

# SCR1 External Architecture Specification

Syntacore, [info@syntacore.com](mailto:info@syntacore.com)

Version 1.2.2, 2019-08-30

# Table of Contents

Revision history .....	1
1. Overview .....	2
1.1. MIMPID (core implementation ID) .....	2
1.2. Features .....	2
1.3. Core configuration .....	3
1.4. Block Diagram .....	5
2. Privilege Levels .....	7
3. Registers .....	8
3.1. General-purpose Integer Registers .....	8
3.2. Control and Status Registers .....	9
3.2.1. Overview and definitions .....	9
3.2.2. CSR Map .....	10
3.2.3. User Mode CSRs .....	12
3.2.4. Machine Mode Standard CSRs .....	13
3.2.4.1. MVENDORID [0xF11] .....	13
3.2.4.2. MARCHID [0xF12] .....	13
3.2.4.3. MIMPID [0xF13] .....	13
3.2.4.4. MHARTID [0xF14] .....	13
3.2.4.5. MSTATUS [0x300] .....	13
3.2.4.6. MISA [0x301] .....	13
3.2.4.7. MIE [0x304] .....	14
3.2.4.8. MTVEC [0x305] .....	14
3.2.4.9. MSCRATCH [0x340] .....	15
3.2.4.10. MEPC [0x341] .....	15
3.2.4.11. MCAUSE [0x342] .....	15
3.2.4.12. MTVAL [0x343] .....	16
3.2.4.13. MIP [0x344] .....	16
3.2.4.14. MCYCLE/MCYCLEH [0xB00/0xB80] .....	17
3.2.4.15. MINSTRET/MINSTRETH [0xB02/0xB82] .....	17
3.2.5. Standard read/write debug CSRs [0x7A0..0x7AF] .....	17
3.2.6. Debug-mode-only CSRs [0x7B0..0x7BF] .....	17
3.2.7. Machine Mode Non-standard CSRs .....	18
3.2.7.1. MCOUNTEN [0x7E0] .....	18
3.2.7.2. IPIC registers [0xBF0..0xBF7] .....	18
3.2.8. Memory-mapped CSRs .....	19
3.2.8.1. TIMER_CTRL [TIMER_BASE] .....	19
3.2.8.2. TIMER_DIV [TIMER_BASE + 0x4] .....	19
3.2.8.3. MTIME/MTIMEH [TIMER_BASE + 0x8/TIMER_BASE + 0xC] .....	19

3.2.8.4. MTIMECMP/MTIMECMPH [TIMER_BASE + 0x10/TIMER_BASE + 0x14] .....	19
4. Memory Model .....	21
4.1. Bit and byte order .....	21
4.2. Data access width and alignment .....	21
4.3. Stack behavior .....	22
4.4. Memory access ordering .....	22
4.5. System memory map .....	22
4.6. Tightly-Coupled Memory .....	24
5. Exceptions and Interrupts .....	25
6. Pipeline theory of operations .....	27
6.1. Instruction execution phases .....	27
6.1.1. Request to Instruction Memory .....	27
6.1.2. Instruction fetch .....	27
6.1.3. Instruction decode .....	27
6.1.4. Operand fetch .....	28
6.1.5. Arithmetical and logical operations .....	28
6.1.6. Load/store operations .....	28
6.1.7. Instruction flow control .....	28
6.1.8. Commit point .....	29
6.2. Pipeline configurations .....	29
6.2.1. 2-stage pipeline .....	29
6.2.2. 3-stage pipeline .....	30
6.2.3. 4-stage pipeline .....	31
6.3. Hazards handling .....	32
6.3.1. Data hazards .....	32
6.3.2. Structural hazards .....	32
6.3.3. Control hazards .....	32
7. Integrated Programmable Interrupt Controller .....	33
7.1. Introduction .....	33
7.2. IPIC Block Diagram and description .....	34
7.3. IPIC Programming Model .....	34
7.3.1. Register Map .....	34
7.4. Detailed IPIC Registers Description .....	35
7.4.1. IPIC_CISV: Current Interrupt Vector in Service .....	35
7.4.2. IPIC_CICSR: Current Interrupt Control Status Register .....	36
7.4.3. IPIC_IPR: Interrupt Pending Register .....	36
7.4.4. IPIC_ISVR: Interrupt Serviced Register .....	37
7.4.5. IPIC_EOI: End Of Interrupt .....	37
7.4.6. IPIC_SOI: Start Of Interrupt .....	37
7.4.7. IPIC_IDX: Index Register .....	38
7.4.8. IPIC_ICSR: Interrupt Control Status register .....	38

7.5. IPIC timing diagrams . . . . .	40
8. Debug . . . . .	41
8.1. Overview . . . . .	41
8.2. TAP Controller (TAPC) . . . . .	42
8.2.1. JTAG frequency requirement . . . . .	42
8.2.2. TAPC Instruction Register (IR) . . . . .	42
8.2.3. TAPC Instructions . . . . .	43
8.2.4. TAPC Data Registers . . . . .	43
8.2.4.1. IDCODE . . . . .	43
8.2.4.2. BYPASS . . . . .	43
8.2.4.3. DTMCS (DTM Control and Status) . . . . .	44
8.2.4.4. DMI_ACCESS (DMI) . . . . .	45
8.2.4.5. SCU_ACCESS . . . . .	47
8.3. System Control Unit (SCU) . . . . .	49
8.3.1. Overview . . . . .	49
8.3.2. Block Diagram . . . . .	49
8.3.3. Registers . . . . .	52
8.3.3.1. CONTROL . . . . .	53
8.3.3.2. MODE . . . . .	53
8.3.3.3. STATUS . . . . .	54
8.3.3.4. STICKY_STATUS . . . . .	54
8.4. Debug Module (DM) . . . . .	55
8.4.1. Overview . . . . .	55
8.4.2. Debug Module Interface (DMI) . . . . .	56
8.4.3. Hart States . . . . .	56
8.4.4. Reset Control . . . . .	56
8.4.5. Run Control . . . . .	57
8.4.6. Abstract Commands . . . . .	57
8.4.6.1. Supported Abstract Commands . . . . .	58
8.4.6.2. Access Register . . . . .	58
8.4.6.3. Access Memory . . . . .	60
8.4.7. Program Buffer . . . . .	61
8.4.8. DM Registers . . . . .	62
8.4.8.1. Register Map . . . . .	62
8.4.8.2. Debug Module Control (DMCONTROL) . . . . .	62
8.4.8.3. Debug Module Status (DMSTATUS) . . . . .	63
8.4.8.4. Hart Info (HARTINFO) . . . . .	64
8.4.8.5. Halt Summary 0 (HALTSUM0) . . . . .	65
8.4.8.6. Abstract Control and Status (ABSTRACTCS) . . . . .	65
8.4.8.7. Abstract Command (COMMAND) . . . . .	66
8.4.8.8. Abstract Command Autoexec (ABSTRACTAUTO) . . . . .	67

8.4.8.9. Abstract Data 0/1 (DATA0/1) .....	67
8.4.8.10. Program Buffer [0:5] (PROGBUF[0:5]) .....	68
8.5. Hart Debug Unit (HDU) .....	68
8.5.1. Overview .....	68
8.5.2. Debug Mode .....	68
8.5.3. Reset .....	69
8.5.4. Single Step .....	69
8.5.5. Debug CSRs .....	70
8.5.5.1. Register Map .....	70
8.5.5.2. Debug Control and Status (DCSR) .....	70
8.5.5.3. Debug PC (DPC) .....	72
8.5.5.4. Debug Scratch Register 0 (DSCRATCH0) .....	72
9. Hardware Trigger Module .....	73
9.1. Overview .....	73
9.2. Reset .....	73
9.3. Operation Basics .....	73
9.3.1. Enumeration .....	73
9.4. Trigger CSRs .....	73
9.4.1. Register Map .....	74
9.4.2. Trigger Select (TSELECT) .....	74
9.4.3. Trigger Data 1 (TDATA1) .....	75
9.4.4. Match Control (MCONTROL) .....	75
9.4.5. Instruction Count (ICOUNT) .....	77
9.4.6. Trigger Data 2 (TDATA2) .....	78
9.4.7. Trigger Info (TINFO) .....	78
10. External Interfaces .....	80
10.1. AHB-Lite Interface .....	80
10.2. AHB-Lite Timing diagrams .....	81
10.3. AXI4 Interface .....	82
10.4. AXI4 Timing diagrams .....	89
10.5. Control Interface .....	92
10.6. JTAG Interface .....	92
10.7. IRQ Interface .....	92
11. Clocks and Resets .....	94
11.1. Clock Distribution .....	94
11.2. Power saving features .....	96
11.2.1. Global clock gating in wait-for-interrupt state .....	96
11.2.2. Software control of performance counters .....	96
11.3. Core Reset Circuit .....	96
12. Initialization .....	98
12.1. Reset .....	98

12.2. C-runtime code example.....	99
13. Instruction set summary .....	102
Referenced documents .....	107

# Revision history

Revision	Date	Description
1.0.0	2017-05-08	Initial version
1.0.1	2017-05-09	The reference links fixed, Instruction List tables updated
1.1.0	2017-07-12	Updated to comply with privileged ISA specification v1.10 and user ISA specification v2.2
1.1.1	2017-08-18	Changed debug scratch CSR address; updated core configuration chapter; TAPC Target_ID register added
1.1.2	2017-09-07	Changes to timer registers
1.1.3	2017-09-14	Partially writable MTVEC CSR
1.1.4	2017-10-10	Added AXI4 interface
1.1.5	2018-01-30	Added initial pipeline theory of operations; updated Exceptions and Interrupts chapter and IPIC chapter
1.1.6	2018-02-19	Updated Clocks and Resets chapter; added core counters description
1.1.7	2018-03-14	Updated MIMPID, vectored interrupts on by default
1.1.8	2018-05-07	Updated MIMPID
1.1.9	2018-09-19	RTL configurations update
1.1.10	2018-10-09	Updated MIMPID
1.1.11	2018-11-07	Changed MARCHID to 0x7
1.1.12	2019-01-17	Instruction retirement delay tables update
1.2.0	2019-03-19	Changed MARCHID to 0x8 Debug subsystem updated to comply with 0.13 debug spec Reset sub-system is modified
1.2.1	2019-03-04	Corrected SCU block-diagram Updated MIMPID
1.2.2	2019-08-30	Updated MIMPID=0x19083000

# 1. Overview

## 1.1. MIMPID (core implementation ID)

This specification is relevant for SCR1 core with MIMPID value of 0x19083000.

## 1.2. Features

Summary of key features:

- Harvard architecture (separate instruction and data buses)
- Machine privilege level
- 32 or 16 32-bit general purpose integer registers
- Instruction set is RV32I/RV32E with optional M and C extensions
  - 47 Integer (32-bit) instructions
  - 27 Compact (16-bit) instructions
  - 8 Multiply/Divide instructions
- Configurable high-performance or area-optimized multiply/divide unit
- Configurable 2 to 4 stage pipeline implementation
- 32-bit AXI4/AHB-Lite external memory interface
- Tightly coupled memory support
- Optional Integrated Programmable Interrupt Controller
  - Low interrupt latency
  - up to 16 IRQ lines
- Optional RISC-V Debug Module with JTAG interface
- Optional Hardware Trigger Module
- 3 embedded 64bit performance counters
  - Real time clock
  - Cycle counter
  - Instructions-retired counter
- Optimized for area and power consumption



## 1.3. Core configuration

The core features a number of configurable parameters described in [Table 1](#). These parameters can be changed in `scr1_arch_description.svh` include file.

- for on/off parameters, comment/uncomment the ``define` directive
- for numeric parameters, change the SystemVerilog parameter value

Table 1: SCR1 configurable options

Name	Description
<b>ISA options</b>	
SCR1_RVE_EXT	Enable RV32E base integer instruction set; when this option is disabled, RV32I base is used
SCR1_RVM_EXT	Enable M extension (hardware multiplication and division)
SCR1_RVC_EXT	Enable C extension
<b>Core options</b>	
SCR1_IFU_QUEUE_BYPASS	Pipeline bypass after IFU (see "Pipeline configurations" in <a href="#">docs/scr1_eas.pdf</a> )
SCR1_EXU_STAGE_BYPASS	Pipeline bypass before EXU (see "Pipeline configurations" in <a href="#">docs/scr1_eas.pdf</a> )
SCR1_FAST_MUL	Enable fast one-cycle multiplication; when this option is disabled, multiplication takes 32 cycles
SCR1_CLKCTRL_EN	Enable global clock gating; please note that for synthesis, code in <code>scr1_cg.sv</code> should be replaced with implementation-specific clock gate cod
SCR1_VECT_IRQ_EN	Enable vectored mode (see <a href="#">MTVEC [0x305]</a> )
SCR1_CSR_MCOUNTEN_EN	Enable counter control CSR (see <a href="#">MCOUNTEN [0x7E0]</a> )
SCR1_CSR_MTVEC_BASE_RW_BITS	Number of writable bits in MTVEC BASE field (see <a href="#">MTVEC [0x305]</a> )
<b>Uncore options</b>	
SCR1_DBGC_EN	Enable Debug Subsystem (TAPC, DM, SCU, HDU) (see <a href="#">Debug</a> )
SCR1_BRKM_EN	Enable Trigger Module (see <a href="#">Hardware Trigger Module</a> )
SCR1_BRKM_BRKPT_NUMBER	Number of hardware triggers/breakpoints
SCR1_IPIC_EN	Enable interrupt controller (see <a href="#">Integrated Programmable Interrupt Controller</a> )
SCR1_IPIC_SYNC_EN	Enable 2-stage input synchronizer for IRQ lines
SCR1_CFG_EXCL_UNCORE	Exclude Debug Subsystem, Trigger Module, IPIC
SCR1_TCM_EN	Enable tightly-coupled memory, default size is 64K
SCR1_IMEM_AHB_IN_BP	Enable bypass on instruction memory AHB bridge inputs

Name	Description
SCR1_IMEM_AHB_OUT_BP	Enable bypass on instruction memory AHB bridge outputs
SCR1_DMEM_AHB_IN_BP	Enable bypass on data memory AHB bridge inputs
SCR1_DMEM_AHB_OUT_BP	Enable bypass on data memory AHB bridge outputs
SCR1_IMEM_AXI_REQ_BP	Enable bypass on instruction memory AXI bridge request
SCR1_IMEM_AXI_RESP_BP	Enable bypass on instruction memory AXI bridge response
SCR1_DMEM_AXI_REQ_BP	Enable bypass on data memory AXI bridge request
SCR1_DMEM_AXI_RESP_BP	Enable bypass on data memory AXI bridge response
<b>Address constants</b>	
SCR1_ARCH_RST_VECTOR	User-defined reset vector (default 0x200)
SCR1_ARCH_CSR_MTVEC_BASE	MTVEC BASE field reset value, or constant value for MTVEC BASE bits that are hardwired (default 0x1C0)
SCR1_TCM_ADDR_MASK	Set TCM mask and size; size in bytes is two's complement of the mask value (default 0xFFFF0000)
SCR1_TCM_ADDR_PATTERN	Set TCM address match pattern (default 0x00480000)
SCR1_TIMER_ADDR_MASK	Set timer mask (default 0xFFFFFFE0)
SCR1_TIMER_ADDR_PATTERN	Set timer address match pattern (default 0x00490000)

#### NOTE

Currently Trigger Module requires Debug Subsystem and vice versa, so both options should be either enabled or disabled.

## 1.4. Block Diagram

The core is load-store architecture, where only load and store instructions access memory and arithmetic instructions only operate on integer registers. The core provides a 32-bit user address space that is byte-addressed and little-endian. The execution environment will define what portions of the address space are legal to access.

Block diagram of the core is shown in [Figure 1](#).

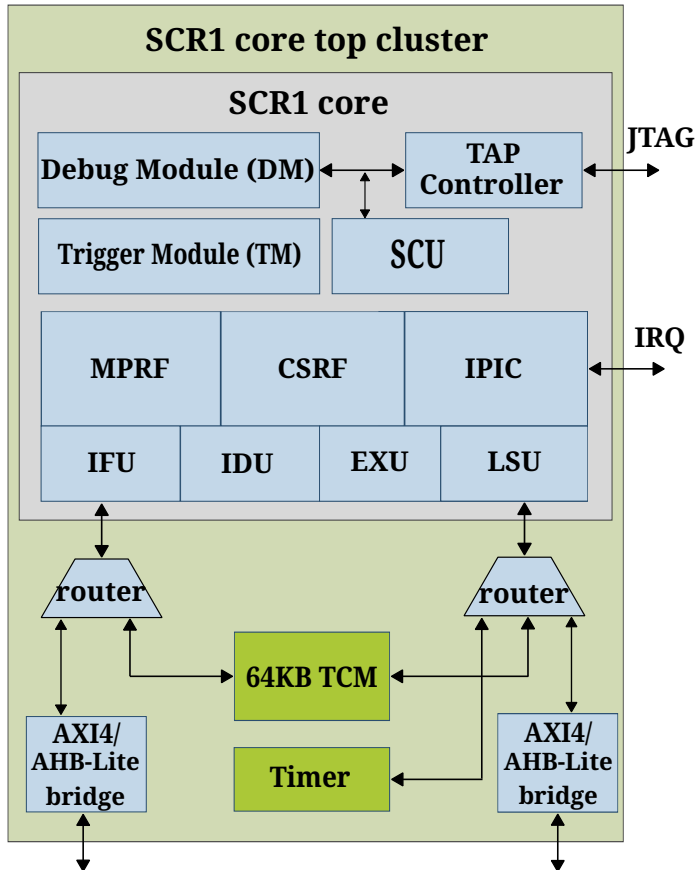


Figure 1: SCR1 Block Diagram

SCR1 core contains:

- Instruction Fetch Unit (IFU)
- Instruction Decode Unit (IDU)
- Execution Unit (incl. integer ALU) (EXU, IALU)
- Load-Store Unit (LSU)
- Multi-port register file (MPRF)
- Control/Status register file (CSRF)
- Integrated programmable interrupt controller (IPIC)
- Trigger Module / Trigger Debug Unit (TM / TDU)
- Tightly-coupled memory (TCM)
- External AXI4/AHB-Lite instruction memory interface

- External AXI4/AHB-Lite data memory interface
- Debug Subsystem:
  - Test access point controller (TAPC)
  - System Control Unit (SCU)
  - Debug Module (DM)

## 2. Privilege Levels

The core implements only one of four RISC-V privilege levels defined in [2] as shown in Table 2.

Table 2: Implemented privilege levels

Numeric level	2-bit encoding	Level name / Mode	Implementation
0	00	User level / U-mode	No
1	01	Supervisor level / S-mode	No
2	10	Hypervisor level / H-mode	No
3	11	Machine level / M-mode	Yes

The machine level has the highest privileges. Code running in machine-mode (M-mode) is inherently trusted, as it has low-level access to all implemented functions of the core.

The core runs any application code in M-mode. Some trap, such as exception or asynchronous external interrupt, forces a switch to a trap handler, which runs in the same privilege mode. The core will then execute the trap handler, which will eventually resume execution at or after the original trapped instruction.

# 3. Registers

## 3.1. General-purpose Integer Registers

Figure 2 shows the user-visible general-purpose integer registers of the core. There are 31 (or 15 for RV32E) general-purpose registers x1-x31 (or x1-x15), which are designed to hold integer values. Register x0 is hardwired to the constant 0 and can be used as a source of constant zero or as a don't care destination register.

Don't care destination x0 is used to ignore the result of instruction execution provided that destination register is mandatory for instruction structure.

All general-purpose registers in the core are 32-bits wide.

The core implements 32-bit pc register, which is used as program counter, meaning that it holds the address of the current instruction.

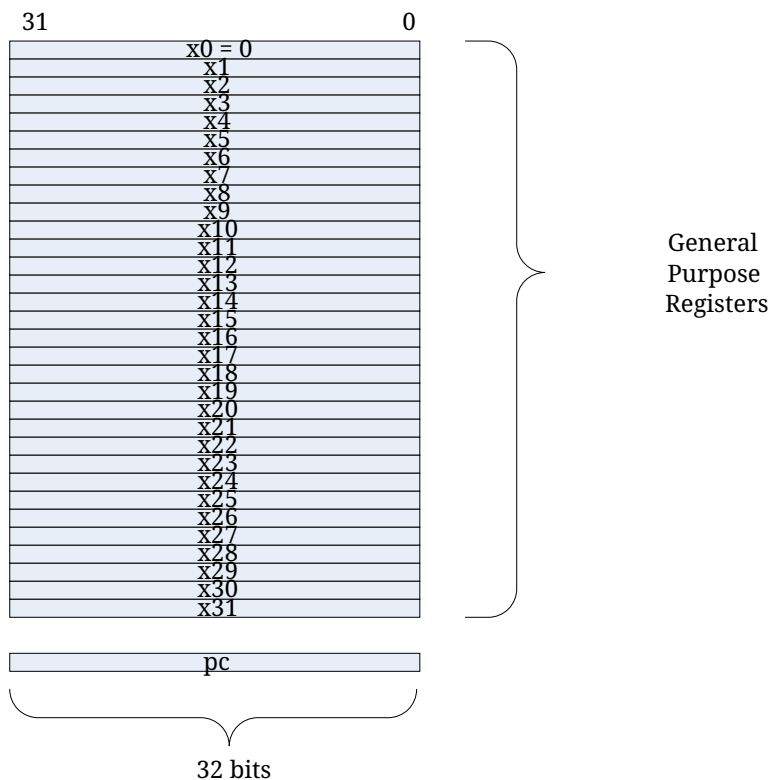


Figure 2: General-purpose integer registers

## 3.2. Control and Status Registers

### 3.2.1. Overview and definitions

Control/status registers (CSR) of the core are accessed atomically using instructions specifically designed for CSR access. CSR access instructions are listed in [Instruction set summary](#) section of this specification.

According to the RISC-V specification [2], the core uses 12-bit encoding space to address up to 4096 control/status registers (CSR) in the instructions which atomically read and modify CSRs. The core implements subset of CSRs according to the mapping shown in the next paragraphs. The core follows RISC-V convention, where the upper 4 bits of the CSR address [11:8] are used to encode the read and write accessibility of the CSRs according to the privilege level. The top two bits [11:10] indicate whether the register is read/write (00, 01, or 10) or read-only (11). The next two bits [9:8] indicate the lowest privilege level that can access the CSR.

The following definitions are used to designate bit or bit field properties throughout the individual CSR descriptions:

- RO - read only (write attempt results in illegal instruction exception)
- QRO - quiet read only (write attempt is ignored)
- RZ - read as zero
- RW - read/write
- RW1S - read/write one to set
- RW1C - read/write one to clear
- RW1P - read/write one to pulse

The core implements the following rules for CSR access:

1. Attempts to access a non-existent CSR raise an illegal instruction exception;
2. Attempts to write a read-only CSR also raise illegal instruction exception;
3. If a read/write register contains some bits that are read-only, then writes to the read-only bits are ignored.

### 3.2.2. CSR Map

Map of control/status registers is shown in [Table 3](#).

All of the standard CSRs do comply with [\[2\]](#), unless explicitly stated otherwise.

Table 3: CSR map

Address	Name
<b>Standard CSRs</b>	
<i>User Counters/Timers (read-only)</i>	
0xC00	CYCLE
0xC01	TIME
0xC02	INSTRET
0xC80	CYCLEH
0xC81	TIMEH
0xC82	INSTRETH
<i>Machine Information Registers (read-only)</i>	
0xF11	MVENDORID
0xF12	MARCHID
0xF13	MIMPID
0xF14	MHARTID
<i>Machine Trap Setup (read-write)</i>	
0x300	MSTATUS
0x301	MISA
0x304	MIE
0x305	MTVEC
<i>Machine Trap Handling (read-write)</i>	
0x340	MSCRATCH
0x341	MEPC
0x342	MCAUSE
0x343	MTVAL
0x344	MIP
<i>Standard read/write debug CSRs (0x7A0..0x7AF)</i>	
0x7A0	TSELECT
0x7A1	TDATA1
0x7A2	TDATA2
...	...
0x7A4	TINFO
...	...



Address	Name
<i>Debug-mode-only CSRs (0x7B0..0x7BF)</i>	
0x7B0	DSCR
0x7B1	DPC
0x7B2	DSCRATCH0
...	...
<i>Machine Counters/Timers (read-write)</i>	
0xB00	MCYCLE
0xB02	MINSTRET
0xB80	MCYCLEH
0xB82	MINSTRETH
<b>Non-standard CSRs (read-write)</b>	
0x7E0	MCOUNTEN
0xBF0..0xBF7	IPIC registers
<b>Memory-mapped CSRs (read-write)</b>	
0x00490000 (default)	TIMER_CTRL
0x00490004 (default)	TIMER_DIV
0x00490008 (default)	MTIME
0x0049000C (default)	MTIMEH
0x00490010 (default)	MTIMECMP
0x00490014 (default)	MTIMECMPH

### 3.2.3. User Mode CSRs

All user-mode CSR registers are implemented in full compliance with the RISC-V specification [2]. Please note that the term "user-mode CSRs" here does not imply support for user mode in the core, but is rather used for coherence with the RISC-V specification.

- CYCLE [0xC00] (read-only mirror of MCYCLE)
- TIME [0xC01] (read-only mirror of MTIME)
- INSTRET [0xC02] (read-only mirror of MINSTRET)
- CYCLEH [0xC80] (read-only mirror of MCYCLEH)
- TIMEH [0xC81] (read-only mirror of MTIMEH)
- INSTRETH [0xC82] (read-only mirror of MINSTRETH)

For more information, see [MCYCLE/MCYCLEH \[0xB00/0xB80\]](#), [MINSTRET/MINSTRETH \[0xB02/0xB82\]](#) and [MTIME/MTIMEH \[TIMER\\_BASE + 0x8/TIMER\\_BASE + 0xC\]](#).

### 3.2.4. Machine Mode Standard CSRs

#### 3.2.4.1. MVENDORID [0xF11]

MVENDORID is hardwired to 0x0.

#### 3.2.4.2. MARCHID [0xF12]

MARCHID is hardwired to 0x8.

#### 3.2.4.3. MIMPID [0xF13]

MIMPID is hardwired to 0x19083000.

Structure of MIMPID register is shown in [Table 4](#).

*Table 4: Structure of MIMPID register*

Bits	Name	Attributes	Description
31..24	Year	RO	BCD-coded value of the year
23..16	Mon	RO	BCD-coded value of the month
15..8	Day	RO	BCD-coded value of the day
7..0	REL	RO	8-bit value of an intra-day release number

#### 3.2.4.4. MHARTID [0xF14]

MHARTID is defined by external fuses.

#### 3.2.4.5. MSTATUS [0x300]

Structure of MSTATUS register is shown in [Table 5](#).

*Table 5: Structure of MSTATUS register*

Bits	Name	Attributes	Description
2..0	RSV	RZ	Reserved
3	MIE	RW	Global interrupt enable
6..4	RSV	RZ	Reserved
7	MPIE	RW	Previous global interrupt enable
10..8	RSV	RZ	Reserved
12..11	MPP	QRO	Previous privilege mode (hardwired to 11)
31..13	RSV	RZ	Reserved

Default value after reset is 0x1880.

#### 3.2.4.6. MISA [0x301]

Structure of MISA register is shown in [Table 6](#).

Table 6: Structure of MISA register

Bits	Name	Attributes	Description
1..0	RSV	RZ	Reserved
2	RVC	QRO	Compressed instruction extension implemented
3	RSV	RZ	Reserved
4	RVE	QRO	RV32E base integer instruction set
7..5	RSV	RZ	Reserved
8	RVI	QRO	RV32I base integer instruction set
11..9	RSV	RZ	Reserved
12	RVM	QRO	Integer Multiply/Divide extension implemented
22..13	RSV	RZ	Reserved
23	RVX	QRO	Non-standard extensions
29..24	RSV	RZ	Reserved
31..30	MXL	QRO	Machine XLEN (hardwired to 01)

### 3.2.4.7. MIE [0x304]

Structure of MIE register is shown in [Table 7](#).

Table 7: Structure of MIE register

Bits	Name	Attributes	Description
2..0	RSV	RZ	Reserved.
3	MSIE	RW	Machine Software Interrupt Enable.
6..4	RSV	RZ	Reserved.
7	MTIE	RW	Machine Timer Interrupt Enable.
10..8	RSV	RZ	Reserved
11	MEIE	RW	Machine External Interrupt Enable.
31..12	RSV	RZ	Reserved

### 3.2.4.8. MTVEC [0x305]

Structure of MTVEC register is shown in [Table 8](#).

Table 8: Structure of MTVEC register

Bits	Name	Attributes	Description
1..0	MODE	RW/RZ	Vector mode (0-direct mode, 1-vector mode)
5..2		RZ	Read as zero
31..6	BASE	RW/QRO	Vector base address (upper 26 bits)

MODE field can be either RW or RZ depending on the SCR1\_VECT\_IRQ\_EN parameter value. BASE field can be QRO, RW, or partially RW depending on SCR1\_CSR\_MTVEC\_BASE\_RW\_BITS parameter

value. SCR1\_ARCH\_CSR\_MTVEC\_BASE\_RST\_VAL parameter is used to set constant values for QRO bits and reset values for RW bits. See [SCR1 configurable options](#) for details.

**NOTE**

In direct mode, all exceptions set PC to BASE. In vectored mode, asynchronous interrupts set PC to BASE+4×cause.

### 3.2.4.9. MSCRATCH [0x340]

Structure of MSCRATCH register is shown in [Table 9](#).

Table 9: Structure of MSCRATCH register

Bits	Name	Attributes	Description
31..0		RW	As defined by the RISC-V specification <a href="#">[2]</a>

### 3.2.4.10. MEPC [0x341]

Structure of MEPC register is shown in [Table 10](#).

Table 10: Structure of MEPC register

Bits	Name	Attributes	Description
0	RSV	RZ	Reserved
31..1		RW	As defined by the RISC-V specification <a href="#">[2]</a>

### 3.2.4.11. MCAUSE [0x342]

Structure of MCAUSE register is shown in [Table 11](#).

Table 11: Structure of MCAUSE register

Bits	Name	Attributes	Description
3..0	EC	RW	Exception Code
30..4	RSV	RZ	Reserved
31	INT	RW	Interrupt

List of MCAUSE Exception Codes is shown in [Table 12](#).

Table 12: List of MCAUSE Exception Codes

INT	EC	Description
0	0	Instruction address misaligned
0	1	Instruction access fault
0	2	Illegal instruction
0	3	Breakpoint
0	4	Load address misaligned
0	5	Load access fault
0	6	Store/AMO address misaligned

INT	EC	Description
0	7	Store/AMO access fault
0	10..8	Not supported
0	11	Ecall from M-mode
0	>=12	Reserved
1	2..0	Reserved
1	3	Machine Software Interrupt
1	6..4	Reserved
1	7	Machine Timer Interrupt
1	10..8	Reserved
1	11	Machine External Interrupt
1	>=12	Reserved

Interrupts have priority over exceptions, as defined by the specification. The priority is determined when the instruction that causes exception is at the decode stage.

#### 3.2.4.12. MTVAL [0x343]

Structure of MTVAL register is shown in [Table 13](#).

*Table 13: Structure of MTVAL register*

Bits	Attributes	Description
31..0	RW	As defined by the RISC-V specification <a href="#">[2]</a>

#### NOTE

MTVAL is written with the faulting instruction bits on an illegal instruction exception.

#### 3.2.4.13. MIP [0x344]

Structure of MIP register is shown in [Table 14](#).

*Table 14: Structure of MIP register*

Bits	Name	Attributes	Description
2..0	RSV	RZ	Reserved.
3	MSIP	QRO	Machine Software Interrupt Pending.
6..4	RSV	RZ	Reserved.
7	MTIP	QRO	Machine Timer Interrupt Pending.
10..8	RSV	RZ	Reserved
11	MEIP	QRO	Machine External Interrupt Pending.
31..12	RSV	RZ	Reserved

#### 3.2.4.14. MCYCLE/MCYCLEH [0xB00/0xB80]

MCYCLE/MCYCLEH CSRs represent the number of clock cycles since some arbitrary point of time in the past, at which both MCYCLE and MCYCLEH were equal to zero, and since which the counting has started. By default, MCYCLE and MCYCLEH are equal to zero after core reset (which also starts counting). Another option to start counting for MCYCLE/MCYCLEH is by writing some value to the MCYCLE/MCYCLEH.

##### NOTE

MCYCLE/MCYCLEH CSRs are optional when RV32E base integer instruction set is used.

Structure of MCYCLE/MCYCLEH registers is shown in [Table 15](#).

Table 15: Structure of MCYCLE/MCYCLEH registers

Bits	Attributes	Description
31..0	RW	As defined by the RISC-V specification <a href="#">[2]</a>

#### 3.2.4.15. MINSTRET/MINSTRETH [0xB02/0xB82]

MINSTRET/MINSTRETH CSRs represent the number of instructions executed by the core from some arbitrary time in the past, at which both MINSTRET and MINSTRETH were equal to zero, and since which the counting has started. By default, MINSTRET and MINSTRETH are equal to zero after core reset (which also starts counting). Another option to start counting for MINSTRET/MINSTRETH is by writing some value to the MINSTRET/MINSTRETH.

##### NOTE

MINSTRET/MINSTRETH value reflects the number of instructions successfully executed by the core, which means instructions that cause exceptions are not counted.

##### NOTE

MINSTRET/MINSTRETH CSRs are optional when RV32E base integer instruction set is used.

Structure of MINSTRET/MINSTRETH registers is shown in [Table 16](#).

Table 16: Structure of MINSTRET/MINSTRETH registers

Bits	Attributes	Description
31..0	RW	As defined by the RISC-V specification <a href="#">[2]</a>

#### 3.2.5. Standard read/write debug CSRs [0x7A0..0x7AF]

For description of Trigger CSRs refer to the [Trigger CSRs](#) section.

#### 3.2.6. Debug-mode-only CSRs [0x7B0..0x7BF]

For description of Debug-mode-only CSRs refer to the [Debug CSRs](#) section.

### 3.2.7. Machine Mode Non-standard CSRs

#### 3.2.7.1. MCOUNTEN [0x7E0]

MCOUNTEN CSR allows to disable counters via software if they are not needed by the application. This CSR does not exist if CYCLE[H] and INSTRET[H] CSRs are disabled. Structure of MCOUNTEN register is shown in [Table 17](#).

*Table 17: Structure of MCOUNTEN register*

Bits	Name	Attributes	Description
0	CY	RW	Enable cycle counter
1	RSV	RZ	Reserved
2	IR	RW	Enable retired instructions counter
31..3	RSV	RZ	Reserved

#### 3.2.7.2. IPIC registers [0xBF0..0xBF7]

For more information, refer to the [Map of IPIC registers](#) section.



### 3.2.8. Memory-mapped CSRs

#### IMPORTANT

Memory-mapped CSRs do not support byte and halfword access, an corresponding attempt will cause a load/store access fault exception.

#### IMPORTANT

Timer memory-mapped CSRs addresses are given below relative to the `TIMER_BASE = SCR1_TIMER_ADDR_PATTERN` (see [SCR1 configurable options](#)).

#### 3.2.8.1. TIMER\_CTRL [TIMER\_BASE]

Structure of `TIMER_CTRL` register is shown in [Table 18](#).

Table 18: Structure of `TIMER_CTRL` register

Bits	Name	Attributes	Description
0	ENABLE	RW	Timer enable
1	CLKSRC	RW	Timer clock source: 0 - internal core clock (default) 1 - external real-time clock
31..2		RZ	Reserved, read as zero

#### 3.2.8.2. TIMER\_DIV [TIMER\_BASE + 0x4]

Structure of `TIMER_DIV` register is shown in [Table 19](#).

Table 19: Structure of `TIMER_DIV` register

Bits	Name	Attributes	Description
9..0	DIV	RW	Timer divider: timer tick occurs every DIV+1 clock ticks
31..10		RZ	Reserved, read as zero

#### 3.2.8.3. MTIME/MTIMEH [TIMER\_BASE + 0x8/TIMER\_BASE + 0xC]

`MTIME/MTIMEH` CSRs represent wall-clock real time (number of timer ticks) from some arbitrary time in the past, at which both `MTIME` and `MTIMEH` were equal to zero, and since which the counting has started. By default, `MTIME` and `MTIMEH` are equal to zero after core reset (which also starts counting). Another option to start counting for `MTIME/MTIMEH` is by writing some value to the `MTIME/MTIMEH`.

Structure of `MTIME/MTIMEH` registers is shown in [Table 20](#).

Table 20: Structure of `MTIME/MTIMEH` registers

Bits	Name	Attributes	Description
31..0		RW	As defined by the RISC-V specification <a href="#">[2]</a>

#### 3.2.8.4. MTIMECMP/MTIMECMPH [TIMER\_BASE + 0x10/TIMER\_BASE + 0x14]

Structure of `MTIMECMP/MTIMECMPH` registers is shown in [Table 21](#).

Table 21: Structure of MTIMECMP/MTIMECMPH registers

Bits	Name	Attributes	Description
31..0		RW	As defined by the RISC-V specification <a href="#">[2]</a>

# 4. Memory Model

## 4.1. Bit and byte order

The core does access instruction and data words in memory assuming generic little endian organization as illustrated in [Figure 3](#). With little-endian format, the byte with the lowest address in a word is the least-significant byte of the word. The byte with the highest address in a word is the most significant. For instance, the byte at address 0 of the data memory bus connects to least significant data lines 7-0.

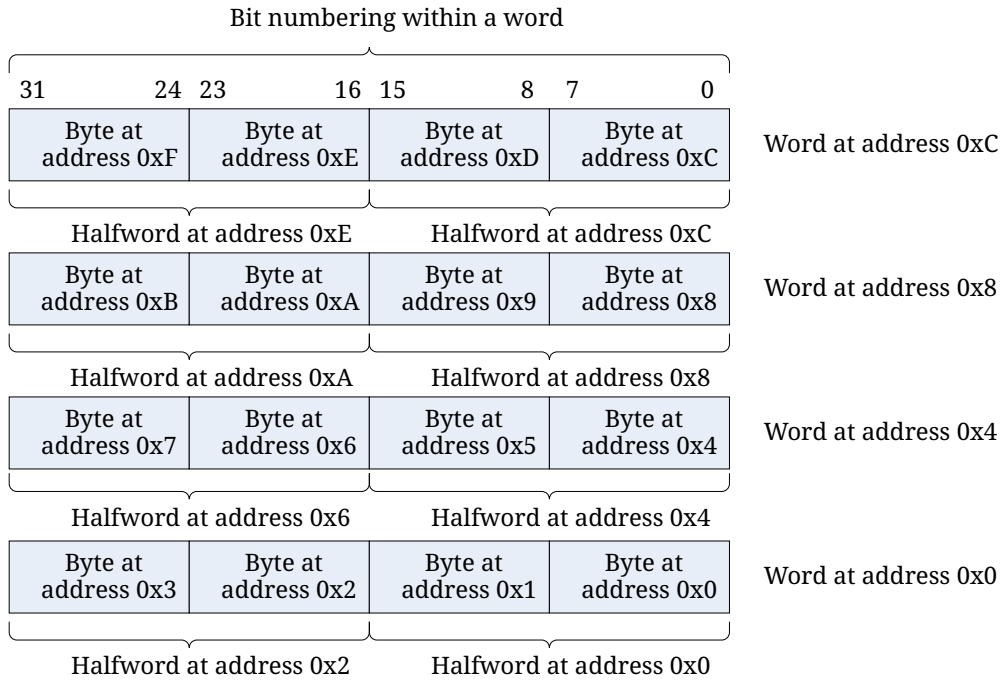


Figure 3: Generic little endian memory organization

Regardless of memory access width the numbering of bits always assumes that bit 0 is least significant bit and it is also rightmost bit in all illustrative diagrams within the specification.

## 4.2. Data access width and alignment

The core supports following memory access widths:

- 32-bit words for instruction and data memory;
- 16-bit halfwords for data memory only;
- 8-bit bytes for data memory only.

The core considers data memory as a contiguous collection of bytes numbered in ascending order in the range 0x00000000-0xFFFFFFFF (32-bit address).

The core considers instruction memory as a contiguous collection of 32-bit words for base 32-bit instruction set (RV32I) or as a contiguous collection of 16-bit halfwords for compact instruction set

(RV32C). Instructions in memory must be aligned to 4-byte boundary or 2-byte boundary correspondingly. Byte numbering in memory starts from 0. In case of compact instruction set the last instruction address is 0xFFFFFFF0. In case of non-compact instruction set the last instruction address is 0xFFFFFFF4. Instruction fetch from memory is physically done as 32-bit words aligned to 4-byte boundary ignoring any unnecessary portion of the word during instruction decode.

## 4.3. Stack behavior

The core supports stack handling with implemented base and compact instruction sets. No special register is used to implement return address link register or stack pointer during subroutine call. However, any subset of general purpose registers x1..x31 can be used for these purposes.

As soon as the register is chosen to be a stack pointer, after appropriate register initialization the implementation of context save/restore or access to local variables during subroutine call becomes straightforward. Standard software calling convention uses register x2 as a stack pointer.

As soon as the register is chosen to be a link register, implemented instruction sets (both base and compact) provide adequate means to memorize the return address during subroutine call and to use this address on return from subroutine. Standard software calling convention uses register x1 to hold the return address during subroutine calls.

## 4.4. Memory access ordering

The core uses strong memory access ordering, meaning that the sequence and the number of memory accesses are guaranteed to correspond one-to-one to underlying sequence of instructions executed. Given that, FENCE instruction is executed as NOP, FENCE.I instruction flushes the instruction fetch queue.

## 4.5. System memory map

The core implements Harward architecture characterized by independent access to instruction memory and data memory through dedicated external memory interfaces.

[Figure 4](#) shows the illustrative view of the system memory map for the core.

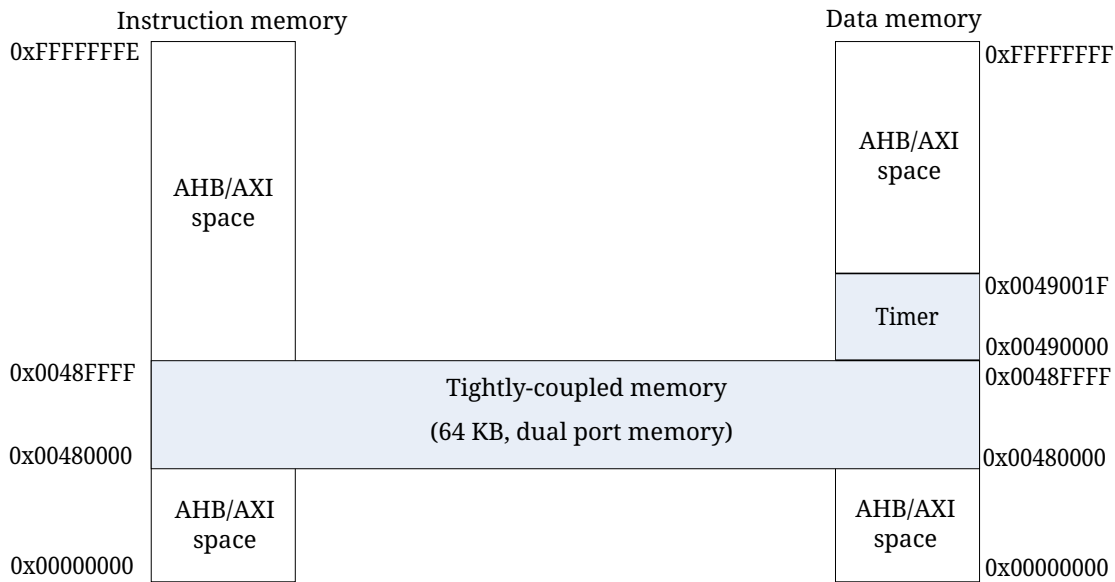


Figure 4: System memory map

The core provides dual-port tightly-coupled memory (TCM) which can be used for both instructions and data. TCM is characterized by short memory response to support time critical code and/or data of the application. TCM is mapped to system memory map with fixed base address 0x00480000. Detailed description of TCM is given in [Tightly-Coupled Memory](#) section of this specification.

## 4.6. Tightly-Coupled Memory

Tightly-Coupled Memory (TCM) is random access memory (RAM) with guaranteed single-cycle response time. TCM is designed for both instruction and data sections of the code which require maximum throughput.

TCM is implemented as dual-port memory with independent access from Instruction and Data memory interfaces (I/F).

Instruction memory I/F does always read TCM as 32-bit words (read only access).

Data memory I/F supports 8/16/32 bits wide access to TCM (read/write access).

TCM size is up to 64 kBytes. TCM base address is 0x00480000.

## 5. Exceptions and Interrupts

The term exception is used to refer to an unusual condition occurring in the core at run time.

The term trap is used to refer to the synchronous transfer of control to a supervising environment when it is caused by an exceptional condition occurring within a core.

The term interrupt is used to refer to the asynchronous transfer of control to a supervising environment caused by an event outside of the core.

Some instructions under certain conditions (as described in [2]) raise an exception during execution. Whether and how these are converted into traps is dependent on the execution environment, though the expectation is that most environments will take a precise trap when an exception is signaled.

Exception codes supported by the core are listed in [Table 22](#).

*Table 22: List of supported exception codes*

Code	Exception cause/description
0	Misaligned instruction fetch address
1	Instruction fetch access fault
2	Illegal instruction
3	Breakpoint
4	Misaligned load address
5	Load access fault
6	Misaligned store address
7	Store access fault
8	Reserved
9	Reserved
10	Reserved
11	Ecall from M-mode
31..12	Reserved

Interrupt codes supported by the core are listed in [Table 23](#). Non-Maskable Interrupts are not implemented in SCR1.

*Table 23: List of supported interrupt codes*

Code	Interrupt cause/description
2..0	Reserved
3	Machine software interrupt
6..4	Reserved
7	Machine timer interrupt

<b>Code</b>	<b>Interrupt cause/description</b>
10..8	Reserved
11	Machine external interrupt
31..12	Reserved



# 6. Pipeline theory of operations

## 6.1. Instruction execution phases

SCR1 has simple in-order pipeline. Functional phases of instruction execution are listed below.

- Request to Instruction Memory
- Instruction fetch
- Instruction decode
- Execution
  - Operand fetch
  - Arithmetical and logical operations
  - Load/store operations
  - Instruction flow control
- Commit point

Depending on the frequency targets, these functional phases can be configured into 2, 3 or 4 stages. 2-stages is the default pipeline configuration.

### 6.1.1. Request to Instruction Memory

In this phase CPU requests instruction words from Instruction Memory using the address contained in the IMEM\_ADDR register. The phase can take arbitrary number of cycles depending on the memory latency. Fetching an instruction word from TCM always takes one clock cycle.

In the SCR1 this phase is implemented in Instruction Fetch Unit (IFU).

### 6.1.2. Instruction fetch

In this phase the instruction words received from the Instruction Memory are placed into the instruction fetch queue, or, in case of queue bypass (SCR1\_IFU\_QUEUE\_BYPASS parameter is defined), are passed directly to the instruction decode unit.

Instruction fetch unit is responsible for assembling the instruction from parts in case when more than one memory access is needed to fetch that instruction.

In SCR1 this phase is implemented in Instruction Fetch Unit (IFU).

### 6.1.3. Instruction decode

Instruction is decoded and provided to the execution unit as a set of control signals and immediate operand. All decoded signals are placed into the queue, or, in case of a queue bypass (SCR1\_EXU\_STAGE\_BYPASS parameter is defined), are passed directly to the execution logic.

In SCR1 this phase is implemented in Instruction Decode Unit (IDU).

#### 6.1.4. Operand fetch

The required operands are fetched from the registers (GPRs or CSRs) or from the immediate field of the instruction buffer. This phase is required only for instructions which have operands.

In SCR1 operand fetch is always a part of the execution stage and is implemented in Execution Unit (EXU), which requests data from MPRF or CSRF.

#### 6.1.5. Arithmetical and logical operations

This covers arithmetical and logical operations with integer values, including multiplication and division operations. This phase is required only for instructions which need the results of arithmetical and logical operations.

Iterative multiplication (configuration with undefined SCR1\_FAST\_MUL parameter) takes 32 clock cycles and division takes 33 clock cycles. Execution of the other operations including 1-cycle multiplication (configuration with defined SCR1\_FAST\_MUL) are implemented on a completely combinatorial logic.

In SCR1 this phase is always a part of execution stage and implemented in Arithmetic Logic Unit (ALU).

#### 6.1.6. Load/store operations

In this phase all operations with Data Memory are executed. This phase is required only for instructions which perform load/store operations.

The phase can take arbitrary number of cycles depending on memory latency (no timeout is implemented). Loads and stores data from/to TCM always take two clock cycles.

In SCR1 this phase is always a part of the execution stage and implemented in Load-Store Unit (LSU).

#### 6.1.7. Instruction flow control

During the normal program flow the next instruction address (PC) is PC+4 for regular instructions or PC+2 for RVC instructions. Below is the list of instructions and events that can alter normal program flow or cause the transition to another state:

- Jump instruction
- Taken branch instruction
- Instruction fence
- Wait for interrupt instruction
- Exception
- Interrupt
- MRET instruction
- Debug mode redirection event

Detection of such cases is function of the instruction flow control phase as well as calculation of all required control signals, the next PC value and the next status of CPU.

MRET instruction and change of MSTATUS or MIE CSRs always takes two clock cycles.

In SCR1 this phase is always part of execution stage and implemented in Execution Unit (EXU).

### 6.1.8. Commit point

Commit point is a moment, when GPRs, CSRs and PC registers are updated with new values, calculated in previous phases. After this point, the instruction is considered completed.

In SCR1 this phase update of GPRs and CSRs is implemented in MPRF, CSRF and update of PC is implemented in the EXU.

## 6.2. Pipeline configurations

Pipeline can be configured for 2, 3 or 4 stages depending on the required target frequency. 2-stages is the default pipeline configuration.

### 6.2.1. 2-stage pipeline

SCR1 2-stage pipeline configuration is shown in the [Figure 5](#). It includes the following stages:

- Request to Instruction Memory stage
- Instruction fetch, decode and execution stage

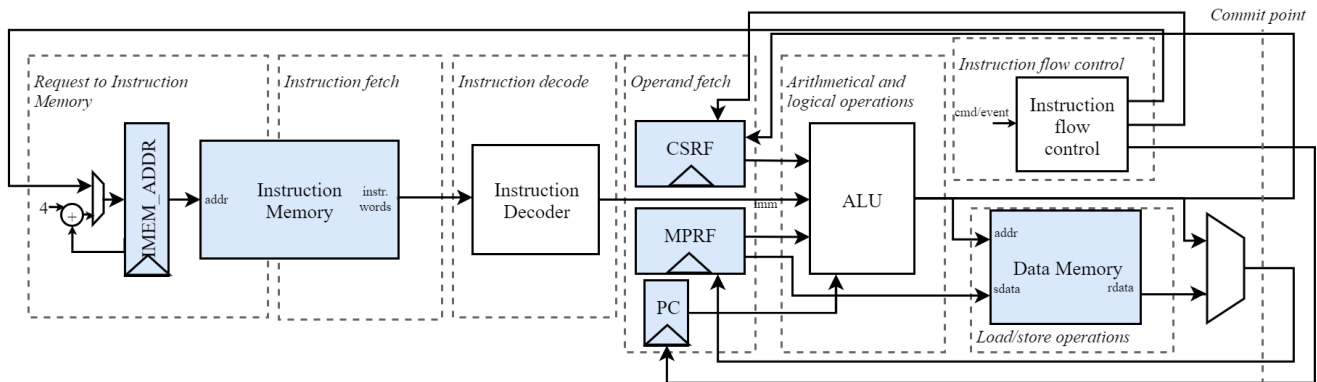


Figure 5: SCR1 2-stage pipeline

Configuration is enabled if both SCR1\_IFU\_QUEUE\_BYPASS and SCR1\_EXU\_STAGE\_BYPASS parameters are defined. This is default pipeline configuration.

```
`define SCR1_IFU_QUEUE_BYPASS // enables bypass between IFU and IDU stages
`define SCR1_EXU_STAGE_BYPASS // enables bypass between IDU and EXU stages
```

Instruction retirement delays in cycles for a 2-stage pipeline are shown in the [Table 24](#). Latency column shows the number of clock cycles from the start of the instruction fetching until the instruction retirement. Throughput column shows the minimum number of clock cycles from the

previous instruction retirement to the current instruction retirement. The given data is valid if TCM is used for instruction fetch, using slower memory will cause additional delays.

Table 24: Instruction execution time for 2-stage pipeline

Instruction	Latency	Throughput	Notes
L[B,BU,H,HU,W], S[B,H,W]	3 + [DMEM Latency]	2 + [DMEM Latency]	DMEM Latency = 0, if TCM is used
MUL[H,HSU,HU] if SCR1_FAST_MUL	2	1	-
MUL[H,HSU,HU] if not SCR1_FAST_MUL	33	32	Throughput = 1, if any of the operands equals zero
DIV[U], REM[U]	34	33	Throughput = 1, if any of the operands equals zero
MRET	3	2	-
CSRR[W,S,C][I] for MSTATUS or MIE	3	2	-
Other instructions	2	1	-

### 6.2.2. 3-stage pipeline

SCR1 3-stage pipeline configuration is shown in the [Figure 6](#). It includes the following stages:

- Request to Instruction Memory stage
- Instruction fetch and decode stage
- Execution stage

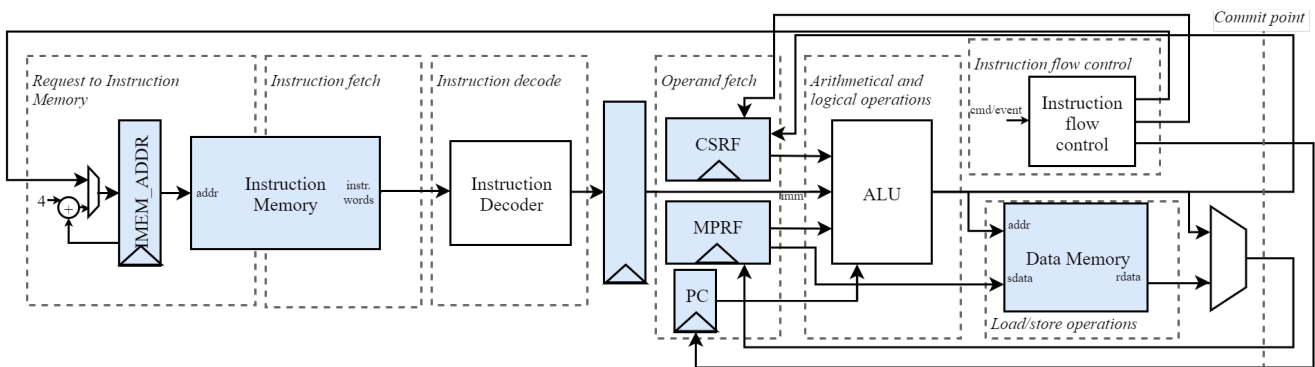


Figure 6: SCR1 3-stage pipeline

3-stage pipeline configuration is enabled if only one of parameters SCR1\_IFU\_QUEUE\_BYPASS or SCR1\_EXU\_STAGE\_BYPASS is defined. Recommended configuration is to define SCR1\_IFU\_QUEUE\_BYPASS and undefine SCR1\_EXU\_STAGE\_BYPASS.

```
`define SCR1_IFU_QUEUE_BYPASS      // enables bypass between IFU and IDU stages
//`define SCR1_EXU_STAGE_BYPASS    // enables bypass between IDU and EXU stages
```

Instruction retirement delays in cycles for a 3-stage pipeline are shown in the [Table 25](#). Latency

column shows the number of clock cycles from the start of the instruction fetching until the instruction retirement. Throughput column shows the minimum number of clock cycles from the previous instruction retirement to the current instruction retirement. The given data is valid if TCM is used for instruction fetch, using slower memory will cause additional delays.

Table 25: Instruction execution time for a 3-stage pipeline

Instruction	Latency	Throughput	Notes
L[B,BU,H,HU,W], S[B,H,W]	4 + [DMEM Latency]	2 + [DMEM Latency]	DMEM Latency = 0, if TCM is used
MUL[H,HSU,HU] if SCR1_FAST_MUL	3	1	-
MUL[H,HSU,HU] if not SCR1_FAST_MUL	34	32	Throughput = 1, if any of the operands equals zero
DIV[U], REM[U]	35	33	Throughput = 1, if any of the operands equals zero
MRET	4	2	-
CSRR[W,S,C][I] for MSTATUS or MIE	4	2	-
Other instructions	3	1	-

### 6.2.3. 4-stage pipeline

SCR1 4-stage pipeline configuration is shown in the [Figure 7](#). It includes the following stages:

- Request to Instruction Memory stage
- Instruction fetch stage
- Instruction decode stage
- Execution stage

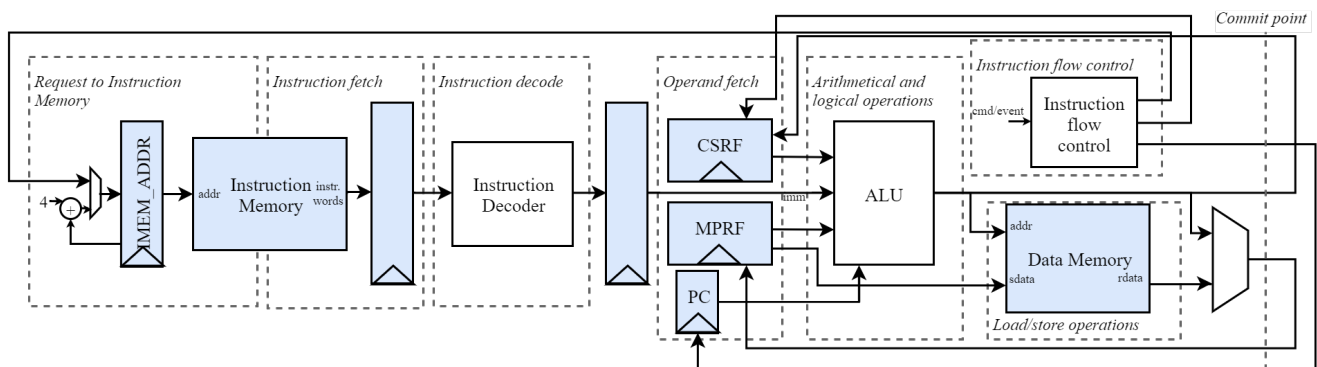


Figure 7: SCR1 4-stage pipeline

4-stage pipeline configuration is enabled if both SCR1\_IFU\_QUEUE\_BYPASS and SCR1\_EXU\_STAGE\_BYPASS parameters are undefined.

```
//`define SCR1_IFU_QUEUE_BYPASS      // enables bypass between IFU and IDU stages
//`define SCR1_EXU_STAGE_BYPASS      // enables bypass between IDU and EXU stages
```

Instruction retirement delays in cycles for a 4-stage pipeline are shown in [Table 26](#). Latency column shows the number of clock cycles from the start of the instruction fetching until the instruction retirement. Throughput column shows the minimum number of clock cycles from the previous instruction retirement to the current instruction retirement. The given data is valid if TCM is used for instruction fetch, using slower memory will cause additional delays.

Table 26: Instruction execution time for 4-stage a pipeline

Instruction	Latency	Throughput	Notes
L[B,BU,H,HU,W], S[B,H,W]	5 + [DMEM Latency]	2 + [DMEM Latency]	DMEM Latency = 0, if TCM is used
MUL[H,HSU,HU] if SCR1_FAST_MUL	4	1	-
MUL[H,HSU,HU] if not SCR1_FAST_MUL	35	32	Throughput = 1, if any of the operands equals zero
DIV[U], REM[U]	36	33	Throughput = 1, if any of the operands equals zero
MRET	5	2	-
CSRR[W,S,C][I] for MSTATUS or MIE	5	2	-
Other instructions	4	1	-

## 6.3. Hazards handling

### 6.3.1. Data hazards

SCR1 pipeline has no data hazards by design, because operand fetch and results commit are executed in the same stage.

### 6.3.2. Structural hazards

Structural hazards in the SCR1 pipeline are resolved as described below: When two or more instructions need the same hardware resource at the same time (structural hazard), the later instructions are stalled till the older instruction finish with the resource and release it.

### 6.3.3. Control hazards

Control hazards in the SCR1 pipeline are resolved as described below: When pipeline is not executing instructions continuously and the new value for the instruction address is defined in the execution phase (is **not** PC+4 or PC+2), all the following instructions are flushed and pipeline is restart from the instruction fetch phase with the new PC value.

# 7. Integrated Programmable Interrupt Controller

## 7.1. Introduction

SCR1 core can optionally include Integrated Programmable Interrupt Controller (IPIC) with low latency IRQ response. IPIC can be configured using IPIC Control Status Registers.

The term Interrupt Line has the meaning of corresponding IPIC external pin where suitable source of external interrupt may be connected to.

The term Interrupt Vector has the meaning of external interrupt number which will be generated by IPIC in response to external interrupt.

IPIC supports maximum 16 Interrupt vectors [0..15] and 16 Interrupt lines [0..15], each line is statically mapped to the corresponding vector.

Interrupt Vectors are given fixed priorities. The lowest Interrupt Vector number has the highest priority.

IPIC supports nested interrupts. Only one interrupt can be serviced at a time.

"Void interrupt vector" is defined as a non-existent vector number 0x10. This value is used to indicate absence of a valid interrupt vector.

### IMPORTANT

Write access to the IPIC control status registers is implemented only through the use of the CSRRW(I) instructions, the CSRRS(I) and CSRRC(I) instructions are not supported.

## 7.2. IPIC Block Diagram and description

Figure 8 shows block diagram of the IPIIC.

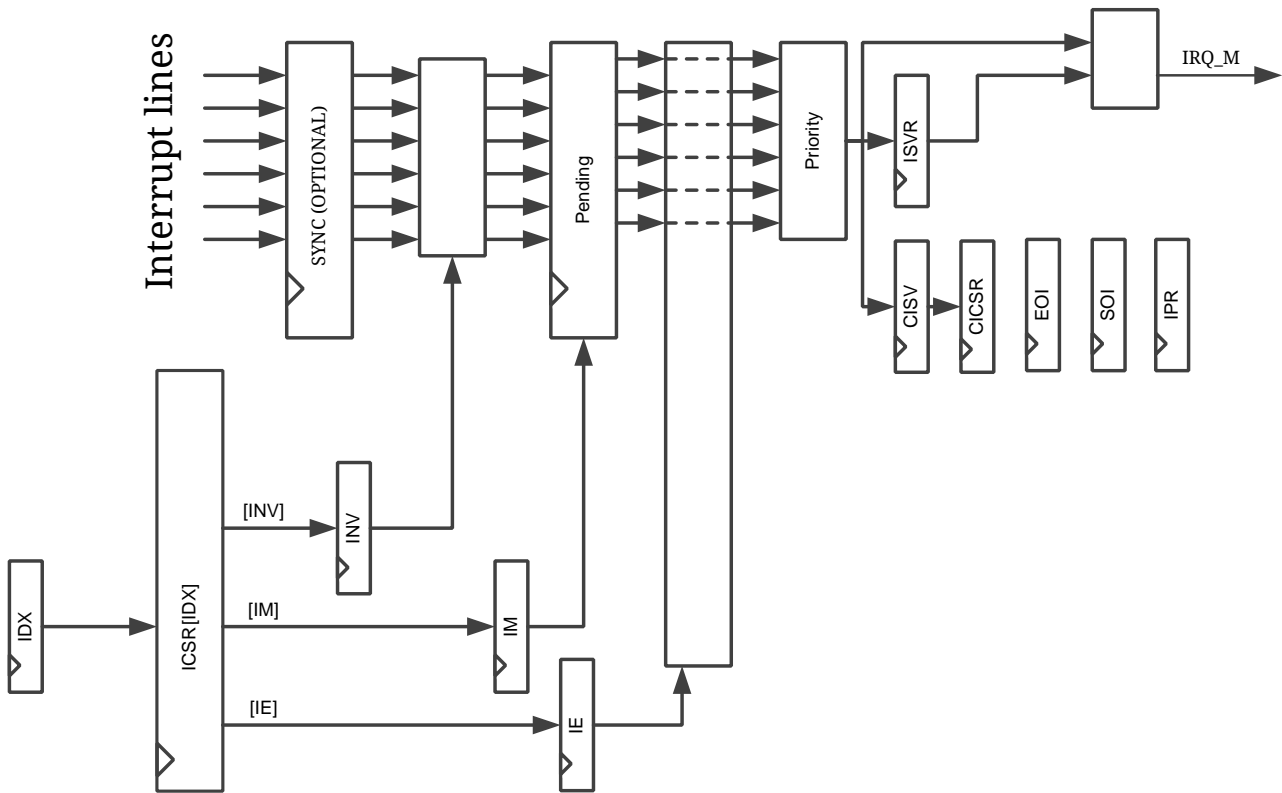


Figure 8: IPIC Block Diagram

## IMPORTANT

IPIC can be configured with (default) or without IRQ lines 2-stage synchronizer.

- Without synchronizer, all IRQ lines must be synchronous to the internal core clock
- With a 2-stage synchronizer, there is a requirement that for IRQ line edge detection, input pulse must be at least 2 clock cycles wide

Depending on the IM (interrupt mode), INV (line inversion) values for each vector, one of four conditions for IP (interrupt pending) bit activation is selected: high level, low level, rising edge, falling edge. Of all vectors with IP and IE (interrupt enable) bits active, the lowest numbered vector has the highest priority. Software is responsible for writing the SOI and EOI registers, thus notifying IPIC of the start and end of interrupt processing, respectively.

### 7.3. IPIC Programming Model

### 7.3.1. Register Map

Following notation is used to specify properties of bit fields within IPIC registers:

- RO - Read Only



- WO - Write Only
- RW - Read/Write
- R/W1S - Read/Write 1 to Set
- R/W1C - Read/Write 1 to Clear

IPIC control status registers file access rights are defined by the current privilege mode. All registers are accessible only from the Machine Mode (M-mode).

IPIC registers in M-mode are mapped relative to the given IPIC base address offset 0xBF0 in the CSR space as shown in [Table 27](#).

*Table 27: Map of IPIC registers*

Offset	Mnemonic	Name
0x00	IPIC_CISV	Current Interrupt Vector in Service
0x01	IPIC_CICSR	Current Interrupt Control Status Register
0x02	IPIC_IPR	Interrupt Pending Register
0x03	IPIC_ISVR	Interrupts in Service Register
0x04	IPIC_EOI	End Of Interrupt
0x05	IPIC_SOI	Start of Interrupt
0x06	IPIC_IDX	Index Register
0x07	IPIC_ICSR	Interrupt Control Status Register

## 7.4. Detailed IPIC Registers Description

### 7.4.1. IPIC\_CISV: Current Interrupt Vector in Service

Structure of IPIC\_CISV register is shown in [Table 28](#).

*Table 28: Structure of IPIC\_CISV register*

Bit number	Attributes	Description
4..0	QRO	Number of the interrupt vector currently in service
31..5	RZ	Reserved

IPIC\_CISV Register contains number of the interrupt vector currently in service (also, it is the number of the lowest assigned bit in the IPIC\_ISVR). When no interrupts are in service, this register contains number of the void interrupt vector (0x10).

### 7.4.2. IPIC\_CICSR: Current Interrupt Control Status Register

Structure of IPIC\_CICSR register is shown in [Table 29](#).

Table 29: Structure of IPIC\_CICSR register

Bit number	Mnemonic	Attributes	Description
0	IP	R/W1C	Interrupt pending:
			0 - no interrupt
			1 - Interrupt pending
1	IE	RW	Interrupt Enable Bit:
			0 - Interrupt disabled
			1 - Interrupt enabled
31..2	Reserved	RZ	Reserved

Control Status register for the interrupt vector currently in service.

This register is RW for IE bits and W1C for IP bit. Register read returns 0 when there are no interrupts currently in service.

### 7.4.3. IPIC\_IPR: Interrupt Pending Register

Structure of IPIC\_IPR register is shown in [Table 30](#).

Table 30: Structure of IPIC\_IPR register

Bit number	Attributes	Description
0	RW1C	Interrupt vector 0 pending status (1- pending)
1	RW1C	Interrupt vector 1 pending status (1- pending)
...		
15	RW1C	Interrupt vector 15 pending status (1- pending)
31..16	RZ	Reserved

Contains aggregated status for all the pending interrupts. Corresponding bits are set to 1 for the pending interrupts.

#### 7.4.4. IPIC\_ISVR: Interrupt Serviced Register

Structure of IPIC\_ISVR register is shown in [Table 31](#).

Table 31: Structure of IPIC\_ISVR register

Bit number	Attributes	Description
0	QRO	Interrupt vector 0 processing status (1- in service)
1	QRO	Interrupt vector 1 processing status (1- in service)
...		
15	QRO	Interrupt vector 15 processing status (1- in service)
31..16	RZ	Reserved

Contains aggregated status of the interrupts vectors, which are currently in service.

In other words, all those vectors, for which processing has started, but is not finished yet, including nested interrupts.

When corresponding bit is set (1) - this interrupt vector is in service. When corresponding bit is in 0 - the interrupt vector is not in service.

#### 7.4.5. IPIC\_EOI: End Of Interrupt

Structure of IPIC\_EOI register is shown in [Table 32](#).

Table 32: Structure of IPIC\_EOI register

Bit number	Attributes	Description
31..0	RZW	End-of-interrupt (any value can be written)

Writing any value to EOI register ends the interrupt which is currently in service.

Register values are updated to reflect the state change:

- IPIC\_CISV is set to its previous value if some interrupt was active prior to the current interrupt, otherwise set to void interrupt vector (0x10).
- IPIC\_CICSR is set to its previous value if some interrupt was active prior to the current interrupt, otherwise set to zero.
- IPIC\_ISVR: a bit corresponding to the current interrupt is cleared.

#### 7.4.6. IPIC\_SOI: Start Of Interrupt

Structure of IPIC\_SOI register is shown in [Table 33](#).

Table 33: Structure of IPIC\_SOI register

Bit number	Attributes	Description
31..0	RZW	Start-of-interrupt (any value can be written)

Writing any value to SOI activates start of interrupt if one of the following conditions is true:

- There is at least one pending interrupt with IE and ISR is zero (no interrupts in service).

- There is at least one pending interrupt with IE and this interrupt has higher priority than the interrupts currently in service.

Register values are updated to reflect the state change:

- IPIC\_CISV is set to the highest priority pending interrupt number.
- IPIC\_CICSR is set to reflect the values for the highest priority pending interrupt.
- IPIC\_IPR: a bit corresponding to the highest priority pending interrupt is cleared.
- IPIC\_ISVR: a bit corresponding to the highest priority pending interrupt is set.

#### 7.4.7. IPIC\_IDX: Index Register

Structure of IPIC\_IDX register is shown in [Table 34](#).

Table 34: Structure of IPIC\_IDX register

Bit number	Attributes	Description
3..0	RW	Interrupt vector index to access through IPIC_ICSR
31..4	RZ	Reserved

The value in IPIC\_IDX register defines the number of interrupt vector which is accessed through the IPIC\_ICSR register.

#### 7.4.8. IPIC\_ICSR: Interrupt Control Status register

Structure of IPIC\_ICSR register is shown in [Table 35](#).

Table 35: Structure of IPIC\_ICSR register

Bit number	Mnemonic	Attributes	Description
0	IP	RW1C	Interrupt pending:
			0 - no interrupt
			1 - Interrupt pending
1	IE	RW	Interrupt Enable Bit:
			0 - Interrupt disabled
			1 - Interrupt enabled
2	IM	RW	Interrupt Mode:
			0 - Level interrupt
			1 - Edge interrupt
3	INV	RW	Line Inversion:
			0 - no inversion
			1 - line inversion
4	IS	RW	In Service
7..5	Reserved	RZ	

Bit number	Mnemonic	Attributes	Description
9..8	PRV	QRO	Privilege mode: hardwired to 11 (machine mode)
11..10	Reserved	RZ	
15..12	LN	QRO	External IRQ Line Number assigned to this interrupt vector. This value is always equal to IPIC_IDX, because of the static line to vector mapping.
31..16	Reserved	RZ	

This is control status register for the interrupt vector, defined by the Index register (IPIC\_IDX).

## 7.5. IPIC timing diagrams

Figure 9, Figure 10 show IPIC and core signals timing to illustrate IRQ latency. See Table 36 for signals description.

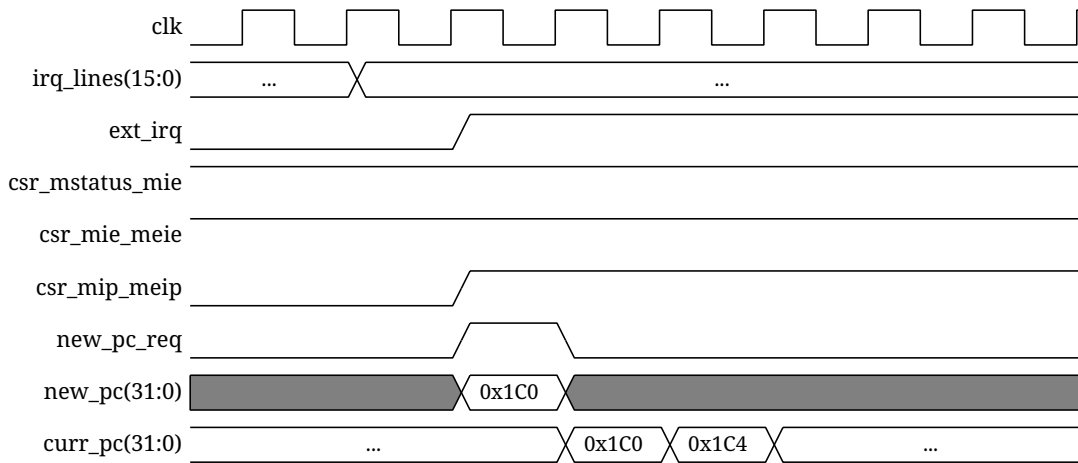


Figure 9: IRQ timing for level IRQs (IPIC synchronizer disabled)

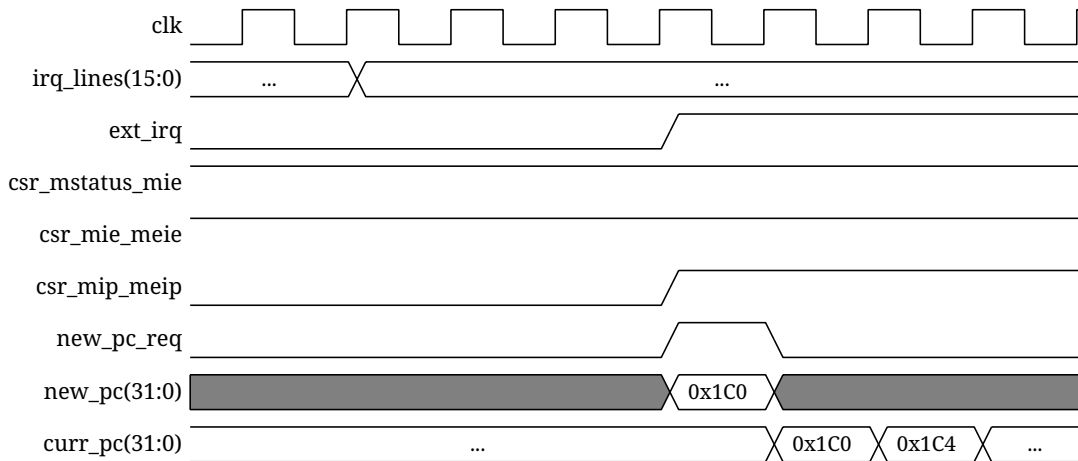


Figure 10: IRQ timing for level IRQs (IPIC synchronizer enabled)

**NOTE** For edge IRQs, latency is increased by one clock cycle.

Table 36: Signals description

Name	Description
clk	Core clock
irq_lines[15:0]	External IPIC IRQ lines
ext_irq	IPIC to core IRQ request
csr_mstatus_mie	Global interrupt enable
csr_mie_meie	External interrupt enable
csr_mip_meip	External interrupt pending
new_pc_req	New program counter request
new_pc[31:0]	New program counter
curr_pc[31:0]	Current program counter

# 8. Debug

## 8.1. Overview

The core's debug sub-system is implemented in compliance with the RISC-V External Debug Support specification [5]. Its block diagram is shown in [Figure 11](#).

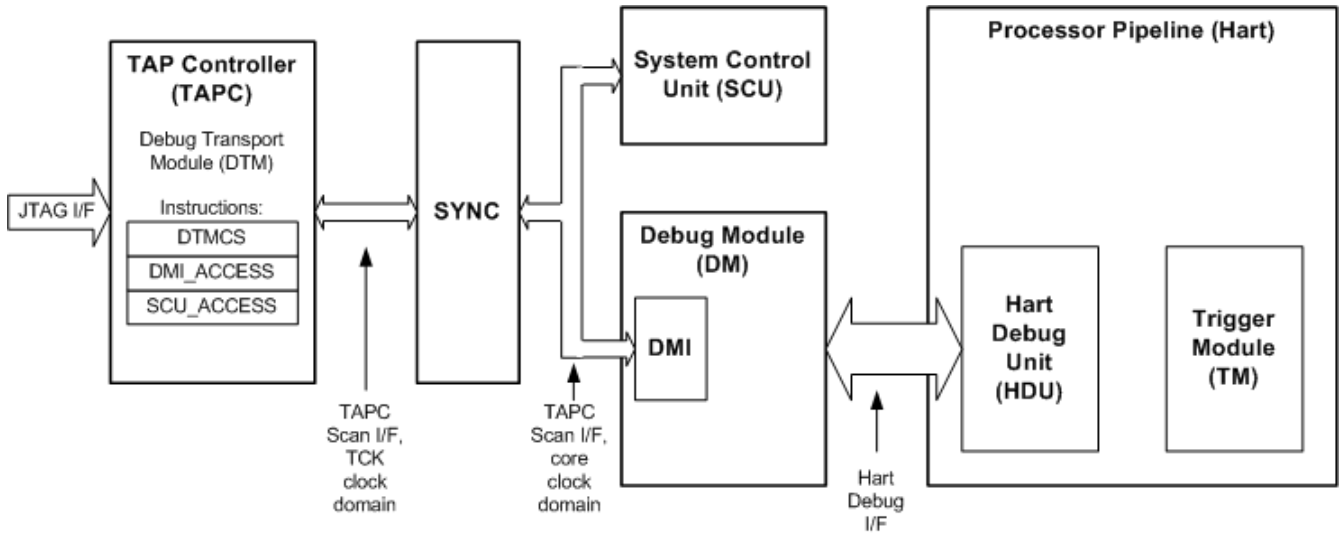


Figure 11: Debug Sub-System Block Diagram

An external debugger communicates with the core's debug sub-system via JTAG interface and TAP Controller (TAPC), playing a role of the Debug Transport Module (DTM) in terms of the RISC-V Debug Specification [5]. The TAPC implements several private TAP instructions allowing debugger to interact with internal debug units:

- DTMCS provides general control over DTM;
- DMI\_ACCESS provides access to the Debug Module (DM);
- SCU\_ACCESS provides access to the System Control Unit (SCU).

Internal connection between TAPC and DM, DM and SCU, is a form of serial scan interface. Source and destination of the TAPC scan interface are in different clock domains: TAPC is fully running in JTAG's TCK clock domain, whereas DM and SCU are in the core clock domain. Therefore, the TAPC scan interface passes through the clock synchronization unit (SYNC).

The System Control Unit (SCU) provides control over implementation-specific reset circuitry, and allows to monitor states of main reset signals. For details about it refer to the "[System Control Unit \(SCU\)](#)" section.

Using the Debug Module Interface (DMI), the Debug Module (DM) exposes a standard register interface to the core's debug features:

- run control of the core's single hart;
- access to its internal registers (GPRs, CSRs);
- access to its memory space;

- capability to execute arbitrary instructions from the Program Buffer.

The register interface is compliant with the RISC-V Debug Specification [5].

Implementation of this debug functionality within the hart is distributed between several units, and from external prospective the most important among them are:

- Hart Debug Unit (HDU) - provides the hart's Debug Interface, connecting the hart with the DM, and contains Debug CSRs;
- Trigger Module (TM) - provides a capability of hardware breakpoints, and contains Trigger CSRs.

## 8.2. TAP Controller (TAPC)

### 8.2.1. JTAG frequency requirement

#### IMPORTANT

The following ratio between System Clock (sys\_clk) and JTAG Clock (tck) frequencies must be met:  $\text{SysClkFreq} / \text{TckFreq} \geq 12$ .

### 8.2.2. TAPC Instruction Register (IR)

Instruction Register has length of 5 bits. After TAPC reset its value is 0x01, selecting the IDCODE instruction.



### 8.2.3. TAPC Instructions

TAP Controller Instructions are listed in [Table 37](#).

Table 37: TAP Controller Instructions

IR code	Mnemonic	Instruction full name	Description
0x00	-	Reserved	Equivalent to BYPASS.
0x01	IDCODE	IDCODE	IDCODE DR Read
0x02 - 0x03	-	Reserved	Equivalent to BYPASS.
0x04	BLD_ID	Build Identifier	BLD_ID DR Read
0x05 - 0x08	-	Reserved	Equivalent to BYPASS.
0x09	SCU_ACCESS	System Control Unit Access	Executes 4 operations over SCU registers: read, write, set bits, clear bits.
0x0A - 0x0F	-	Reserved	Equivalent to BYPASS.
0x10	DTMCS	DTM Control and Status	General control over Debug Transport Module (DTM).
0x11	DMI_ACCESS (DMI)	Debug Module Interface Access	Performs reading/writing of the Debug Module registers via Debug Module Interface.
0x12 - 0x1E	-	Reserved	Equivalent to BYPASS.
0x1F	BYPASS	BYPASS instruction	

### 8.2.4. TAPC Data Registers

#### 8.2.4.1. IDCODE

The IDCODE register is used to capture Device ID as shown in [Table 38](#). It is mandatory IEEE 1149.1 compliant register [3].

Table 38: IDCODE, DR-Capture Value

Bits	Name	Access	Reset Value	Description
0..31	IDCODE	RO	0xDEB11001	IDCODE Value. Current value of the IDCODE register for the core is 0xDEB11001.

#### 8.2.4.2. BYPASS

The BYPASS register is 1-bit mandatory IEEE 1149.1 compliant register [3]. The BYPASS register is described in [Table 39](#).

Table 39: BYPASS, DR-Capture Value

Bits	Name	Access	Reset Value	Description
0	Zero	RO	0	When TAP FSM is in DR-Capture state, the BYPASS register latches zero value at TCK rising edge.

### 8.2.4.3. DTMCS (DTM Control and Status)

The DTMCS register is described in [Table 40](#).

Table 40: DTMCS

Bits	Name	Access	Reset Value	Description
0..3	version	RO	0x1	DTM Version. The value corresponds to the one described in the RISC-V Debug Specification 0.13 (0x1).
4..9	abits	RO	0x7	Address Bits. The size of DMI_ACCESS.address bit field.
10..11	dmistat	RO	0x0	DMI Status. Encoding: <ul style="list-style-type: none"><li>• 0x0 - No error.</li><li>• 0x1 - Reserved. Must be interpreted the same as 2.</li><li>• 0x2 - An operation is failed (resulted in DMI_ACCESS.op of 2).</li><li>• 0x3 - An operation was attempted while a DMI access was still in progress (resulted in DMI_ACCESS.op of 3).</li></ul>
12..14	idle	RO	0x0	<p>This is a hint to the debugger of the minimum number of cycles a debugger should spend in RunTest/Idle after every DMI scan to avoid a "busy" return code (DTMCS.dmistat of 3). A debugger must still check DTMCS.dmistat when necessary.</p> <p>The given DMI implementation does not require entering to the RunTest/Idle state for proper operation, therefore the field indicates zero value.</p>
15	reserved	RO	0x0	Reserved for future use.
16	dmireset	RW1P	0x0	<p>Writing 1 to this bit clears the sticky error state and allows the DTM to retry or complete the previous transaction.</p> <p>Reading of the bit always returns 0.</p>

Bits	Name	Access	Reset Value	Description
17	dmihardreset	RW1P	0x0	<p>Writing 1 to this bit does a hard reset of the DTM, causing the DTM to forget about any outstanding DMI transactions.</p> <p>In general this should only be used when the Debugger has reason to expect that the outstanding DMI transaction will never complete (e.g. a reset condition caused an inflight DMI transaction to be cancelled).</p> <p>Reading of the bit always returns 0.</p>
18.. 31	reserved	RO	0x0	Reserved for future use.

#### 8.2.4.4. DMI\_ACCESS (DMI)

This register allows access to the Debug Module Interface (DMI).

In Capture-DR, the DTM updates DMI\_ACCESS.data with the result from previous operation, updating DMI\_ACCESS.op if the current op isn't sticky.

In Update-DR, the DTM starts the operation specified in DMI\_ACCESS.op unless the current status reported in DMI\_ACCESS.op is sticky.

The DMI\_ACCESS register is described in [Table 41](#).

*Table 41: DMI\_ACCESS (DMI)*

Bits	Name	Access	Reset Value	Description
0..1	op	RW	0	<p>Operation. When the debugger writes this field (in Update-DR state), it has the following meaning:</p> <ul style="list-style-type: none"> <li>• 0 - Ignore DMI_ACCESS.data and DMI_ACCESS.address (nop). Don't send anything over the DMI during Update-DR. This operation should never result in a busy or error response. The address and data reported in the following Capture-DR are undefined.</li> <li>• 1 - Read register specified by DMI_ACCESS.address (read).</li> <li>• 2 - Write data to the register specified by DMI_ACCESS.address (write).</li> <li>• 3 - Reserved.</li> </ul> <p>When the debugger reads this field (in Capture-DR state), it means the following:</p> <ul style="list-style-type: none"> <li>• 0 - The previous operation completed successfully.</li> <li>• 1 - Reserved.</li> <li>• 2 - A previous operation failed. The data scanned into DMI_ACCESS in this access will be ignored. This status is sticky and can be cleared by writing DTMCS.dmireset. This indicates that the DM itself responded with an error. There are no specified cases in which the DM would respond with an error, and DMI is not required to support returning errors.</li> <li>• 3 - An operation was attempted while a DMI request is still in progress. The data scanned into DMI_ACCESS in this access will be ignored. This status is sticky and can be cleared by writing DTMCS.dmireset. If a debugger sees this status, it needs to give the target more TCK edges between Update-DR and Capture-DR. The simplest way to do that is to add extra transitions in Run-Test/Idle.</li> </ul>
2..33	data	RW	0	<p>The data to send to the DM over the DMI during Update-DR, and the data returned from the DM as a result of the previous operation.</p>

Bits	Name	Access	Reset Value	Description
34.. 40	address	RW	0	Address used for DMI access. In Update-DR this value is used to access the DM over the DMI.

#### 8.2.4.5. SCU\_ACCESS

This register allows access to the System Control Unit (SCU). It is described in [Table 42](#).

*Table 42: SCU\_ACCESS*

Bits	Name	Access	Reset Value	Description
0..1	op	RW	0	<p>Operation. When the debugger writes this field (in Update-DR state), it has the following meaning:</p> <ul style="list-style-type: none"> <li>• 0 - Write data to the register specified by SCU_ACCESS.addr (WRITE).</li> </ul> <p>SCU_reg[SCU_ACCESS.addr] := SCU_ACCESS.data</p> <ul style="list-style-type: none"> <li>• 1 - Read the register specified by SCU_ACCESS.addr (READ). The read value (temp_data) could be received by Debugger in Capture-DR state during the next JTAG access cycle. SCU_ACCESS.data field is ignored.</li> </ul> <p>temp_data := SCU_reg[SCU_ACCESS.addr]</p> <ul style="list-style-type: none"> <li>• 2 - Set Bits (SET_BITS). Sets bits in the SCU register specified by SCU_ACCESS.addr (SCU_reg[SCU_ACCESS.addr]) in accordance with the bit mask provided in SCU_ACCESS.data.</li> </ul> <p>SCU_reg[SCU_ACCESS.addr] := SCU_ACCESS.data   SCU_reg[SCU_ACCESS.addr]</p> <ul style="list-style-type: none"> <li>• 3 - Clear Bits (CLR_BITS). Clears bits in the SCU register specified by SCU_ACCESS.addr (SCU_reg[SCU_ACCESS.addr]) in accordance with the bit mask provided in SCU_ACCESS.data.</li> </ul> <p>SCU_reg[SCU_ACCESS.addr] := (~SCU_ACCESS.data) &amp; SCU_reg[SCU_ACCESS.addr]</p> <p>When the debugger reads this field (in Capture-DR state), it contains the SCU_ACCESS.op value of the previous access.</p>
2..3	addr	RW	0	<p>Address used for SCU access. In Update-DR this value is used to specify the SCU register for the access.</p> <p>In Capture-DR this field contains the SCU_ACCESS.addr value used during the previous access.</p>

Bits	Name	Access	Reset Value	Description
4..7	data	RW	0	<p>Data used for SCU access. In Update-DR this value is used to access the SCU register.</p> <p>In Capture-DR this field contains the result of the previous access (temp_data).</p>

## 8.3. System Control Unit (SCU)

### 8.3.1. Overview

The System Control Unit (SCU) contains main components of the core reset sub-system, including a set of control/status registers and reset signals circuitry.

The registers provide the following capabilities:

- assertion/de-assertion of the System Reset;
- control over implementation-specific modes of the core reset signals behavior;
- monitoring of key reset signals' states, including their sticky status;
- convenient set of operations over a register value: read/write/set bits/clear bits.

Access to the SCU registers is performed via the TAP Controller and scan-chain interface (for details refer to the "[SCU\\_ACCESS](#)" section, [Table 42](#)).

The SCU's reset signals circuitry performs input reset signals synchronization, and provides necessary dependencies between reset inputs, internally generated resets and SCU's main product: reset outputs for key core components. It also supports Design-For-Test (DFT) mode of operation.

### 8.3.2. Block Diagram

SCU block diagram is shown in [Figure 12](#).

The SCU contains register file with scan-chain interface, and reset circuitry. The reset circuitry is composed from a set of DFT-friendly cells supporting Test Mode capability. All of them have connection with the SCU's test\_mode and test\_rst\_n inputs.

The reset circuitry has the following reset inputs:

- Power-Up Reset, pin pwrup\_rst\_n - the signal is intended for unconditional resetting of all logic inside the core after switching power on.
- Reset, pin rst\_n - the regular core reset used to put the core into a known state during a normal power session. In accordance with the RISC-V External Debug Support specification [5], the signal may not influence some core's components (e.g., Debug Module). This influence depends on the SCU's MODE register value (refer to the [Table 45](#)).
- CPU reset - the regular hardware reset input for putting the CPU into a known state. It doesn't reset the TAPC, DM logic.

- Non-DM Reset, pin `ndm_rst_n` - the reset signal from the Debug Module (DM), intended to reset all platform's components except DM itself. In the SCU it is used to assert Core Reset.



# System Control Unit (SCU)

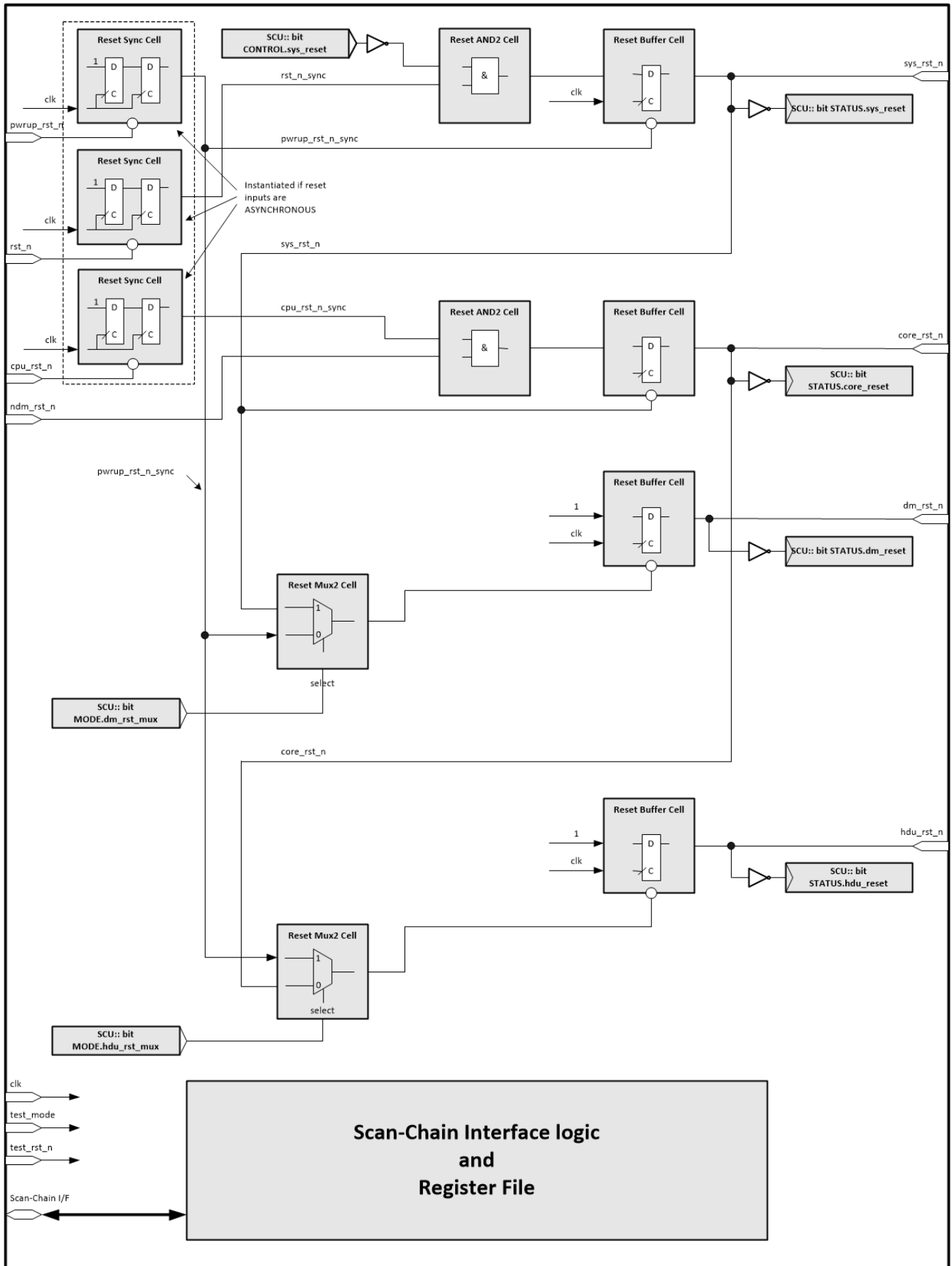


Figure 12: System Control Unit (SCU) Block Diagram

There are two options for Power-Up Reset and Reset inputs:

- **ASYNCHRONOUS** - the inputs are supposed to be asynchronous with the core's clock (clk); in that case Reset Synchronization Cells are instantiated inside SCU for those inputs.
- **SYNCHRONOUS** - the inputs are synchronous with the core's clock; in this case internal synchronous reset nets are connected immediately to the corresponding reset inputs.

SCR1 Processor Clusters have SCU instance with the **SYNCHRONOUS** Reset Inputs option.

The SCU generates the following key reset signals:

- **System Reset (pin sys\_rst\_n)** - the signal is used for regular resetting of all core's logic except DM and hart's debug components in certain reset sub-system modes (refer to the [Table 45](#)). Additionally to Power-Up Reset and Reset inputs, the signal can be asserted also by software via the CONTROL.sys\_reset bit ([Table 44](#)).
- **Core Reset (pin core\_rst\_n)** - the signal for resetting of the Pipeline (Hart) and all dependent units like memory sub-system, timer etc. It is influenced by the System Reset, CPU Reset and Non-DM Reset.
- **DM Reset (pin dm\_rst\_n)** - the signal for the Debug Module resetting. There are two modes of its operation in dependance of the MODE.dm\_rst\_mux bit value:
  - 0 (default/normal) - only Power-Up Reset activates the reset;
  - 1 (special) - the reset might be activated by the System Reset.
- **HDU Reset (pin hdu\_rst\_n)** - the reset signal for debug units inside the Pipeline (Hart): Hart Debug Unit (HDU) and Trigger Module (TM). This reset has also two operational modes controlled by the MODE.hdu\_rst\_mux bit:
  - 0 (default) - Core Reset influences the hart debug units;
  - 1 (special) - Core Reset DOES NOT activate resetting of the hart debug units. The mode might be convenient for use of all debug facilities like HW breakpoints through the core reset cycling process.

The reset circuitry provides capability of asynchronous/synchronous assertion and synchronous de-assertion of all output reset signals.

The SCU also contains logic for monitoring of the states of those generated output reset signals. Their instant states are reflected in the STATUS register ([Table 46](#)), and result of event accumulation - in the STICKY\_STATUS register ([Table 47](#)).

### 8.3.3. Registers

SCU registers are listed in [Table 43](#).

*Table 43: SCU Register Map*

Address	Mnemonic	Full name
0x0	CONTROL	SCU Control Register
0x1	MODE	SCU Mode Register
0x2	STATUS	SCU Status Register

Address	Mnemonic	Full name
0x3	STICKY_STATUS	SCU Sticky Status Register

### 8.3.3.1. CONTROL

The CONTROL register is described in [Table 44](#).

*Table 44: SCU CONTROL Register*

Bits	Name	Access	Reset Value	Description
0	sys_reset	RW	0	System Reset. If 1, activates System Reset of the core (equivalent to activation of the core's hardware rst_n input). Reading returns just state of the bit.
1..3	rsrv0	RW	0	Reserved for future use.

### 8.3.3.2. MODE

The MODE register is described in [Table 45](#).

*Table 45: SCU MODE Register*

Bits	Name	Access	Reset Value	Description
0	dm_rst_mux	RW	0	DM Reset Multiplexor. Encoding: <ul style="list-style-type: none"> <li>• 0 - System Reset DOES NOT activate hardware DM Reset.</li> <li>• 1 - System Reset DOES activate hardware DM Reset.</li> </ul> Reading returns state of the bit.
1	hdu_rst_mux	RW	0	HDU Reset Multiplexor. Encoding: <ul style="list-style-type: none"> <li>• 0 - HART reset DOES activate reset of Hart Debug Unit (HDU) and Trigger Module (TM) inside HART.</li> <li>• 1 - HART reset DOES NOT affect HDU and TM, so Debug/Trigger CSRs stay intact.</li> </ul> Reading returns state of the bit.
2..3	rsrv0	RW	0	Reserved for future use.

### 8.3.3.3. STATUS

The STATUS register is described in [Table 46](#).

*Table 46: SCU STATUS Register*

Bits	Name	Access	Reset Value	Description
0	sys_reset	RO	0	System Reset. Reading returns current state of the System Reset: <ul style="list-style-type: none"><li>• 0 - de-asserted;</li><li>• 1 - asserted.</li></ul>
1	core_reset	RO	0	Core Reset. Reading returns current state of the Core Reset: <ul style="list-style-type: none"><li>• 0 - de-asserted;</li><li>• 1 - asserted.</li></ul>
2	dm_reset	RO	0	DM Reset. Reading returns current state of the DM Reset: <ul style="list-style-type: none"><li>• 0 - de-asserted;</li><li>• 1 - asserted.</li></ul>
3	hdu_reset	RO	0	HDU Reset. Reading returns current state of the HDU Reset: <ul style="list-style-type: none"><li>• 0 - de-asserted;</li><li>• 1 - asserted.</li></ul>

### 8.3.3.4. STICKY\_STATUS

The STICKY\_STATUS register is described in [Table 47](#).

*Table 47: SCU STICKY\_STATUS Register*

Bits	Name	Access	Reset Value	Description
0	sys_reset	RW1C	0	System Reset. Reading returns sticky state of the System Reset: <ul style="list-style-type: none"> <li>• 0 - the reset has not been asserted after the last bit clearing;</li> <li>• 1 - the reset has been asserted at least once. Clearing of the bit should be performed via CLR_BITS operation.</li> </ul>
1	core_reset	RW1C	0	Core Reset. Reflects sticky state of the Core Reset. Behavior is the same as for STICKY_STATUS.sys_reset bit.
2	dm_reset	RW1C	0	DM Reset. Reflects sticky state of the DM Reset. Behavior is the same as for STICKY_STATUS.sys_reset bit.
3	hdu_reset	RW1C	0	HDU Reset. Reflects sticky state of the HDU Reset. Behavior is the same as for STICKY_STATUS.sys_reset bit.

Note: the register supports only READ and CLR\_BITS operations.

## 8.4. Debug Module (DM)

### 8.4.1. Overview

The Debug Module implements a translation interface between abstract debug operations and their specific implementation. It might support the following operations:

1. Give the debugger necessary information about the implementation.
2. Allow the core's single hart to be halted and resumed.
3. Provide status if the hart is halted.
4. Provide abstract read and write access to a halted hart's GPRs.
5. Provide access to a reset signal that allows debugging from the very first instruction after reset.
6. Provide a mechanism to allow debugging hart immediately out of reset (regardless of the reset cause).
7. Provide abstract access to non-GPR hart registers.
8. Provide a Program Buffer to force the hart to execute arbitrary instructions.
9. Allow memory access from a hart's point of view.

### 8.4.2. Debug Module Interface (DMI)

Debug Module is a slave to a virtual bus called the Debug Module Interface (DMI). The master of the bus is the TAP Controller (TAPC) playing a role of the Debug Transport Module (DTM).

The DMI in the given core implements 7 address bits. It supports read and write operations. The bottom of the address space is used for the DM.

The Debug Module is controlled via register accesses to its DMI address space.

### 8.4.3. Hart States

In accordance with the RISC-V Debug Specification 0.13 every hart is in exactly one of four states. Which state the hart is in is reflected by `DMSTATUS.allnonexistent`, `anynonexistent`, `allunavail`, `anyunavail`, `allrunning`, `anyrunning`, `allhalted`, and `anyhalted`.

Harts are nonexistent if they will never be part of this system, no matter how long a user waits. Harts are unavailable if they might exist/become available at a later time, or if there are other harts with higher indexes than this one. The given core has only single hart, and it is always existent and available, so `DMSTATUS.allnonexistent`, `anynonexistent`, `allunavail`, `anyunavail` bits are hardwired to zero.

The single hart is running when it is executing normally, as if no debugger was attached. This includes being in a low power mode or waiting for an interrupt, as long as a halt request will result in the hart being halted.

The hart is halted when it is in Debug Mode, only performing tasks on behalf of the debugger.

### 8.4.4. Reset Control

The Debug Module controls a global reset signal, `ndmreset` (non-debug module reset), which can reset, or hold in reset, every component in the platform, except for the Debug Module and Debug Transport Modules. Exactly what is affected by this reset is implementation dependent, as long as it is possible to debug programs from the first instruction executed. The Debug Module's own state and registers should only be reset at power-up and while `dmactive` in `dmcontrol` is 0. The halt state of harts should be maintained across system reset provided that `dmactive` is 1, although trigger CSRs may be cleared.

Due to clock and power domain crossing issues, it is not possible to perform arbitrary DMI accesses across system reset. While `ndmreset` or any external reset is asserted, the only supported DM operation is accessing `dmcontrol`. The behavior of other accesses is undefined.

There is no requirement on the duration of the assertion of `ndmreset`. The implementation must ensure that a write of `ndmreset` to 1 followed by a write of `ndmreset` to 0 triggers system reset. The system may take an arbitrarily long time to come out of reset, as reported by `allunavail`, `anyunavail`.

When hart has been reset, it sets a sticky `havereset` state bit. The conceptual `havereset` state bits can be read for the hart in `DMSTATUS.anyhavereset` and `allhavereset`. These bits are set regardless of the cause of the reset. The `havereset` bits for the hart can be cleared by writing 1 to `DMCONTROL.ackhavereset`. The `havereset` bits are cleared when `DMCONTROL.dmactive` is low.

## 8.4.5. Run Control

For the hart, the Debug Module tracks 2 conceptual bits of state: halt request and resume ack. These 2 bits reset to 0. The DM receives from the hart the following status signals: halted, running and havereset. The debugger can observe the state of resume ack in allresumeack and anyresumeack, and the state of halted, running, and havereset signals in allhalted, anyhalted, allrunning, anyrunning, allhavereset, and anyhavereset. The state of the other bits cannot be observed directly.

When a debugger writes 1 to DMCONTROL.haltreq, the hart's halt request bit is set. When a running hart sees its halt request bit high, it responds by halting, deasserting its running signal, and asserting its halted signal. Halted hart ignore their halt request bit.

When a debugger writes 1 to DMCONTROL.resumereq, the hart's resume ack bit is cleared and halted hart is sent a resume request. Hart responds by resuming, clearing its halted signal, and asserting its running signal. At the end of this process the resume ack bit is cleared. These status signals of the hart are reflected in DMSTATUS.allresumeack, anyresumeack, allrunning, and anyrunning. Resume request is ignored by running hart.

## 8.4.6. Abstract Commands

The DM supports a set of abstract commands, including those ones which might be performed when the hart is not halted. Debuggers can only determine which abstract commands are supported by the hart in a given state by attempting them and then looking at ABSTRACTCS.cmderr to see if they were successful. Commands may be supported with some options set, but not with other options set. If a command has unsupported options set, the DM sets ABSTRACTCS.cmderr to 2 (not supported).

Debuggers execute abstract commands by writing them to COMMAND. They can determine whether an abstract command is complete by reading ABSTRACTCS.busy. After completion, ABSTRACTCS.cmderr indicates whether the command was successful or not. Commands may fail because a hart is not halted, not running, unavailable, or because they encounter an error during execution.

If the command takes arguments, the debugger must write them to the DATA0/1 registers before writing to COMMAND. If a command returns results, the Debug Module puts them in the DATA0/1 registers before ABSTRACTCS.busy is cleared. Which DATA registers are used for the arguments is described in [Table 48](#). In all cases the least-significant word is placed in the lowest-numbered DATA register. The argument width depends on the command being executed.

*Table 48: Use of DATA Registers*

Argument Width	arg0/return value	arg1
32	DATA0	DATA1

Before starting an abstract command, a debugger must ensure that DMCONTROL.haltreq, resumereq, and ackhavereset are all 0.

While an abstract command is executing (ABSTRACTCS.busy is high), a debugger must not write 1 to DMCONTROL.haltreq, resumereq, or ackhavereset.

If an abstract command does not complete in the expected time and appears to be hung, the following procedure can be attempted to abort the command: first the debugger resets the hart (using DMCONTROL.ndmreset), and then it resets the Debug Module (using DMCONTROL.dmactive).

The Abstract Command interface is designed to allow a debugger to write commands as fast as possible, and then later check whether they completed without error. In the common case the debugger will be much slower than the target and commands succeed, which allows for maximum throughput. If there is a failure, the interface ensures that no commands execute after the failing one. To discover which command failed, the debugger has to look at the state of the DM (e.g. contents of DATA0) or hart (e.g. contents of a register modified by a Program Buffer program) to determine which one failed.

#### 8.4.6.1. Supported Abstract Commands

Each abstract command is a 32-bit value. The top 8 bits contain cmdtype bit field which determines the kind of command. The core supports two commands listed in [Table 49](#).

Table 49: Supported Abstract Commands

cmdtype	Command	Description Reference
0	Access Register	<a href="#">Access Register</a>
1	-	Not supported
2	Access Memory	<a href="#">Access Memory</a>

#### 8.4.6.2. Access Register

This command gives the debugger access to hart registers and allows it to execute the Program Buffer. Its format and fields are described in [Table 50](#).

Table 50: Access Register Command

Bits	Name	Description
0.. 15	regno	Number of the register to access, as described in the <a href="#">Table 51</a> . DPC may be used as an alias for PC when this command is supported on a non-halted hart.
16	write	When transfer is set: <ul style="list-style-type: none"><li>• 0 - Copy data from the specified register into arg0 portion of data.</li><li>• 1 - Copy data from arg0 portion of data into the specified register.</li></ul>



Bits	Name	Description
17	transfer	<p>Encoding:</p> <ul style="list-style-type: none"> <li>• 0 - Don't do the operation specified by write.</li> <li>• 1 - Do the operation specified by write.</li> </ul> <p>Zeroing of this bit can be used to just execute the Program Buffer without having to worry about placing valid values into aarsize or regno.</p>
18	postexec	<p>Encoding:</p> <ul style="list-style-type: none"> <li>• 0 - No effect.</li> <li>• 1 - Execute the program in the Program Buffer exactly once after performing the transfer, if any.</li> </ul>
19	aarpostincrement	The feature is not supported. So, the bit must be zero.
20.. 22	aarsize	<p>Encoding:</p> <ul style="list-style-type: none"> <li>• 2 - Access the lowest 32 bits of the register.</li> <li>• others - not supported.</li> </ul> <p>If aarsize specifies a size larger than the register's actual size, then the access fails. If a register is accessible, then reads of aarsize less than or equal to the register's actual size is supported. This field controls the Argument Width as referenced in the <a href="#">Table 48</a>.</p>
23	rsrv0	Reserved for future use. Must be zero.
24.. 31	cmdtype	This is 0 to indicate Access Register Command.

Mapping of regno indexes into the hart registers is listed in [Table 51](#).

*Table 51: Abstract Register Numbers Mapping*

regno value	Hart registers category
0x0000 .. 0x0FFF	CSRs. The "PC" can be accessed here through DPC.
0x1000 .. 0x101F	GPRs
0xC000 .. 0xFFFF	Reserved for future use.

The command performs the following sequence of operations:

1. If write is clear and transfer is set, then copy data from the register specified by regno into the arg0 region of data, and perform any side effects that occur when this register is read from M-mode.
2. If write is set and transfer is set, then copy data from the arg0 region of data into the register

specified by regno, and perform any side effects that occur when this register is written from M-mode.

3. Execute the Program Buffer, if postexec is set.

If any of these operations fail, ABSTRACTCS.cmderr is set and none of the remaining steps are executed. If the failure is that the requested register does not exist in the hart, cmderr is set to 3 (exception).

The Debug Module supports read and write access to all GPRs, CSRs and FPRs when the hart is halted. Besides of that, the Debug Module supports reading of the following registers, when the hart is running:

- MISA CSR (0x301)
- MVENDORID CSR (0xF11)
- MARCHID CSR (0xF12)
- MIMPID CSR (0xF13)
- MHARTID CSR (0xF14)
- MVENDORID CSR (0xF11)
- DPC CSR (0x7B1) - used as alias for PC register for its sampling.

#### 8.4.6.3. Access Memory

This command lets the debugger perform memory accesses, with the exact same memory view and permissions as the hart has. This includes access to hart-local memory-mapped registers, etc. Its format and fields are described in [Table 52](#).

Table 52: Access Memory Command

Bits	Name	Description
0.. 15	rsrv0	Reserved for future use. Must be zero.
16	write	Encoding: <ul style="list-style-type: none"><li>• 0 - Copy data from the memory location specified in arg1 into arg0 portion of data.</li><li>• 1 - Copy data from arg0 portion of data into the memory location specified in arg1.</li></ul>
17.. 18	rsrv1	Reserved for future use. Must be zero.
19	aampostincrement	The feature is not supported. So, the bit must be zero.

Bits	Name	Description
20.. 22	aamsize	<p>Encoding:</p> <ul style="list-style-type: none"> <li>• 0 - Access the lowest 8 bits of the memory location.</li> <li>• 1 - Access the lowest 16 bits of the memory location.</li> <li>• 2 - Access the lowest 32 bits of the memory location.</li> <li>• others - not supported.</li> </ul> <p>This field controls the Argument Width as referenced in the <a href="#">Table 48</a>.</p>
23	aamvirtual	The core supports only physical addresses. So, the bit is hardwired to zero.
24.. 31	cmdtype	This is 2 to indicate Access Memory Command.

The command performs the following sequence of operations:

1. Copy data from the memory location specified in arg1 into the arg0 portion of data, if write is clear.
2. Copy data from the arg0 portion of data into the memory location specified in arg1, if write is set.

If any of these operations fail, ABSTRACTCS.cmderr is set and none of the remaining steps are executed. An access may only fail if the hart, running M-mode code, might encounter that same failure when it attempts the same access.

The Debug Module supports read and write access to memory locations only when the hart is halted.

This command modifies arg0 only when memory is read. The other data registers are not changed.

#### 8.4.7. Program Buffer

To support executing arbitrary instructions on a halted hart, the Debug Module includes the Program Buffer that a debugger can write small programs to.

A debugger can write a small program to the Program Buffer, and then execute it exactly once with the Access Register Abstract Command, setting the postexec bit in command. The debugger can write whatever program it likes (including jumps out of the Program Buffer), but the program must end with ebreak or c.ebreak. The core implementation supports an implied ebreak that is executed when a hart runs off the end of the Program Buffer. This is indicated by DMSTATUS.impebreak.

ABSTRACTCS.progbufsize indicates the actual size of the Program Buffer. It is possible that the Program Buffer can hold only one 32- or 16-bit instruction, so the debugger must only write a single instruction in this case, regardless of its size. This instruction can be a 32-bit instruction, or a compressed instruction in the lower 16 bits accompanied by a compressed nop in the upper 16 bits.

While these programs are executed, the hart does not leave Debug Mode. If an exception is

encountered during execution of the Program Buffer, no more instructions are executed, the hart remains in Debug Mode, and ABSTRACTCS.cmderr is set to 3 (exception error). If the debugger executes a program that doesn't terminate with an ebreak instruction, the hart will remain in Debug Mode and the debugger will lose control of the hart.

Executing the Program Buffer does not clobber DPC. However, the debugger must attempt to save DPC between halting and executing a Program Buffer, and then restore DPC before leaving Debug Mode.

## 8.4.8. DM Registers

### 8.4.8.1. Register Map

DM registers are listed in [Table 53](#).

*Table 53: DM Register Map*

Address	Mnemonic	Full name
0x00.. 0x03	-	Reserved
0x04	DATA0	Abstract Data 0
0x05	DATA1	Abstract Data 1
0x06.. 0x0F	-	Reserved
0x10	DMCONTROL	Debug Module Control
0x11	DMSTATUS	Debug Module Status
0x12	HARTINFO	Hart Info
0x13.. 0x15	-	Reserved
0x16	ABSTRACTCS	Abstract Control and Status
0x17	COMMAND	Abstract Command
0x18	ABSTRACTAUTO	Abstract Command Autoexec
0x19.. 0x1F	-	Reserved
0x20.. 0x25	PROGBUF[0:5]	Program Buffer 0 .. Program Buffer 5
0x26.. 0x3F	-	Reserved
0x40	HALTSUM0	Halt Summary 0
0x41.. 0x7F	-	Reserved

### 8.4.8.2. Debug Module Control (DMCONTROL)

The DMCONTROL register is described in [Table 54](#).

*Table 54: DMCONTROL Register*

Bits	Name	Access	Reset Value	Description
0	dmactive	RW	0	<p>This bit serves as a reset signal for the Debug Module itself. Meaning:</p> <ul style="list-style-type: none"> <li>• 0 - The module's state, including authentication mechanism, takes its reset values. In that state the DMCONTROL.dmactive bit is the only bit which can be written to something other than its reset value.</li> <li>• 1 - The module functions normally.</li> </ul> <p>A debugger may pulse this bit low to get the Debug Module into a known state.</p>
1	ndmreset	RW	0	This bit controls the reset signal from the DM to the rest of the system. To perform a system reset the debugger writes 1, and then writes 0 to deassert the reset.
2.. 15	rsrv0	RO	0	Reserved for future use.
16.. 25	hartsello	RO	0	The low 10 bits of hartsel: the DM-specific index of the hart to select. In the given implementation core has only 1 HART, therefore the field is hardwired to zero.
26.. 27	rsrv1	RO	0	Reserved for future use.
28	ackhavereset	RW1P	0	Writing 0 has no effect. Writing 1 clears DMSTATUS.havereset.
29	rsrv2	RO	0	Reserved for future use.
30	resumereq	RW1P	0	<p>Writing 1 causes the hart to resume once, if it is halted when the write occurs. It also clears the resume ack bit.</p> <p>DMCONTROL.resumereq is ignored if haltreq is set.</p>
31	haltreq	RW1P	0	<p>Writing 0 clears the halt request bit. This may cancel outstanding halt request for the hart.</p> <p>Writing 1 sets the halt request bit for hart. Running hart will halt whenever its halt request bit is set.</p>

#### 8.4.8.3. Debug Module Status (DMSTATUS)

The DMSTATUS register is described in [Table 55](#).

*Table 55: DMSTATUS Register*

Bits	Name	Access	Reset Value	Description
0.. 3	version	RO	2	The value of 2 means that Debug Module conforms to the RISC-V Debug Spec version 0.13.
4.. 6	rsrv0	RO	0	Reserved for future use.
7	authenticated	RO	1	The bit is hardwired to 1, as authentication is not implemented.
8	anyhalted	RO	-	This field is 1 when the hart is halted.
9	allhalted	RO	-	This field is 1 when the hart is halted.
10	anyrunning	RO	-	This field is 1 when the hart is running.
11	allrunning	RO	-	This field is 1 when the hart is running.
12	anyunavail	RO	0	The bit is hardwired to 0 as the only hart is always available.
13	allunavail	RO	0	The bit is hardwired to 0 as the only hart is always available.
14	anynonexistent	RO	0	The bit is hardwired to 0 as the only hart is always existent.
15	allnonexistent	RO	0	The bit is hardwired to 0 as the only hart is always existent.
16	anyresumeack	RO	0	This field is 1 when the hart has acknowledged its last resume request.
17	allresumeack	RO	0	This field is 1 when the hart has acknowledged its last resume request.
18	anyhavereset	RO	-	This field is 1 when the hart has been reset, and reset has not been acknowledged for the hart.
19	allhavereset	RO	-	This field is 1 when the hart has been reset, and reset has not been acknowledged for the hart.
20.. 21	rsrv1	RO	0	Reserved for future use.
22	impebreak	RO	1	If 1, then there is an implicit ebreak instruction at the non-existent word immediately after the Program Buffer. This saves the debugger from having to write the ebreak itself, and allows the Program Buffer to be one word smaller.
29.. 31	rsrv2	RO	0	Reserved for future use.

#### 8.4.8.4. Hart Info (HARTINFO)

The HARTINFO register is described in [Table 56](#).

*Table 56: HARTINFO Register*

Bits	Name	Access	Reset Value	Description
0.. 11	dataaddr	RO	0x7b2	The fields indicates the index of the first CSR dedicated to shadowing the DM DATA registers.
12.. 15	datasize	RO	1	The fields indicates the number of CSRs dedicated to shadowing the DM DATA registers.
16	dataaccess	RO	0	The field is hardwired to 0 to indicate that the DM DATA registers are shadowed in the hart by CSRs.
17.. 19	rsrv0	RO	0	Reserved for future use.
20.. 23	nscratch	RO	1	The field reflects the number of DSCRATCH CSRs available for the debugger to use during Program Buffer execution, starting from DSCRATCH0. The debugger can make no assumptions about the contents of these registers between commands.
24.. 31	rsrv1	RO	0	Reserved for future use.

#### 8.4.8.5. Halt Summary 0 (HALTSUM0)

The HALTSUM0 register is described in [Table 57](#).

*Table 57: HALTSUM0 Register*

Bits	Name	Access	Reset Value	Description
0	hart0	RO	0	The bit indicates whether the only existent hart (hart0) is halted or not.
1.. 31	rsrv0	RO	0	Reserved for future use.

#### 8.4.8.6. Abstract Control and Status (ABSTRACTCS)

The ABSTRACTCS register is described in [Table 58](#).

*Table 58: ABSTRACTCS Register*

Bits	Name	Access	Reset Value	Description
0.. 3	datacount	RO	2	The fields indicates the number of Abstract Data registers that are implemented as part of the abstract command interface.
4.. 7	rsrv0	RO	0	Reserved for future use.

Bits	Name	Access	Reset Value	Description
8.. 10	cmderr	RW1C	0	<p>Gets set if an abstract command fails. The bits in this field remain set until they are cleared by writing 1 to them. No abstract command is started until the value is reset to 0. This field only contains a valid value if busy is 0.</p> <p>Encoding:</p> <ul style="list-style-type: none"> <li>• 0 (none) - No error.</li> <li>• 1 (busy) - An abstract command was executing while command or abstractcs was written, or when one of the DATA or PROGBUF registers was read or written. This status is only written if ABSTRACTCS.cmderr contains 0.</li> <li>• 2 (not supported) - The requested command is not supported, regardless of whether the hart is running or not.</li> <li>• 3 (exception) - An exception occurred while executing the command (e.g. while executing the Program Buffer).</li> <li>• 4 (halt/resume) - The abstract command couldn't execute because the hart wasn't in the required state (running/halted), or unavailable.</li> <li>• 7 (other) - The command failed for another reason.</li> </ul>
11	rsrv1	RO	0	Reserved for future use.
12	busy	RO	0	If 1, an abstract command is currently being executed. This bit is set as soon as COMMAND is written, and is not cleared until that command has completed.
13.. 23	rsrv2	RO	0	Reserved for future use.
24.. 28	progbufsize	RO	6	Size of the Program Buffer, in 32-bit words.
29.. 31	rsrv3	RO	0	Reserved for future use.

#### 8.4.8.7. Abstract Command (COMMAND)

Writes to this register cause the corresponding abstract command to be executed.

Writing this register while an abstract command is executing causes ABSTRACTCS.cmderr to be set to 1 (busy) if it is 0.



If ABSTRACTCS.cmderr is non-zero, writes to this register are ignored.

The COMMAND register is described in [Table 59](#).

*Table 59: COMMAND Register*

Bits	Name	Access	Reset Value	Description
0..23	control	W	0	This field is interpreted in a command-specific manner, described for each abstract command.
24..31	cmdtype	W	0	The type determines the overall functionality of this abstract command.

#### 8.4.8.8. Abstract Command Autoexec (ABSTRACTAUTO)

This register is intended to make burst accesses more efficient.

Writing this register while an abstract command is executing causes ABSTRACTCS.cmderr to be set to 1 (busy) if it is 0.

The ABSTRACTAUTO register is described in [Table 60](#).

*Table 60: ABSTRACTAUTO Register*

Bits	Name	Access	Reset Value	Description
0..1	autoexecdata	RW	0	When a bit in this field is 1, read or write accesses to the corresponding DATA word cause the command in COMMAND to be executed again.
2..31	rsrv0	RO	0	Reserved for future use.

#### 8.4.8.9. Abstract Data 0/1 (DATA0/1)

DATA0/DATA1 are basic read/write registers that may be read or changed by abstract commands.

Accessing these registers while an abstract command is executing causes ABSTRACTCS.cmderr to be set to 1 (busy) if it is 0.

Attempts to write them while busy is set does not change their value.

The values in these registers may not be preserved after an abstract command is executed. The only guarantees on their contents are the ones offered by the command in question. If the command fails, no assumptions can be made about the contents of these registers.

The DATA registers are described in [Table 61](#).

*Table 61: DATA Register*

Bits	Name	Access	Reset Value	Description
0.. 31	data	RW	0	The field contains the data used in abstract command.

#### 8.4.8.10. Program Buffer [0:5] (PROGBUF[0:5])

PROGBUF[0:5] provide read/write access to the Program Buffer.

Accessing the registers while an abstract command is executing causes ABSTRACTCS.cmderr to be set to 1 (busy) if it is 0.

Attempts to write them while busy is set does not change their value.

The PROGBUF[0:5] registers are described in [Table 62](#).

*Table 62: PROGBUF[0:5] Registers*

Bits	Name	Access	Reset Value	Description
0.. 31	instr	RW	0	The field contains an instruction to be executed by the hart in Debug Mode as a part of abstract command execution (if postexec bit is set).

## 8.5. Hart Debug Unit (HDU)

### 8.5.1. Overview

Hart Debug Unit (HDU) is a component inside a hart implementing control over its debug features and providing interface to the Debug Module for that. It drives transitions between hart debug states (reset/running/halted), as well as process of execution instructions from the Program Buffer.

HDU is the unit where Debug CSRs are situated.

### 8.5.2. Debug Mode

Debug Mode is a special processor mode used only when a hart is halted for external debugging.

When executing code from the Program Buffer, the hart stays in Debug Mode and the following apply:

1. All operations are executed at machine mode privilege level, except that MSTATUS.mprv may be ignored according to DCSR.mprven.
2. All interrupts (including NMI) are masked.
3. Exceptions don't update any registers. That includes CAUSE, EPC, TVAL, DPC, and MSTATUS. They do end execution of the Program Buffer.
4. No action is taken if a trigger matches.
5. Counters are not stopped in the given core implementation.

6. Timers are not stopped in the given core implementation.
7. The wfi instruction acts as a nop.
8. Almost all instructions that change the privilege level have undefined behavior. This includes ecall, mret, and uret. The only exception is ebreak. When that is executed in Debug Mode, it halts the hart again but without updating DPC or DCSR.
9. Completing Program Buffer execution is considered output for the purpose of fence instructions.

### 8.5.3. Reset

There are two modes of hart reset behavior:

1. Hardware hart reset signal influences all hart registers, including Debug CSRs (Table 63) and Trigger CSRs (Table 68). This mode is default after Power-Up Reset.
2. Hardware hart reset signal as usual influences architectural hart registers but DOES NOT reset Debug and Trigger CSRs keeping them intact. The mode is activated when SCU's `MODE.hdu_rst_mux = 1`.

The 2nd mode allows for debugger software to utilize debug features (e.g., hardware breakpoints) over hart reset cycling.

HDU also provides a mechanism to allow debugging the hart immediately out of reset. If the halt signal (driven by the hart's halt request bit in the Debug Module) is asserted when the hart comes out of reset, the hart enters Debug Mode before executing any instructions, but after performing any initialization that would usually happen before the first instruction is executed.

### 8.5.4. Single Step

A debugger can cause a halted hart to execute a single instruction and then re-enter Debug Mode by setting `DCSR.step` before setting `DMCONTROL.resumereq`.

If executing or fetching that instruction causes an exception, Debug Mode is re-entered immediately after the PC is changed to the exception handler and the appropriate TVAL and CAUSE registers are updated.

If executing or fetching the instruction causes a trigger to fire, Debug Mode is re-entered immediately after that trigger has fired. In that case `DCSR.cause` is set to 2 (trigger) instead of 4 (single step). In the given core implementation the instruction caused that trigger firing, is not executed.

If the instruction that is executed causes the PC to change to an address where an instruction fetch causes an exception, that exception does not occur until the next time the hart is resumed. Similarly, a trigger at the new address does not fire until the hart actually attempts to execute that instruction.

If the instruction being stepped over is wfi and would normally stall the hart, then instead the instruction is treated as nop.

### 8.5.5. Debug CSRs

They are CSRs, accessible using the RISC-V csr opcodes and optionally also using abstract debug commands.

These registers are only accessible from Debug Mode.

#### 8.5.5.1. Register Map

Debug CSRs are listed in [Table 63](#).

*Table 63: Hart Debug CSRs Map*

Address	Mnemonic	Full name
0x7B0	DCSR	Debug Control and Status
0x7B1	DPC	Debug PC
0x7B2	DSCRATCH0	Debug Scratch Register 0
0x7B3.. 0x7BF	-	Reserved

#### 8.5.5.2. Debug Control and Status (DCSR)

The DCSR register is described in [Table 64](#).

*Table 64: DCSR Register*

Bits	Name	Access	Reset Value	Description
0.. 1	prv	RW	3	Contains the privilege level the hart was operating in when Debug Mode was entered. The SCR1 core has the field hardwired to 3 - Machine Mode.
2	step	RW	0	When set and not in Debug Mode, the hart will only execute a single instruction and then enter Debug Mode. If the instruction does not complete due to an exception, the hart will immediately enter Debug Mode before executing the trap handler, with appropriate exception registers set. The debugger must not change the value of this bit while the hart is running.
3.. 5	rsrv0	RO	0	Reserved for future use.

Bits	Name	Access	Reset Value	Description
6.. 8	cause	RO	0	<p>Explains why Debug Mode was entered. When there are multiple reasons to enter Debug Mode in a single cycle, hardware should set cause to the cause with the highest priority.</p> <p>Encoding:</p> <ul style="list-style-type: none"> <li>• 1 - An ebreak instruction was executed (priority 3);</li> <li>• 2 - The Trigger Module caused a breakpoint exception (priority 4, highest);</li> <li>• 3 - The debugger requested entry to Debug Mode using haltreq (priority 1);</li> <li>• 4 - The hart single stepped because step was set (priority 0, lowest);</li> </ul> <p>Other values are reserved for future use</p>
9.. 10	rsrv1	RO	0	Reserved for future use.
11	stepie	RW	0	<p>Encoding:</p> <ul style="list-style-type: none"> <li>• 0 - Interrupts are disabled during single stepping.</li> <li>• 1 - Interrupts are enabled during single stepping.</li> </ul> <p>The debugger must not change the value of this bit while the hart is running.</p>
12.. 14	rsrv2	RO	0	Reserved for future use.
15	ebreakm	RW	0	<p>Encoding:</p> <ul style="list-style-type: none"> <li>• 0 - ebreak instructions in M-mode behave as described in the Privileged Spec.</li> <li>• 1 - ebreak instructions in M-mode enter Debug Mode.</li> </ul>
16.. 27	rsrv3	RO	0	Reserved for future use.
28.. 31	xdebugver	RO	4	The field's value (4) indicates that debug support exists as described in the RISC-V Debug Spec version 0.13.

### 8.5.5.3. Debug PC (DPC)

Upon entry to debug mode, DPC is updated with the virtual address of the next instruction to be executed. The behavior is described in more detail in [Table 65](#).

Table 65: Virtual address in DPC upon Debug Mode Entry

Cause	Virtual Address in DPC
ebreak	Address of the ebreak instruction.
Single Step	Address of the instruction that would be executed next if no debugging was going on. Ie. PC + 4 for 32-bit instructions that don't change program flow, the destination PC on taken jumps/branches, etc.
Trigger Module	The address of the instruction which caused the trigger to fire (as MCONTROL.timing is always 0 in the given core).
Halt request	Address of the next instruction to be executed at the time that debug mode was entered.

When resuming, the hart's PC is updated to the virtual address stored in DPC. A debugger may write DPC to change where the hart resumes.

The DPC register is described in [Table 66](#).

Table 66: DPC Register

Bits	Name	Access	Reset Value	Description
0..31	dpc	RW	-	The field contains the Debug PC value.

### 8.5.5.4. Debug Scratch Register 0 (DSCRATCH0)

The DSCRATCH0 register is described in [Table 67](#).

Table 67: DSCRATCH0 Register

Bits	Name	Access	Reset Value	Description
0..31	data	RW	0	The field contains the data might be used by instructions executed from the Program Buffer.

# 9. Hardware Trigger Module

## 9.1. Overview

Triggers can cause a breakpoint exception or entry into Debug Mode. This makes them invaluable when debugging code from ROM. They can trigger on execution of instructions at a given memory address, or on the address in loads/stores. These are all features that can be useful without having the Debug Module present, so the Trigger Module is represented as a separate unit that can be implemented separately.

Triggers do not fire while in Debug Mode.

## 9.2. Reset

There are two modes of reset behavior:

1. Hardware hart reset signal influences all hart registers, including Debug CSRs ([Table 63](#)) and Trigger CSRs ([Table 68](#)). This mode is default after Power-Up Reset.
2. Hardware hart reset signal as usual influences architectural hart registers but DOES NOT reset Debug and Trigger CSRs keeping them intact. The mode is activated when SCU's `MODE.hdu_rst_mux = 1`.

The 2nd mode allows for debugger software to utilize debug features (e.g., hardware breakpoints) over hart reset cycling.

## 9.3. Operation Basics

### 9.3.1. Enumeration

Each RISC-V Debug Spec compliant trigger may support a variety of features. A debugger can build a list of all triggers and their features as follows:

1. Write 0 to TSELECT.
2. Read back TSELECT and check that it contains the written value. If not, exit the loop.
3. Read TINFO.
4. If that caused an exception, the debugger must read TDATA1 to discover the type. (If type is 0, this trigger doesn't exist. Exit the loop.)
5. If TINFO.info is 1, this trigger doesn't exist. Exit the loop.
6. Otherwise, the selected trigger supports the types discovered in TINFO.info.
7. Repeat, incrementing the value in TSELECT.

## 9.4. Trigger CSRs

These registers are CSRs, accessible using the RISC-V csr opcodes and optionally also using abstract

debug commands.

Some combinations of activated features might be unsupported. All TDATA registers follow write-any-read-legal semantics. If a debugger writes an unsupported configuration, the register will read back a value that is supported (which may simply be a disabled trigger). This means that a debugger must always read back values it writes to TDATA registers, unless it already knows already what is supported. Writes to one TDATA register may not modify the contents of other TDATA registers, nor the configuration of any trigger besides the one that is currently selected.

The trigger registers are only accessible in Machine and Debug Mode to prevent untrusted user code from causing entry into Debug Mode without the OS's permission.

### 9.4.1. Register Map

Trigger CSRs are listed in [Table 68](#).

Table 68: Hart Trigger CSRs Map

Address	Mnemonic	Full name
0x7A0	TSELECT	Trigger Select
0x7A1	TDATA1 / MCONTROL / ICOUNT	Trigger Data 1 / Match Control / Instruction Count
0x7A2	TDATA2	Trigger Data 2
0x7A3	-	Reserved
0x7A4	TINFO	Trigger Info
0x7A5.. 0x7AF	-	Reserved

### 9.4.2. Trigger Select (TSELECT)

This register determines which trigger is accessible through the other trigger registers. The set of accessible triggers starts at 0, and is contiguous.

Writes of values greater than or equal to the number of supported triggers may result in a different value in this register than what was written. To verify that what they wrote is a valid index, debuggers can read back the value and check that tselect holds what they wrote.

Since triggers can be used both by Debug Mode and M-mode, the debugger must restore this register if it modifies it.

The TSELECT register is described in [Table 69](#).

Table 69: TSELECT Register

Bits	Name	Access	Reset Value	Description
0.. 1	index	RW	0	The field determines which trigger is accessible through the other trigger registers.  Maximal index supported in the given core is 2.



Bits	Name	Access	Reset Value	Description
2.. 31	rsrv0	RO	0	Reserved for future use.

### 9.4.3. Trigger Data 1 (TDATA1)

The TDATA1 register is described in [Table 70](#).

*Table 70: TDATA1 Register*

Bits	Name	Access	Reset Value	Description
0.. 26	data	RW	0	Trigger-specific data.
27	dmode	RW	0	Encoding: <ul style="list-style-type: none"> <li>• 0 - Both Debug and M-mode can write the TDATA registers at the selected TSELECT.</li> <li>• 1 - Only Debug Mode can write the TDATA registers at the selected TSELECT. Writes from other modes are ignored. This bit is only writable from Debug Mode.</li> </ul>
28.. 31	type	RW	0	Encoding: <ul style="list-style-type: none"> <li>• 0 - There is no trigger at this TSELECT.</li> <li>• 2 - The trigger is an address match trigger. The remaining bits in this register act as described in MCONTROL.</li> <li>• 3 - The trigger is an instruction count trigger. The remaining bits in this register act as described in ICOUNT.</li> <li>• 15 - This trigger exists (so enumeration shouldn't terminate), but is not currently available.</li> </ul> <p>Other values are reserved for future use.</p>

### 9.4.4. Match Control (MCONTROL)

This register is accessible as tdata1 when type is 2.

The MCONTROL register is described in [Table 71](#).

*Table 71: MCONTROL Register*

Bits	Name	Access	Reset Value	Description
0	load	RW	0	When set, the trigger fires on the virtual address of a load.
1	store	RW	0	When set, the trigger fires on the virtual address of a store.
2	execute	RW	0	When set, the trigger fires on the virtual address of an instruction that is executed.
3.. 5	rsrv0	RO	0	Reserved for future use.
6	m	RW	0	When set, enable this trigger in M-mode.
7.. 10	match	RW	0	Encoding: <ul style="list-style-type: none"> <li>• 0 - Matches when the value equals TDATA2.</li> </ul> Other values are reserved for future use.
11	chain	RW	0	Encoding: <ul style="list-style-type: none"> <li>• 0 - When this trigger matches, the configured action is taken.</li> <li>• 1 - While this trigger does not match, it prevents the trigger with the next index from matching.</li> </ul> <p>A trigger chain starts on the first trigger with chain = 1 after a trigger with chain = 0, or simply on the first trigger if that has chain = 1. It ends on the first trigger after that which has chain = 0. This final trigger is part of the chain. The action on all but the final trigger is ignored. The action on that final trigger will be taken if and only if all the triggers in the chain match at the same time.</p> <p>Because chain affects the next trigger, hardware must zero it in writes to MCONTROL that set MCONTROL.dmode to 0 if the next trigger has dmode of 1. In addition hardware should ignore writes to MCONTROL that set dmode to 1 if the previous trigger has both dmode of 0 and chain of 1. Debuggers must avoid the latter case by checking chain on the previous trigger if they're writing MCONTROL.</p>

Bits	Name	Access	Reset Value	Description
12.. 15	action	RW	0	<p>The action to take when the trigger fires. Encoding:</p> <ul style="list-style-type: none"> <li>• 0 - Raise a breakpoint exception. (Used when software wants to use the trigger module without an external debugger attached.)</li> <li>• 1 - Enter Debug Mode. (Only supported when the trigger's dmode is 1.)</li> <li>• 2..5 - Reserved for use by the trace specification.</li> <li>• Others - Reserved for future use.</li> </ul>
16.. 17	szelo	RO	0	The field is hardwired to zero. Thus, the trigger will attempt to match against an access of any size.
18	timing	RO	0	<p>The bit is hardwired to 0.</p> <p>That means the action for this trigger will be taken just before the instruction that triggered it is executed, but after all preceding instructions are committed.</p>
19	select	RO	0	<p>The bit is hardwired to 0.</p> <p>That means the trigger performs a match only on the virtual address.</p>
20	hit	RW	0	<p>The hardware sets this bit when the given trigger matches. The trigger's user can set or clear it at any time.</p> <p>It is used to determine which trigger(s) matched.</p>
21.. 26	rsrv1	RO	0	Reserved for future use.
27	dmode	RW	0	The bit is described as a part of TDATA1 register.
28.. 31	type	RW	0	The bit field is described as a part of TDATA1 register.

### 9.4.5. Instruction Count (ICOUNT)

This register is accessible as TDATA1 when type is 3.

This trigger type is intended to be used as a single step that's useful both for external debuggers and for software monitor programs. For that case count must be equal 1.

The ICOUNT register is described in [Table 72](#).

*Table 72: ICOUNT Register*

Bits	Name	Access	Reset Value	Description
0.. 5	action	RW	0	<p>The action to take when the trigger fires. Encoding:</p> <ul style="list-style-type: none"> <li>• 0 - Raise a breakpoint exception. (Used when software wants to use the trigger module without an external debugger attached.)</li> <li>• 1 - Enter Debug Mode. (Only supported when the trigger's dmode is 1.)</li> <li>• 2..5 - Reserved for use by the trace specification.</li> <li>• Others - Reserved for future use.</li> </ul>
6.. 8	rsrv0	RO	0	Reserved for future use.
9	m	RW	0	When set, every instruction completed or exception taken in M-mode decrements count by 1.
10.. 23	count	RW	1	When count is decremented to 0, the trigger fires.
24	hit	RW	0	<p>The hardware sets this bit when the given trigger matches. The trigger's user can set or clear it at any time.</p> <p>It is used to determine which trigger(s) matched.</p>
25.. 26	rsrv1	RO	0	Reserved for future use.
27	dmode	RW	0	The bit is described as a part of TDATA1 register.
28.. 31	type	RW	0	The bit field is described as a part of TDATA1 register.

#### 9.4.6. Trigger Data 2 (TDATA2)

The TDATA2 register is described in [Table 73](#).

*Table 73: TDATA2 Register*

Bits	Name	Access	Reset Value	Description
0.. 31	data	RW	-	The field contains trigger-specific data.

#### 9.4.7. Trigger Info (TINFO)

The TINFO register is described in [Table 74](#).

*Table 74: TINFO Register*

Bits	Name	Access	Reset Value	Description
0.. 15	info	RO	0	<p>The bit field indicates supported types for the selected trigger: one bit for each possible type enumerated in TDATA1. Bit N corresponds to type N. If the bit is set, then that type is supported by the currently selected trigger.</p> <p>If the currently selected trigger doesn't exist, this field contains 1.</p> <p>Trigger capabilities in the given core are distributed as follows:</p> <ul style="list-style-type: none"> <li>• 0..1 - support trigger of type = 2 only (MCONTROL), info = 0x04;</li> <li>• 2 - supports trigger of type = 3 only (ICOUNT), info = 0x08;</li> <li>• other indexes indicate info = 0x01.</li> </ul>
16.. 31	rsrv0	RO	0	Reserved for future use.

# 10. External Interfaces

## 10.1. AHB-Lite Interface

AHB-Lite external interface consists of two separate AHB-Lite master buses for instructions and data. Interface signals are listed in [Table 75](#).

Table 75: AHB-Lite external interface

Name	Direction	Description
<b>AHB-Lite instruction interface</b>		
imem_hprot[3:0]	output	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection
imem_hburst[2:0]	output	Indicates if the transfer forms part of a burst
imem_hsize[2:0]	output	Indicates the size of the transfer
imem_htrans[1:0]	output	Indicates the type of the current transfer
imem_hmastlock	output	Indicates that the current transfer is part of a locked sequence
imem_haddr[31:0]	output	The 32-bit address bus
imem_hready	input	When '1' the HREADY signal indicates that a transfer has finished on the bus
imem_hrdata[31:0]	input	The read data bus is used to transfer data from bus slaves to the bus master during read operations
imem_hresp[1:0]	input	The transfer response provides additional information on the status of a transfer
<b>AHB-Lite data interface</b>		
dmem_hprot[3:0]	output	The protection control signals provide additional information about a bus access and are primarily intended for use by any module that wishes to implement some level of protection
dmem_hburst[2:0]	output	Indicates if the transfer forms part of a burst
dmem_hsize[2:0]	output	Indicates the size of the transfer
dmem_htrans[1:0]	output	Indicates the type of the current transfer
dmem_hmastlock	output	Indicates that the current transfer is part of a locked sequence
dmem_haddr[31:0]	output	The 32-bit address bus
dmem_hwrite	output	1 - write transfer; 0 - read transfer
dmem_hwdata[31:0]	output	The write data bus is used to transfer data from the master to the bus slaves during write operations
dmem_hready	input	When '1' the HREADY signal indicates that a transfer has finished on the bus

Name	Direction	Description
dmem_hrdata[31:0]	input	The read data bus is used to transfer data from bus slaves to the bus master during read operations
dmem_hresp[1:0]	input	The transfer response provides additional information on the status of a transfer

Both AHB-Lite bridges (instruction and data) have optional input and output registers, which can be switched on to meet design timing requirements. The registers are disabled by default. See [SCR1 configurable options](#) for details.

## 10.2. AHB-Lite Timing diagrams

Figure 13 shows example of data memory AHB-Lite read/write.

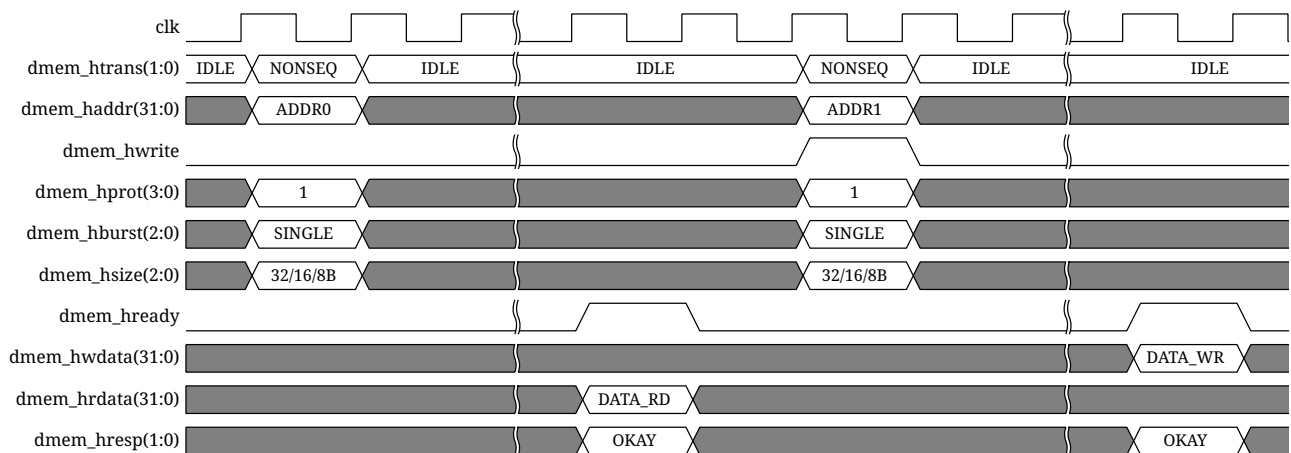


Figure 13: Data memory AHB-Lite read/write

### IMPORTANT

SCR1 does not perform sequential read or write requests to **data memory**, it always waits for a transaction to finish before initiating another one.

Figure 14 shows example of instruction memory AHB-Lite read with delay.

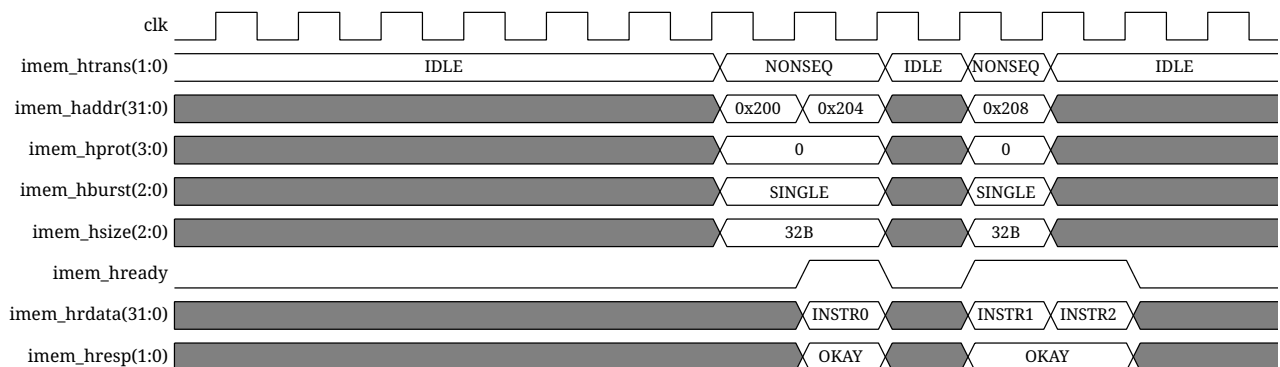


Figure 14: Instruction memory AHB-Lite read with delay

## 10.3. AXI4 Interface

Instruction memory AXI4 write data channel signals are shown in [Table 76](#).

The IMEM AXI4 write data channel, IMEM AXI4 write response channel, and IMEM AXI4 write address channel are provided for compatibility with AXI4 specification. All output ports of these three channels are hardwired to 0. All input ports of these three channels must be connected to constant 0.

*Table 76: IMEM AXI4 write data channel signals*

Name	Direction	Description
io_axi_imem_wdata[31:0]	output	Master Write data
io_axi_imem_wstrb[3:0]	output	Master Write strobes. This signal indicates which byte lanes hold valid data. There is one write strobe bit for each eight bits of the write data bus
io_axi_imem_wlast	output	Master Write last. This signal indicates the last transfer in a write burst
io_axi_imem_wuser[3:0]	output	Master User signal. Optional User-defined signal in the write data channel
io_axi_imem_wvalid	output	Master Write valid. This signal indicates that valid write data and strobes are available
io_axi_imem_wready	input	Slave Write ready. This signal indicates that the slave can accept the write data

Instruction memory AXI4 write response channel signals are shown in [Table 77](#).

*Table 77: IMEM AXI4 write response channel signals*

Name	Direction	Description
io_axi_imem_bid[3:0]	input	Slave Response ID tag. This signal is the ID tag of the write response
io_axi_imem_bresp[1:0]	input	Slave Write response. This signal indicates the status of the write transaction
io_axi_imem_bvalid	input	Slave Write response valid. This signal indicates that the channel is signaling a valid write response
io_axi_imem_buser[3:0]	input	Slave User signal. Optional User-defined signal in the write response channel
io_axi_imem_bready	output	Master Response ready. This signal indicates that the master can accept a write response

Instruction memory AXI4 write address channel signals are shown in [Table 78](#).

*Table 78: IMEM AXI4 write address channel signals*

Name	Direction	Description
io_axi_imem_awid[3:0]	output	Master Write address ID. This signal is the identification tag for the write address group of signals



Name	Direction	Description
io_axi_imem_awaddr[31:0]	output	Master Write address. The write address gives the address of the first transfer in a write burst transaction
io_axi_imem_awlen[7:0]	output	Master Burst length. The burst length gives the exact number of transfers in a burst. This information determines the number of data transfers associated with the address
io_axi_imem_awsiz[2:0]	output	Master Burst size. This signal indicates the size of each transfer in the burst
io_axi_imem_awburst[1:0]	output	Master Burst type. The burst type and the size information, determine how the address for each transfer within the burst is calculated
io_axi_imem_awlock	output	Master Lock type. Provides additional information about the atomic characteristics of the transfer
io_axi_imem_awcache[3:0]	output	Master Memory type. This signal indicates how transactions are required to progress through a system
io_axi_imem_awprot[2:0]	output	Master Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access
io_axi_imem_awregion[3:0]	output	Master Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces
io_axi_imem_awuser[3:0]	output	Master User signal. Optional User-defined signal in the write address channel
io_axi_imem_awqos[3:0]	output	Master Quality of Service, QoS. The QoS identifier sent for each write transaction
io_axi_imem_awvalid	output	Master Write address valid. This signal indicates that the channel is signaling valid write address and control information
io_axi_imem_awready	input	Slave Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals

Instruction memory AXI4 read address channel signals are shown in [Table 79](#).

Table 79: IMEM AXI4 read address channel signals

Name	Direction	Description
io_axi_imem_arid[3:0]	output	Master Read address ID. This signal is the identification tag for the read address group of signals. This output is hardwired to constant 0
io_axi_imem_araddr[31:0]	output	Master Read address. The read address gives the address of the first transfer in a read burst transaction
io_axi_imem_arlen[7:0]	output	Master Burst length. This signal indicates the exact number of transfers in a burst. This output is hardwired to constant 0

Name	Direction	Description
io_axi_imem_arsize[2:0]	output	Master Burst size. This signal indicates the size of each transfer in the burst. This output is hardwired to constant 2
io_axi_imem_arburst[1:0]	output	Master Burst type. The burst type and the size information determine how the address for each transfer within the burst is calculated. This output is hardwired to constant 1
io_axi_imem_arlock	output	Master Lock type. This signal provides additional information about the atomic characteristics of the transfer. This output is hardwired to constant 0
io_axi_imem_arcache[3:0]	output	Master Memory type. This signal indicates how transactions are required to progress through a system. This output is hardwired to constant 2
io_axi_imem_arprot[2:0]	output	Master Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access. This output is hardwired to constant 0
io_axi_imem_arregion[3:0]	output	Master Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. This output is hardwired to constant 0
io_axi_imem_aruser[3:0]	output	Master User signal. Optional User-defined signal in the read address channel. This output is hardwired to constant 0
io_axi_imem_arqos[3:0]	output	Master Quality of Service, QoS. QoS identifier sent for each read transaction. This output is hardwired to constant 0
io_axi_imem_arvalid	output	Master Read address valid. This signal indicates that the channel is signaling valid read address and control information
io_axi_imem_arready	input	Slave Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals

Instruction memory AXI4 read data channel signals are shown in [Table 80](#).

*Table 80: IMEM AXI4 read data channel signals*

Name	Direction	Description
io_axi_imem_rid[3:0]	input	Slave Read ID tag. This signal is the identification tag for the read data group of signals generated by the slave
io_axi_imem_rdata[31:0]	input	Slave Read data
io_axi_imem_rresp[1:0]	input	Slave Read response. This signal indicates the status of the read transfer
io_axi_imem_rlast	input	Slave Read last. This signal indicates the last transfer in a read burst
io_axi_imem_ruser[3:0]	input	Slave User signal. Optional User-defined signal in the read data channel

Name	Direction	Description
io_axi_imem_rvalid	input	Slave Read valid. This signal indicates that the channel is signaling the required read data
io_axi_imem_rready	output	Master Read ready. This signal indicates that the master can accept the read data and response information

Data memory AXI4 write address channel signals are shown in [Table 81](#).

*Table 81: DMEM AXI write address channel signals*

Name	Direction	Description
io_axi_dmem_awid[3:0]	output	Master Write address ID. This signal is the identification tag for the write address group of signals. This output is hardwired to constant 1
io_axi_dmem_awaddr[31:0]	output	Master Write address. The write address gives the address of the first transfer in a write burst transaction
io_axi_dmem_awlen[7:0]	output	Master Burst length. The burst length gives the exact number of transfers in a burst. This output is hardwired to constant 0
io_axi_dmem_awsize[2:0]	output	Master Burst size. This signal indicates the size of each transfer in the burst
io_axi_dmem_awburst[1:0]	output	Master Burst type. The burst type and the size information, determine how the address for each transfer within the burst is calculated. This output is hardwired to constant 1
io_axi_dmem_awlock	output	Master Lock type. Provides additional information about the atomic characteristics of the transfer. This output is hardwired to constant 0
io_axi_dmem_awcache[3:0]	output	Master Memory type. This signal indicates how transactions are required to progress through a system. This output is hardwired to constant 2
io_axi_dmem_awprot[2:0]	output	Master Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access. This output is hardwired to constant 0
io_axi_dmem_awregion[3:0]	output	Master Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. This output is hardwired to constant 0
io_axi_dmem_awuser[3:0]	output	Master User signal. Optional User-defined signal in the write address channel. This output is hardwired to constant 0
io_axi_dmem_awqos[3:0]	output	Master Quality of Service, QoS. The QoS identifier sent for each write transaction. This output is hardwired to constant 0
io_axi_dmem_awvalid	output	Master Write address valid. This signal indicates that the channel is signaling valid write address and control information

Name	Direction	Description
io_axi_dmem_awready	input	Slave Write address ready. This signal indicates that the slave is ready to accept an address and associated control signals

Data memory AXI4 write data channel signals are shown in [Table 82](#).

*Table 82: DMEM AXI write data channel signals*

Name	Direction	Description
io_axi_dmem_wdata[31:0]	output	Master Write data
io_axi_dmem_wstrb[3:0]	output	Master Write strobes. This signal indicates which byte lanes hold valid data. There is one write strobe bit for each eight bits of the write data bus
io_axi_dmem_wlast	output	Master Write last. This signal indicates the last transfer in a write burst
io_axi_dmem_wuser[3:0]	output	Master User signal. Optional User-defined signal in the write data channel. This output is hardwired to constant 0
io_axi_dmem_wvalid	output	Master Write valid. This signal indicates that valid write data and strobes are available
io_axi_dmem_wready	input	Slave Write ready. This signal indicates that the slave can accept the write data

Data memory AXI4 write response channel signals are shown in [Table 83](#).

*Table 83: DMEM AXI4 write response channel signals*

Name	Direction	Description
io_axi_dmem_bid[3:0]	input	Slave Response ID tag. This signal is the ID tag of the write response
io_axi_dmem_bresp[1:0]	input	Slave Write response. This signal indicates the status of the write transaction
io_axi_dmem_bvalid	input	Slave Write response valid. This signal indicates that the channel is signaling a valid write response
io_axi_dmem_buser[3:0]	input	Slave User signal. Optional User-defined signal in the write response channel
io_axi_dmem_bready	output	Master Response ready. This signal indicates that the master can accept a write response

Data memory AXI4 read address channel signals are shown in [Table 84](#).

*Table 84: DMEM AXI read address channel signals*

Name	Direction	Description
io_axi_dmem_arid[3:0]	output	Master Read address ID. This signal is the identification tag for the read address group of signals. This output is hardwired to constant 0

Name	Direction	Description
io_axi_dmem_araddr[31:0]	output	Master Read address. The read address gives the address of the first transfer in a read burst transaction
io_axi_dmem_arlen[7:0]	output	Master Burst length. This signal indicates the exact number of transfers in a burst. This output is hardwired to constant 0
io_axi_dmem_arsize[2:0]	output	Master Burst size. This signal indicates the size of each transfer in the burst
io_axi_dmem_arburst[1:0]	output	Master Burst type. The burst type and the size information determine how the address for each transfer within the burst is calculated. This output is hardwired to constant 1
io_axi_dmem_arlock	output	Master Lock type. This signal provides additional information about the atomic characteristics of the transfer. This output is hardwired to constant 0
io_axi_dmem_arcache[3:0]	output	Master Memory type. This signal indicates how transactions are required to progress through a system. This output is hardwired to constant 2
io_axi_dmem_arprot[2:0]	output	Master Protection type. This signal indicates the privilege and security level of the transaction, and whether the transaction is a data access or an instruction access. This output is hardwired to constant 0
io_axi_dmem_arregion[3:0]	output	Master Region identifier. Permits a single physical interface on a slave to be used for multiple logical interfaces. This output is hardwired to constant 0
io_axi_dmem_aruser[3:0]	output	Master User signal. Optional User-defined signal in the read address channel. This output is hardwired to constant 0
io_axi_dmem_arqos[3:0]	output	Master Quality of Service, QoS. QoS identifier sent for each read transaction. This output is hardwired to constant 0
io_axi_dmem_arvalid	output	Master Read address valid. This signal indicates that the channel is signaling valid read address and control information
io_axi_dmem_arready	input	Slave Read address ready. This signal indicates that the slave is ready to accept an address and associated control signals

Data memory AXI4 read data channel signals are shown in [Table 85](#).

*Table 85: DMEM AXI4 read data channel signals*

Name	Direction	Description
io_axi_dmem_rid[3:0]	input	Slave Read ID tag. This signal is the identification tag for the read data group of signals generated by the slave
io_axi_dmem_rdata[31:0]	input	Slave Read data
io_axi_dmem_rresp[1:0]	input	Slave Read response. This signal indicates the status of the read transfer

Name	Direction	Description
io_axi_dmem_rlast	input	Slave Read last. This signal indicates the last transfer in a read burst
io_axi_dmem_ruser[3:0]	input	Slave User signal. Optional User-defined signal in the read data channel
io_axi_dmem_rvalid	input	Slave Read valid. This signal indicates that the channel is signaling the required read data
io_axi_dmem_rready	output	Master Read ready. This signal indicates that the master can accept the read data and response information

## 10.4. AXI4 Timing diagrams

The AXI4 interface of the core defines the following independent transaction channels between the core (master) and external memory (slave) [4]:

- read address channel;
- read data channel;
- write address channel;
- write data channel;
- write response channel.

An address channel carries control information that describes the nature of the data to be transferred. The data is transferred between master and slave using either:

- A write data channel to transfer data from the master to the slave. In a write transaction, the slave uses the write response channel to signal to the master the completion of the transfer;
- A read data channel to transfer data from the slave to the master.

The AXI4 interface of the core permits address information to be issued ahead of the actual data transfer.

Figure 15 shows read and write transaction from perspective of used channels.

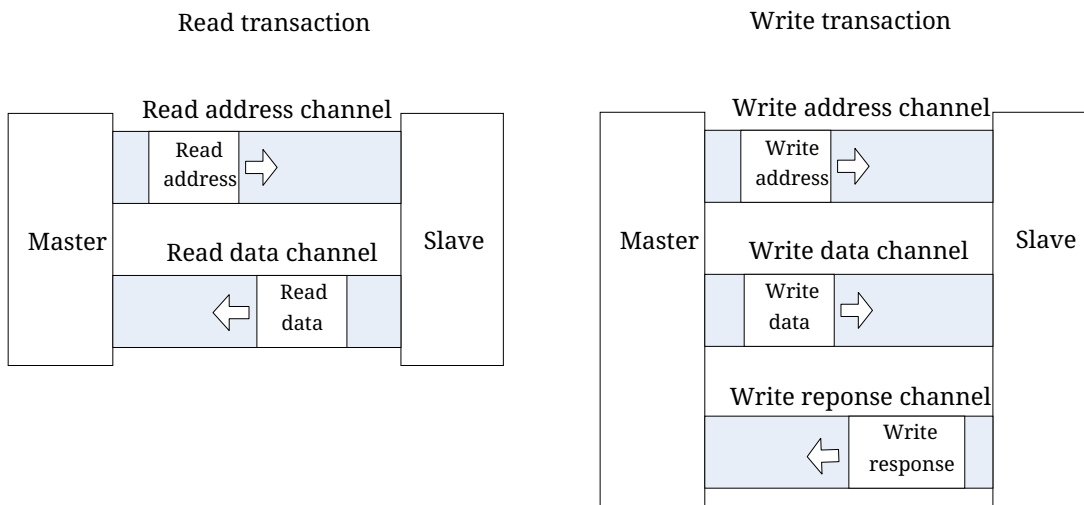


Figure 15: AXI4 read and write channels

Following timing diagrams show example of two read transactions and two write transactions that could happen in the following sequence scenario:

- 1) Read RD0 data word from address RA0;
- 2) Read RD1 data word from address RA1;
- 3) Write WD0 data word to address WA0;
- 4) Write WD1 data word to address WA1.

Figure 16 shows read word transaction from perspective of interface signals.

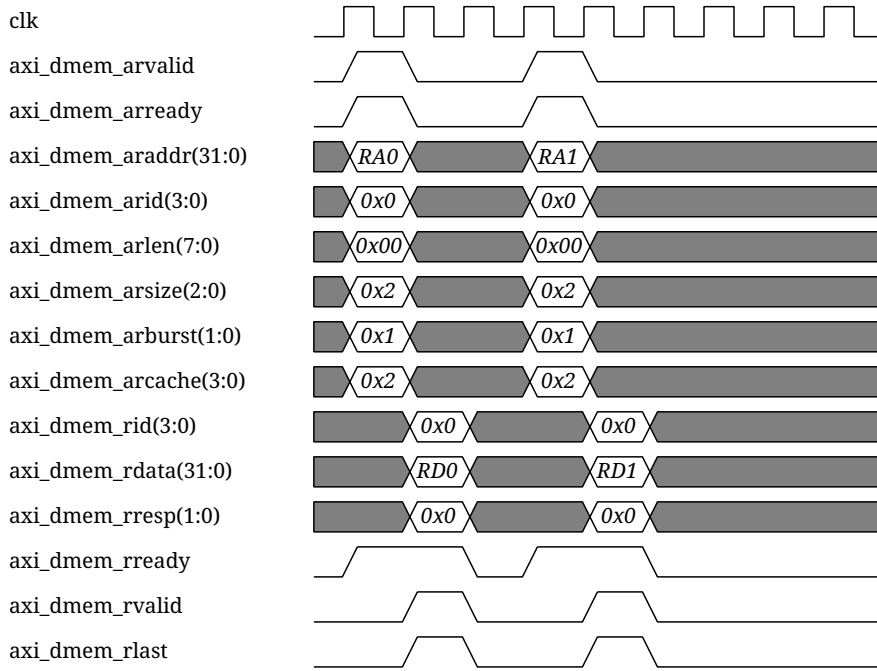


Figure 16: AXI4 read word transaction

Figure 17 shows write word transaction from perspective of interface signals.

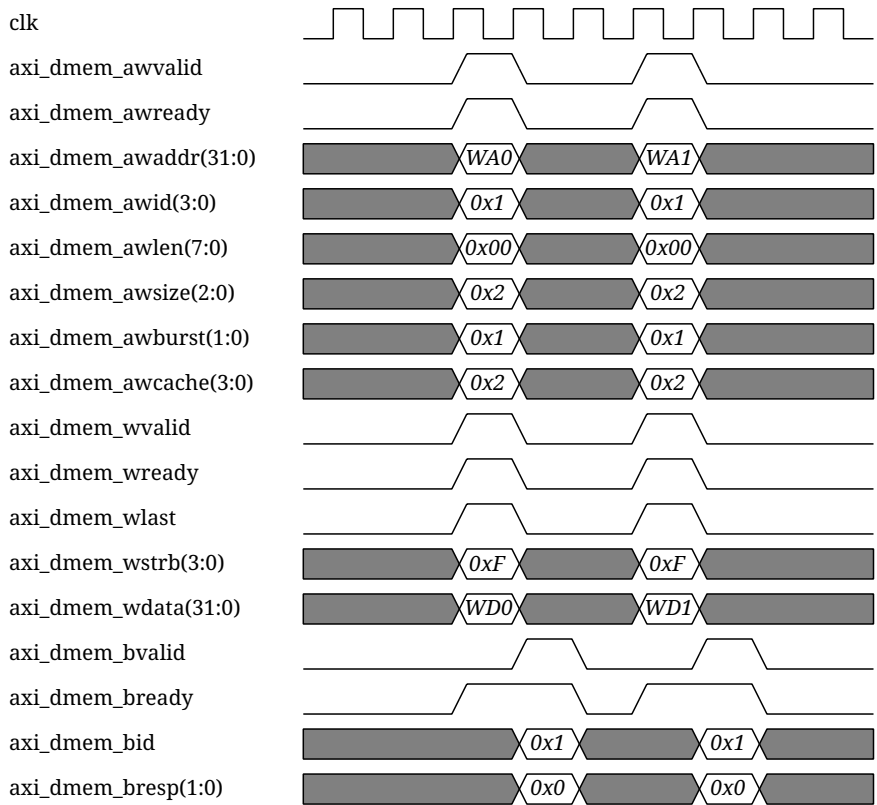


Figure 17: AXI4 write word transaction



AXI4 interface has no ordering restrictions between read and write transactions. They can complete in any order, even if the `axi_dmem_arid` value of a read transaction is the same as the `axi_dmem_awid` value of a write transaction. If a master requires a given relationship between a read transaction and a write transaction then it must ensure that the earlier transaction is complete before it issues the later transaction.

A master can only consider the earlier transaction is complete when:

- for a read transaction, it receives the last of the read data;
- for a write transaction, it receives the write response.

Sending all of the write data for the write transaction must not be considered as completion of that transaction.

AXI4 interface is able to flag as an error the DECERR and SLVERR types of responses that may appear in `bresp[1:0]` or `rresp[1:0]` signals, as presented in [4]. The EXOKAY type of response is also considered as error since the exclusive access is never requested by the core. All types of errors are processed in the same way and flagged as memory access fault, resulting in instruction, load or store access fault synchronous exception in the core.

## 10.5. Control Interface

Control interface signals of the SCR1 core are shown in [Table 86](#).

Table 86: Control interface signals

Name	Direction	Description
pwrup_rst_n	input	Power-Up Reset
rst_n	input	Reset (regular reset for entire cluster)
cpu_rst_n	input	CPU Reset (does not affect AXI bridges and TCM)
test_mode	input	DFT Test Mode
test_rst_n	input	DFT Test Reset
clk	input	System clock
ndm_rst_n_out	output	Non-Debug Module Reset Output (from DM for peripherals)
rtc_clk	input	Real-time clock
fuse_mhartid[31:0]	input	CPU Hardware Thread ID (HART ID)
fuse_idcode[31:0]	input	CPU TAPC IDCODE

## 10.6. JTAG Interface

Standard JTAG interface is provided by SCR1 core to access TAP registers and DBGC module registers. JTAG interface signals do comply with IEEE 1149.1 [3]. JTAG interface signals are shown in [Table 87](#).

Table 87: JTAG Interface Signals

Name	Direction	Description
trst_n	input	Test reset (active low)
tck	input	Test clock
tms	input	Test mode select
tdi	input	Test data input
tdo	output	Test data output
tdo_en	output	Test data output enable

## 10.7. IRQ Interface

IRQ interface signals are shown in [Table 88](#).

Table 88: IRQ Interface Signals

Name	Direction	Description
soft_irq *	input	Software interrupt
ext_irq *	input	External interrupt (only with IPIC disabled)
irq_lines[15:0]	input	External IRQ lines (only with IPIC enabled)

\* Must be synchronous to the internal clock.

# 11. Clocks and Resets

## 11.1. Clock Distribution

As shown in [Figure 18](#) and [Figure 19](#), the core supports three clock domains, :

- Core clock domain (clk);
- Real-Time clock domain (rtc\_clk);
- TAP controller (TAPC) clock domain (tck).

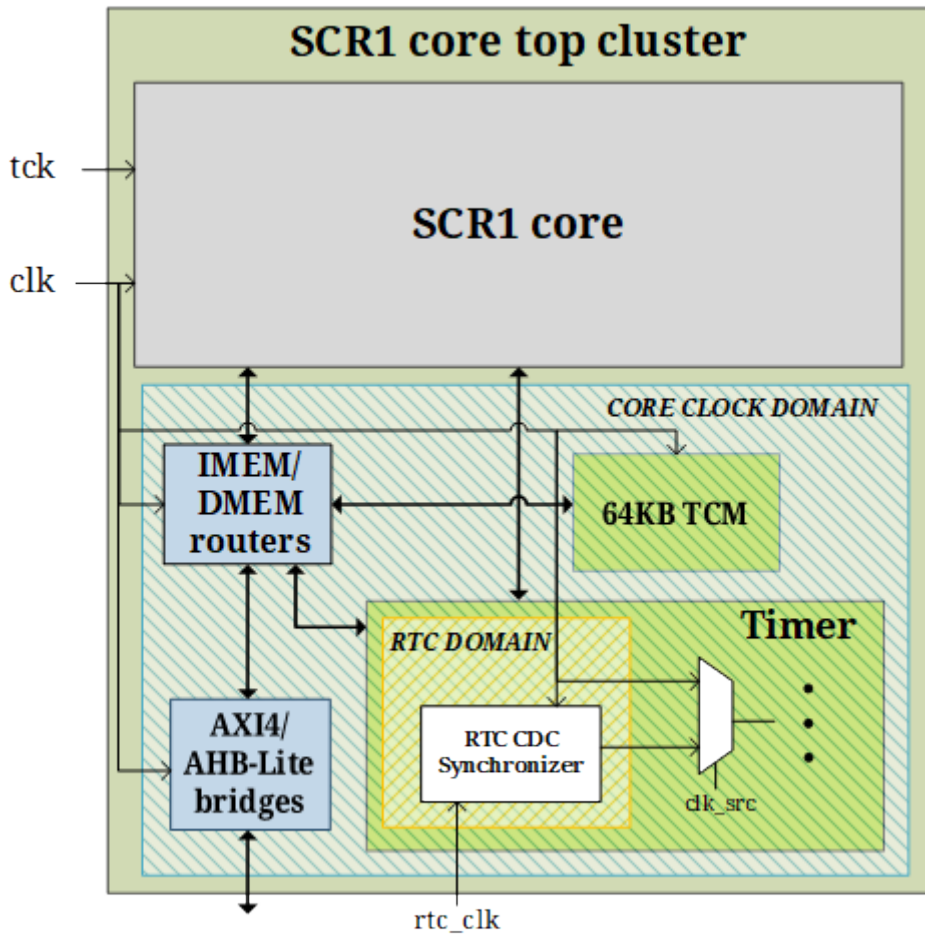


Figure 18: Clock distribution in SCR1 core top cluster

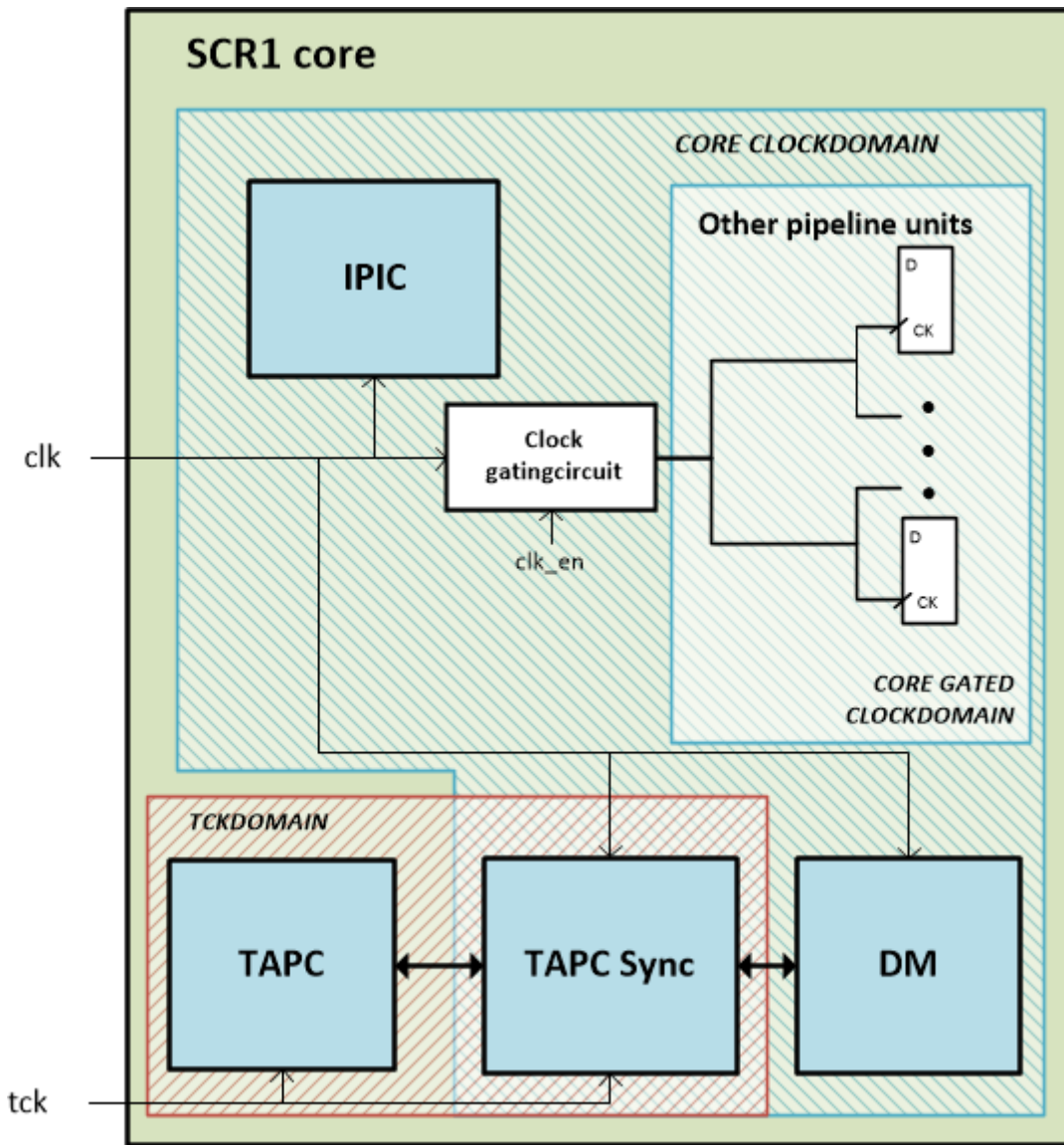


Figure 19: Clock distribution in SCR1 core

Different clock domains have clocks which may have a different frequency, a different phase (due to either differing clock latency or a different clock source), or both. Either way, the relationship between the clock edges in the various domains cannot be relied upon and may cause undesired metastability in some cases. The core assumes that clk frequency is higher than frequency of both rtc\_clk and tck.

Synchronizing a single bit signal to a clock domain with a higher frequency is accomplished by registering the signal through a flip-flop that is clocked by the source domain, thus holding the signal long enough to be detected by the higher frequency clocked destination domain.

Real-Time clock is always sampled by the core clock. Synchronized rtc\_clk can be used instead of clk as a clock source for the timer, if RTC is selected in TIMER\_CTRL CSR. For more information, see [TIMER\\_CTRL \[TIMER\\_BASE\]](#).

The TAP Controller works on tck, but the Debug Module, which is integrated into the core, and which interfaces with the TAP Controller, is timed by the core clock. The signals between TAP Controller and Debug Module are going through the synchronization logic. For its proper functioning JTAG clock frequency has to fulfill the following relation:  $\text{SysClkFreq}/\text{TckFreq} \geq 12$ .

## 11.2. Power saving features

The core has power saving features that can be used for low-power applications.

### 11.2.1. Global clock gating in wait-for-interrupt state

The clock gating circuit (controlled by the `SCR1_CLKCTRL_EN` configurable parameter) allows to optimize the energy efficiency by switching off the main system clock. If WFI instruction is executed and no enabled pending interrupts are present at the moment, the core switches into the sleep mode. In this mode, the system clock is stopped from the entire core logic except the following modules:

- Clock Gating Circuit;
- MCYCLE counter;
- IPIC, to generate external interrupt for the core;
- DM, to allow debug;
- All the top level modules, which can be controlled directly from the top level (MTIMER, TCM, AXI/AHB bridges).

The core returns to the normal operation after any enabled interrupt becomes pending.

When debug session is in progress, i.e., `DMCONTROL.dmactive = 1`, pipeline sleep mode is disabled.

By default, the clock gating is not enabled in the core configuration (`SCR1_CLKCTRL_EN` parameter is not defined). In this case, WFI instruction causes the pipeline to stop without pruning the clock tree.

### 11.2.2. Software control of performance counters

It is possible to switch off individual performance counters (TIMER, CYCLE, INSTRET) via software by modifying `TIMER_CTRL` and `MCOUNTEN` CSRs. By default, after reset, all three performance counters are switched on. `MCOUNTEN` CSR is available if `SCR1_CSR_MCOUNTEN_EN` parameter is defined (enabled in default core configuration). For more information, see [MCOUNTEN \[0x7E0\]](#), [TIMER\\_CTRL \[TIMER\\_BASE\]](#).

## 11.3. Core Reset Circuit

The core reset circuit is shown in [Figure 20](#).

The core may receive reset signal from the following sources:

- Power-Up Reset (input `pwrup_rst_n`) - the signal unconditionally resets all logic inside the core after powering up.
- Reset (input `rst_n`) - the regular hardware reset input for putting the core into a known state. It doesn't reset the TAPC and DM logic. The active clock is required for resetting the core.
- CPU reset - the regular hardware reset input for putting the CPU into a known state. It doesn't

reset the TAPC, DM logic, TCM and AXI bridges. The active clock is required for resetting the core.

- System Reset (SCU, bit CONTROL.sys\_reset) - software-generated reset signal, equivalent to the regular Reset signal.
- TAP Reset (trst\_n) - hardware reset input for the TAP Controller.
- Test Reset (test\_rst\_n) - the reset signal being used in the Test Mode (DFT requirements, activated by test\_mode input).

The reset circuit generates also ndm\_rst\_n\_out signal which is intended for resetting of hardware components outside of the core. It might be used as a HW platform reset in small systems.

The main part of the reset circuitry is placed within the System Control Unit (SCU). For its description refer to the "[System Control Unit \(SCU\)](#)" section.

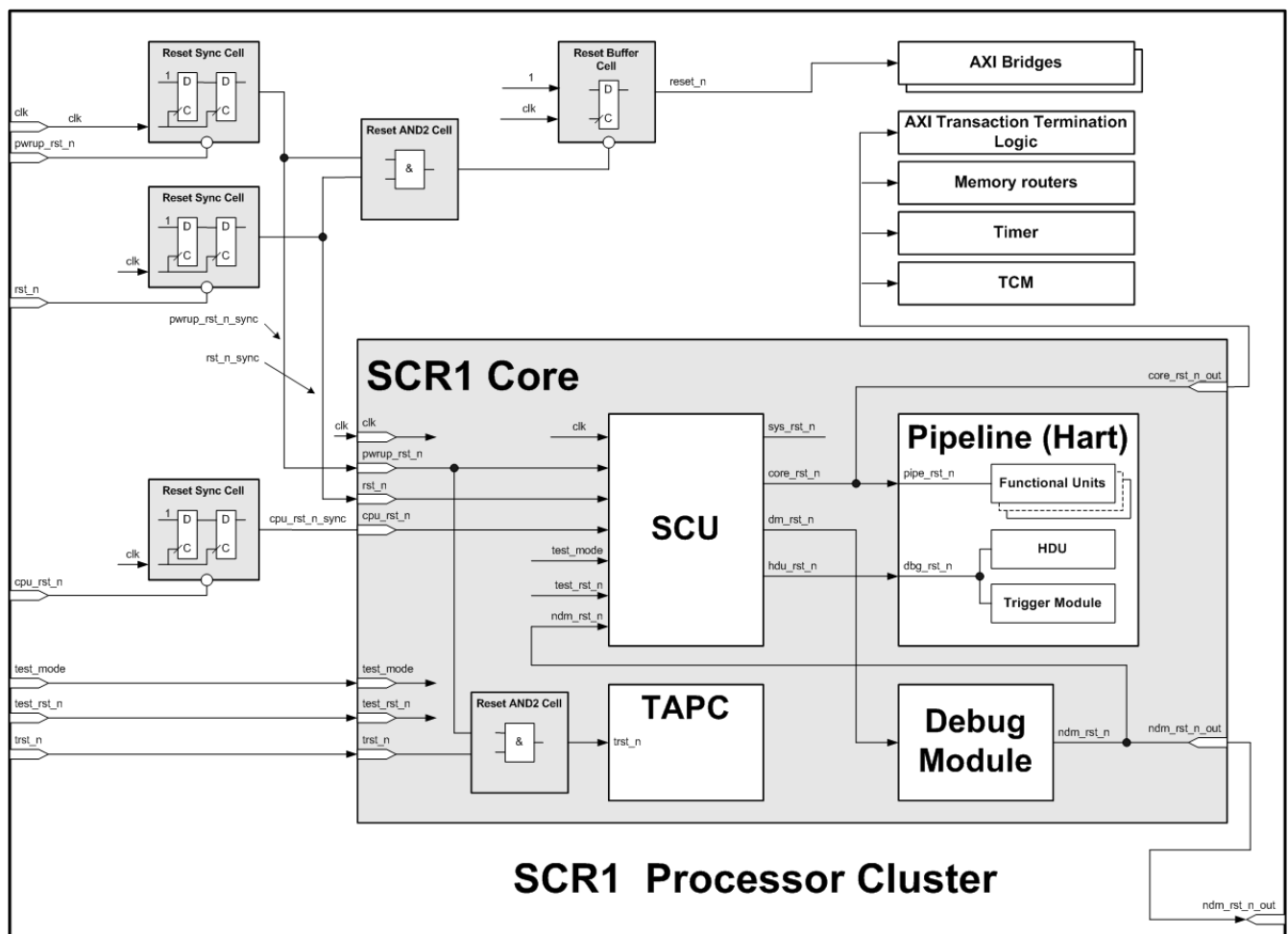


Figure 20: Core reset circuit

# 12. Initialization

## 12.1. Reset

After reset signal is de-asserted, the following happens:

- Core begins instruction fetch at address 0x200
- General-purpose registers are reset to zero
- Control and status registers are reset to their default values ([Table 89](#))

Table 89: CSR reset values

CSR name	Reset value
MSTATUS	0x1880
MIE	0
MTVEC	0x1C0
MSCRATCH	0
MEPC	0
MCAUSE	0
MTVAL	0
MIP	0
MCYCLE[H]	0
MINSTRET[H]	0
MTIME[H]	0
MTIMECMP[H]	0
TIMER_CTRL	0x1
TIMER_DIV	0
DBG_SCRATCH	0
MCOUNTEN	0x5

[Figure 21](#) shows reset de-assertion and instruction fetch start on the AHB-Lite bus.



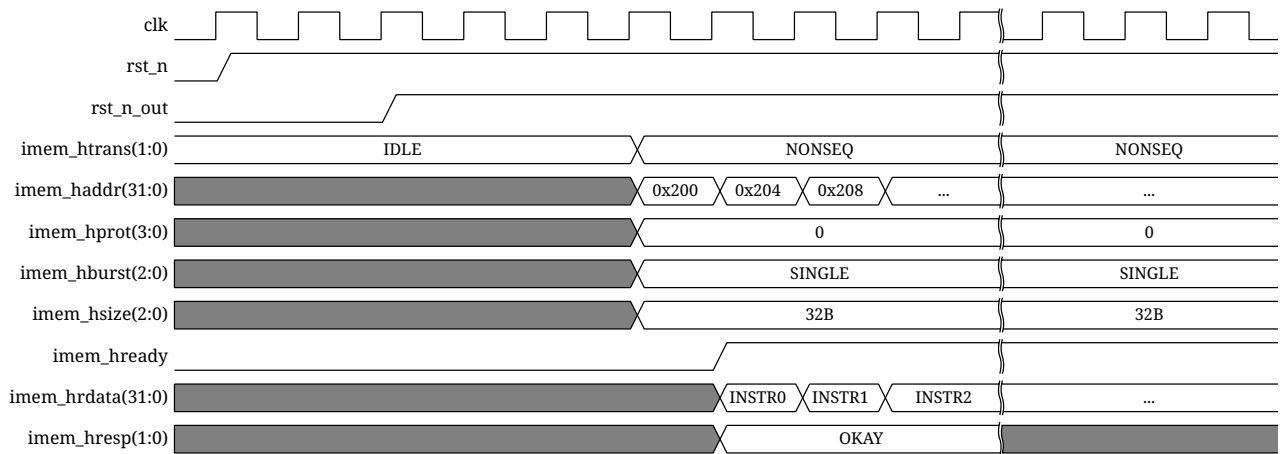


Figure 21: Reset timing diagram

## 12.2. C-runtime code example

The following is a CRT code example, which can be used to initialize the core (assuming that .text section is linked at 0x1C0).

```
#include "riscv_csr_encoding.h"

# define LREG lw
# define SREG sw
# define REGBYTES 4

.globl _start
.globl main
.globl trap_entry
.globl handle_trap
.globl sc_exit
.weak trap_entry, handle_trap, sc_exit

.text
.align 6
machine_trap_entry:
    j trap_entry

.align 6
_start:
    auipc    gp, %hi(_gp)
    addi     gp, gp, %lo(_gp)
    # clear bss
    la      a1, __BSS_START__
    la      a2, __BSS_END__
    j       4f
3:  sw      zero, 0(a1)
    add     a1, a1, 4
4:  bne     a1, a2, 3b
    la      sp, __C_STACK_TOP__
```

```

// Timer init
li      t0, mtime_ctrl
li      t1, (1 << SCR1_MTIME_CTRL_EN)
sw      t1, (t0)
li      t0, mtime_div
li      t1, (100-1)
sw      t1, (t0)
li      t0, mtimecmp
li      t1, -1
sw      t1, (t0)
sw      t1, 4(t0)

li      a0, 0
li      a1, 0
jal     main
j       sc_exit

```

```

trap_entry:
    addi sp, sp, -124

```

```

SREG x1, 1*REGBYTES(sp)
SREG x2, 2*REGBYTES(sp)
SREG x3, 3*REGBYTES(sp)
SREG x4, 4*REGBYTES(sp)
SREG x5, 5*REGBYTES(sp)
SREG x6, 6*REGBYTES(sp)
SREG x7, 7*REGBYTES(sp)
SREG x8, 8*REGBYTES(sp)
SREG x9, 9*REGBYTES(sp)
SREG x10, 10*REGBYTES(sp)
SREG x11, 11*REGBYTES(sp)
SREG x12, 12*REGBYTES(sp)
SREG x13, 13*REGBYTES(sp)
SREG x14, 14*REGBYTES(sp)
SREG x15, 15*REGBYTES(sp)
#ifdef __RVE_EXT
SREG x16, 16*REGBYTES(sp)
SREG x17, 17*REGBYTES(sp)
SREG x18, 18*REGBYTES(sp)
SREG x19, 19*REGBYTES(sp)
SREG x20, 20*REGBYTES(sp)
SREG x21, 21*REGBYTES(sp)
SREG x22, 22*REGBYTES(sp)
SREG x23, 23*REGBYTES(sp)
SREG x24, 24*REGBYTES(sp)
SREG x25, 25*REGBYTES(sp)
SREG x26, 26*REGBYTES(sp)
SREG x27, 27*REGBYTES(sp)
SREG x28, 28*REGBYTES(sp)
SREG x29, 29*REGBYTES(sp)

```

```

    SREG x30, 30*REGBYTES(sp)
    SREG x31, 31*REGBYTES(sp)
#endif // __RVE_EXT

    csrr a0, mcause
    csrr a1, mepc
    mv a2, sp
    jal handle_trap

    LREG x1, 1*REGBYTES(sp)
    LREG x2, 2*REGBYTES(sp)
    LREG x3, 3*REGBYTES(sp)
    LREG x4, 4*REGBYTES(sp)
    LREG x5, 5*REGBYTES(sp)
    LREG x6, 6*REGBYTES(sp)
    LREG x7, 7*REGBYTES(sp)
    LREG x8, 8*REGBYTES(sp)
    LREG x9, 9*REGBYTES(sp)
    LREG x10, 10*REGBYTES(sp)
    LREG x11, 11*REGBYTES(sp)
    LREG x12, 12*REGBYTES(sp)
    LREG x13, 13*REGBYTES(sp)
    LREG x14, 14*REGBYTES(sp)
    LREG x15, 15*REGBYTES(sp)
#ifdef __RVE_EXT
    LREG x16, 16*REGBYTES(sp)
    LREG x17, 17*REGBYTES(sp)
    LREG x18, 18*REGBYTES(sp)
    LREG x19, 19*REGBYTES(sp)
    LREG x20, 20*REGBYTES(sp)
    LREG x21, 21*REGBYTES(sp)
    LREG x22, 22*REGBYTES(sp)
    LREG x23, 23*REGBYTES(sp)
    LREG x24, 24*REGBYTES(sp)
    LREG x25, 25*REGBYTES(sp)
    LREG x26, 26*REGBYTES(sp)
    LREG x27, 27*REGBYTES(sp)
    LREG x28, 28*REGBYTES(sp)
    LREG x29, 29*REGBYTES(sp)
    LREG x30, 30*REGBYTES(sp)
    LREG x31, 31*REGBYTES(sp)
#endif // __RVE_EXT

    addi sp, sp, 124
    mret

handle_trap:
sc_exit:
1:  wfi
    j 1b

```

# 13. Instruction set summary

Table 90 and Table 91 present the summary for RV32I instruction set.

Table 90: RV32I instruction set summary

31.....25	24.....20	19.....15	14....12	11.....7	6.....0	Name
imm[31:12]				rd	0110111	LUI
imm[31:12]				rd	0010111	AUIPC
imm[20   10:1   11   19:12]				rd	1101111	JAL
imm[11:0]		rs1	000	rd	1100111	JALR
imm[12   10:5]	rs2	rs1	000	imm[4:1   11]	1100011	BEQ
imm[12   10:5]	rs2	rs1	001	imm[4:1   11]	1100011	BNE
imm[12   10:5]	rs2	rs1	100	imm[4:1   11]	1100011	BLT
imm[12   10:5]	rs2	rs1	101	imm[4:1   11]	1100011	BGE
imm[12   10:5]	rs2	rs1	110	imm[4:1   11]	1100011	BLTU
imm[12   10:5]	rs2	rs1	111	imm[4:1   11]	1100011	BGEU
imm[11:0]		rs1	000	rd	0000011	LB
imm[11:0]		rs1	001	rd	0000011	LH
imm[11:0]		rs1	010	rd	0000011	LW
imm[11:0]		rs1	100	rd	0000011	LBU
imm[11:0]		rs1	101	rd	0000011	LHU
imm[11:5]	rs2	rs1	000	imm[4:0]	0100011	SB
imm[11:5]	rs2	rs1	001	imm[4:0]	0100011	SH
imm[11:5]	rs2	rs1	010	imm[4:0]	0100011	SW
imm[11:0]		rs1	000	rd	0010011	ADDI
imm[11:0]		rs1	010	rd	0010011	SLTI
imm[11:0]		rs1	011	rd	0010011	SLTIU
imm[11:0]		rs1	100	rd	0010011	XORI
imm[11:0]		rs1	110	rd	0010011	ORI
imm[11:0]		rs1	111	rd	0010011	ANDI

Table 91: RV32I instruction set summary (continued)

31.....25		24.....20	19.....15	14....12	11.....7	6.....0	Name
0000000		shamt	rs1	001	rd	0010011	SLLI
0000000		shamt	rs1	101	rd	0010011	SRLI
0100000		shamt	rs1	101	rd	0010011	SRAI
0000000		shamt	rs1	000	rd	0110011	ADD
0100000		shamt	rs1	000	rd	0110011	SUB
0000000		shamt	rs1	001	rd	0110011	SLL
0000000		shamt	rs1	010	rd	0110011	SLT
0000000		shamt	rs1	011	rd	0110011	SLTU
0000000		shamt	rs1	100	rd	0110011	XOR
0000000		shamt	rs1	101	rd	0110011	SRL
0100000		shamt	rs1	101	rd	0110011	SRA
0000000		shamt	rs1	110	rd	0110011	OR
0000000		shamt	rs1	111	rd	0110011	AND
0000	pred	succ	00000	000	00000	0001111	FENCE
0000	0000	0000	00000	001	00000	0001111	FENCE.I
000000000000			00000	000	00000	1110011	ECALL
000000000001			00000	000	00000	1110011	EBREAK
001100000010			00000	000	00000	1110011	MRET
000100000101			00000	000	00000	1110011	WFI
csr			rs1	001	rd	1110011	CSRRW
csr			rs1	010	rd	1110011	CSRRS
csr			rs1	011	rd	1110011	CSRRC
csr			zimm	101	rd	1110011	CSRRWI
csr			zimm	110	rd	1110011	CSRRSI
csr			zimm	111	rd	1110011	CSRRCI

Table 92 presents the summary for RV32M instruction set.

Table 92: RV32M instruction set summary

31.....25	24.....20	19.....15	14....12	11.....7	6.....0	Name
0000001	rs2	rs1	000	rd	0110011	MUL
0000001	rs2	rs1	001	rd	0110011	MULH
0000001	rs2	rs1	010	rd	0110011	MULHSU
0000001	rs2	rs1	011	rd	0110011	MULHU
0000001	rs2	rs1	100	rd	0110011	DIV
0000001	rs2	rs1	101	rd	0110011	DIVU
0000001	rs2	rs1	110	rd	0110011	REM
0000001	rs2	rs1	111	rd	0110011	REMU

Table 93 and Table 94 present the summary for RVC instruction set.

Table 93: RVC instruction set summary

15..13	12	11	10	9	8	7	6	5	4	3	2	1.0	Name	
Quadrant 0														
000	0								0		00	Illegal instruction		
000	nzimm[5:4   9:6   2   3]								rd'		00	C.ADDI4SPN (RES,nzimm=0)		
010	imm[5:3]			rs1'			imm[2   6]		rd'		00	C.LW		
110	imm[5:3]			rs1'			imm[2   6]		rs2'		00	C.SW		
Quadrant 1														
000	0	0			0				01		C.NOP			
000	nzimm[5]	rs1/rd≠0			nzimm[4:0]				01		C.ADDI (HINT,nzimm=0)			
001	offset[11   4   9:8   10   6   7   3:1   5]										01		C.JAL (RV32)	
010	imm[5]	rs1/rd≠0			imm[4:0]				01		C.LI (HINT,rd=0)			
011	nzimm[9]	2			nzimm[4   6   8:7   5]				01		C.ADDI16SP (RES,nzimm=0)			
011	nzimm[17]	rs1/rd≠{0, 2}			nzimm[16:12]				01		C.LUI (RES,nzimm=0; HINT,rd=0)			
100	nzimm[5]	00	rs1'/rd'			nzimm[4:0]				01		C.SRLI (RV32 NSE,nzimm[5]=1)		
100	nzimm[5]	01	rs1'/rd'			nzimm[4:0]				01		C.SRAI (RV32 NSE,nzimm[5]=1)		
100	imm[5]	10	rs1'/rd'			imm[4:0]				01		C.ANDI		
100	0	11	rs1'/rd'			00		rs2'		01		C.SUB		
100	0	11	rs1'/rd'			01		rs2'		01		C.XOR		
100	0	11	rs1'/rd'			10		rs2'		01		C.OR		
100	0	11	rs1'/rd'			11		rs2'		01		C.AND		
101	offset[11   4   9:8   10   6   7   3:1   5]										01		C.J	
110	offset[8   4:3]			rs1'			offset[7:6   2:1   5]				01		C.BEQZ	
111	offset[8   4:3]			rs1'			offset[7:6   2:1   5]				01		C.BNEZ	

Table 94: RVC instruction set summary (continued)

15..13	12	11	10	9	8	7	6	5	4	3	2	1.0	Name	
Quadrant 2														
000	nzimm[5]	rd#0			nzimm[4:0]				10	C.SLLI (HINT,rd=0; RV32 NSE,nzimm[5]=1)				
010	imm[5]	rd#0			imm[4:2   7:6]				10	C.LWSP (RES,rd=0)				
100	0	rs1#0			0				10	C.JR (RES,rs1=0)				
100	0	rd#0			rs2#0				10	C.MV (HINT,rd=0)				
100	1	0			0				10	C.EBREAK				
100	1	rs1#0			0				10	C.JALR				
100	1	rd#0			rs2#0				10	C.ADD (HINT,rd=0)				
110	imm[5:2   7:6]					rs2				10	C.SWSP			



# Referenced documents

- [[[1]]] The RISC-V Instruction Set Manual Volume I: User-Level ISA Version 2.2  
<https://riscv.org/specifications/>
- [[[2]]] The RISC-V Instruction Set Manual Volume II: Privileged Architecture Version 1.10  
<https://riscv.org/specifications/privileged-isa/>
- [[[3]]] IEEE Std-1149.1 Standard Specification for boundary-scan  
<http://standards.ieee.org/findstds/standard/1149.1-2001.html>
- [[[4]]] AMBA AXI and ACE Protocol Specification (Issue E)  
<http://infocenter.arm.com/help/index.jsp?topic=/com.arm.doc.ih0022e/index.html>
- [[[5]]] RISC-V External Debug Support, Version 0.13-DRAFT  
<https://github.com/riscv/riscv-debug-spec>