Open SoC Debug Documentation Library

Release 0.1

The Open SoC Debug Contributors

Mar 28, 2020

Documentation Parts

The Open SoC Debug Project provides a full-stack debug solution. The documentation provided caters to different audiences, from people interested in a high-level overview, to users of Open SoC Debug, to developers implementing OSD in their projects, or even contributing to OSD itself.

The *Overview Documents* provide a general big-picture introduction to the ideas and concepts of OSD. They are written for a wide technical and non-technical audience.

The *Open SoC Debug Specification* describes the architecture and components of OSD. It is written for developers implementing OSD in their own designs, or extending OSD with custom components.

In addition to the specification, the Open SoC Debug Project also produces an extensible reference implementation of the OSD Specification. This reference implementation, consisting both of hardware IP components and software tools, is also documented here.

The *User Guides* describe how to use the tools provided by the OSD reference implementation. They are written for developers using OSD to debug software.

The *Implementer Guides* aim at people who want to integrate OSD into their own SoC designs, or want to develop software for an OSD-enabled SoC.

The *Identifier Registry* lists all vendor identifiers used by OSD devices and products.

CHAPTER 1

Overview Documentation

Understanding the general concepts behind Open SoC Debug is key to effectively use and implement it. In this part of the documentation, we give an high-level overview of OSD.

1.1 Open SoC Debug Primer

1.1.1 About Open SoC Debug

Systems-on-Chip (SoCs) have become embedded deeply into our lives. Most of the time we enjoy the way they serve their purpose without getting in the way. Until they don't. In those moments, we as engineers are reminded of the complex interplay between software and hardware in SoCs. We might pose questions like "How does my software execute on the chip?" or "Why is it showing this exact behavior?" To answer these questions we need insight into the system that executes the software. We gain this insight through the debug infrastructure integrated into the SoC.

Even though debug infrastructure is an essential part of any SoC design, most people consider creating it more of a necessary chore than an exciting endeavor. Therefore, most vendors today include debug infrastructure that follows one of two major specifications: ARM CoreSight[1] and NEXUS 5001[2] (officially called "IEEE-ISTO 5001"). Unfortunately, none of these specifications are fully open, they cannot be used without any money involved.

The Open SoC Debug (OSD) specification was created to close this gap. Three key messages guide its design.

- **OSD is a truly open (source) specification.** Without any committee membership required or royalty fees to be payed, anyone can freely
  - share and modify the specification itself, and
  - create and distribute implementations of the specification for any purpose.
- **OSD is for debugging and tracing.** A debugging infrastructure by itself is not a solution, but a toolbox providing the right tool for the task. Some bugs are best hunted using run-control debugging, some are better found using tracing. OSD supports both, enabling hardware and software developers to pick what's best for their needs.
- **OSD provides the common and enables the special.** SoCs came to live because they allow reuse of components and specialization at the same time, letting hardware designers focus on the unique challenges

[1] http://www.arm.com/products/system-ip/debug-trace/
[2] http://nexus5001.org/

without re-inventing the wheel. OSD follows this lead. Common IP blocks, interfaces and software tools can be re-used, and multiple extension vectors allow for easy customization where necessary.

Scope

By implementing Open SoC Debug, a SoC gains the following features (to a varying and implementation-defined degree).

- Support for run-control debugging, i.e. setting breakpoints and watchpoints and reading register values. In short, all you need to attach a debugger like GDB to the SoC.
- Support for tracing, i.e. non-intrusively observing the program execution on the SoC.
- Support for remotely controlling the SoC during development, e.g. starting the CPUs, resetting the system, and reading and writing the memories.

To provide these features, this specification defines

- an extensible debug system architecture, covering both hardware and host software,
- templates with well-defined interfaces for debug and trace IP blocks ("debug modules"),
- a set of common debug modules for the most frequent run-control debug and tracing tasks,
- a host-side software programming interface (API) for debug tools to interact with an OSD-enabled debug system.

In addition, implementations of many components described in the OSD specification are made available under a permissive open source license which can be used directly in custom designs.

Current Status

OSD is an evolving effort. Currently, we target the first release of the base specification and module specification, that contain the following parts:

- Basic interfaces and transport protocols
- A generic and mandatory memory map for all debug modules to allow enumeration, capabilities and versioning
- Basic modules for run-control and trace debugging

This is just the start that covers the very basic functionality, but more features are planned to be added to the specification in the near future: tracing to memory instead of host, device traces, module triggering, cross-triggers, on-chip aggregation and filtering, sophisticated interconnects, just to mention a few.

1.1.2 About This Document

This document gives an overview of Open SoC Debug. The goal is to provide interested designers SoC hardware components as well as developers of debugging software tools a good understanding of the overall picture and the reasoning behind the design of OSD. This document is not the specification itself. Please refer to the individual sub-documents for the exact wording of the specification.

1.1.3 High-Level Features

By implementing OSD, a SoC can easily be enhanced with advanced debug functionality. This section describes these features in more detail.

Run-Control Debug

Run-control debugging, a.k.a. breakpoint debugging, "stop-and-stare" debugging, or just "debugging," is the most common way of finding problems in software at early stages of development. Using software tools like the GNU Debugger (gdb) breakpoints can be set in the software code. If this point in the program reached, the program execution is stalled and the program control is handed over to the debugger. Using the debugger, a developer can now read register or memory values, print stack traces, and much more. To be efficient, run-control debugging functionality needs hardware support to stop the program execution at a given time. In addition, run-control debugging on SoC platforms is usually done remotely, i.e. the system is controlled from a host PC, as opposed to running the debugger directly on the SoC.

OSD contains all parts to add run-control debug support to a SoC. On the hardware side, OSD interfaces with the CPU core(s) to control its behavior. On the host side, OSD provides a daemon that GDB can connect to. The actual debugging is then handled by GDB and the usage of OSD is transparent to the software developer.

Tracing

Today's heterogeneous multi-core designs present new challenges to software developers. Concurrent software distributed across multiple CPUs and hardware accelerators, interacting with complex I/O interfaces and strict real-time requirements is the new normal. This results in new classes of bugs which are hard to find, like race conditions, deadlocks, and severely degraded performance for no obvious reason. To find such bugs, run-control debugging is not applicable: setting a breakpoint disturbs the temporal relationship between the different threads of execution. This disturbance to the program execution is called "probe effect" and can cause the original problem to disappear when searching for it, a phenomenon known as "Heisenbug."

Tracing avoids these problems by unobtrusively monitoring the program execution and transferring the observations off-chip. There, the program flow can be reconstructed and the program behavior analyzed.

OSD comes with components to enable tracing for not only CPU cores, but also for any component in the SoC, such as memories, hardware accelerators, and interconnects.

Memory Access

Reading and writing memories is an essential tool during bring-up and debugging of a SoC. A typical use case is to write software to a program memory from the host PC, to avoid writing it for example to a SD card or flash memory and then resetting the system.

OSD ships with a module that can be attached to a memory to support reads and writes from and to memories.

System Discovery

Users of today's debug systems know the pain: setting up a debugger on a host PC to communicate with the hardware often requires obscure configuration settings, secret switches and a bit of magic sauce to make it all work.

OSD is designed to be plug-and-play. All hardware components are self-describing. When a host connects to the system, it first enumerates all available components, and reads necessary configuration bits.

Timestamping

Timestamps are monotonically increasing numbers which are attached to events generated by the debug system. (They usually do not correspond in any way to wall-clock time.) Timestamps enable correlation of events in different parts of the chip with each other. Additionally, they can be used to restore order to events which are (for some reason) out of order when they arrive.

While timestamps are useful in many cases, adding them to all events generated by the debug system can significantly increase the overhead of such events.

Currently OSD supports timestamps which are full numbers of configurable width. Some debug modules can be configured to enable or disable timestamp generation.

The timestamping method used in OSD is referred to as "source timestamps" in some debug systems. Timestamps are added to the trace data at the source, as opposed to (e.g.) adding timestamps when the data is received by a debug adapter hardware between the SoC and the host PC.

Security and Authentication

Any debug system, by nature, exposes much of the system internals to the outside world. To prevent abuse of the debug system, production devices often require a developer to authenticate towards the system before being able to use the debug system.

OSD provides the infrastructure to implement such features.

1.1.4 OSD By Example

Before we dive into the details of the OSD architecture, this section discusses two typical usage examples of OSD. The first example only shows run-control debugging, the second one presents a full tracing infrastructure.

OSD for Run-Control Debugging

Many smaller single-core designs traditionally only support run-control debugging through custom JTAG-based debug infrastructure. OSD supports this use case well. Its modular architecture makes it easy to implement only essential debug modules to support run-control debug, and to add advanced features such as trace later without major changes.

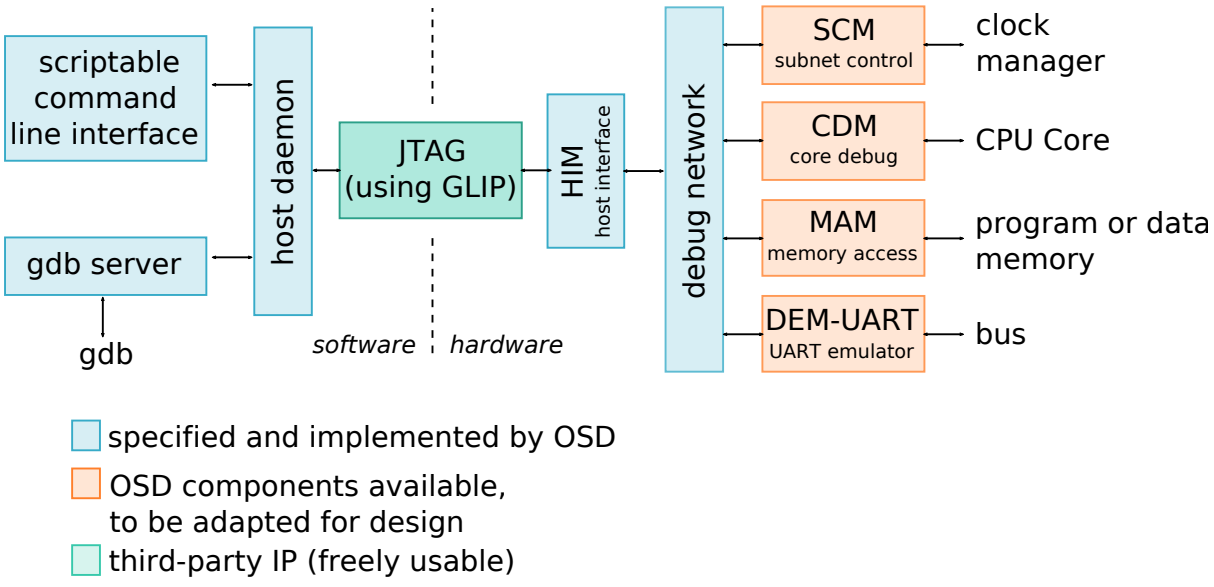

Fig. 1.1: An example system using OSD for run-control debugging

Figure 1.1 shows an example configuration of OSD for a small run-control debug scenario. The functional system (to be attached on the right side) consists of a single-core CPU, a memory and a bus interconnect. To this functional system the debug modules are attached.

- The Subnet Control Module (SCM) module allows to control the system remotely: reset the system, halt the system, reset the CPUs, etc.
- The Core Debug Module (CDM) provides all functionality expected from a run-control debug system: setting breakpoints and reading CPU registers.

- The Memory Access Module (MAM) gives access to the chip's memories for loading the memories during debugging (e.g. with the program code), to verify the memory contents, or to read out memory contents during debugging.
- To show the benefits of using OSD, the example system adds another module, the Device Emulation Module UART (DEM-UART). This module behaves on the functional hardware side, and on the software side like a usual UART device. But instead of using dedicated pins, the data is transported through the debug connection.

For all mentioned components, OSD includes a full specifications which enables a custom implementation, as well as a hardware implementation that can be used unmodified or adapted to fit the interface to the custom functional system.

The debug modules are all connected to a debug network. The OSD specification does not require a specific network topology or implementation type. However, usually OSD implementations use a 16-bit wide, unidirectional ring network on chip (NoC), as it presents a good trade-off between area usage and performance.

To connect with a host PC, three further components are needed: the Host Interface Module (HIM) on the hardware side, a GLIP transport module, and a software daemon on the host side.

The transport of data between host and device is handled by GLIP[3]. GLIP is a library which abstracts the data transport between hardware and software with a bi-directional FIFO interface. The data transport itself can happen through different physical interfaces, such as UART, JTAG, USB or PCI Express (PCIe). In the presented example, a JTAG connection is used. A possibly existing JTAG boundary scan interface can be re-used and a new Test Access Point (TAP) is added to the JTAG chain for the debug connection.

The Host Interface Module (HIM) connects the debug network to the FIFO-interface of GLIP.

On the software side, the OSD host daemon encapsulates the communication to the device and provides a API for various tools communicating with the debug system. A scriptable command line interface can be used to control the system (such as reset, halt, etc.) and to read and write memories. A gdb server provides an interface to the core debug functionality that the GNU Debugger (gdb) can connect to. In the end, software can be debugged with an unmodified gdb (and other gdb-enabled IDEs, such as Eclipse CDT).

OSD for Tracing

Today's debug system architectures strictly separate between run-control debugging and tracing. The example below shows how OSD units the two worlds with a common interface, thus reducing development and maintainance effort. Since most of the architecture is shared between run-control debugging and tracing, upgrading an existing design from run-control debugging to tracing is not a large step.

Figure 1.2 shows an example architecture of a OSD system with tracing support for a dual-core design. Most of the architecture is identical to the previous example: the host daemon, the HIM, the debug network and the SCM, CDM and MAM debug modules. New in this example are the following parts.

- The GLIP transport library now uses USB 2.0 instead of JTAG for communication. This allows for higher off-chip transfer speeds to get improved visibility into the system by tracing.
- The Core Trace Module (CTM) provides program trace (a.k.a. instruction trace) support. It is attached to the CPU core next to the CDM.
- A graphical trace viewer can be attached to the host daemon to view the traces. Currently, OSD does not come with such a tool, but all interfaces are provided to easily write such a tool.

The two examples in this section have already shed a light on what is possible with OSD. In the remainder of this document, we'll discuss OSD in more depth, starting with a more general overview of the architecture.

[3] http://glip.io

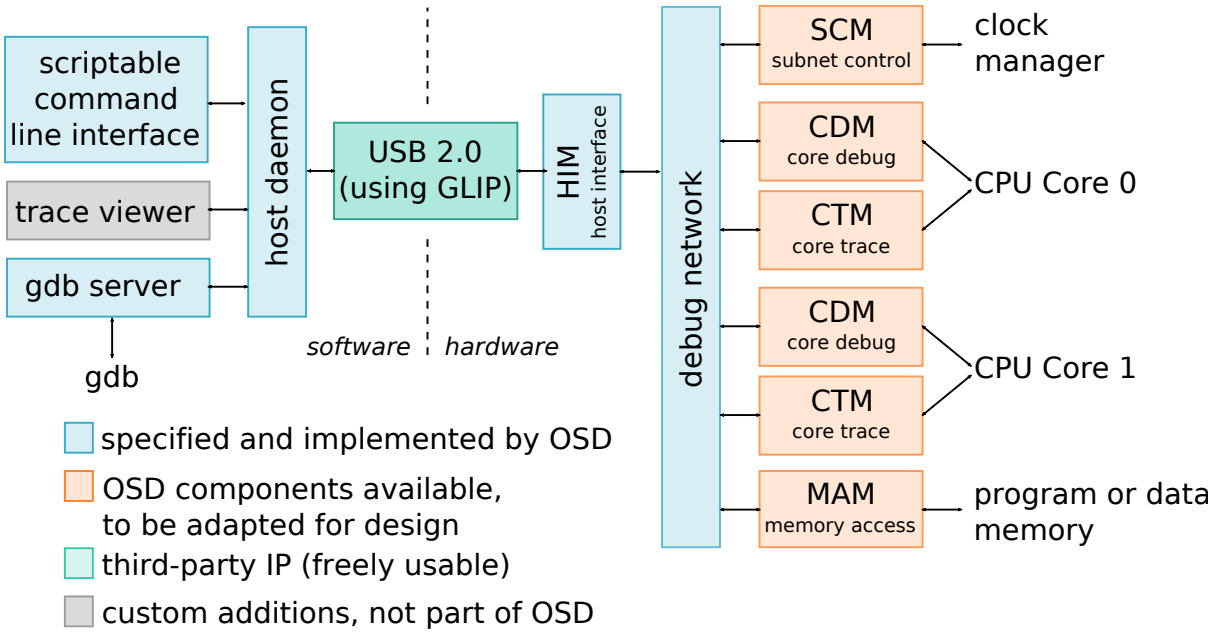

Fig. 1.2: An example system dual-core system using OSD tracing

CHAPTER 2

The Open SoC Debug Specification

2.1 Preface

2.1.1 About this specification

This specification describes Open SoC Debug (OSD), an extensible infrastructure adding debug and trace support to a System-on-Chip.

Target audience

This specification targets all people involved in the design and implementation of software or hardware products using Open SoC Debug. Explicitly, this specification targets

- Hardware designers integrating Open SoC Debug in their System-on-Chip designs.
- Hardware designers extending Open SoC Debug, for example by writing own OSD modules.
- Software developers writing software tools interacting with an OSD-enabled SoC.

2.1.2 Conventions

The key words “MUST”, “MUST NOT”, “REQUIRED”, “SHALL”, “SHALL NOT”, “SHOULD”, “SHOULD NOT”, “RECOMMENDED”, “NOT RECOMMENDED”, “MAY”, and “OPTIONAL” in this document are to be interpreted as described in **RFC 2119**[4].

Unless noted otherwise, numbers are written in decimal (base 10) representation. Hexadecimal numbers (base 16) are prefixed with “0x”, binary numbers (base 2) are prefixed with “0b”.

Bit fields given in the form MSB:LSB, as commonly used in Verilog and VHDL. Both MSB and LSB are included in the range.

Ranges are given as UPPER BOUND .. LOWER BOUND. Both bounds are included in the range.

[4] https://tools.ietf.org/html/rfc2119.html

2.1.3 Terms

- **Target**. The component which is being observed through Open SoC Debug. Usually, this is a physical chip (e.g. an ASIC or an FPGA).
- **Host**. The component which observes the target. Typically, this is a regular PC, but it the same functionality could also be performed by a special-purpose hardware unit.
- **Conforming implementation**. A implementation of the Open SoC Debug specification which includes all required functionality as defined in this specification.

2.2 Introduction

To develop, debug and improve software running on Systems-on-Chip (SoC), developers need access to the SoC's internal structures. Developers need to observe the program execution to understand how the software is executed, and they need control over the software execution from a remote system. The Open SoC Debug (OSD) is an extensible, modular specification which enhances SoCs with such debug and tracing functionality.

The functionality of Open SoC Debug can be placed in three groups:

- Run-Control debug functionality. The software execution on the SoC is temporarily suspended and control over the execution flow is handed over to a software tool, the "debugger."
- Tracing functionality. The software execution is observed and the resulting observations are transferred out of the chip, but the execution is not halted or otherwise modified.
- Internal system access. During the process of software development, fast and hassle-free access to the SoC is required. For example, the program code can be written into the RAM directly, instead of loading it from persistent storage, such as flash memory.

Open SoC Debug is a modular and extensible specification, acknowledging the fact that every SoC and every target market has its different design goals and thus requires different trade-offs.

This document, the Open SoC Debug specification, contains both required and optional parts. The required parts MUST be implemented by any conforming implementation of Open SoC Debug, the optional parts MAY be implemented.

In addition to this specification, the Open SoC Debug Contributors have also developed a reference implementation of many components described in this specification. It is provided as a starting point for own developments. However, independent implementations following this specification are encouraged.

2.3 Open SoC Debug Architecture

2.3.1 Architecture Overview

In Open SoC Debug, software and hardware components form together an extensible architecture.

Fig. 2.1: High-level overview of the Open SoC Debug architecture.

Figure 2.1 shows the different components in a typical Open SoC Debug-based debug system.

- **Debug modules** (shown on the right) monitor or interact with the functional components of the SoC. Towards the functional SoC the interface is implementation-specific. Towards the debug interconnect the modules conform to a standardized interface, which is specified in this document. Also included in this specification is a description of common debug modules, some of which are must be implemented by any conforming implementation, others which are optional. Debug modules are self-describing and discoverable from the target at runtime through a standardized programmer interface, which is also described in this specification. All debug modules are given an address, which is used in all communication with the module.
- The **debug interconnect** is used to exchange data between debug modules, and between the target and the host. The format of the data transmitted over the debug interconnect (the Debug Packets, DP), as well as the interface between debug modules and the debug interconnect (the Debug Interconnect Interface, DII) are specified in this document.
- The **physical transport** connects the target to the host. The OSD specification does not cover the physical transport. However, an example implementation is given as part of the reference implementation.
- **Host software** implements the low-level interface to connect and interact with an OSD-enabled system. While the OSD project has created a reference implementation for the host software, its usage is not mandatory and it is not part of this specification.
- Finally, **debug tools** use the debug system to perform debugging and analysis tasks, ranging from run-control debugging to tracing and runtime verification. Debug tools can be implemented on top of Open SoC Debug, but are not part of this specification.

The OSD specification is designed with extensibility in mind. Well-defined extension vectors can be used to customize the behavior of OSD and adapt OSD to different target systems. If little or no customization is required, the reference implementation can serve as a good starting point to reduce engineering cost.

2.4 Data Exchange Formats

2.4.1 Introduction

Interaction between components in OSD requires that a common "language" is spoken. The data exchange formats described in this chapter are this "language," they are employed between OSD modules, as well as between the host and the OSD-enabled target.

2.4.2 General Considerations

In Open SoC Debug, the native **word width is 16 bit**. The byte ordering is **big endian**. Exceptions are mentioned explicitly in the specification.

2.4.3 The Debug Packet

The main data exchange format in OSD is the Debug Packet (DP). A Debug Packet is a routable and typed piece of information, consisting of

- a destination address
- a source address
- a type
- a payload

The payload format depends on the type of the packet. For some types the payload format is strictly specified, while other types can be used to transfer data which is not specified in this specification.

Length Limitations

Each debug packet consists of at least three 16 bit wide words. A packet MUST NOT consist of more than $2^{16} - 1 = 65535$ words, including all headers. Implementations MAY impose a lower limit, but the limit may not be less than 12 words.

The maximum packet length within a subnet can be determined by reading the `MAX_PKT_LEN` register from the SCM module.

Debug Packet Structure

A Debug Packet consists of a multiple words as described in the table below.

Table 2.1: Debug Packet (DP) Structure

Word	Name	Description
0	`DEST`	Packet destination address
1	`SRC`	Packet source address
2	`FLAGS`	Packet flags
3 .. *packet size*	`PAYLOAD`	Payload (content) of the Debug Packet

Table 2.2: Field Reference: `FLAGS`

Bit(s)	Field	Description
15:14	`TYPE`	**Packet Type** **0b00: REG (Register Access)** Access to a register in a debug module. Register accesses are synchronous read and write operations on a debug module. **0b01: RESERVED for future use** Implementations MUST discard packets of this type. **0b10: EVENT (Debug Event)** Unsolicited debug event generated by any of the debug modules. The payload of Debug Packets of this type is module-specific. **0b11: RESERVED for future use** Implementations MUST discard packets of this type.
13:10	`TYPE_SUB`	**Packet Subtype** The packet subtype refines the packet type (`TYPE`). Allowed values depend on the `TYPE` field.
9:0	`RESERVED`	**Reserved** Reserved space for future extensions. Senders must set this field to zero, receivers must ignore its contents.

Register access (`TYPE == REG`)

Register accesses are Debug Packets which access a single register in a debug module. All accesses are synchronous: read requests trigger a read response, write requests are acknowledged.

All register accesses must set the `TYPE` field of a Debug Packet to `REG`. The field `TYPE_SUB` describes the type of register access, allowed values are listed in the following table.

Table 2.3: Reference of Debug Packet subtypes for register accesses

Field Name	TYPE_SUB Value	Description
REQ_READ_REG_16	0b0000	16 bit register read request
REQ_READ_REG_32	0b0001	32 bit register read request
REQ_READ_REG_64	0b0010	64 bit register read request
REQ_READ_REG_128	0b0011	128 bit register read request
RESP_READ_REG_SUCCESS_16	0b1000	16 bit register read response
RESP_READ_REG_SUCCESS_32	0b1001	32 bit register read response
RESP_READ_REG_SUCCESS_64	0b1010	64 bit register read response
RESP_READ_REG_SUCCESS_128	0b1011	128 bit register read response
RESP_READ_REG_ERROR	0b1100	register read failure
REQ_WRITE_REG_16	0b0100	16 bit register write request
REQ_WRITE_REG_32	0b0101	32 bit register write request
REQ_WRITE_REG_64	0b0110	64 bit register write request
REQ_WRITE_REG_128	0b0111	128 bit register write request
RESP_WRITE_REG_SUCCESS	0b1110	the preceding write request was successful
RESP_WRITE_REG_ERROR	0b1111	the preceding write request failed

Register read request (TYPE_SUB == REQ_READ_REG_*)

Read from a single register. Reads from 16, 32, 64 and 128 bit wide registers are supported, the appropriate DP Subtype (TYPE_SUB) must be used to select the register width. The address ADDR must be 16 bit wide. ADDR addresses 16 bit and must be aligned to the register size.

A debug module MUST respond with a RESP_READ_REG_SUCCESS_* of the same size as the read in case of a successful read, or a RESP_READ_REG_ERROR Debug Packet in case of an error.

Table 2.4: Debug Packet payload for register read requests (TYPE == REG && TYPE_SUB == REQ_READ_REG_*)

Payload word	Field name	Description
0	ADDR	Register address to read from

Register read response (TYPE_SUB == RESP_READ_REG_SUCCESS_*)

The preceding register read request (TYPE_SUB == REQ_READ_REG_*) was successful, the payload is the data read from the address given in the request.

Table 2.5: Debug Packet payload for a response to a 16 bit register read request (TYPE_SUB == RESP_READ_REG_SUCCESS_16)

Payload word	Field name	Description
0	DATA[15:0]	data word read from the register

Table 2.6: Debug Packet payload for a response to a 32 bit register read request (TYPE_SUB == RESP_READ_REG_SUCCESS_32)

Payload word	Field name	Description
0	DATA[31:16]	bits 31 to 16 of the data read from the register (most significant word)
1	DATA[15:0]	bits 15 to 0 of the data read from the register (least significant word)

Table 2.7: Debug Packet payload for a response to a 64 bit register read request (TYPE_SUB == RESP_READ_REG_SUCCESS_64)

Payload word	Field name	Description
0	DATA[63:48]	bits 63 to 48 of the data read from the register (most significant word)
1	DATA[47:32]	bits 47 to 32 of the data read from the register
2	DATA[31:16]	bits 31 to 16 of the data read from the register
3	DATA[15:0]	bits 15 to 0 of the data read from the register (least significant word)

Table 2.8: Debug Packet payload for a response to a 128 bit register read request (TYPE_SUB == RESP_READ_REG_SUCCESS_128)

Payload word	Field name	Description
0	DATA[127:112]	bits 127 to 112 of the data read from the register (most significant word)
1	DATA[111:96]	bits 111 to 96 of the data read from the register
2	DATA[95:80]	bits 95 to 80 of the data read from the register
3	DATA[79:64]	bits 79 to 64 of the data read from the register
4	DATA[63:48]	bits 63 to 48 of the data read from the register
5	DATA[47:32]	bits 47 to 32 of the data read from the register
6	DATA[31:16]	bits 31 to 16 of the data read from the register
7	DATA[15:0]	bits 15 to 0 of the data read from the register (least significant word)

Register error read response (TYPE_SUB == RESP_READ_REG_ERROR)

The preceding register read request to this module failed for some reason.

Register write request (TYPE_SUB == REQ_WRITE_REG_*)

Writes to a register. Writes to 16, 32, 64 and 128 bit wide registers are supported, the appropriate DP Subtype (TYPE_SUB) must be used to select the register width. The address ADDR must be 16 bit wide. ADDR addresses 16 bit and must be aligned to the register size.

A debug module MUST respond with a RESP_WRITE_REG_SUCCESS Debug Packet in case the write was executed successfully, or a RESP_WRITE_REG_ERROR Debug Packet if the write failed.

Table 2.9: Debug Packet payload for 16 bit register write requests (TYPE == REG && TYPE_SUB == REQ_WRITE_REG_16)

Payload word	Field name	Description
0	ADDR	Register address to write to
1	DATA[15:0]	data word to be written to the register

Table 2.10: Debug Packet payload for 32 bit register write requests (TYPE == REG && TYPE_SUB == REQ_WRITE_REG_32)

Payload word	Field name	Description
0	ADDR	Register address to write to
1	DATA[31:16]	bits 31 to 16 of the data to be written to the register (most significant word)
2	DATA[15:0]	bits 15 to 0 of the data to be written to the register (least significant word)

Table 2.11: Debug Packet payload for 64 bit register write requests (`TYPE == REG && TYPE_SUB == REQ_WRITE_REG_64`)

Payload word	Field name	Description
0	`ADDR`	Register address to write to
1	`DATA[63:48]`	bits 63 to 48 of the data to be written to the register (most significant word)
2	`DATA[47:32]`	bits 47 to 32 of the data to be written to the register
3	`DATA[31:16]`	bits 31 to 16 of the data to be written to the register
4	`DATA[15:0]`	bits 15 to 0 of the data to be written to the register (least significant word)

Table 2.12: Debug Packet payload for 128 bit register write requests (`TYPE == REG && TYPE_SUB == REQ_WRITE_REG_128`)

Payload word	Field name	Description
0	`ADDR`	Register address to write to
1	`DATA[127:112]`	bits 127 to 112 of the data to be written to the register (most significant word)
2	`DATA[111:96]`	bits 111 to 96 of the data to be written to the register
3	`DATA[95:80]`	bits 95 to 80 of the data to be written to the register
4	`DATA[79:64]`	bits 79 to 64 of the data to be written to the register
5	`DATA[63:48]`	bits 63 to 48 of the data to be written to the register
6	`DATA[47:32]`	bits 47 to 32 of the data to be written to the register
7	`DATA[31:16]`	bits 31 to 16 of the data to be written to the register
8	`DATA[15:0]`	bits 15 to 0 of the data to be written to the register (least significant word)

Register write response: successful (`TYPE_SUB == RESP_WRITE_REG_SUCCESS`)

The preceding register write request to the module was successful (write acknowledgement).

Unless explicitly documented in the module documentation, the `RESP_WRITE_REG_SUCCESS` implies that all actions triggered by the corresponding register write have been executed fully.

Register write response: error (`TYPE_SUB == RESP_WRITE_REG_ERROR`)

The preceding register write request to the module was not successful.

Event Debug Packets (`EVENT`)

Debug Events are Debug Packets sent by a debug module without being explicitly triggered by the host or by another module. The main purpose of Debug Events is to transport trace data, but they can also be used for other purposes. Hence this specification does not attempt to specify the payload of event packets strictly.

Table 2.13: Reference of Debug Packet subtypes for event packets

Field Name	`TYPE_SUB` Value	Description
`EV_LAST`	0b0000	Standalone event packet, or in split event transmissions, the last packet in the transmission
`EV_CONT`	0b0001	A non-last event packet in a split event transmission
`EV_OVERFLOW`	0b0005	An overflow happened

Split Event Transmissions

All event packets must set the TYPE field of a Debug Packet to EVENT. Event packets must be not larger then the maximum packet length, which can be obtained from the SCM module. If payload should be transmitted which is larger than the maximum DI packet size the payload can be "split" into two or more packets. In this case, all but the last event packets set TYPE_SUB to 1,

For the TYPE_SUB The field TYPE_SUB and the packet payload are defined by the debug module itself.

Overflows

If the debug system is overloaded, events may be dropped by the producer. If this happens, the producer counts the dropped packet and sents an overflow packet once it is able to transmit again. The overflow packet contains the number of dropped packets as only data word. Overflow packets have TYPE_SUB set to OVERFLOW.

2.5 Programmer Interface

2.5.1 Debug Module Base Register Map

To enable functionality like the discovery of debug modules, all debug modules MUST implement the "Open SoC Debug Status & Control" registers and MUST follow the base address map.

All registers are accessed through Debug Packets of TYPE == REG.

Debug module base address map

Address Range	Description
0x0000 - 0x01ff	Open SoC Debug Base Registers
0x0200 - 0xffff	Module-specific registers

All debug modules MUST implement the Open SoC Debug base registers.

Debug modules MAY implement any additional registers in the module-specific register space (register addresses between 0x0200 and 0xffff).

Open SoC Debug Base Registers

All base registers are 16 bit wide.

Table 2.14: Open SoC Debug base register map

address	name	description
0x0000	MOD_VENDOR	module vendor
0x0001	MOD_TYPE	module type identifier
0x0002	MOD_VERSION	module version
0x0003	MOD_CS	module control and status
0x0004	MOD_EVENT_DEST	destination of debug events

Module Vendor Identifier (MOD_VENDOR)

- Address: 0x0000
- Reset Value: implementation defined

- Access: read-only

The module vendor identifier is a 16 bit value. Vendor identifiers MUST be assigned by the Open SoC Debug project before they can be used. The Open SoC Debug project SHALL provide a publicly accessible list of all known vendors.

Note: A list of assigned vendor IDs is available online at *Vendor Identifier Registry*.

Module type identifier (MOD_TYPE)

- Address: 0x0001
- Reset Value: implementation defined
- Access: read-only

The module type identifier describes the module type. It is assigned by the module vendor. The combination of `MOD_VENDOR` and `MOD_TYPE` must be descriptive for a given debug module across all conforming implementations.

Module Version (MOD_VERSION)

- Address: 0x0002
- Reset Value: implementation defined
- Access: read-only

The versions are plain numbers that identify the module version. The module version can be used by the host software to adapt the communication protocol to the API specific to a module version. A module's API MUST NOT change in incompatible ways as long as the same module version is used.

Control and Status (MOD_CS)

- Address: 0x0003
- Reset Value: *see the table below*
- Access: *see the table below*

Module control and status register.

Table 2.15: Field Reference: MOD_CS

Bit(s)	Field	Access	Reset Value	Description
15:1	RESERVED	r/w	0x0	**Reserved for future use** This field is reserved for future use. Implementations MUST ignore the contents of this field.
0	MOD_CS_ACTIVE	r/w	0b0	**Activate or stall the debug module** **0b0: Module is stalled** The module is stalled. A stalled module MAY NOT send any debug events, i.e. packets of TYPE == EVENT. **0b1: Module is active** The module is active. An active event MAY send debug events, i.e. packets of TYPE == EVENT.

Event Destination (MOD_EVENT_DEST)

- Address: 0x0004
- Reset Value: *see the table below*
- Access: *see the table below*

Table 2.16: Field Reference: MOD_EVENT_DEST

Bit(s)	Field	Access	Reset Value	Description
15:10	RESERVED	r/w	0x0	**Reserved for future use** This field is reserved for future use. Implementations MUST ignore the contents of this field.
9:0	MOD_EVENT_DEST_ADDR	r/w	0x0	**Event Packet Destination** Address of the module in the Debug Interconnect to which all event packets (TYPE == EVENT) should be sent. Changing the destination address MAY not take immediate effect, but MUST take effect soon after it has been set (e.g. after a buffer has been cleared). The exact timing behavior is implementation-defined.

2.6 Component Architecture

2.6.1 Debug Interconnect

Todo: A specification for two debug interconnects, one for control and one for tracing is still missing.

To route debug information to the correct debug module and to the host, OSD uses a simple packet-based protocol. All data exchanged on the Debug Interconnect is formatted as Debug Packet. The *Debug Interconnect Interface (DII)* defines the the hardware interface and the flow control mechanism for hardware modules interacting with the Debug Interconnect.

One key property of the transport & switching in the Open SoC Debug specification is that it generally allows that debug modules exchange packets between them. This enables on-chip trace processing, run-control debugging from a special core or other methods to reduce the traffic on the host interface, which is the most critical resource in modern debugging.

The Debug Interconnect is only loosely specified to allow implementors to choose an interconnect implementation suitable for their target system.

General Requirements

In general, the interconnect implementation is not specified in this document, as long as it fulfills two basic properties: - It provides strict ordering of packets with the same (source, destination) tuple. This property forbid debug packets from one source to one destination to overtake each other in the interconnect, which is useful to allow payload data to span multiple packets. - It is free of deadlocks.

Topology

Implementors MAY choose any interconnect topology. Figure 2.2 shows favored topologies. The baseline implementation is a simple ring interconnect. The ring balances well between clock speed, required chip area and most importantly flexibility. It can easily span the entire chip without dominating a design.

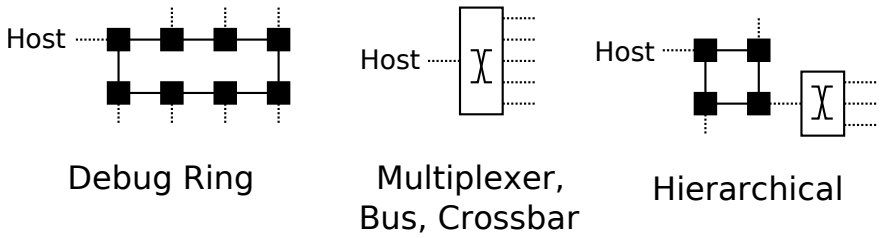

Fig. 2.2: Debug ring and other interconnects

Alternatively, other topologies may be favored. For example a low count of debug modules favors a multiplexer interconnect. Especially if the debug modules are all trace debugging or all run-control debugging a bus or similar can be favorable for low debug module counts. When the modules also communicate with each other a crossbar may be used for high throughput, but with the disadvantage of large area overhead.

Finally, we believe once some first tests with larger systems in the real world have been performed, hierarchical topologies may become favorable. Beside optimizing the resource utilization, aggregating modules may bridge subsets of trace modules to the actual debug interconnect to perform size optimizations on the aggregated packet stream.

Debug Interconnect Interface (DII)

The *Debug Interconnect Interface (DII)* is the synchronous interface between the Debug Interconnect (DI) and the debug modules. It is used to transfer Debug Packets.

The DII is a FIFO-like interface with data and handshake signals. All debug components must conform to the DII when accessing the Debug Interconnect.

Table 2.17: Debug Interconnect Interface description (master view)

signal name	driver	width (bit)	description
`data`	master	16	a word of data of the debug packet
`last`	master	1	Set to 0b1 by the master to indicate that `data` is the last word in a Debug Packet. Set to 0b0 otherwise.
`valid`	master	1	set to 0b1 by the master to indicate that `data` is valid and should be transferred
`ready`	slave	1	set to 0b1 by the slave to acknowledge the transfer

The following rules and restrictions apply:

- The `valid` signal must not depend on the `ready` signal. This means you cannot have a combinational dependency of the `valid` signal on the `ready` signal in one cycle.
- A transfer was succesfull iff `valid` and `ready` were set.
- The `last` signal indicates that `data` is the last word in a Debug Packet.

2.6.2 Debug Modules

Most functionality in Open SoC Debug is implemented as debug module. A debug module is connected to the Debug Interconnect on the one side, and usually to a component in the functional system on the other side (such as a CPU or a memory). The task of a debug module to collect data from or to interact with the functional system.

Debug modules MUST provide one Debug Interconnect Interface, and MUST implement the required parts of the Programmer API, especially the *Debug Module Base Register Map*.

2.7 System Architecture

2.7.1 Implementation Aspects

Overflow Handling

Todo: make this a bit less vague and about the future, but about what we have now.

In case the trace events are generated at a faster rate than the host interface can transfer. This problem becomes crucial with the increasing number of trace modules. Generally, this can be done on the level of the debug system by a sophisticated flow control that will be specified in later revisions. An overflow occurs if a trace event is generated, but cannot be transferred or buffered due to backpressure from the interconnect, but backpressure cannot be generated to the system module. In the current specification the trace infrastructure detects this situation, counts how many packets could not be transfered and then transfers a `missed_events` event once it the interface is available again.

Clock and Power Domains

Todo: copy clock and power domain aspects from architecture doc

2.7.2 Physical Interfaces

Note: This version of the Open SoC Debug specification does not describe any physical interface.

Todo: leave this out of the spec!? or put it next to HIM?

The physical interface is abstracted in Open SoC Debug as a FIFOs which transmit data between the host and the device.

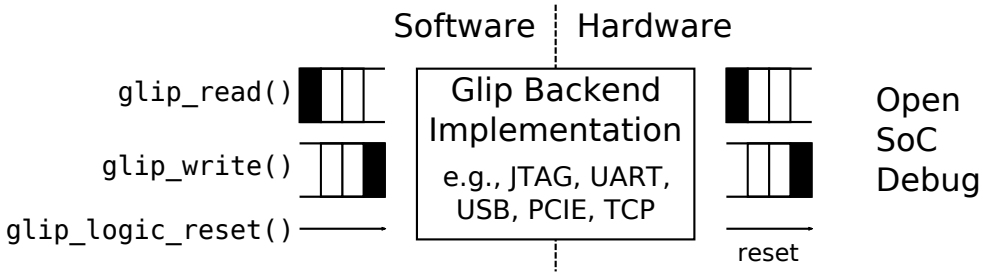

Fig. 2.3: GLIP as abstraction layer from the physical interface

While not required by OSD, we recommend building on top of GLIP[5] as depicted in @fig:glip_overview. GLIP provides a generic FIFO interface that reliably transfers data between the host and the system. Multiple alternatives for simulations and prototyping hardware exist. In a silicon device, a high-speed serial interface is most probably favorable.

2.8 Core Debug Modules

Open SoC Debug specifies a set of modules with commonly used functionality.

2.8.1 Host Interface Module (HIM)

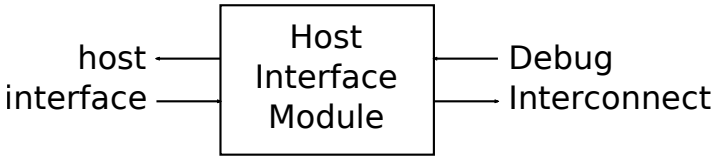

Fig. 2.4: Host Interface Module

The *Host Interface Module (HIM)* converts the debug packets to a `length-value` encoded data stream, that is transferred using the glip interconnect. This format is simple and contains the length of the debug packet in one data item followed by the debug packet.

Alternatively, the HIM can be configured to store the debug packets to the system memory using the memory interface.

Debug Transport Datagram (DTD)

[5] http://www.glip.io

Todo: Do we really want to specify this format in here, or should we leave it as implementation-defined until we find a better solution which can cope with variable-length data in a streaming fashion (i.e. without buffering the whole packet first to determine its length)?

The Debug Transport Datagram (DTD) encapsulates the Debug Packet into a 16-bit wide packet. The Debug Packet data is prepended with the size of the packet in 16 bit words.

Table 2.18: Debug Interconnect Packet (DI Packet) Structure

word index	data
0	size *n* of the Debug Packet in 16 bit words
1	word 0 of the Debug Packet
2	word 1 of the Debug Packet
...	...
n + 1	word *n* of the Debug Packet

Note: Due to the native width the Debug Transport Datagram is used when transferring a Debug Packet off-chip.

2.8.2 Subnet Control Module (SCM)

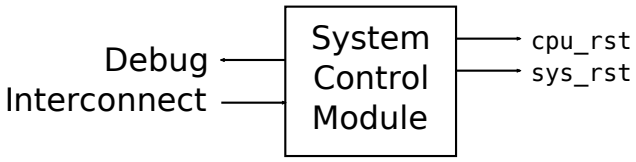

Fig. 2.5: Subnet Control Module

The *Subnet Control Module (SCM)* is always mapped to the local address 0 in a subnet of the DI. The SCM provides an description of the subnet (such as its vendor or the number of debug modules available in the subnet). In addition, the SCM can be used to control the whole subnet, like resetting and starting or stopping its CPUs.

Programmer Interface: Control Registers

The Subnet Control Module implements the *Debug Module Base Register Map*. The reset values are listed below.

Table 2.19: SCM base register reset values

address	name	description	reset value
0x0000	`MOD_VENDOR`	module vendor	0x0001
0x0001	`MOD_TYPE`	module type identifier	0x0001
0x0002	`MOD_VERSION`	module version	0x0000
0x0003	`MOD_CS`	module control and status	0x0000
0x0004	`MOD_EVENT_DEST`	destination of debug events	impl.-specific

Additionally, it implements the following control registers for module-specific functionality.

Table 2.20: SCM Register Map

address	name	description
0x0200	SYSTEM_VENDOR_ID	Vendor ID
0x0201	SYSTEM_DEVICE_ID	Device ID
0x0202	NUM_MOD	Debug module count
0x0203	MAX_PKT_LEN	Maximum packet length
0x0204	SYSRST	System Reset

System ID (SYSTEM_VENDOR_ID)

- Address: 0x0200
- Reset Value: *implementation specific*
- Access: read-only

The vendor ID identifies the entity creating/producing the device (e.g. the chip) that contains the OSD implementation. Vendor IDs are assigned by the Open SoC Debug Project. Unassigned vendor IDs may not be used.

Note: A list of assigned vendor IDs is available online at *Vendor Identifier Registry*.

Device ID (SYSTEM_DEVICE_ID)

- Address: 0x0201
- Reset Value: *implementation specific*
- Access: read-only

Number identifying the device (e.g. the chip) that contains the OSD implementation. The device ID must be uniquely describe the device design as it is visible through the debug system.

Device IDs are assigned by the device vendor, identified by SYSTEM_VENDOR_ID.

Debug module count (NUM_MOD)

- Address: 0x0202
- Reset Value: *implementation specific*
- Access: read-only

The number of debug modules, including the SCM module itself (which is always assigned address 0 in the subnet). Since all module addresses must be continguous, this value also describes the highest module address available in the debug system as NUM_MOD - 1.

Maximum packet length (MAX_PKT_LEN)

- Address: 0x0203
- Reset Value: *implementation specific*
- Access: read-only

Maximum length of debug packets in 16 bit words, including all headers. MAX_PKT_LEN must be at least 12 to enable the transmission of all register access packets within one packet.

System reset (SYSRST)

- Address: 0x0204
- Reset Value: *implementation specific*
- Access: read-only

Reset the (parts) of the system.

Table 2.21: Field Reference: CONF

Bit(s)	Field	Access	Reset Value	Description
15:2	RES	r/w	0x0	**Reserved**
1	CPU_RST	*impl.-spec.*	w	**CPU Reset** Reset all units executing code (e.g. CPUs) in the system. **0b0: Deactivate the CPU reset** The CPU reset signal is set to the deactivated state. **0b1: Activate the CPU reset** The CPU reset signal is set to the activated state, resetting all CPUs. The reset signal must be explicitly deactivated again with another register write.
0	SYS_RST	*impl.-spec.*	w	**System Reset** Put the device, excluding the debug system. **0b0: Deactivate the system reset** The system reset signal is set to the deactivated state. **0b1: Activate the system reset** The system reset signal is set to the activated state, resetting the device. The reset signal must be explicitly deactivated again with another register write.

2.8.3 Host Authentication Module (HAM)

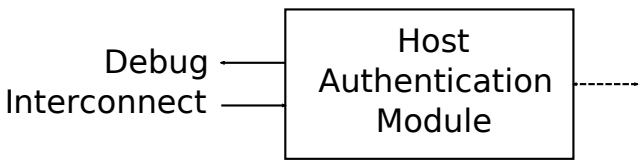

Fig. 2.6: Host Authentication Module

Todo: This module is not really specified yet.

The system can require the host to authenticate before connecting to the debug system, because the debug can expose confidential information. A *HAM* implementation can for example require a token to match or a sophisticated challenge-response protocol. If configured the HIM will wait for the HAM to allow the host to communicate with modules other than the HAM.

2.8.4 Memory Access Module (MAM)

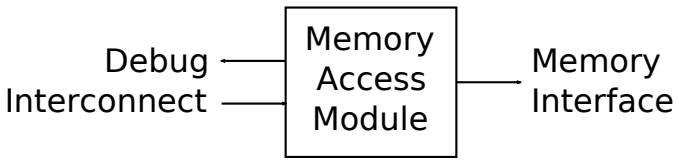

Fig. 2.7: Memory Access Module

The Memory Access Module (MAM) gives read and write access to a memory in the system. Typical use cases include the initialization of a memory with a program, or the inspection of memory post-mortem or during run-control debugging.

The module is either connected to the system memory, other memory blocks, or the last level cache.

System Interface

There is a generic interface between the MAM and the system:

Signal	Direction	Description
`req_valid`	MAM->System	Start a new memory access request
`req_ready`	MAM->System	Acknowledge the new memory access request
`req_we`	MAM->System	Write enable. `0`: Read, `1`: Write
`req_addr`	MAM->System	Request base address
`req_burst`	MAM->System	`0` for single beat access, `1` for incremental burst
`req_size`	MAM->System	Burst size in number of words
`write_valid`	MAM->System	Next write data is valid
`write_data`	MAM->System	Write data
`write_strb`	MAM->System	Byte strobe if `req_burst==0`
`write_ready`	System->MAM	Acknowledge this data item
`read_valid`	System->MAM	Next read data is valid
`read_data`	System->MAM	Read data
`read_ready`	MAM->System	Acknowledge this data item

Programmer Interface: Control Registers

The Memory Access Module implements the *Debug Module Base Register Map*. The reset values are listed below.

Table 2.22: MAM base register reset values

address	name	description	reset value
0x0000	`MOD_VENDOR`	module vendor	0x0001
0x0001	`MOD_TYPE`	module type identifier	0x0003
0x0002	`MOD_VERSION`	module version	0x0000
0x0003	`MOD_CS`	module control and status	0x0001
0x0004	`MOD_EVENT_DEST`	destination of debug events	0x0000 (unused, read-only)

Additionally, it implements the following control registers for MAM-specific functionality.

Table 2.23: MAM register map

address	name	description
0x0200	AW	address width of the attached memory in bits. Valid values are 16, 32 and 64.
0x0201	DW	data width of the attached memory in bits. Valid values are 16, 32 and 64.
0x0202	REGIONS	number of memory regions
0x0280	REGION0_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 0
0x0281	REGION0_BASEADDR_1	Bits [31:16] of the base address of region 0
0x0282	REGION0_BASEADDR_2	Bits [47:32] of the base address of region 0
0x0283	REGION0_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 0
0x0284	REGION0_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 0
0x0285	REGION0_MEMSIZE_1	Bits [31:16] of the memory size of region 0
0x0286	REGION0_MEMSIZE_2	Bits [47:32] of the memory size of region 0
0x0287	REGION0_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 0
0x0290	REGION1_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 1
0x0291	REGION1_BASEADDR_1	Bits [31:16] of the base address of region 1
0x0292	REGION1_BASEADDR_2	Bits [47:32] of the base address of region 1
0x0293	REGION1_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 1
0x0294	REGION1_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 1
0x0295	REGION1_MEMSIZE_1	Bits [31:16] of the memory size of region 1
0x0296	REGION1_MEMSIZE_2	Bits [47:32] of the memory size of region 1
0x0297	REGION1_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 1
0x02A0	REGION2_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 2
0x02A1	REGION2_BASEADDR_1	Bits [31:16] of the base address of region 2
0x02A2	REGION2_BASEADDR_2	Bits [47:32] of the base address of region 2
0x02A3	REGION2_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 2
0x02A4	REGION2_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 2
0x02A5	REGION2_MEMSIZE_1	Bits [31:16] of the memory size of region 2
0x02A6	REGION2_MEMSIZE_2	Bits [47:32] of the memory size of region 2
0x02A7	REGION2_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 2
0x02B0	REGION3_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 3
0x02B1	REGION3_BASEADDR_1	Bits [31:16] of the base address of region 3
0x02B2	REGION3_BASEADDR_2	Bits [47:32] of the base address of region 3
0x02B3	REGION3_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 3
0x02B4	REGION3_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 3
0x02B5	REGION3_MEMSIZE_1	Bits [31:16] of the memory size of region 3
0x02B6	REGION3_MEMSIZE_2	Bits [47:32] of the memory size of region 3
0x02B7	REGION3_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 3
0x02C0	REGION4_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 4
0x02C1	REGION4_BASEADDR_1	Bits [31:16] of the base address of region 4
0x02C2	REGION4_BASEADDR_2	Bits [47:32] of the base address of region 4
0x02C3	REGION4_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 4
0x02C4	REGION4_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 4
0x02C5	REGION4_MEMSIZE_1	Bits [31:16] of the memory size of region 4
0x02C6	REGION4_MEMSIZE_2	Bits [47:32] of the memory size of region 4
0x02C7	REGION4_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 4
0x02D0	REGION5_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 5
0x02D1	REGION5_BASEADDR_1	Bits [31:16] of the base address of region 5
0x02D2	REGION5_BASEADDR_2	Bits [47:32] of the base address of region 5

Continued on next page

Table 2.23 – continued from previous page

address	name	description
0x02D3	REGION5_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 5
0x02D4	REGION5_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 5
0x02D5	REGION5_MEMSIZE_1	Bits [31:16] of the memory size of region 5
0x02D6	REGION5_MEMSIZE_2	Bits [47:32] of the memory size of region 5
0x02D7	REGION5_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 5
0x02E0	REGION6_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 6
0x02E1	REGION6_BASEADDR_1	Bits [31:16] of the base address of region 6
0x02E2	REGION6_BASEADDR_2	Bits [47:32] of the base address of region 6
0x02E3	REGION6_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 6
0x02E4	REGION6_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 6
0x02E5	REGION6_MEMSIZE_1	Bits [31:16] of the memory size of region 6
0x02E6	REGION6_MEMSIZE_2	Bits [47:32] of the memory size of region 6
0x02E7	REGION6_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 6
0x02F0	REGION7_BASEADDR_0	Bits [15:0] (least significant bits) of the base address of region 7
0x02F1	REGION7_BASEADDR_1	Bits [31:16] of the base address of region 7
0x02F2	REGION7_BASEADDR_2	Bits [47:32] of the base address of region 7
0x02F3	REGION7_BASEADDR_3	Bits [63:48] (most significant bits) of the base address of region 7
0x02F4	REGION7_MEMSIZE_0	Bits [15:0] (least significant bits) of the memory size of region 7
0x02F5	REGION7_MEMSIZE_1	Bits [31:16] of the memory size of region 7
0x02F6	REGION7_MEMSIZE_2	Bits [47:32] of the memory size of region 7
0x02F7	REGION7_MEMSIZE_3	Bits [63:48] (most significant bits) of the memory size of region 7

Address Width (AW)

- Address: 0x0200
- Reset Value: *implementation specific*
- Access: read-only

The Address Width (AW) register contains the width of a memory address in bits. Address Width is guaranteed to be a multiple of 16.

Data Width (DW)

- Address: 0x0201
- Reset Value: *implementation specific*
- Access: read-only

The Data Width (DW) register contains the width of a data word in bits. Data Width is guaranteed to be a multiple of 16.

Number of Memory Regions (REGIONS)

- Address: 0x0202
- Reset Value: *implementation specific*
- Access: read-only

The Regions (REGIONS) register holds the number of memory regions available, as set during design time. At least 1 region is available.

Region Memory Base Address (REGION*_BASEADDR_*)

- Address: *see full register map above*
- Reset Value: *implementation specific*
- Access: read-only

The base address of a region 0-7 is given in the REGION*_BASEADDR_* registers. The base address is a 64 bit number stored in big endian format in four configuration registers.

For example, the base address of region 0 can be determined by the following operation:

```
region0_baseaddr = REGION0_BASEADDR_3 << 48 | REGION0_BASEADDR_2 << 32 | REGION0_
↪BASEADDR_1 << 16 | REGION0_BASEADDR_0
```

Note: For any given region, the corresponding base address register is only present if the region actually exists. You must read the REGIONS register first to determine how many regions are available.

Region Memory Size (REGION*_MEMSIZE_*)

- Address: *see full register map above*
- Reset Value: *implementation specific*
- Access: read-only

The memory size of a region 0-7 is given in the REGION*_MEMSIZE_* registers. The memory size is a 64 bit number stored in big endian format in four configuration registers.

For example, the memory size of region 0 can be determined by the following operation:

```
region0_memsize = REGION0_MEMSIZE_3 << 48 | REGION0_MEMSIZE_2 << 32 | REGION0_
↪MEMSIZE_1 << 16 | REGION0_MEMSIZE_0
```

Note: For any given region, the corresponding memory size register is only present if the region actually exists. You must read the REGIONS register first to determine how many regions are available.

Programmer Interface: Data

Reading and writing of the memory happens through a MAM-specific protocol inside DI packets of type EVENT. Data transfers can either be a read or a write access. Both burst and single word accesses are supported.

Figure 2.8 shows how data, which should be written to or read from a memory address is encapsulated.

In a first step, a **MAM Transfer Request** is formed, which consists of a header, the address to write to/read from, and (for write accesses) the data itself. In a second step, the MAM Transfer Request is split into parts each not exceeding the maximum payload size of the Debug Interconnect. Out of each of the resulting chunks a DI Packet of type EVENT is created.

In case of read accesses or acknowledged (synchronous) write accesses, a response is sent from the MAM to the source of the MAM Transfer Request.

In the following, we first explaing the structure of the different types of MAM transfers, and then its packetization into DI Packets.

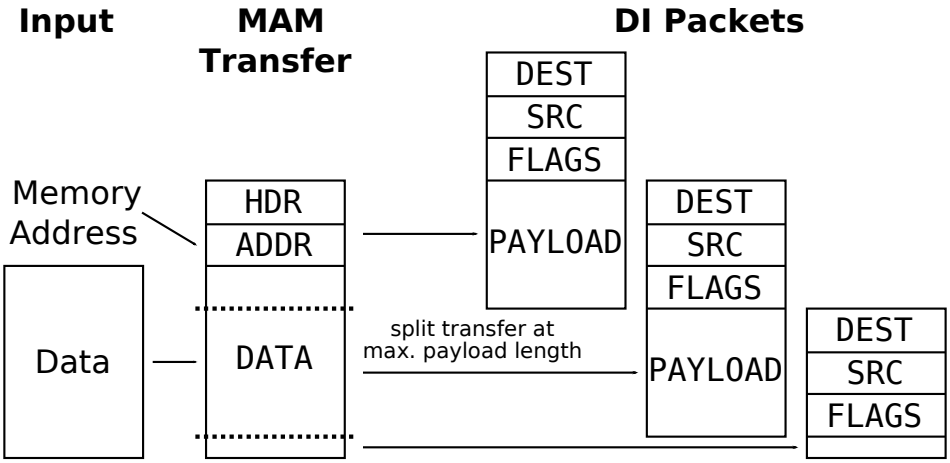

Fig. 2.8: MAM data transfer encapsulation

MAM Transfer Request

A MAM Transfer Request is used to read or write *s* bytes of data, starting at the byte address *addr*.

Note: The address *addr* must be word-aligned according to the data width `DW`. To access non-aligned data the byte-select field `SELSIZE` can be used.

A transfer request is structured as a sequence of bytes, consisting of a header, the memory byte address, and (in case of a write request) the write data. The structure of a MAM Transfer Request is given below.

The following variables are used:

- `AW` and `DW` are the address and data width, respectively, of the memory. The values for these variables can be read from the MAM control registers. The protocol supports address and data width values of 16, 32 and 64 bit.
- *s* is the number of bytes to transfer.
- *a* is the size of a memory address in bytes, calculated as *a* = `AW` / 8.

Table 2.24: Structure of a MAM Transfer Request

byte	name	description
MAM Transfer Request Header		
0	`HDR0`	MAM Transfer Request Header (part 1)
1	`HDR1`	MAM Transfer Request Header (part 2)
Address		
2	`ADDR(0)`	most significant byte of the read/write address, i.e. `addr[AW-1 : AW-8]`
...	...	...
1 + *a*	`ADDR(a-1)`	least significant byte of the read/write address, i.e. `addr[7 : 0]`
Write Data		
1 + *a* + 1	`D(0)`	the first data byte to be transferred, to be written to address *addr*.
1 + *a* + 2	`D(1)`	the second data byte to be transferred, to be written to address (*addr* + 1).
...	...	...
1 + *a* + *s*	`D(s-1)`	the last data byte to be transferred, to be written to address (*addr* + *s* - 1).

MAM Transfer Request Header, Part 1 (HDR0)

Table 2.25: Field Reference: HDR0

Bit(s)	Field	Description
7	WE	**Write Enable** **0: Read** read from memory **1: Write** write to memory
6	BURST	**Burst or Single Word Access Mode** This flag switches between burst and single word access. **0: Single Word Access** Use single word access. **1: Burst Access** Read or write from a continuous region of memory.
5	SYNC	**Use Synchronous Writes** **0: Asynchronous Writes** Asynchronous writes are not acknowledged by the MAM, thus other modules cannot know when a write has finished and the data has reached the attached memory. However, subsequent reads from the same MAM will return the newly written data. **1: Synchronous Writes** Synchronous writes are acknowledged by the MAM. The acknowledgement is an empty read response.
4:0	RESERVED	**Reserved for future extensions**

MAM Transfer Request Header, Part 2 (HDR1)

Table 2.26: Field Reference: HDR1

Bit(s)	Field	Description
7:0	SELSIZE	**Burst Size/Byte Select** This field has a different meaning depending on the value of the HDR0.BURST field. **If HDR0.BURST = 1: Burst Size** The number of words the transfer consists of, i.e. (*s* / DW). **If HDR0.BURST = 0: Byte Select** Only relevant for writes (HDR0.WE = 1): byte select. SELSIZE contains a bit mask, a data byte is only written if a corresponding bit in the mask is set to 1. For example, set SELSIZE[0] := 1 to write D0.

MAM Transfer Response

Table 2.27: Structure of a MAM Transfer Response

byte	name	description
Read Data		
$1 + a + 1$	D(0)	the first data byte read from the memory at address *addr*.
$1 + a + 2$	D(1)	the second data byte read from the memory at address (*addr* + 1).
...	...	...
$1 + a + s$	D(s-1)	the last data byte read from the memory at address (*addr* + *s* - 1).

Packetization

A MAM Transfer (both request and response) is packetized into DI event packets for transmission over the debug interconnect. Towards this goal, a MAM Transfer is split into chunks of each (MAX_PAYLOAD_LEN * 2) bytes. Each such chunk is sent as `PAYLOAD` in a DI packet.

The maximum number of payload words in a Debug Packet (`MAX_PAYLOAD_LEN`) can be determined by reading the `MAX_PKT_LEN` register of the SCM module and subtracting 3 to account for the header words.

The following fields in the header of the DI packet are set:

- `FLAGS.TYPE` is set to `EVENT`
- `FLAGS.TYPE_SUB` is set to 0

Table 2.28: MAM Packet Structure

payload word	description
0	[15 : 8] := `D(0)`, [7 : 0] := `D(1)`
1	[15 : 8] := `D(2)`, [7 : 0] := `D(3)`
...	...
`MAX_PAYLOAD_LEN` - 1	...

All packets except the last one should be of size `MAX_PKT_LEN` to reduce overhead.

2.8.5 Software Trace Module (STM)

The *Software Trace Module (STM)* emits trace events that are emitted by the software execution. Such an STM event is a tuple `(id,value)`. There are generally two classed: user-defined and system-generated trace events.

User-defined trace events are added by the user by instrumenting the source code with calls to an API like `TRACE(short id, uint64_t value)`. A debug tool can map the trace events to a visualization.

Different user threads can emit trace events interleaved. Beside this the operating system can emit relevant trace information too. For both reasons, there are system-generated events.

There are two ways to emit a software trace event. First there is a set of *special purpose registers* or similar techniques used to emit trace events. Most importantly, each trace event must be emitted atomically. Secondly, the processor core can have hardware to emit software trace events. For example a mode change can be emitted without much overhead.

The generic trace interface is `enable`, `id` and `value` at the core level and the STM handles the filtering, aggregation and packetization as described above.

System Interface: Software Trace Port

The software trace port is a simple data port with an `enable` signal. There is no backpressure as the debug infrastructure is not supposed to influence the processor execution.

The interface is defined as:

Name	Width	Description
`id`	16	Trace identifier
`value`	`VALWIDTH`	Trace value, width of CPU general purpose registers
`enable`	1	Trace an event this cycle

Trace generation

The method of emitting a trace event depends on the micro-architecture. Examples for existing processor core architectures are given in the following.

Software Trace Port: OpenRISC

In OpenRISC an interesting property of the instruction set is used: The no-operation `l.nop` has a parameter `K` of 16 bit width. The specification defines this parameter to be used for simulation purposes, and it is here used to emit the trace value.

We use this operation for the trace identifier. As the compiler emits `l.nop 0x0`, the user-defined value of `0x0000` is not available in this specification.

The trace value is defined to be written to the general purpose register `r3` with the properties described before. As a general purpose register is restored after interrupts, the atomicity property holds. Finally, the register `r3` is the first function parameter register in the ABI which eases efficient implementation of library functions for trace events.

In the hardware implementation the writeback stage must be observed and whenever a write to register `r3` is observed, the same value is stored into the register `value`. When completion of an `l.nop` operation is observed, the operand `K` (if not equal to 0) and the `value` are emitted on the trace port for one cycle.

Finally, the following extension is required to support the trace event `THREAD_SWITCH`: All writes to register `r10` must be tracked and if a value is written, the trace event is emitted. The register is historically reserved and in the Linux port used as thread-local storage (TLS), which is unique for concurrently executed threads.

Software Trace Port: RISC-V

In RISC-V an additional control register is added to emit a trace event (non-standard for the moment). A write to this register triggers the emission of the trace event for one cycle.

Beside this, the general purpose register `x10` (`a0`) is tracked for updates as the trace event value, identical to the reasoning for OpenRISC.

Finally the register `x4` (`tp`) may also be tracked and a `THREAD_SWICH` trace event is emitted on updates to the register.

Software Trace Port: Other cores

The method described for the RISC-V microarchitecture should be applicable to a variety of RISC cores.

Software Trace Port: Out-of-Order

With out-of-order cores it is important to track the accesses to the two data items properly, which can be enforced by a memory fence.

In an out-of-order implementation the software trace port may be implemented more efficiently at stages where the trace event may still be canceled. If that is the case, the software trace port should hold back the value until it can be safely emitted or aborted beforehand.

Programmer Interface: Control Registers

The System Trace Module implements the *Debug Module Base Register Map*. The reset values are listed below.

Table 2.29: STM base register reset values

address	name	description	reset value
0x0000	`MOD_VENDOR`	module vendor	0x0001
0x0001	`MOD_TYPE`	module type identifier	0x0004
0x0002	`MOD_VERSION`	module version	0x0000
0x0003	`MOD_CS`	module control and status	0x0000
0x0004	`MOD_EVENT_DEST`	destination of debug events	0x0000

Additionally, it implements the following control registers for module-specific functionality.

Table 2.30: STM register map

address	name	description
0x0200	VALWIDTH	width of the value data items emitted by this module, in bit. Typically identical with the register width of the processor the STM module is connected to. Valid values are 16, 32 and 64.

Programmer Interface: Data

The STM module generates two types of event packets: trace packets and overflow packets. Trace packets contain the trace data in the form of (id,value) tuples. Overflow packets indicate that trace events were missed, usually if more events are generated than the module can send out to the DI.

Trace Packets

A Trace Packet encapsulates a trace event, which consists of an identifier id (always 16 bit wide) and an associated value value (VALWIDTH bit wide).

The following fields in the header of the DI packet are set:

- FLAGS.TYPE is set to EVENT
- FLAGS.TYPE_SUB is set to 0

Table 2.31: STM Trace Packet Structure

payload word	description
0	timestamp[15:0]
1	timestamp[31:16]
2	id
3	value[15:0]
...	...
2 + VALWIDTH / 16	value[VALWIDTH-1:VALWIDTH-16]

Overflow Packets

The following fields in the header of the DI packet are set:

- FLAGS.TYPE is set to EVENT
- FLAGS.TYPE_SUB is set to 0x5

Table 2.32: STM Overflow Packet Structure

payload word	description
0	number of missed events

Trace Events

Trace events are a tuple (id, value). The identifier (id) can be used to identify the type of event. To achieve interoperability between implementations, some identifiers are specified, while others can be used for vendor extensions.

Generally, the trace identifiers are divided into these groups:

Identifiers	Group	Description
0x0000	N/A	Not used
0x0001 - 0x3fff	USER	User-defined trace events
0x4000 - 0x7fff	COM	Commonly defined trace events
0x8000 - 0xbfff	N/A	Reserved
0xc000 - 0xffff	SYS	System-generated trace events

The implementation of all trace events is optional, both for software and for hardware.

User-defined trace events (USER)

Trace events from this group are generated from software execution. There are many possibilities to implement them in hardware, but from the ABI there are generally two data items that a software writes to emit a trace event: first it writes the `value` and then it writes the `id`.

As those are two memory accesses in two distinct operations the following properties must hold:

- Sequential consistency: The write to the second data item must occur after the first data item. This property is usually enforced with memory fences.
- Atomicity: The first data item must not be changed in multithreaded systems or by interrupt processing in general.

For more details see the section *Trace Generation* in the following.

Commonly defined trace events (COM)

Those are trace events identical to the user-defined trace events, but that have a commonly defined semantic meaning. Their semantic meaning is therefore still transparent to the hardware module, but common to all platforms. This eases implementation of trace debugger tools.

Identifier	Name	Description
0x4000	THREAD_NAME	Emit a thread name, emitted repeatedly

System-generated trace events (SYS)

This group of trace events are generated by the hardware or by the operating/runtime system. For the latter the same method as user-defined trace events is used. For hardware-generated events the method of emitting the trace event is core-specific and examples are described in the the section *Trace Generation*.

Identifier	Name	Description
0x8000	THREAD_SWITCH	Unique value of the scheduled thread

2.8.6 Core Debug Module (CDM)

The Core Debug Module (CDM) provides access to the run-control debug functionality of a single CPU core. The CDM targets CPU cores which provide a memory-mapped interface to their registers which control the debugging procedure. In its current form the CDM targets the 32 bit or1k CPU core and other cores with a similar interface.

Through the CDM a debugging tool, e.g. GDB, can access the Special Purpose Registers (SPRs) of the CPU core to control the debugging process. Additionally the debugging tool uses the Memory Access Module (MAM) to read and write data from/to the memory. Debugging-related events (e.g. “core has stalled”) are signalled through a OSD event packet.

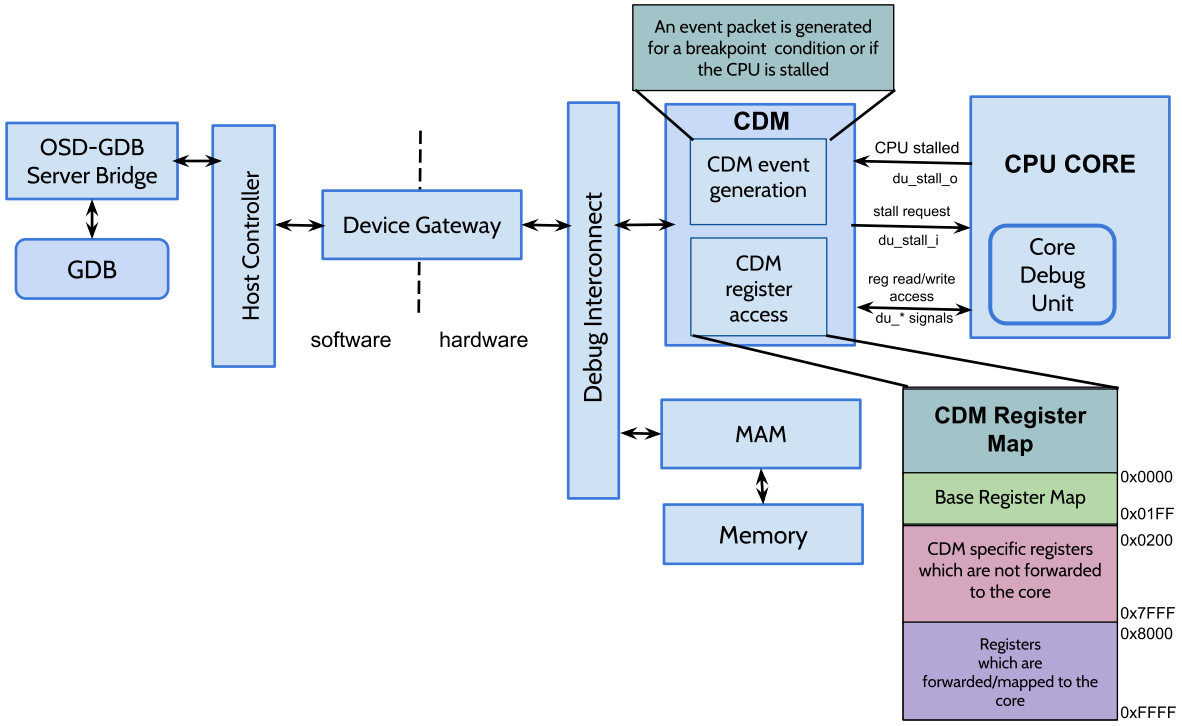

Fig. 2.9: High Level Overview of the Core Debug Module (CDM)

System Interface

The following signals are required from the CPU core in order to interface to the CDM module:

Signal	Width	Direction	Description
du_stall_i	1	CDM->CPU	Logic '1' causes CPU to stall
du_stall_o	1	CPU->CDM	Indicates CPU has reached breakpoint condition
du_stb_i	1	CDM->CPU	Access to the core debug interface
du_ack_o	1	CPU->CDM	Complete access to the core
du_adr_i	16	CDM->CPU	Address of CPU register to be read or written
du_we_i	1	CDM->CPU	Write cycle when true, read cycle when false
du_dat_i	32	CDM->CPU	Write data
du_dat_o	32	CPU->CDM	Read data

Reference: https://opencores.org/usercontent/doc/1242694069 (Section 2.2.1)

Programmer Interface: Control Registers

The Control Debug Module implements the *Debug Module Base Register Map*.

Table 2.33: CDM base register reset values

address	name	description	reset value
0x0000	MOD_VENDOR	module vendor	0x0001
0x0001	MOD_TYPE	module type identifier	0x0006
0x0002	MOD_VERSION	module version	0x0000
0x0003	MOD_CS	module control and status	0x0000
0x0004	MOD_EVENT_DEST	destination of debug events	0x0000

Additionally, the CDM implements the following registers.

Table 2.34: CDM register map

address	name	width (bit)	description
0x0200	CORE_CTRL	16	Control the CPU core
0x0201	CORE_REG_UPPER	16	Most significant bits of the SPR address (see below)
0x0202	CORE_DATA_WIDTH	16	Register data width of the attached CPU core in bits
0x8000-0xFFFF		32	Access to the SPRs of the CPU core (see below)

Core Control Register (CORE_CTRL)

- Address: 0x0200
- Reset Value: 0
- Data Width: 16 bit
- Access: read-write

Table 2.35: Field Reference: CORE_CTRL

Bit(s)	Field	Access	Reset Value	Description
15:1	RES	r/w	0x0	**Reserved**
0	STALL	r/w	*impl.-spec.*	**Core Stall** Stall the attached CPU core. **0b1: Stall the core** The core is stalled. **0b0: Unstall the core** The core is unstalled.

Core Upper Register (CORE_REG_UPPER)

- Address: 0x0201
- Reset Value: 0
- Data Width: 16 bit
- Access: read-write

The most significant bit of the SPR register address. See the section "Access to core registers" for more details.

Core Data Width (CORE_DATA_WIDTH)

- Address: 0x0202
- Reset Value: 0
- Data Width: 16 bit
- Access: read-write

The register width of the CPU core (in bits) the CDM module is connected to. Valid values are 16, 32 and 64 and 128.

Access to core registers

- Address: 0x8000-0xFFFF
- Reset Value: *implementation specific*
- Data Width: 32 bit
- Access: read-write

Accesses to CDM registers between 0x8000 and 0xFFFF are forwarded to the SPRs of the attached CPU core. The register address of the accessed SPR can be determined with the help of the `CORE_REG_UPPER` value using the following rule:

```
spr_reg_addr = CORE_REG_UPPER << 15 | cdm_reg_addr - 0x8000
```

Consult the specification of the attached CPU core for a further description of the register accessed, and possible access limitations (e.g. read-only registers).

Programmer Interface: Data

The CDM module generates only one type of event packets: **CPU debug stall packets**. These packets contain the data in the form of `stall` payload word.

CPU Debug Stall packet

A CPU Debug Stall Packet encapsulates a breakpoint or watchpoint event. Whenever the program counter in the CPU core matches with the watchpoint/breakpoint address, CPU core stalls and this event packet is generated. It notifies the debugger, i.e. GDB that CPU has reached a breakpoint or watchpoint condition and the CPU core is stalled.

The following fields in the header of the DI packet are set:

- `FLAGS.TYPE` is set to `EVENT`
- `FLAGS.TYPE_SUB` is set to 0

Table 2.36: CPU Debug Stall Packet Structure

payload word	description
0	`stall` Bit '0': Logic 1 indicates the debugger that the CPU core is stalled.

2.8.7 Core Trace Module (CTM)

The *Core Trace Module (CTM)* captures trace events generated from the processor core, and sends them in compressed form as event packets.

Which events are available depends on the observed processor core. Typically the following events are traced:

- executed instructions (instruction trace)
- branch predictor status
- memory access delays
- cache miss rates

Note: The CTM module is in a very early preview state and has significant limitations. It currently focuses on function call traces and has been tested only on RISC-V and or1k (OpenRISC) ISAs. No trace compression mechanisms are employed.

Programmer Interface: Control Registers

The Core Trace Module implements the *Debug Module Base Register Map*. The reset values are listed below.

Table 2.37: CTM base register reset values

address	name	description	reset value
0x0000	MOD_VENDOR	module vendor	0x0001
0x0001	MOD_TYPE	module type identifier	0x0005
0x0002	MOD_VERSION	module version	0x0000
0x0003	MOD_CS	module control and status	0x0000
0x0004	MOD_EVENT_DEST	destination of debug events	0x0000

Additionally, it implements the following control registers for module-specific functionality.

Table 2.38: CTM register map

address	name	description
0x0200	ADDR_WIDTH	width of memory addresses in bit. Valid values are 16, 32 and 64.
0x0201	DATA_WIDTH	width of a data word in bit. Valid values are 16, 32 and 64.

Programmer Interface: Data

The CTM module generates two types of event packets: trace packets and overflow packets. Trace packets contain the data traced from the processor core. Overflow packets indicate that trace events were missed, usually if more events are generated than the module can send out to the DI.

Trace Packets

A Trace Packet encapsulates a single trace event.

The following fields in the header of the DI packet are set:

- FLAGS.TYPE is set to EVENT
- FLAGS.TYPE_SUB is set to 0

Table 2.39: CTM Trace Packet Structure

payload word	description
0	`timestamp[15:0]`
1	`timestamp[31:16]`
2	Next Program Counter `npc[15:0]`
...	...
1 + `ADDR_WIDTH` / 16	Next Program Counter `npc[ADDR_WIDTH-1:ADDR_WIDTH-16]`
1 + `ADDR_WIDTH` / 16 + 1	Program Counter `pc[15:0]`
...	...
1 + 2 * (`ADDR_WIDTH` / 16 + 1)	Program Counter `pc[ADDR_WIDTH-1:ADDR_WIDTH-16]`
1 + 2 * (`ADDR_WIDTH` / 16 + 1) + 1	[1:0] `mode`: The privilege mode of the executed instruction. [2] `ret`: The executed instruction returned from a subroutine (e.g. Jump and Link Register (*jalr*) on RISC, *ret* on x86). [3] `call`: The executed instruction called a subroutine (e.g. Jump and Link (*jal*) on RISC, *call* on x86). [4] `modechange`: The executed instruction changed the privilege mode (e.g. from user to kernel space). [15:5] reserved

Overflow Packets

The following fields in the header of the DI packet are set:

- `FLAGS.TYPE` is set to `EVENT`
- `FLAGS.TYPE_SUB` is set to 0x5

Table 2.40: CTM Overflow Packet Structure

payload word	description
0	number of missed events

2.8.8 UART Device Emulation Module (DEM-UART)

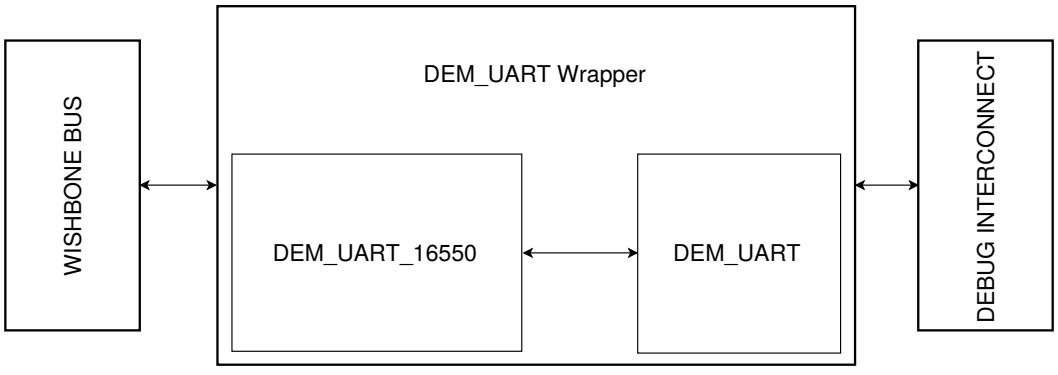

Fig. 2.10: High-Level Overview of the Device Emulation Module for UART (DEM-UART)

The UART Device Emulation Module connects to the bus of a given CPU on one side, and to the Debug Interconnect on the other. Towards the CPU it behaves like a UART-16550A device, but instead of transmitting information over a UART-Interface it instead passes it to a host PC via the Debug Interconnect.

Programmer Interface: Control Registers

The Device Emulation modules implements the Base Register Map. (*Debug Module Base Register Map*) The reset values are listed below.

Table 2.41: DEM-UART base register reset values

address	name	description	reset value
0x0000	MOD_VENDOR	module vendor	0x0001
0x0001	MOD_TYPE	module type identifier	0x0002
0x0002	MOD_VERSION	module version	0x0000
0x0003	MOD_CS	module control and status	0x0000
0x0004	MOD_EVENT_DEST	destination of debug events	impl.-specific

There are no additional registers implemented in this module.

Programmer Interface: Data

The Device Emulation Module only generates one type of event packet: **DEM-UART Data Packet**

DEM-UART Data Packet

A DEM-UART Data packet contains one 8-bit character, that has been sent to a UART interface by a CPU, as the only payload. All packets of this type are sent to the DI Address MOD_EVENT_DEST.

The following fields in the header of the DI packet are set:

- FLAGS.TYPE is set to EVENT
- FLAGS.TYPE_SUB is set to 0x00

The resulting Debug Interconnect packet has the following structure.

Table 2.42: Structure of a DEM-UART data packet

Word	Field	Description
1	CHARACTER	Contains the character that is being sent, the MSB is always 0x00

16550 UART Registers

The following UART registers are implemented and accessible via the bus, the address mapping is in accordance with the UART-16550(A) standard as specified in this Datasheet[6]. No FIFOs are present in hardware. No Modem or DMA-Mode related features, registers or interrupts are implemented. Writing to a register that is not implemented has no effect, reading from such a register will always return 0x00.

All registers are 8 bit wide.

[6] http://caro.su/msx/ocm_de1/16550.pdf

Table 2.43: 16550 UART Registers

Address	Register	Access Type	Reset Value	Description
0x00	RBR	Read only	0x00	Receiver Buffer Register
0x00	THR	Write only	0x00	Transmitter Holding Register
0x01	IER	Read/Write	0x00	Enable(1)/Disable(0) interrupts. See this[7] for more details on each interrupt.
0x02	IIR	Read only	0x01	Information which interrupt occurred
0x02	FCR	Write only	0x00	Control behavior of the internal FIFOs. Currently writing to this Register has no effect.
0x03	LCR	Read/Write	0x00	The only bit in this register that has any meaning is LCR7 aka the DLAB, all other bits hold their written value but have no meaning.
0x05	LSR	Read only	0x60	Information about state of the UART. After the UART is reset, 0x60 indicates when it is ready to transmit data.

System Interface

The DEM-UART module provides a generic bus interface, which can be used by wrapper modules to support actual bus interfaces. We specify a Wishbone wrapper interface below.

Signal	Width (bit)	Direction	Description
bus_req	1	CPU->DEM	1 indicates an active request from the CPU
bus_addr	3	CPU->DEM	Address to be used with write/read operation
bus_write	1	CPU->DEM	1 indicates a register write request
bus_wdata	8	CPU->DEM	Data to be written into the register
bus_ack	1	DEM->CPU	Acknowledge last request
bus_rdata	8	DEM->CPU	Data that was read from the register

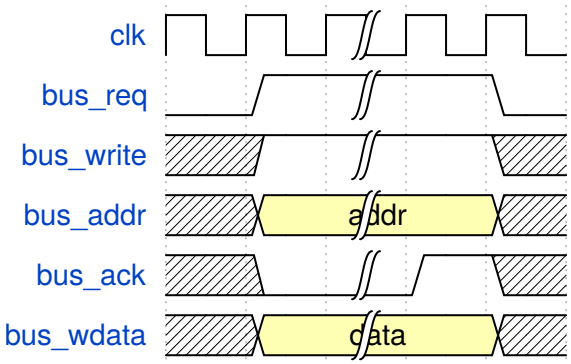

Fig. 2.11: A typical write cycle

A new request is made by asserting bus_req, unless bus_req is asserted, no other signal is valid. bus_write indicates whether it is a read (0) or a write (1) request. bus_addr may be any of the values documented under *16550 UART Registers*. Finally bus_ack is asserted to confirm the request and end the cycle.

bus_req, bus_addr and bus_write are asserted in the same cycle, if it is a write cycle bus_wdata is also set in the same cycle. bus_ack may be asserted any number of cycles after bus_req has been asserted. bus_rdata is only valid when bus_ack is asserted and bus_write is negated.

[7] http://caro.su/msx/ocm_de1/16550.pdf

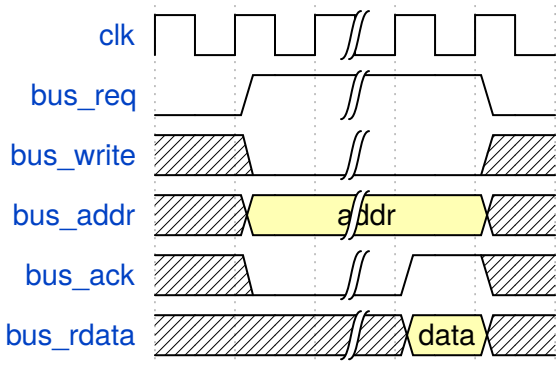

Fig. 2.12: A typical read cycle

Wishbone Bus Interface

If a wishbone interface is present, it should wrap around the generic bus described above and take care of translating all the signals. The following signals MUST be present on a compatible WISHBONE bus.

Signal	Width (bit)	Direction	Description
`wb_adr_i`	3	CPU->DEM	Address to be used with write/read operation
`wb_cyc_i`	1	CPU->DEM	`1` indicates valid bus cycle is in progress
`wb_dat_i`	32	CPU->DEM	Data to be written into the register
`wb_sel_i`	4	CPU->DEM	Bitfield indicating validity of data on dat_i
`wb_stb_i`	1	CPU->DEM	`1` indicates that DEM is selected
`wb_we_i`	8	CPU->DEM	`1` indicates a write request by the WB-Master
`wb_ack_o`	1	DEM->CPU	`1` indicates termination of normal bus cycle
`wb_dat_o`	32	DEM->CPU	Data that was read from the register

For more information see the Wishbone B3 specification[8]

2.9 Revision History

2.9.1 Preview 2 (to be released)

2nd preview of initial version for discussion.

- Revised document structure.
- Added more introduction and clarified design goals.

2.9.2 Preview 1 (released 2016-02-01)

1st preview of initial version for discussion.

A full revision history in all detail is available in our Git repository[9].

[8] https://cdn.opencores.org/downloads/wbspec_b3.pdf
[9] https://github.com/opensocdebug/documentation

CHAPTER 3

User Guides

You have a SoC design which contains an Open SoC Debug-enabled debug system? The OSD User Guides describe how to make use of OSD, answering questions like

- How can I connect to an OSD-enabled SoC?
- How can I initialize the chip's memories with my software program?
- How can I start and stop the software execution?
- How can I produce a trace of all functions?

Todo: This part of the documentation remains to be written.

CHAPTER 4

Implementer Documentation

Open SoC Debug comes with many hooks and options to implement your own ideas on top of it. This part of the documentation targets developers working on integrating Open SoC Debug software or hardware into their products, as well as people contributing to OSD itself.

4.1 Style Guides

4.1.1 SystemVerilog Coding Guidelines

We try to maintain SV compatibility for 4 tools: Verilator, VCS, ISim (the Vivado simulator) and Vivado Synthesizer. To increase the code compatibility, please following the following guidance until tools are improved.

1. Avoid using arrayed interface any where. (port or inside a module)

   Existing issues in: ISim, Vivado

2. Avoid using functions/tasks in interfaces.

   Existing issues in: VCS

3. Avoid using simple names, such as "length", "size", "out", "in". They can be mistaken into functions.

   Existing issues in: ISim

4. Avoid "assign" member elements of a struct, use always_comb instead.

   Existing issues in: ISim

5. Avoid using interface as data buffer inside a module.

   Existing issues in: Potentially all as interface is not supposed to be used in this way.

6. Avoid using interface to connect multiple hierarchical ports. Such as connect `A.data -> B.data -> C.data`, where B is a sub-module of A and C is a sub-module of B.

   Existing issues in: ISim (surprisingly it does not support this basic feature!)

7. Avoid using interface as top-level ports. Interfaces are flattened after synthesis, which causes port mismatch between behavioural DUT and post-syn DUT. Arguably avoid using interface modport. Some tool cannot correctly check the modport input/output definition anyway.

   Existing issues in: ISim(no check), Verilator(no check)

8. Use `always_comb` rather than `always_comb @(*)`. Even wild-cased sensitive list is an error in VCS.

   Existing issues in: VCS

4.1.2 Documentation Style Guide

The Open SoC Debug documentation is written in reStructuredText (rST). It is then processed by Sphinx[10] to generate various output formats, including HTML (for online documentation) and PDF (for offline reading and printing).

A good syntax overview is included in the Sphinx documentation reStructuredText Primer[11].

For the OSD docs, we have a couple of additional conventions.

Headlines and Sections

rST does not have strict rules regarding the formatting of section headers. To keep things consistent, we follow the rules recommended by the Python Style Guide[12]. That is:

- # with overline, for parts
- * with overline, for chapters
- =, for sections
- -, for subsections
- ^, for subsubsections
- ", for paragraphs

Tables

Tables are a tricky business, especially if both the PDF and the HTML output should look reasonable and editing should be easy. We included the Sphinx extension "flat-table" extension, which is also used by the Linux Kernel documentation, as an easier way of writing tables as nested list. This makes tables a bit harder to read in source code, but easier to edit and clearer in diffs. In addition, we use the tabularcolumns[13] directive to explicitly specify the width of the table columns in LaTeX/PDF output.

The following code snippet shows how to use flat-table together with a LaTeX column width specification required for good-looking PDF output:

```
.. tabularcolumns:: |p{\dimexpr 0.20\linewidth-2\tabcolsep}|p{\dimexpr 0.
↪20\linewidth-2\tabcolsep}|p{\dimexpr 0.60\linewidth-2\tabcolsep}|
.. flat-table:: Example Table Title
  :widths: 2 2 6
  :header-rows: 1

  * - address
    - name
    - description

  * - 0x0000
    - ``MOD_ID``
    - module type identifier
```

Notes:

- Specify the width as fractions of 1 (equal to 100 %) in the tabularcolumn directive.

[10] http://www.sphinx-doc.org
[11] http://www.sphinx-doc.org/en/stable/rest.html
[12] https://docs.python.org/devguide/documenting.html#sections
[13] http://www.sphinx-doc.org/en/stable/markup/misc.html#directive-tabularcolumns

- Specify the width of columns as fraction of 10 in the `::widths:` parameter.
- Do not change anything in the `tabularcolumns` directive except for the width unless you have double-checked that both HTML and PDF output look fine.
- Give tables a caption.

Figures

Pixel Graphics

You can use PNG and JPEG images as usual. Make sure to provide them in sufficiently large resolution (> 150 dpi) to be suitable for printing.

Vector Graphics

For vector graphics, use SVG.

If you include the figure in your document, use `.*` instead of `.svg` as file extension. This instructs Sphinx to use the most appropriate extension for the output format. For HTML, SVG images are included directly. For PDF/LaTeX, the images are first converted to PDF using Inkscape.

Example:

```
.. figure:: img/overview.*
   :alt: Open SoC Debug architecture overview
   :name: fig:spec:architecture:overview

   High-level overview of the Open SoC Debug architecture.

:numref:`Figure %s <fig:spec:architecture:overview>` shows the different␣
↪components in a typical Open SoC Debug-based debug system.
```

Notes:

- Make sure not to use any non-standard font inside the graphic, as the SVG will be displayed as-is to users who might not have the font installed (SVG does not embed fonts; the only thing you can do is create a path out of the text, removing the ability to edit the text.)
- SVG images do not show up in PDFs generated by ReadTheDocs at the moment.

CHAPTER 5

Vendor Identifier Registry

Vendor Identifiers are numbers assigned to an entity building Open SoC Debug components or products/devices and serve as an identifier namespace for that entity. In combination with another identifier (such as a device identifier or a module identifier) unique identifiers can be created. Vendor IDs are used in different places in OSD, most notably:

- to describe the type of a debug module (`MOD_VENDOR` and `MOD_ID`) and
- to identify the full OSD-enabled device type (`SYSTEM_VENDOR_ID` and `SYSTEM_DEVICE_ID`)

The vendor identifier is issued by the Open SoC Debug Project and listed on this page. Other identifiers are usually assigned independently by the vendor.

To register a new vendor ID please open an issue (or pull request) on GitHub for this document. No fees or documents are needed.

Table 5.1: OSD Vendor IDs

vendor ID	name
0x0001	The Open SoC Debug Project
0x0002	The OpTiMSoC Project
0x0003	LowRISC

CHAPTER 6

Licensing

Open SoC Debug is licensed in a way which allows you to use, modify, and distribute OSD itself and products based on it without paying any royalties. At the same time, The OSD Contributors do not give any warranty that it works as expected (or at all). The legal details are layed out in the licenses that govern the individual parts of OSD. Due to the different nature of code and documentation, we use different licenses.

Note: Only the licenses and other legal documents itself are binding, this page only gives a human-readable and not legally binding overview. We cannot provide any legal advise if OSD is usable for your use case. Please speak to a lawyer in such cases.

6.1 Documentation License

This documentation, including the OSD specification itself, is licensed under the Creative Commons Attribution-ShareAlike 4.0 International License. To view a copy of this license, visit http://creativecommons.org/licenses/by-sa/4.0/ or send a letter to Creative Commons, PO Box 1866, Mountain View, CA 94042, USA.

6.2 Reference Implementation License

We try to keep the reference implementation free for use even in commercial closed-source settings. Towards this goal, the software is generally licensed under the MIT license[14], while the hardware reference implementation carries the Solderpad License[15], a variant of the Apache 2.0 software license adjusted for hardware implementations.

Please see the individual source code repositories and files for more detailled information and possible exceptions.

[14] https://opensource.org/licenses/MIT
[15] http://solderpad.org/licenses/

Index

R