Project acronym:	**TRESCCA**
Project title:	TRustworthy Embedded systems for Secure Cloud Computing
Project number:	European Commission – 318036
Call identifier:	FP7-ICT-2011.1.4
Start date of project:	01 Oct. 2012 **Duration:** 36 months

Document reference number:	**D2.4**
Document title:	Hardware Security Module
Version:	1.1
Due date of document:	31st of March 2015
Submission date:	6th of July 2015
Lead beneficiary:	IMT
Participants:	Jérémie BRUNEL (IMT), Guillaume DUC (IMT), Salaheddine OUAARAB (IMT), Renaud PACALET (IMT), Abdelmalek SI MERABET (IMT)
Reviewer:	

Project co-funded by the European Commission within the 7th Framework Programme

DISSEMINATION LEVEL		
PU	Public	**X**
PCA	Public with confidential annex	
CO	Confidential, only for members of the consortium (including Commission Services)	

EXECUTIVE SUMMARY

This document, part of the deliverable D2.4, describes the content of the archive containing the source code of the Hardware Security Module (HSM-Mem) and how to use it.

CONTENTS

LIST OF FIGURES

LIST OF TABLES

1 INTRODUCTION

The deliverable D2.4 of the TRESCCA project consists of an archive file containing the VHDL source files of the Hardware Security Module (HSM-Mem), simulation and synthesis scripts, and of this document that describes the content of the archive and how it can be used.

The most recent version of the archive can be downloaded from the SecBus project website: https://secbus.telecom-paristech.fr/raw-attachment/wiki/Downloading/secbus-0.1.tgz.

The Hardware Security Module for memory protection (HSM-Mem) is responsible for enciphering and deciphering the data read/written from/to the external memories and for managing and checking their integrity. It sits on-chip, between the central interconnect and the memory controller. The full description of the architecture of the HSM-Mem and a SystemC model are included into the deliverable D2.2.

The first part of this document is a brief reminder of the HSM-Mem architecture. Compared to D2.2, it does not provide any new information. It is given here to such that this document is as self-contained as possible. The second part presents the organization and the content of the archive. The third part describes how to use the different scripts and makefiles to test and synthesize the HSM-Mem.

1.1 Document Versions Sheet

Version	Date	Description, modifications, authors
1.0	2015-04-17	Initial version for Technical Review. J. BRUNEL (IMT), G. DUC (IMT), S. OUAARAB (IMT), R. PACALET (IMT), A. SI MERABET (IMT)
1.1	2015-07-06	Add functional description of HSM-Mem. J. BRUNEL (IMT), G. DUC (IMT), S. OUAARAB (IMT), R. PACALET (IMT), A. SI MERABET (IMT)

2 HSM-MEM ARCHITECTURE

2.1 Position and role in the global TRESCCA platform

The TRESCCA client platform is a modular and flexible HW/SW architecture that is adaptable to different application use cases ranging from embedded systems over smart phones and tablets to set top boxes. TRESCCA itself does not specify or propose a specific HW/SW architecture but provides a set of HW and SW components that can be integrated into typical System-on-Chip (SoC) designs.

The HW architecture of the platform, as shown in Fig. 2.1 is based on existing off-the-shelf SoC designs (e.g. multi-core ARM-based SoCs) which are extended by hardware security modules (HSMs). These HSM significantly improve the security of the systems by protecting the external memory bus (HSM-mem) and by controlling the access and sharing of internal SoC IP components by Virtual Machines. This document is about HSM-mem only. Please refer to deliverable D2.3 *Security Hardware with Support for Virtualization* for a description of the HSM-NoC.

One of the demonstration targets for the HSMs is based on the Zynq cores from Xilinx[1]. Figure 2.1 shows how the two HSMs are inserted in a Zynq-based prototyping platform (like, for instance, the ZedBoard[2]). The different address ranges used by the processor to access its address space are shown and explain how the memory accesses can be routed through the Programmable Logic (PL) where the HSMs are mapped.

The Hardware Security Module for memory protection (HSM-mem) is responsible for enciphering and deciphering the data read/written from/to the external memories and for managing and checking their integrity. It sits on-chip, between the central interconnect and the memory controller. It is driven by a small set of interface registers (as any hardware peripheral) and by control data structures stored in external memories, a bit like a Memory Management Unit (MMU) is driven by tables of Page Table Entries (PTE) also stored in external memories. Each access to the external memories issued by the System on Chip (SoC) flows through the HSM-mem before reaching the memory controller. Upon read accesses, the returned data flow through the HSM-mem before reaching the central interconnect. The HSM-mem uses the physical addresses of the memory accesses to identify what Security Policy (SP) to apply, both in terms of confidentiality and integrity. The association between physical memory pages and SPs is specified by a table of Page Security Parameter Entries (PSPEs) stored in external memory. PSPEs contain several fields among which one finds the index of a SP. SPs are also stored in a table in external memory. The HSM-mem is capable a walking through these tables of control data structures autonomously.

2.2 Internals of the HSM-mem

Figure 2.3 illustrates the global architecture of the HSM and the interconnections between the different sub-modules.

The HSM embeds three types of modules:

- Interface modules handle the requests coming from the SoC interconnect, check whether protection is required or not, and route the requests-responses accordingly.
  - `VciSplit`

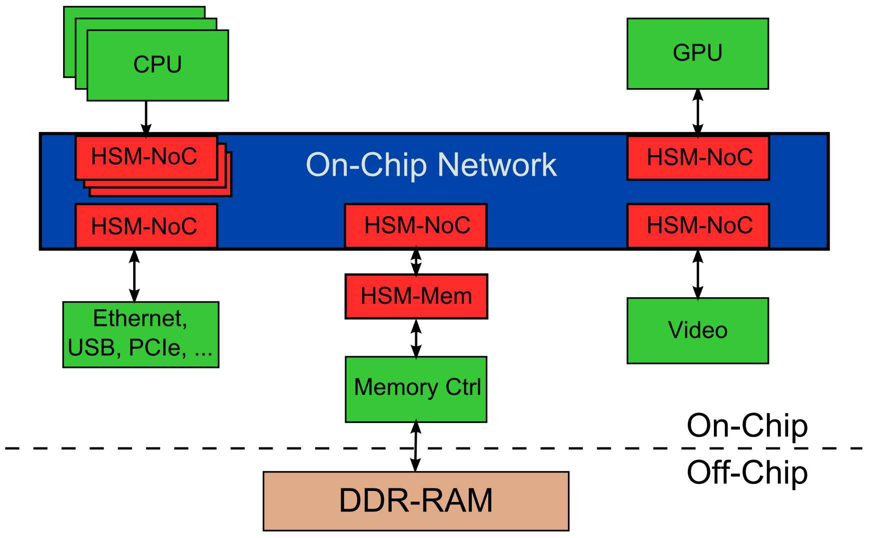
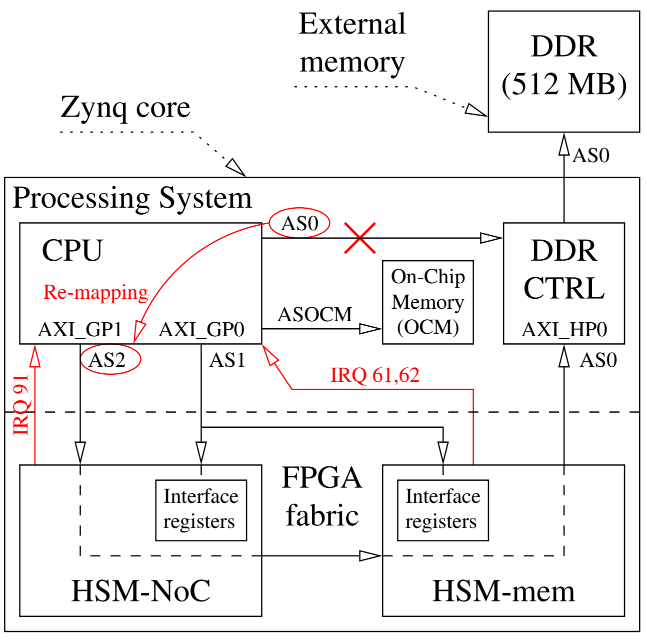


Figure 2.1: TRESCCA client HW architecture with HSMs

- `VciMerge`
- `VciInputCtrl`
- `VciMemCtrl`
- `MemArbiter`

- Protection modules are in charge of managing or applying the cryptographic primitives.
  - `SecurityCtx_Ctrl`
  - `Security_Ctrl`
  - `MT_Ctrl`
  - `MTCache_Ctrl`
  - `MS_Ctrl`
  - `MSCache_Ctrl`
  - `CryptoEngine_conf`
  - `CryptoEngine_int`
  - `CryptoArbiter`
  - `ScArbiter`
  - `IrqHandler`
- Miscellaneous (internal caches, general purpose 256-bits registers $R0$ to $R4$, FIFOs, multiplexers...)

The VHDL source code of all these modules and of their assembly as the complete HSM-mem is given in the archive, as will be explained in chapter 3.



AS0: `[0...1G[`
AS1: `[1G...2G[`
AS2: `[2G...3G[`
ASOCM: `[4G-256K...4G[`

Figure 2.2: Example HSM-NoC and HSM-mem prototype on a ZedBoard.

2.3 Control and status registers of the HSM-mem

The HSM is controlled through a set of interface registers and a set of data structures stored in external memory. The HSM low-level software driver offers a small set of software primitives to access both. Before listing these primitives we will explore the interface registers and explain their role. In the following the interface registers are read-write, unless otherwise stated. An unused register's field is represented as a grey area. Reading an unused field always returns a zero value and writing it has no effect. When reading or writing a register with unused fields it is recommended to assume zero values and to write zero values in unused fields because if future versions make use of these fields the zero value will always be the default one, corresponding to the current behaviour.

The configuration register

The *configuration register* (`cfg`, figure 2.4 and table 2.1) defines the global configuration of the HSM (address of the Master Block in external memory, various enable flags, definition of the protected memory area). It is mainly used at HSM initialization. The interrupts enable flag can also be set/unset during execution.

Table 2.1: Hardware Security Module `cfg` register fields

Name	**Width**	**Long name**	**Description**
`mbba`	8 bits	Master Block Base Address	Aligned multiple of 16MB. 8 MSBs only. Must be set prior use of external memory.
`en`	1 bits	hsm ENable	0=disable, 1=enable.

Name	**Width**	**Long name**	**Description**
`ie`	1 bits	Interrupt Enable	0=disable, 1=enable.
`pce`	1 bits	Pspe Cache Enable	0=disable, 1=enable.
`spce`	1 bits	SP Cache Enable	0=disable, 1=enable.
`msce`	1 bits	Mac Set Cache Enable	0=disable, 1=enable.
`mtce`	1 bits	Mac Tree Cache Enable	0=disable, 1=enable.
`ive`	1 bits	IV Enable	0=disable, 1=enable.
`ivce`	1 bits	IV Cache Enable	0=disable, 1=enable.
`psiz`	3 bits	Protected SIZe	Size of protected memory area: 1=64MB, 2=256MB, 3=1GB, 4=4GB.
`padd`	8 bits	Protected ADDress	Start address of protected memory. Aligned multiple of 16MB. 8 MSBs only.

The status register

The *status register* (`status`, figure 2.5 and table 2.2) is read only. It contains indicators about the current state of the HSM. Reading the status register clears the pending interrupts flag.

Table 2.2: Hardware Security Module `status` register fields

Name	**Width**	**Long name**	**Description**
`busy`	1 bits	BUSY flag	0=idle, 1=busy.
`errt`	3 bits	ERRor Type	If not 0 on HSM interrupt, indicates the type of error: 0=none, 1=PSPE invalid, 2=SP invalid, 3=integrity violation (MAC sets), 4=integrity violation (MAC trees).
`errc`	1 bits	ERRor Cause	Type of access that caused error: 0=read, 1=write.
`erra`	27 bits	ERRor Address	Address of group which access caused an error (27 MSBs).

The master integrity key register

The *master integrity key register* (`mik`, figure 2.6 and table 2.3) is write only and is used at start-up to set the key used to compute the MAC nodes of the MAC trees (Master MAC tree and MAC trees protecting regular memory pages).

Table 2.3: Hardware Security Module `mik` register fields

Name	**Width**	**Long name**	**Description**
`ikey0`	32 bits	Integrity KEY0	32 LSBs of MIK.K.
`ikey1`	24 bits	Integrity KEY1	24 MSBs of MIK.K (most significant byte ignored).
`ikey2`	32 bits	Integrity KEY2	32 LSBs of MIK.K1.
`ikey3`	32 bits	Integrity KEY3	32 MSBs of MIK.K1.
`ikey4`	32 bits	Integrity KEY4	32 LSBs of MIK.K2.
`ikey5`	32 bits	Integrity KEY5	32 MSBs of MIK.K2.

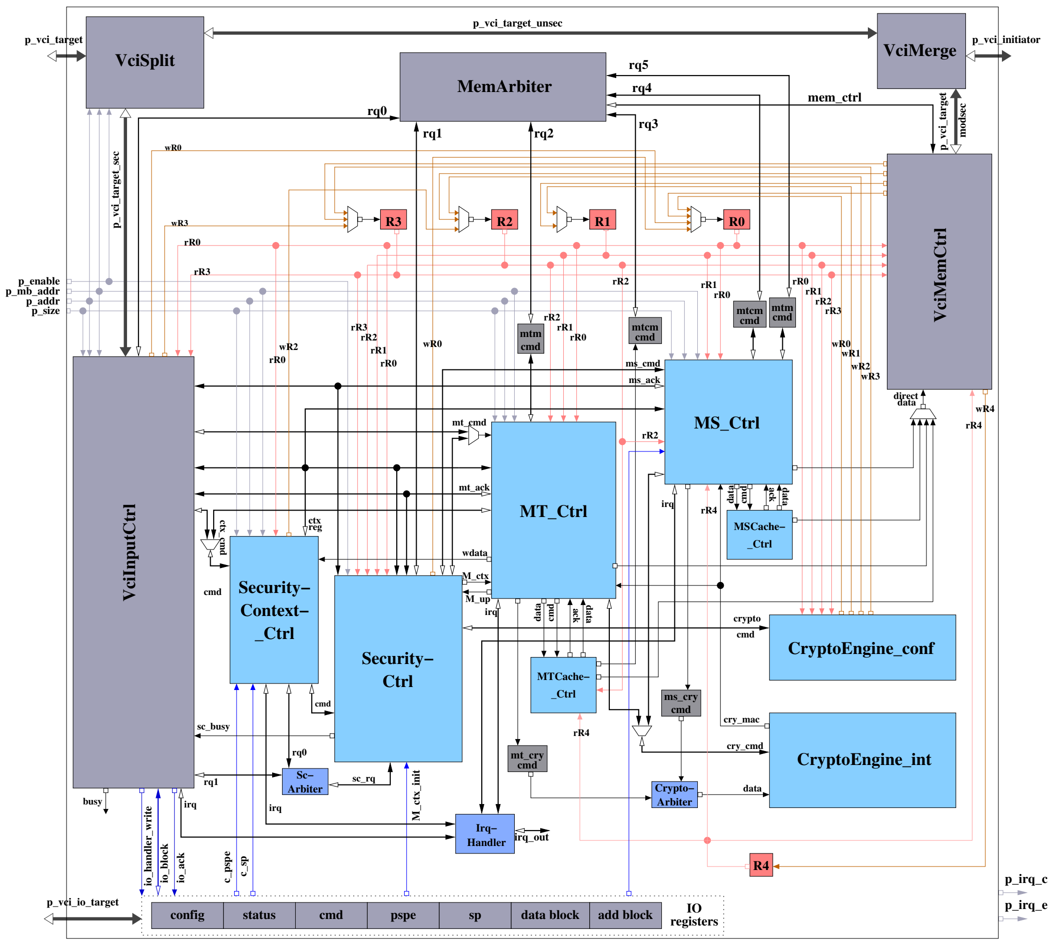


Figure 2.3: The internal architecture of the HSM-mem

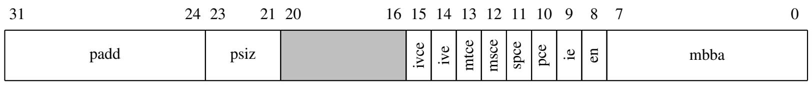


Figure 2.4: Hardware Security Module `cfg` register layout: ConFiGuration register

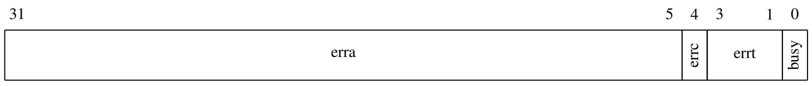


Figure 2.5: Hardware Security Module `status` register layout: STATUS register

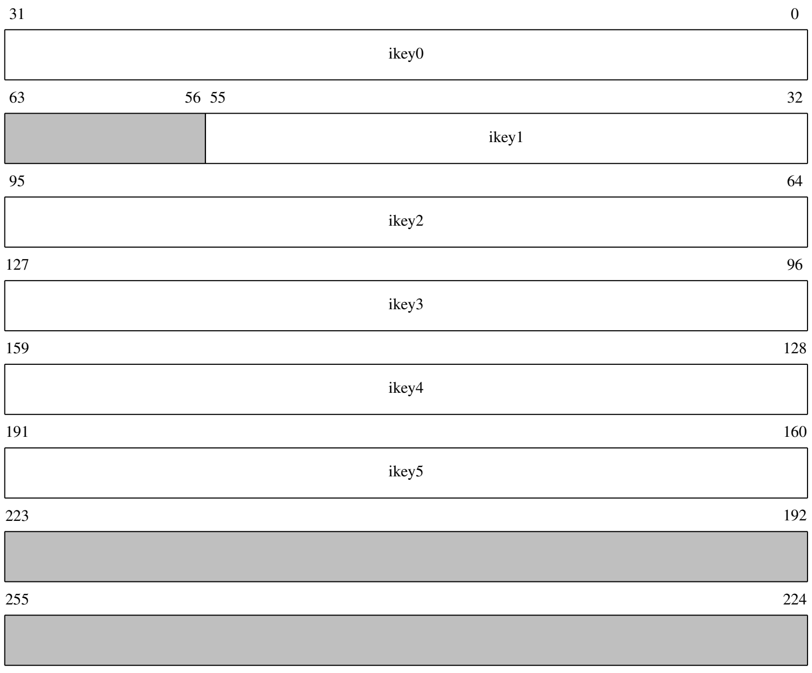


Figure 2.6: Hardware Security Module `mik` register layout: Master Integrity Key

The master confidentiality key register

The *master confidentiality key register* (`mck`, figure 2.7 and table 2.4) is write only and is used at start-up to set the key used to encipher / decipher the Security Policy area of the Master Block.

Table 2.4: Hardware Security Module `mck` register fields

Name	Width	Long name	Description
`ckey0`	32 bits	Confidentiality KEY0	32 LSBs of MCK.K.

Name	**Width**	**Long name**	**Description**
`ckey1`	24 bits	Confidentiality KEY1	24 MSBs of MCK.K (most significant byte ignored).
`ckey2`	32 bits	Confidentiality KEY2	32 LSBs of MCK.K1.
`ckey3`	32 bits	Confidentiality KEY3	32 MSBs of MCK.K1.
`ckey4`	32 bits	Confidentiality KEY4	32 LSBs of MCK.K2.
`ckey5`	32 bits	Confidentiality KEY5	32 MSBs of MCK.K2.

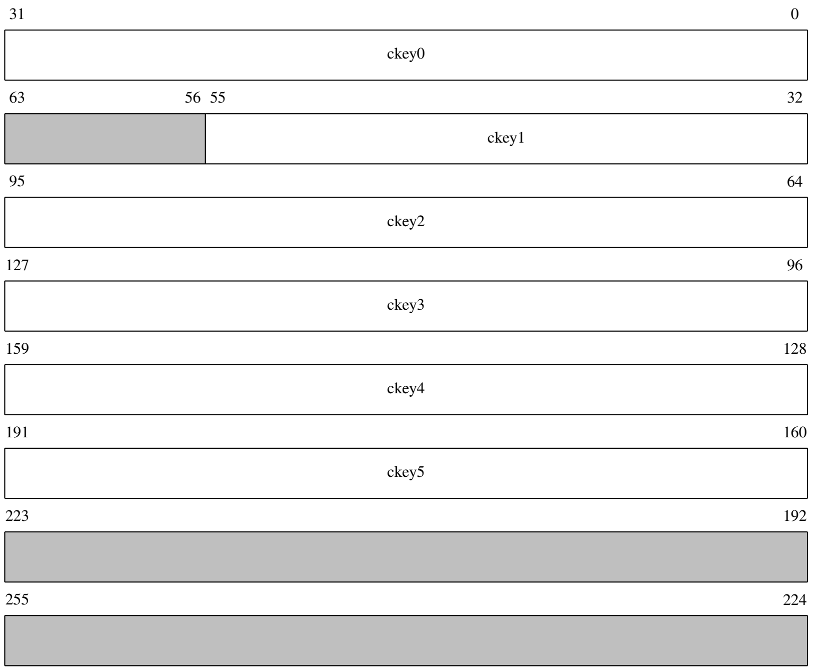


Figure 2.7: Hardware Security Module `mck` register layout: Master Confidentiality Key

The group (or block) atomic read-write operations

The HSM offers atomic operations to securely access an aligned 64-bits double word or an aligned 256-bits group in external memory. The 256-bits atomic accesses are required for proper initialization of read-only memory pages protected by the block cipher in counter mode (confidentiality) and / or MAC sets (integrity). They are the only way to guarantee the write-once property[1]. The atomic accesses are also used to efficiently access PSPEs (64-bits) and SPs (2 $\times$ 256-bits). Atomic accesses in the PSPE area of the Master Block are always 64-bits. Accesses elsewhere in memory are always 256-bits. Requesting an atomic access is done by setting a set of interface registers (see below); writing the `agrwcmd` register launches the access (and must thus be the last register setting of a request). Upon read accesses the read 64 or 256 bits are retrieved from the `agrwdata` register. When the HSM performs the requested atomic access it automatically applies the defined Security Policy, based of the target address, as for regular load-store operations. Note: regular load-store accesses in the Master Block are forbidden. Accessing the Master Block must absolutely be done through the atomic operations.

The same set of registers is also used to initialize the MAC tree of a newly allocated read-write memory page that must be integrity-protected. The only relevant parameter for the MAC tree initialization is the byte base address of the protected regular page. The associated PSPE and SP provide all the other parameters. Two different commands are dedicated to this MAC tree initialization:

- If the MAC tree to initialize is the first of its page of MAC trees, the topmost levels of the other MAC trees in the same page of MAC trees are not verified when computing the root MAC of the page of MAC trees.

[1]If the initial write of the 256-bits group was not atomic, it could lead to multiple enciphering and / or MAC computations with a partly initialized group.

- If the page of MAC trees already contains initialized MAC trees, the topmost levels of the other MAC trees in the same page of MAC trees are verified when computing the root MAC of the page of MAC trees.

The atomic group read-write address register

The *atomic group read-write address register* (`agrwadd`, figure 2.8 and table 2.5) is used to set the byte address of the 64-bits double word or 256-bits group to access atomically.

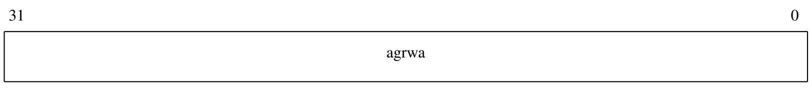


Figure 2.8: Hardware Security Module `agrwadd` register layout: Atomic Group Read-Write ADDress

Table 2.5: Hardware Security Module `agrwadd` register fields

Name	Width	Long name	Description
`agrwa`	32 bits	Atomic Group Read-Write Address	Group's or block's byte address for atomic group read-write operations. Aligned on group's or block's boundary: LSBs are ignored. Block atomic access if address falls in PSPEs, else group atomic access.

The atomic group read-write data register

The *atomic group read-write data register* (`agrwdata`, figure 2.9 and table 2.6) is used to store the data to write or to retrieve the read data of an atomic access. Upon 64-bits accesses (PSPEs), only one quarter of this 256-bits register is used and the quarter used depends on the alignment of the 64-bits double word in the 256-bits group.

Table 2.6: Hardware Security Module `agrwdata` register fields

Name	Width	Long name	Description
`data0`	32 bits	DATA0	Read data or data to write (lowest address in memory).
`data1`	32 bits	DATA1	Read data or data to write.
`data2`	32 bits	DATA2	Read data or data to write.
`data3`	32 bits	DATA3	Read data or data to write.
`data4`	32 bits	DATA4	Read data or data to write.
`data5`	32 bits	DATA5	Read data or data to write.
`data6`	32 bits	DATA6	Read data or data to write.
`data7`	32 bits	DATA7	Read data or data to write (highest address in memory).

The atomic group read-write command register

The *atomic group read-write command register* (`agrwcmd`, figure 2.10 and table 2.7) is used to set the requested command:

- read (of a 64-bits PSPE or a 256-bits group),

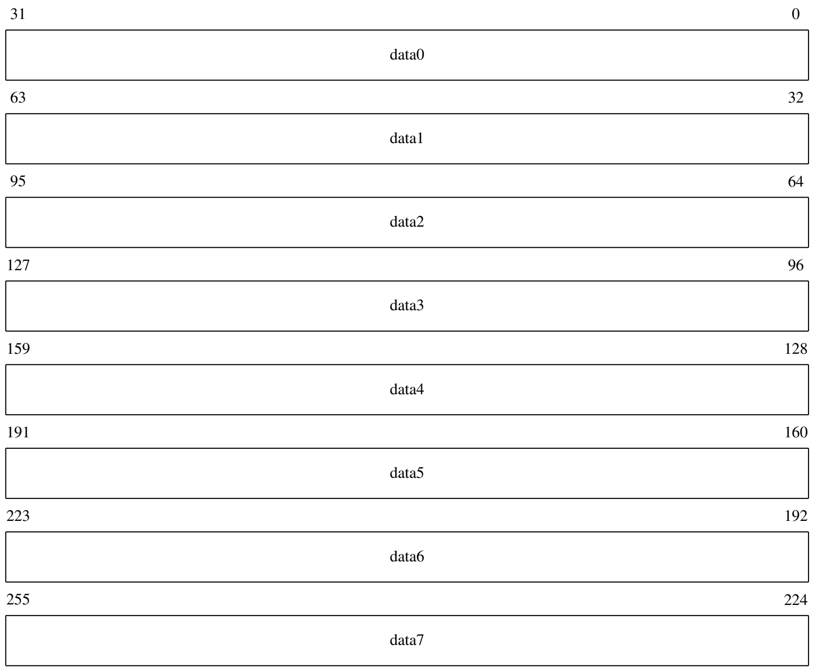


Figure 2.9: Hardware Security Module `agrwdata` register layout: Atomic Group Read-Write DATA

- write (of a 64-bits PSPE or a 256-bits group).
- initialize first MAC tree of a page of MAC trees
- initialize a MAC tree that is not the first of its page of MAC trees

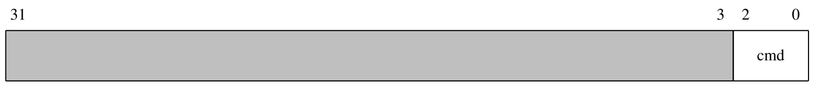


Figure 2.10: Hardware Security Module `agrwcmd` register layout: Atomic Group Read-Write CoMmanD

Table 2.7: Hardware Security Module `agrwcmd` register fields

Name	**Width**	**Long name**	**Description**
`cmd`	3 bits	Atomic Group Read-Write CoMmanD	0: none, 1: read, 2: write, 3: init, 4: continue. HSM applies SP defined for target group or block.

3 ORGANIZATION AND CONTENT OF THE ARCHIVE

The archive (as version 0.1) is organized as follow:

- `COPYING` and `COPYING-FR`: These two files contain the license (in English and in French) under which the source code of the HSM-Mem is distributed. The CeCILL version 2.1, a free and open-source software license (similar to the well-known GPL) was chosen.
- `Makefile`: This is the main makefile to launch the tests or the synthesis of the different parts of the HSM-Mem
- `scripts`: This directory contains all the scripts used to launch simulations, tests, synthesis...
- `bitfields`: This directory contains the definition of the different data structure (SP, PSPE, configuration and status registers...).
- `src`: This directory contains the VHDL source files of the HSM-Mem and its submodules:
  - `arbiters`: This directory contains the code of the different arbiters (example: the module `MemArbiter` (file `mem_arbiter.vhd`) is in charge of arbiter the access to the module `VciMemCtrl`).
  - `axi_bridge`: This directory contains the package axi_bridge with the definitions of the AXI interfaces used by the HSM.
  - `axi_secbus_bridge`: This directory contains the AXI SecBus bridge module (the HSM-Mem with its AXI interfaces for the Zedboard with some test features) and the synthesis script for Vivado.
  - `axi_vci`: This directory contains the AXI-to-VCI and VCI-to-AXI bridges.
  - `bc`: This directory contains the block cipher (DES-X) and its modes of operation.
  - `caches`: This directory contains the different caches used in the HSM-Mem (MS, MT, PSPE, SP).
  - `crypto`: This directory contains the modules `CryptoEngine_conf` (cryptographic engine for confidentiality) and `CryptoEngine_int` (cryptographic engine for integrity).
  - `des`: This directory contains the package DES with all the constants and functions used by the DES-X algorithm.
  - `fifo`: This directory contains a simple FIFO module.
  - `global`: This directory contains packages with structures, interfaces and functions used by the other modules.
  - `io_input`: This directory contains the module `IOInputCtrl` that responds to commands (Load, Store, Init, InitPage) sent via the IO registers of the HSM-Mem.
  - `mem_ctrl`: This directory contains the module `MemoryCtrl` that handles the read-write requests from the different HSM modules to the external memory.

- `ms_ctrl`: This directory contains the module `MS_Ctrl` which manages the integrity protection and verification using MAC sets.
- `mt_ctrl`: This directory contains the module `MT_Ctrl` which manages the integrity protection and verification using MAC Trees.
- `random`: This directory contains a random number generator for testing purpose.
- `register`: This directory contains the module `reg_data` that encapsulates the behavior of the internal registers of the HSM-Mem.
- `sec_ctrl`: This directory contains the module `Security_Ctrl` which is one of the main sub-modules of the HSM-Mem. It manages the read-write accesses to the protected region of the external memory, including the Master Block.
- `sec_ctx`: This directory contains the module `Security_Context_Ctrl` which manages the security contexts associated with memory pages, that is PSPEs and SPs.
- `vci`: This directory contains a VCI pattern generator for testing purpose.
- `vci_input`: This directory contains the module `VciInputCtrl` which handles read-write requests from the processor to/from the protected memory area.
- `vci_merge`: This directory contains the module `VciMerge` which multiplexes requests from `VciSplit` and `MemoryCtrl` to the memory controller.
- `vci_secbus`: This directory contains the top modules `vci_secbus` and `axi_secbus` (the full HSM-Mem module with VCI or AXI interfaces). It also contains test patterns (`axi_ini_in.txt`, `axi_tgt_in.txt`, `vci_ini_in.txt`, `vci_tgt_in.txt`) used to validate the HSM-Mem.
- `vci_io_target`: This directory contains the module `vci_io_target` which implements the VCI IO target of the HSM-Mem and manages the IO registers.
- `vci_ram`: This directory contains a RAM model used in several tests.
- `vci_split`: This directory contains the module `VciSplit` which receives VCI requests through its target interface, checks whether they fall in the protected region of the external memory and, depending on the check, routes them through one or the other of its two VCI initiator interfaces.

4 USE OF THE ARCHIVE

This section describes how to use the content of the archive.

4.1 Tests

Two sets of regression tests can be launched using the Makefile provided in the archive:

- the target `ms-tests` launches the compilation regression tests that verifies whether all design units of all modules compiler without error;
- the target `ms-sim-tests` launches the simulation regression tests.

These tests require *Modelsim* from *Mentor Graphics* (tested with Modelsim SE-64 version 10.4 on Linux).

4.1.1 Compilation regression tests

```
secbus-0.1 % make ms-tests
Modelsim compilation non-regression test:
make[1]: Entering directory ’/scratch/secbus-0.1/src/vci_ram’
Modelsim compilation non-regression test:
  vci_ram: OK
  ram: OK
  axi_ram: OK
make[1]: Leaving directory ’/scratch/secbus-0.1/src/vci_ram’
make[1]: Entering directory ’/scratch/secbus-0.1/src/crypto’
Modelsim compilation non-regression test:
  cryptoConf: OK
  cryptoConf_sim: OK
  cryptoInt: OK
  cryptoInt_sim: OK
make[1]: Leaving directory ’/scratch/secbus-0.1/src/crypto’
make[1]: Entering directory ’/scratch/secbus-0.1/src/bc’
Modelsim compilation non-regression test:
  bc: OK
  bc_sim: OK
  bc_sim_pkg: OK
  desx: OK
make[1]: Leaving directory ’/scratch/secbus-0.1/src/bc’
make[1]: Entering directory ’/scratch/secbus-0.1/src/sec_ctrl’
Modelsim compilation non-regression test:
  security_ctrl: OK
  security_ctrl_sim: OK
make[1]: Leaving directory ’/scratch/secbus-0.1/src/sec_ctrl’
make[1]: Entering directory ’/scratch/secbus-0.1/src/caches’
Modelsim compilation non-regression test:
  mt_cache: OK
  sp_cache: OK
  sp_cache_sim: OK
  ms_cache: OK
  ms_cache_sim: OK
  rnd_cache_gen: OK
  ram: OK
  pspe_cache: OK
  pspe_cache_sim: OK
make[1]: Leaving directory ’/scratch/secbus-0.1/src/caches’
make[1]: Entering directory ’/scratch/secbus-0.1/src/vci_input’
Modelsim compilation non-regression test:
  rnd_vci_initiator: OK
  rnd_ctx_gen: OK
  vci_input_ctrl_sim: OK
  vci_input_ctrl: OK
  rnd_sec_gen: OK
make[1]: Leaving directory ’/scratch/secbus-0.1/src/vci_input’
make[1]: Entering directory ’/scratch/secbus-0.1/src/axi_bridge’
Modelsim compilation non-regression test:
  axi_bridge_pkg: OK
```

```
make[1]: Leaving directory '/scratch/secbus-0.1/src/axi_bridge'
make[1]: Entering directory '/scratch/secbus-0.1/src/axi_secbus_bridge'
Modelsim compilation non-regression test:
  axi_secbus_bridge: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/axi_secbus_bridge'
make[1]: Entering directory '/scratch/secbus-0.1/src/ms_ctrl'
Modelsim compilation non-regression test:
  sr_ff: OK
  ms_ctrl: OK
  ms_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/ms_ctrl'
make[1]: Entering directory '/scratch/secbus-0.1/src/axi_vci'
Modelsim compilation non-regression test:
  axi4lite_2_vci: OK
  axi4_2_vci: OK
  vci_2_axi4: OK
  axilite_vci_sim: OK
  axi_vci_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/axi_vci'
make[1]: Entering directory '/scratch/secbus-0.1/src/global'
Modelsim compilation non-regression test:
  global: OK
  utils: OK
  numeric_std: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/global'
make[1]: Entering directory '/scratch/secbus-0.1/src/register'
Modelsim compilation non-regression test:
  reg_data: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/register'
make[1]: Entering directory '/scratch/secbus-0.1/src/mt_ctrl'
Modelsim compilation non-regression test:
  mt_ctrl: OK
  mt_cache_ctrl_sim: OK
  mt_cache_ctrl: OK
  mt_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/mt_ctrl'
make[1]: Entering directory '/scratch/secbus-0.1/src/io_input'
Modelsim compilation non-regression test:
  io_input_ctrl: OK
  io_input_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/io_input'
make[1]: Entering directory '/scratch/secbus-0.1/src/random'
Modelsim compilation non-regression test:
  rnd: OK
  random_pkg: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/random'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_io_target'
Modelsim compilation non-regression test:
  vci_io_target: OK
  vci_io_target_sim: OK
  rnd_io_handler_tgt: OK
  rnd_vci_io_init: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_io_target'
make[1]: Entering directory '/scratch/secbus-0.1/src/fifo'
Modelsim compilation non-regression test:
  fifo: OK
  fifo_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/fifo'
make[1]: Entering directory '/scratch/secbus-0.1/src/arbiters'
Modelsim compilation non-regression test:
  direct_data_arbiter: OK
  crypto_int_arbiter: OK
  crypto_int_arbiter_sim: OK
  mt_arbiter: OK
  reg_arbiter: OK
  mem_arbiter: OK
  sc_arbiter: OK
  irq_arbiter: OK
  ctx_arbiter: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/arbiters'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_merge'
Modelsim compilation non-regression test:
  vci_merge: OK
  vci_merge_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_merge'
make[1]: Entering directory '/scratch/secbus-0.1/src/sec_ctx'
Modelsim compilation non-regression test:
  security_ctx_ctrl: OK
  security_ctx_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/sec_ctx'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_split'
Modelsim compilation non-regression test:
  vci_split: OK
  vci_split_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_split'
make[1]: Entering directory '/scratch/secbus-0.1/src/des'
Modelsim compilation non-regression test:
  des_pkg: OK
  des_pkg_sim: OK
```

```
make[1]: Leaving directory '/scratch/secbus-0.1/src/des'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci'
Modelsim compilation non-regression test:
  rnd_vci_initiator: OK
  vci_pack: OK
  rnd_vci_target: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_secbus'
Modelsim compilation non-regression test:
  axi_secbus_wrapper: OK
  axi_secbus_sim: OK
  vci_secbus: OK
  vci_secbus_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_secbus'
make[1]: Entering directory '/scratch/secbus-0.1/src/mem_ctrl'
Modelsim compilation non-regression test:
  rnd_mem_gen: OK
  vci_mem_ctrl: OK
  vci_mem_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/mem_ctrl'
```

4.1.2 Simulation regression tests

The archive contains unit regression tests for all the important submodules of the HSM-Mem (files `*_sim.vhd`). It also contains tests for the HSM-Mem module itself (both VCI and AXI versions).

The tests of the VCI version of the HSM-Mem are based on VCI transactions recorded using the virtual platform and the HSM-Mem SystemC model. These transactions are provided to the VHDL implementation of the HSM-Mem and the test environment verifies that it behaves as expected.

```
secbus-0.1 % make ms-sim-tests
Modelsim simulation non-regression test:
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_ram'
Modelsim simulation non-regression test:
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_ram'
make[1]: Entering directory '/scratch/secbus-0.1/src/crypto'
Modelsim simulation non-regression test:
  cryptoConf_sim: OK
  cryptoInt_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/crypto'
make[1]: Entering directory '/scratch/secbus-0.1/src/bc'
Modelsim simulation non-regression test:
  bc_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/bc'
make[1]: Entering directory '/scratch/secbus-0.1/src/sec_ctrl'
Modelsim simulation non-regression test:
  security_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/sec_ctrl'
make[1]: Entering directory '/scratch/secbus-0.1/src/caches'
Modelsim simulation non-regression test:
  pspe_cache_sim: OK
  sp_cache_sim: OK
  ms_cache_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/caches'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_input'
Modelsim simulation non-regression test:
  vci_input_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_input'
make[1]: Entering directory '/scratch/secbus-0.1/src/axi_bridge'
Modelsim simulation non-regression test:
make[1]: Leaving directory '/scratch/secbus-0.1/src/axi_bridge'
make[1]: Entering directory '/scratch/secbus-0.1/src/axi_secbus_bridge'
Modelsim simulation non-regression test:
make[1]: Leaving directory '/scratch/secbus-0.1/src/axi_secbus_bridge'
make[1]: Entering directory '/scratch/secbus-0.1/src/ms_ctrl'
Modelsim simulation non-regression test:
  ms_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/ms_ctrl'
make[1]: Entering directory '/scratch/secbus-0.1/src/axi_vci'
Modelsim simulation non-regression test:
  axi_vci_sim: OK
  axilite_vci_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/axi_vci'
make[1]: Entering directory '/scratch/secbus-0.1/src/global'
Modelsim simulation non-regression test:
make[1]: Leaving directory '/scratch/secbus-0.1/src/global'
make[1]: Entering directory '/scratch/secbus-0.1/src/register'
Modelsim simulation non-regression test:
make[1]: Leaving directory '/scratch/secbus-0.1/src/register'
make[1]: Entering directory '/scratch/secbus-0.1/src/mt_ctrl'
Modelsim simulation non-regression test:
  mt_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/mt_ctrl'
```

```
make[1]: Entering directory '/scratch/secbus-0.1/src/io_input'
Modelsim simulation non-regression test:
  io_input_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/io_input'
make[1]: Entering directory '/scratch/secbus-0.1/src/random'
Modelsim simulation non-regression test:
make[1]: Leaving directory '/scratch/secbus-0.1/src/random'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_io_target'
Modelsim simulation non-regression test:
  vci_io_target_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_io_target'
make[1]: Entering directory '/scratch/secbus-0.1/src/fifo'
Modelsim simulation non-regression test:
  fifo_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/fifo'
make[1]: Entering directory '/scratch/secbus-0.1/src/arbiters'
Modelsim simulation non-regression test:
  crypto_int_arbiter_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/arbiters'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_merge'
Modelsim simulation non-regression test:
  vci_merge_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_merge'
make[1]: Entering directory '/scratch/secbus-0.1/src/sec_ctx'
Modelsim simulation non-regression test:
  security_ctx_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/sec_ctx'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_split'
Modelsim simulation non-regression test:
  vci_split_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_split'
make[1]: Entering directory '/scratch/secbus-0.1/src/des'
Modelsim simulation non-regression test:
  des_pkg_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/des'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci'
Modelsim simulation non-regression test:
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci'
make[1]: Entering directory '/scratch/secbus-0.1/src/vci_secbus'
Modelsim simulation non-regression test:
  vci_secbus_sim: OK
  axi_secbus_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/vci_secbus'
make[1]: Entering directory '/scratch/secbus-0.1/src/mem_ctrl'
Modelsim simulation non-regression test:
  vci_mem_ctrl_sim: OK
make[1]: Leaving directory '/scratch/secbus-0.1/src/mem_ctrl'
```

4.2 Synthesis

The HSM-Mem can be synthesized for the ZedBoard using *Xilinx Vivado*. Software stack (including the Software Security Module) and demonstration applications will be provided as part of WP4.

The synthesis can be launched with the command `make axi_secbus_bridge.vsyn` inside the directory `src/axi_secbus_bridge`. It has been tested with *Vivado* version v2014.4 64-bit on Linux.

5 CONCLUSION

This deliverable (D2.4) contains the VHDL code of the HSM-Mem and the simulation and synthesis environment. This hardware component requires a software driver (the Software Security Module) that has been developed in WP3 (deliverable D3.1).

Demonstrations of a full system, including the HSM-Mem, are being developed in WP4.

BIBLIOGRAPHY

[1] Xilinx all programmable socs: `http://www.xilinx.com/products/silicon-devices/soc.html`.

[2] Digilent. ZedBoard: http://zedboard.org/product/zedboard. `http://zedboard.org/product/zedboard`.