Select to include reset control logic 0: Disable 1: Enable
Include BFM	0 or 1	0	To select the BFM 0: Disable 1: Enable

6 Tool Flow

SoftConsole is the free-of-charge Microsemi software development environment that allows quick turn-around for C and C++ based projects targeting CoreCortexM1 and other processor-based platforms available for use in Microsemi devices. SoftConsole allows users to create a project and transparently manages all compilation stages in order to generate a binary file ready to be used with the CoreCortexM1 processor. SoftConsole includes a fully integrated debugger that offers easy access to memory contents, registers, and single- instruction execution. Programs developed with SoftConsole can be debugged on a target board or in the tool's simulator using the same uniform interface.

SoftConsole provides a flexible and easy-to-use graphical user interface for managing your software development projects. The tool gives you the ability to quickly develop and debug software programs and to implement them in Microsemi devices. SoftConsole enables users to edit and debug software programs, organize files, and configure settings in a project. This tool provides simultaneous access to multiple tool windows and the ability to quickly switch editing, debug, and synchronization views.

The compilation tools can be used to build C or C++ programs.

The following tools are included in SoftConsole:

- SoftConsole Eclipse-based IDE
- GCC Compiler
- GDB Debugger
- Support for program download and debug with FlashPro4

There are other tools available from ARM and third-party companies that can be purchased, including the RealView Development Suite, RealView Microcontroller Development Kit, and embedded workbench tools from IAR. The RealView and IAR tools feature compilers that offer a higher level of efficiency than the GCC compiler included in SoftConsole. If higher code density is required for an application than what can be achieved using GNU, Microsemi recommends that one of these other tools be purchased and used.

6.1 License

CoreCortexM1 is licensed under the terms of the ARM Cortex-M1 end-user license agreement.

6.1.1 RTL

Complete obfuscated encrypted netlist is provided for the core and test bench.

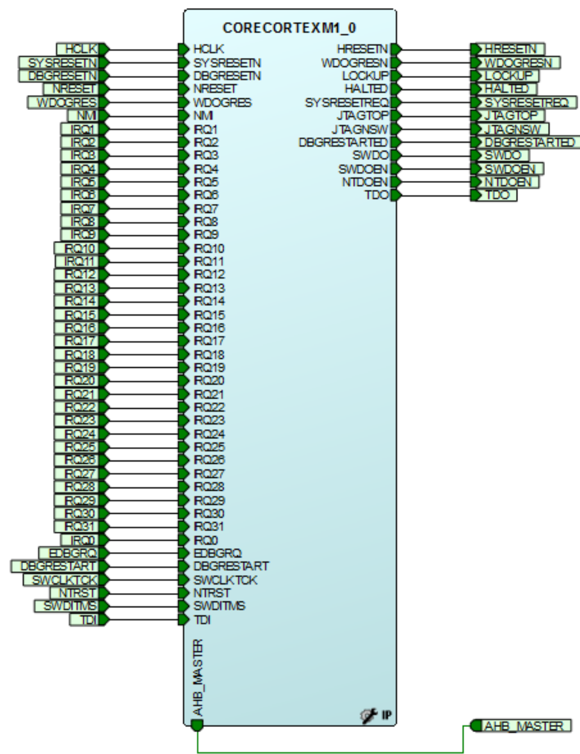
6.2 SmartDesign

CoreCortexM1 is available for download in the Libero IP catalog through the web repository. Once it is listed in the catalog, the core can be instantiated using the SmartDesign flow. For information on using SmartDesign to configure, connect, and generate cores, refer to the Libero online help. An example instantiated view is shown in Figure 12.

After configuring and generating the core instance, basic functionality can be simulated using the test-bench supplied with the CoreCortexM1. The testbench parameters automatically adjust to the CoreCortexM1 configuration. The CoreCortexM1 can be instantiated as a component of a larger design.

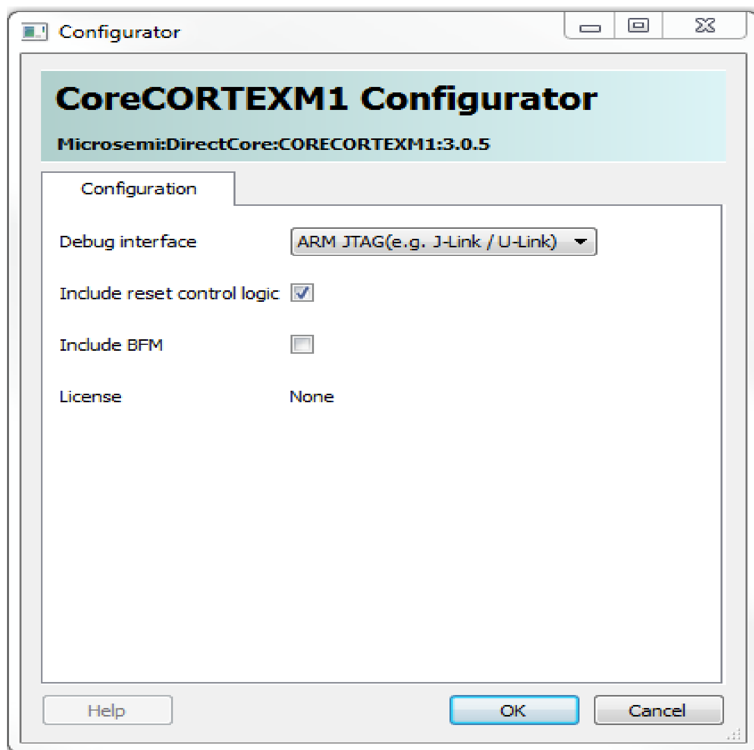
CoreCortexM1 is compatible with Libero SoC. For more information on using SmartDesign to instantiate and generate cores, refer to the Using DirectCore in Libero® System-on-Chip (SoC) User Guide or consult the Libero SoC online help.

Figure 12 • **SmartDesign CoreCortexM1 Instance View**



6.2.1 Configuring CoreCortexM1 in SmartDesign

Figure 13 • **SmartDesign CoreCortexM1 Configuration window**



6.3 Simulation Flows

The BusFunctional Model (BFM) for CoreCortexM1 is included in all releases.

To run simulations, select the **BFM** and click **Save** and **Generate** on the **Generate** pane.

When SmartDesign generates the Libero SoC project, it installs the BFM files.

To run the **BFM**, click the **Simulation** icon in the Libero SoC design flow window. This invokes ModelSim® and automatically run the simulation.

6.4 Synthesis in Libero

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

6.5 Place-and-Route in Libero

Click the **Layout** icon in the Libero SoC to invoke Designer. CoreCortexM1 requires no special place-and-route settings.

7 BFM Usage Flow

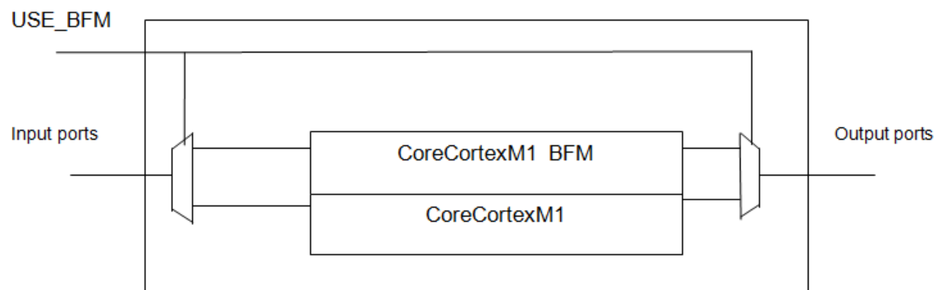
The BFM is part of an overall system test strategy, this section describes the usage of BFM. It is assumed that the processor subsystem has been specified by selecting the processor, bus fabric, IP blocks, and the memory system using SmartDesign.

Based on the bus connections made, SmartDesign can build up a memory map for the CoreCortexM1 system.

SmartDesign generates the following outputs:

- Encrypted core
- CoreCortexM1 BFM
- BFM test bench script
- System-level skeleton testbench

Figure 14 • **Simulation Environment for a CoreCortexM1 System Inside Libero IDE Board Level System**



Set the USE_BFM parameter from the Configurator to use the CoreCortexM1 BFM in pre-synthesis simulation. The BFM acts as a replacement for the CoreCortexM1 in the project system. It initiates cycle-accurate bus transactions on the native CoreCortexM1 bus. It has no knowledge about the real CoreCortexM1 instructions.

At this point, the BFM may be used to run a basic test of the system using the skeleton system with the BFM script serving as a stimulus for the simulation. This script does a write to and/or read from all accessible locations. It has knowledge of whether registers are read-only, read/write, clear- on-read, or write-only. From this, it can decide what the expected data should be on reads.

The system Verilog/VHDL can be edited to add new design blocks. The system level testbench can be edited to include tasks that test any newly added functionality, or for adding stubs to allow more complex system testing involving the IP cores. The BFM input scripts may also be manually enhanced, so that, you can test access to register locations in newly added logic. Using this method, stimuli can be provided to the system from the inside (through the CortexM1 BFM), as well as from the outside (testbench tasks).

Note: The simulator executes automatically a TCL script that converts a BFM script (*.bfm) into binary instructions (*.vec files).

Note: The *.vec files are automatically read by the CoreCoreCortexM1 BFM and executed.

The BFM generates output messages to the console of the simulation tool.

7.1 Functionality

This section describes the specific functionality of the CoreCortexM1 BFM. The BFM models transactions on the external (AHB-Lite) bus of CoreCortexM1.

7.1.1 CoreCortexM1 Pin Compatibility

The BFM model is pin-for-pin compatible with the CoreCortexM1. This allows the model to be dropped into the space that would be occupied by the processor core in the system testbench.

7.1.2 CoreCortexM1 Bus Cycle Accuracy

The bus cycle timings for the CoreCortexM1 external bus signals are specified in the CoreCortexM1 Technical Reference Manual. The CoreCortexM1 BFM models these bus cycles .

7.1.3 Scripting

In order to provide a simple and extensible mechanism for providing stimuli to the BFM, a BFM scripting language is defined (see "BFM Script Language" section on page 32). This allows initiating writes to system resources, reads from system resources (with or without checking of expected data), and waiting for interrupt events.

7.1.4 Self-Checking

The BFM gives a pass/fail indication at the end of a test run. This is based on whether or not any of the expected data read checks failed.

7.1.5 Endianness

The BFM supports both big and little-endian memory configurations. For byte and halfword transfers, it reads and writes data from/to the appropriate data lanes.

7.1.6 Interrupt Support

The BFM has the ability to wait for the CoreCortexM1 interrupt lines to be triggered before proceeding with the remainder of the test script.

7.1.7 Log File Generation

The BFM generates output messages to the console of the simulation tool, and also generates an HTML log file. The messages in this file are color-coded so that any errors can be easily identified.

7.1.8 BFM Script Language

The following script commands are defined for use by the BFM:

7.1.8.1 Write

The write command causes the BFM to perform a write to a specified offset, within the memory map range of a specified system resource.

Syntax

```
write width resource_name byte_offset data;
```

Width

This takes on the enumerated values of W, H or B, for word, halfword, or byte respectively.

resource_name

This is a string containing the user-friendly instance name of the resource being accessed.

byte_offset

This is the offset from the base of the resource, in bytes. It is specified as a hexadecimal value.

Data

This is the data to be written. It is specified as a hexadecimal value.

Example

```
write W videoCodec 20 11223344;
```

7.1.8.2 Read

The read command causes the BFM to perform a read of a specified offset, within the memory map range of a specified system resource.

Syntax

```
read width resource_name byte_offset;
```

Width

This takes on the enumerated values of W, H or B, for word, halfword, or byte respectively.

resource_name

This is a string containing the user-friendly instance name of the resource being accessed.

byte_offset

This is the offset from the base of the resource, in bytes. It is specified as a hexadecimal value.

Example

```
read W videoCodec 20;
```

7.1.8.3 Readcheck

The readcheck command causes the BFM to perform a read of a specified offset, within the memory map range of a specified system resource and to compare the read value with the expected value provided.

Syntax

```
readcheck width resource_name byte_offset data;
```

Width

This takes on the enumerated values of W, H or B, for word, halfword, or byte respectively.

resource_name

This is a string containing the user-friendly instance name of the resource being accessed.

byte_offset

This is the offset from the base of the resource, in bytes. It is specified as a hexadecimal value.

Data

This is the expected read data. It is specified as a hexadecimal value.

Example

```
readcheck W videoCodec 20 11223344;
```

7.1.8.4 poll

This command continuously reads a specified location until a requested value is obtained. This command allows one or more bits of the read data to be masked out. This allows, for example, poll waiting for a ready bit to be set, while ignoring the values of the other bits in the location being read.

Syntax

```
poll width resource_name byte_offset data_bitmask;
```

width

This takes on the enumerated values of W, H, or B, for word, halfword, or byte.

resource_name

This is a string containing the user-friendly instance name of the resource being accessed.

byte_offset

This is the offset from the base of the resource, in bytes. It is specified as a hexadecimal value.

bitmask

The bitmask is ANDed with the read data and the result is then compared to the bitmask itself. If equal, then all the bits of interest are at their required value and the poll command is complete. If not equal, then the polling continues.

7.1.8.5 wait

This command causes the BFM script to stall for a specified number of clock periods.

Syntax

```
wait num_clock_ticks;
```

num_clock_ticks

This is the number of CoreCortexM1 clock periods, during which the BFM stalls (doesn't initiate any bus transactions).

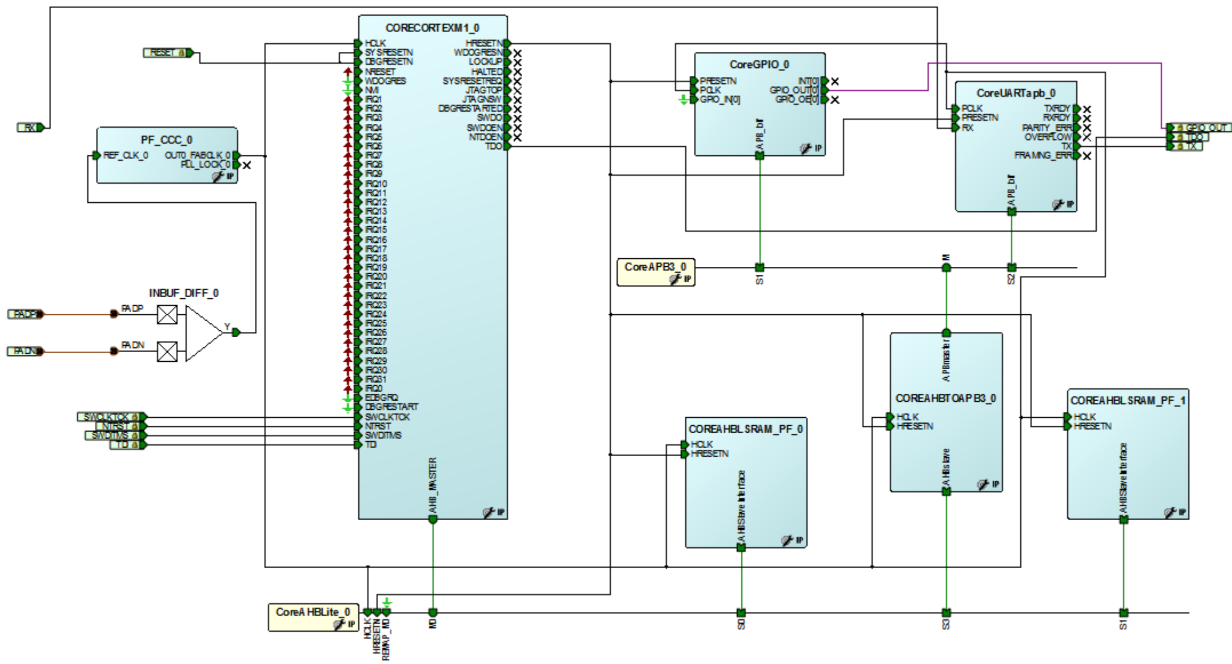
8 System Integration

8.1 Integration of CoreCortexM1 with the example design

This section provides hints to ease the integration of CoreCortexM1.

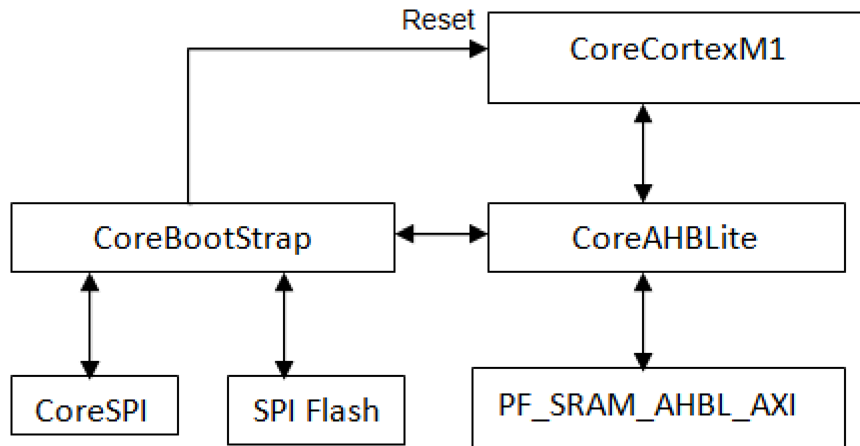
- The example design contains CORECORTEXM1_0, this core is interfaced with CoreGPIO_0 and CoreUARTapb_0 through CoreAHBLite_0, CoreAHBtoAPB3_0, and CoreAPB3_0.
- External reset pin (RESET) is used to reset CORECORTEXM1_0 and the HRESETN of CORECORTEXM1_0 is used to reset all other modules in the design.
- The CORECORTEXM1_0 has HCLK of 50MHz, which is driven from PF_CCC_0/OUT0_FABCLK_0.
- GPIO_OUT [0] pin of CoreGPIO_0 is connected to LED on the board and CoreUARTapb_0 is used for the console purpose.
- The application was not tested with this design.

Figure 15 • **CoreCortexM1 Example Design**



8.1.1 Interface with Core Bootstrap

The CoreBootStrap is Microsemi DirectCore IP which can optionally be used as a first level boot loader for the CoreCortexM1. The CoreBootStrap has a master interface to connect to a CoreAHBLite bus. The diagram below explains how the CoreBootStrap can be used in the CoreCortexM1 system. The CoreBootStrap has a SPI master interface. It uses this interface to access the on-board SPI Flash via CoreSPI. The CoreBootStrap IP acts as a first-stage boot loader. It holds the CoreCortexM1 in reset while it copies the application image from on-board SPI flash to the PF_SRAM_AHBL_AXI memory after system reset. It then release the reset to the CoreCortexM1 allowing it to boot from the PF_SRAM_AHBL_AXI memory. The interface diagram is as shown in below figure.

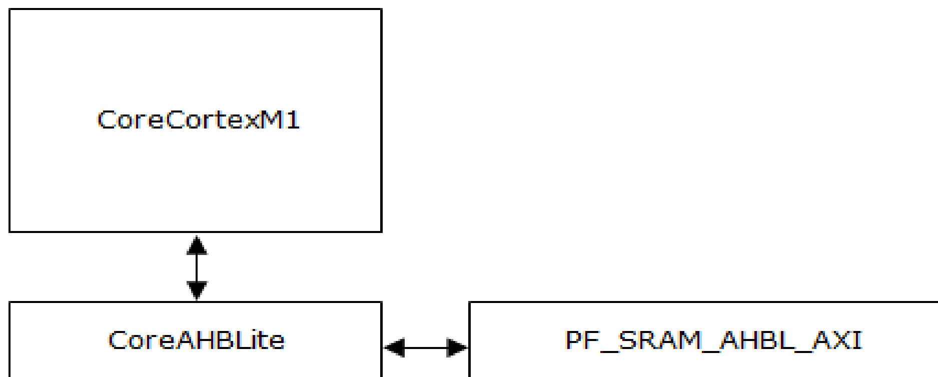


Below is the sequence of actions:

- Executable code should be available in SPI Flash.
- On System reset, the CoreBootStrap asserts the reset line of the CoreCortexM1 processor, holding it in reset as it copies the processor's executable program from an external SPI flash device into the processor's instruction memory space.
- Assuming no errors, the CoreBootStrap then releases the CoreCortexM1 processor's reset line, allowing the CoreCortexM1 to boot directly from on-chip memory.

8.1.2 Interface with PF_SRAM_AHBL_AXI

The PF_SRAM_AHBL_AXI is used in the design to run the executable code. This executable code is downloaded into PF_SRAM_AHBL_AXI memory by using debugger (FlashPro). The PF_SRAM_AHBL_AXI should be interfaced with CoreCortexM1 through CoreAHBLite bridge. The interface diagram is as shown in below figure



If PF_SRAM_AHBL_AXI is used to store the executable code for CoreCortexM1, then PF_SRAM_AHBL_AXI should fall in the memory space as mentioned in memory map section of CoreCortexM1 and need to follow the below steps for loading the application on PF_SRAM_AHBL_AXI.

- Import the executable file output from SoftConsole/IAR project in the PF_SRAM_AHBL_AXI IP block using the memory initialization menu in the configurator.
- Generate the UIC script as the memory client, using “Design and Memory Initialization” menu in the Design flow.
- Generate the bit-stream and load it on to the board. This will now include the memory client which will be used to initialize the PF_SRAM_AHBL_AXI on reset.
- The CoreCortexM1 will now execute your executable file output from SoftConsole/IAR project from PF_SRAM_AHBL_AXI IP after reset

9 Ordering Information

9.1 Ordering Codes

CoreCortexM1 is available for free but the user needs to fill up the license agreement to download the ordering code given below.

Use the following number convention when ordering: CoreCortexM1. XX is listed in Table 9.

Table 9 • **Ordering Codes**

XX	Description
OM	Available as obfuscated encrypted netlist only.