Trusted Communication Unit – Specification

Version 2.0.0

Nils Asmussen

October 15, 2024

Contents

Chapter 1

System Overview

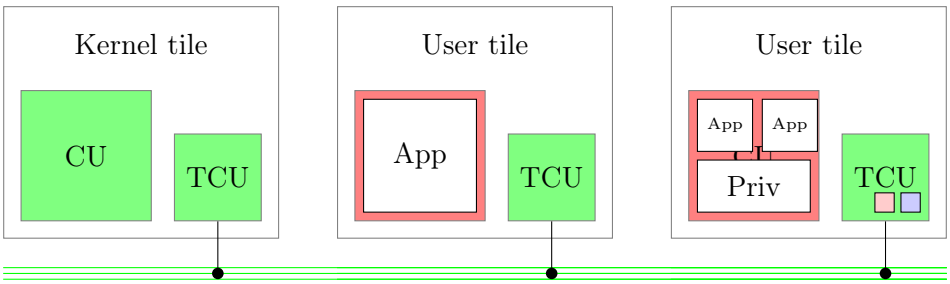


Figure 1.1: System Overview.

The trusted communication unit (TCU) is a building block that can be used to construct secure systems. As shown in Figure 1.1, the system is based upon a tiled architecture and each tile contains a compute unit (CU) and a TCU. The tiles are linked through some interconnect (e.g., a network-on-chip) and are split into kernel tiles and user tiles. The former are privileged and manage the TCUs of the user tiles, whereas the latter are unprivileged.

In this system, the TCU, the interconnect, and the kernel tiles are trusted (shown green), whereas the CUs in the user tiles are untrusted (shown red). By default, all tiles are isolated from each other, but communication channels between tiles can be established. These communication channels can only be establish by kernel tiles, but can be used afterwards by user tiles. Which tiles are kernel tiles is defined by the TCU's FEATURES register. At boot, all tiles are kernel tiles, which can be changed by the kernel booting on one specific tile before the other tiles start.

User tiles can have different complexities, mostly driven by the used CU. The left user tile uses a simple core with scratchpad memory and a basic TCU without extensions. The right user tile uses a complex core with caches and virtual memory support and a privileged software (Priv) that enables multiple applications on the same tile and/or virtual memory. These features require TCU extensions for virtual memory ( ) and/or tile sharing ( ). This specification highlights the portions that are required by only one extension in its

color and portions that are required by either extension as ■.

1.1 Tile Internals

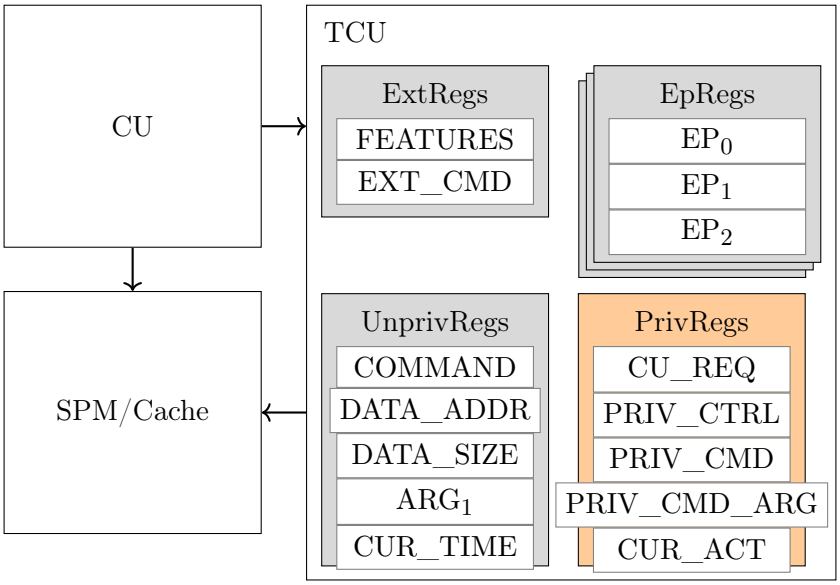


Figure 1.2: The internal organization of a tile.

Figure 1.2 shows the internals of a tile. The compute unit (CU) is connected to the trusted communication unit (TCU) and can access the TCU's registers via memory mapped input/output (MMIO). Additionally, the CU is connected to the local memory. The TCU is also connected to the local memory (SPM or cache) to, for example, access messages. These components are not necessarily arranged in this way. For example, the TCU might interpose itself between the CU and local memory.

To support the security model introduced in Chapter 1, the registers are split into different groups, each group having different access permissions. All registers are generally readable. External registers can only be written externally, that is, from a remote tile. They are intended for the kernel tile to manipulate the state of remote TCUs. Endpoint registers can be written externally and locally by the kernel tile. Unprivileged registers can be written by the CU. In consequence, only kernel tiles can establish communication channels by writing endpoint registers, but user tiles can use these communication channels through the unprivileged registers.

The privileged registers are intended for privileged software running on tiles that are multiplexed among multiple activities and/or use virtual memory. The privileged software should make sure that unprivileged software running on the same tile cannot access the privileged registers.

1.2 Registers

The TCU has several registers that are accessible through memory-mapped input/output (MMIO). The memory interface from CU to TCU is expected to be 64-bit wide. The MMIO region of the TCU is defined as follows:

Address	Register	Group	Description
0xF000_0000	FEATURES	ExtRegs	Contains feature flags
0xF000_0008	TILE_DESC	ExtRegs	Contains the tile description
0xF000_0010	EXT_CMD	ExtRegs	Triggers external commands
0xF000_0018	COMMAND	UnprivRegs	Triggers unprivileged commands
0xF000_0020	DATA_ADDR	UnprivRegs	Specifies the data address for commands
0xF000_0028	DATA_SIZE	UnprivRegs	Specifies the data size for commands
0xF000_0030	ARG_1	UnprivRegs	Additional argument for commands
0xF000_0038	CUR_TIME	UnprivRegs	Yields the current time in nanoseconds
0xF000_0040	PRINT	UnprivRegs	Triggers a debug print
0xF000_0048	$EP0_0$	EpRegs	First register of EP0
0xF000_0050	$EP0_1$	EpRegs	Second register of EP0
0xF000_0058	$EP0_2$	EpRegs	Third register of EP0
0xF000_0060	$EP1_0$	EpRegs	First register of EP1
0xF000_0068	$EP1_1$	EpRegs	Second register of EP1
0xF000_0070	$EP1_2$	EpRegs	Third register of EP1
		...	
0xF000_0630	$EP63_0$	EpRegs	First register of EP63
0xF000_0638	$EP63_1$	EpRegs	Second register of EP63
0xF000_0640	$EP63_2$	EpRegs	Third register of EP63
0xF000_0648	PR0	PrintRegs	First print register
0xF000_0650	PR1	PrintRegs	Second print register
		...	
0xF000_0740	PR31	PrintRegs	Thirty-second print register
0xF000_2000	CU_REQ	PrivRegs	TCU-CU interactions
0xF000_2008	PRIV_CTRL	PrivRegs	Configures the privileged interface
0xF000_2010	PRIV_CMD	PrivRegs	Triggers privileged commands
0xF000_2018	PRIV_CMD_ARG	PrivRegs	Argument for privileged commands
0xF000_2020	CUR_ACT	PrivRegs	Currently running activity

1.3 Error List

The TCU uses the following error codes to provide feedback to the software for failed commands:

- NONE (0): no error (success),

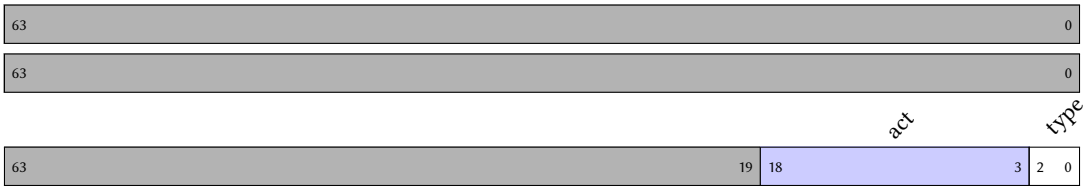
- NO_MEP (1): no memory endpoint,
- NO_SEP (2): no send endpoint,
- NO_REP (3): no receive endpoint,
- FOREIGN_EP (4): the endpoint belongs to a different activity,
- SEND_REPLY_EP (5): SEND command with a reply EP or REPLY with send EP,
- RECV_GONE (6): receiver gone (received message at non-receive EP),
- RECV_NO_SPACE (7): receiver buffer full,
- REPLIES_DISABLED (8): replies are disabled,
- OUT_OF_BOUNDS (9): the offset and/or size is out of bounds,
- NO_CREDITS (10): no credits to send a message,
- NO_PERM (11): insufficient permissions,
- INV_MSG_OFF (12): invalid message offset,
- TRANSLATION_FAULT (13): translation of address for data transfer failed,
- ABORT (14): a command was aborted,
- UNKNOWN_CMD (15): unknown command,
- RECV_OUT_OF_BOUNDS (16): message too large for receive buffer,
- RECV_INV_RPL_EPS (17): invalid reply EPs in receive EP,
- SEND_INV_CRD_EP (18): invalid credit EP in send EP,
- SEND_INV_MSG_SZ (19): invalid value in msg_sz field (> 11),
- TIMEOUT_MEM (20): timeout while waiting for memory response,
- TIMEOUT_NOC (21): timeout while waiting for NoC response,
- PAGE_BOUNDARY (22): data to transfer contains a page boundary,
- MSG_UNALIGNED (23): message to send is not 16-byte aligned,
- TLB_MISS (24): entry in TLB not found,
- TLB_FULL (25): TLB contains only fixed entries.
- NO_PMP_EP (26): no PMP endpoint,

Chapter 2

Endpoints

The TCU has a number of endpoints (EPs) to establish communication channels, which can be configured to three different EP types: send EPs and receive EPs are used for message passing, whereas memory EPs are used for RDMA-like memory access. Each EP is represented by a TCU register and can be configured (at runtime) to one of these EP types. Each EP consists of 192 bits, starting with 3 bits for the endpoint type (T), 16 bits for the activity that can use the EP and 173 bits (shown as dark grey below), whose meaning depends on the EP type:

Register 2.1: Endpoint

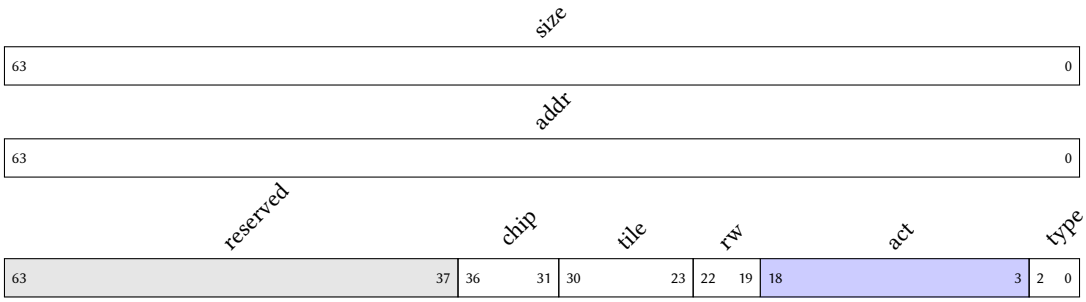


act the id of the activity that can use this endpoint

type the endpoint type: INVALID (0), SEND (1), RECEIVE (2), or MEMORY (3)

2.1 Memory Endpoint

Register 2.2: Memory EP



size the size of the region at the destination

addr the base address of the region at the destination

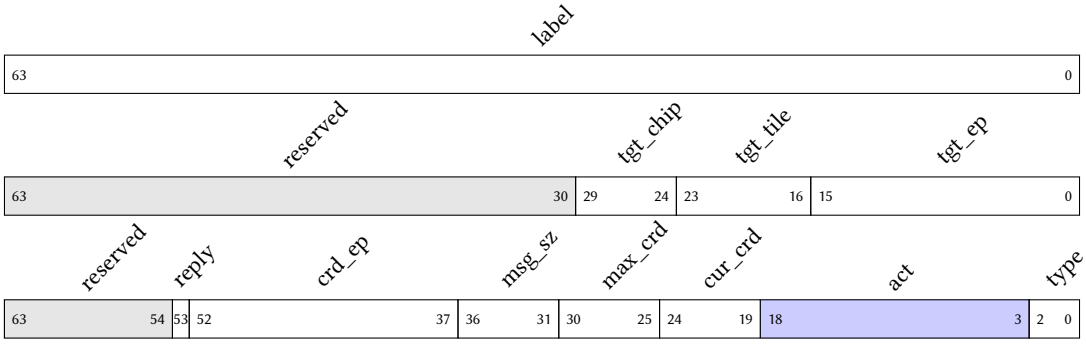
chip the destination chip ID

tile the destination tile ID

rw the permission bits (read = 1, write = 2)

2.2 Send Endpoint

Register 2.3: Send EP



label the label the TCU puts into the header of each sent message

tgt_chip the ID of the target chip

tgt_tile the ID of the target tile

tgt_ep the ID of the receive EP at the target tile

reply whether this is a reply EP

crd_ep for reply EPs: the send EP at sender-side to receive credits

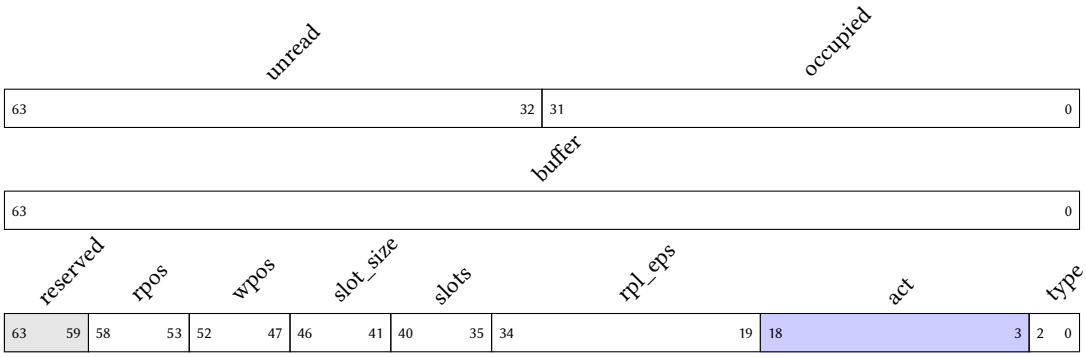
max_crd the initially received (=max) credits (in messages)

cur_crd the currently owned credits (in messages)

msg_sz the maximum message size supported by the receiver (as power of 2)

2.3 Receive Endpoint

Register 2.4: Receive EP



unread a bitmask with the unread (not yet fetched) messages in the buffer

occupied a bitmask with the occupied slots in the buffer

buffer the physical address of the receive buffer, must be 8-byte aligned

rpos the read position (for message fetches) within the receive buffer

wpos the write position (for message receptions) within the receive buffer

slot_size the size of one slot as a power of 2

slots the number of slots in the receive buffer as a power of 2

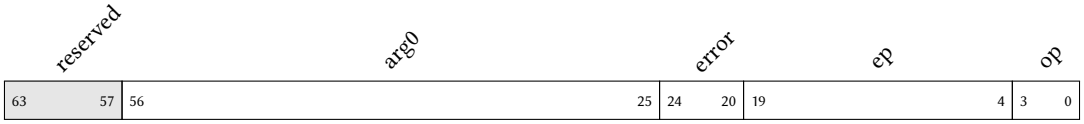
rpl_eps the offset of the reply EPs

Chapter 3

Unprivileged Interface

The unprivileged interface of the TCU is available to unprivileged software. Most importantly, it allows to use the TCU's endpoints via unprivileged commands. The unprivileged registers are used to pass input arguments for a command to the TCU, start a command, and wait until the command is finished. The following unprivileged registers are used:

Register 3.1: COMMAND (0xF000_0018)



arg0 the first argument for the command

error the error code (0 = no error)

ep the endpoint to use for the command

op the opcode of the command

A write to the COMMAND register starts the command with opcode COMMAND.op and the DATA_ADDR and DATA_SIZE registers specify the source or destination data in local memory. The meaning of the two arguments (COMMAND.arg0 and ARG1) depends on the opcode.

3.1 Command List

The TCU supports the following unprivileged commands with the opcodes in parentheses. Other opcodes lead to an UNKNOWN_CMD error.

- IDLE (0): don't do anything,
- SEND (1): send a message via a send EP to a receive EP,

- REPLY (2): reply on a message that has been received earlier via a receive EP,
- READ (3): read data from a region defined in a memory EP into local memory,
- WRITE (4): write data from local memory into a region defined in a memory EP,
- FETCH (5): fetch a new message from a receive EP,
- ACK_MSG (6): acknowledge that the processing of a message has been completed.

3.2 Pseudo Code Building Blocks

The following sections use pseudo code to describe the behavior of the TCU commands, based on several building blocks:

- read_ep(id) -> EP:
  read the TCU-internal EP register with the given id. If the id is out of bounds, an invalid EP is returned.
- write_ep(id, EP):
  write EP to the TCU-internal EP register with given id (assumed to be within bounds)
- read_local(phys, size) -> (data, Error):
  read size bytes from given physical address in local memory into data and return the error code (0 = success)
- write_local(data, phys, size) -> Error:
  write data of size bytes to given physical address in local memory and return the error code (0 = success)
- read_remote(chip, tile, phys, size) -> (data, Error):
  read size bytes from the given tile on given chip at given the physical address into data and return the error code (0 = success)
- write_remote(data, chip, tile, phys, size) -> Error:
  write size bytes from data to the given physical address in the given tile on given chip and return the error code (0 = success)
- send_msg(msg, chip, tile, ep):
  send msg to endpoint ep at given tile on given chip
- send_ack(error):
  send ACK to the sending TCU and report given error back
- wait_for_ack() -> Error:
  wait for the ACK the receiving TCU sends upon successfully storing the message into the receive buffer or an error occurred

- find_slot(mask, pos, slots, val) -> idx:
  searches for a bit with value val in the given mask between bit 0 and bit (1 << slots) – 1, starting at pos and rotating around. The function returns the position of the bit or –1 if none was found.

- unpriv_stop(error):
  stop the execution of the unprivileged command by setting the opcode in COMMAND to 0 and the error code to the given error.

- queue_foreign_msg_req(ep, act):
  append a new foreign-message request to the queue of CU requests (see algorithm 16 for more details).

- lookup_tlb(act, virt, perm) -> (Phys, Error):
  lookup the given virtual address in the TLB for given activity and return the physical address. On misses or missing permissions, the TLB_MISS or NO_PERM error is returned, respectively. perm is either READ or WRITE and denotes what permission bit needs to be present in the TLB entry. Note that the virtual address is not page aligned and the page offset should be added to the resulting physical address, depending on the page size set in the TLB entry.

3.3 Memory Access

Memory access is performed with a memory EP based on the commands READ and WRITE. The commands behave as follows:

3.3.1 READ

Algorithm 1: The TCU's READ command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.type != MEMORY then
3  |  unpriv_stop(NO_MEP)
4  if ep.act != CUR_ACT.id then
5  |  unpriv_stop(FOREIGN_EP)
6  if ep.rw & READ == 0 then
7  |  unpriv_stop(NO_PERM)
8  if DATA_SIZE == 0 then
9  |  unpriv_stop(NONE)
10 if DATA_SIZE + ARG_1 > ep.size then
11 |  unpriv_stop(OUT_OF_BOUNDS)
12 if (DATA_ADDR & 0xFFF) + DATA_SIZE > 0x1000 then
13 |  unpriv_stop(PAGE_BOUNDARY)
14 phys ← DATA_ADDR;
15 (phys, err) ← lookup_tlb( CUR_ACT.id, DATA_ADDR, WRITE);
16 if err != 0 then
17 |  unpriv_stop(TRANSLATION_FAULT)
18 (data, err) ← read_remote(ep.chip, ep.tile, ep.addr + ARG_1, DATA_SIZE);
19 if err != 0 then
20 |  unpriv_stop(ABORT)
21 err ← write_local(data, phys, DATA_SIZE);
22 if err != 0 then
23 |  unpriv_stop(ABORT)
24 COMMAND ← 0;
```

3.3.2 WRITE

Algorithm 2: The TCU's WRITE command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.type != MEMORY then
3  |  unpriv_stop(NO_MEP)
4  if ep.act != CUR_ACT.id then
5  |  unpriv_stop(FOREIGN_EP)
6  if ep.rw & WRITE == 0 then
7  |  unpriv_stop(NO_PERM)
8  if DATA_SIZE == 0 then
9  |  unpriv_stop(NONE)
10 if DATA_SIZE + ARG_1 > ep.size then
11 |  unpriv_stop(OUT_OF_BOUNDS)
12 if (DATA_ADDR & 0xFFF) + DATA_SIZE > 0x1000 then
13 |  unpriv_stop(PAGE_BOUNDARY)
14 phys ← DATA_ADDR;
15 (phys, err) ← lookup_tlb( CUR_ACT.id,  DATA_ADDR, READ);
16 if err != 0 then
17 |  unpriv_stop(TRANSLATION_FAULT)
18 (data, err) ← read_local(phys, DATA_SIZE);
19 if err != 0 then
20 |  unpriv_stop(ABORT)
21 err ← write_remote(data, ep.chip, ep.tile, ep.addr + ARG_1, DATA_SIZE);
22 if err != 0 then
23 |  unpriv_stop(ABORT)
24 COMMAND ← 0;
```

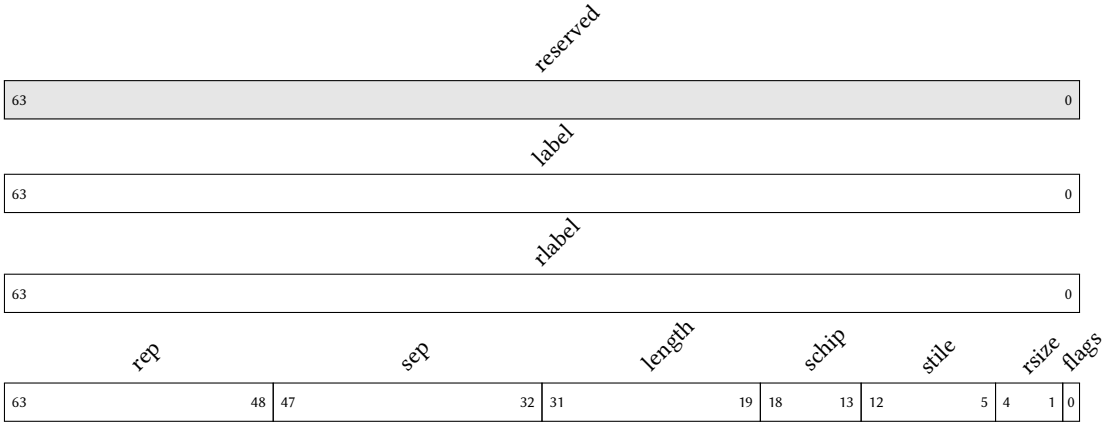
3.4 Message Passing

Message passing is performed between a send EP and a receive EP. Each send EP is connected to exactly one receive EP, whereas each receive EP can receive from multiple send EPs. The send EP supports the command SEND, whereas the receive EP supports REPLY, FETCH, and ACK_MSG.

For flow control and to prevent denial-of-service attacks on recipients, the TCU uses a credit system. The idea is to let the recipient hand out credits to its senders, decrease the credits on sent messages and increase them again on received replies.

Each message consists of a header and a payload. The header is built by the TCU and the payload is given by the application. The TCU stores both header and payload into the receive buffer in memory. The header is defined as:

Register 3.2: Message Header



label the label of the sender

rlabel the label the receiver should use for the reply

rep the receive endpoint ID for the reply at the sender side

sep the sender endpoint ID

length the payload size in bytes

schip the sender chip ID

stile the sender tile ID

rsize the size of the reply message as a power of 2

flags contains the following flags:

- REPLY (1): the message is a reply

The commands and the message reception behave as follows:

3.4.1 SEND

Note that DATA_SIZE == 0 is allowed and sends only the message header to the receiver without payload. Note also the placement of read_local: this transfer can fail due to command abortions, but we haven't changed any state before. After this transfer, we reduce the credits and send the message. The latter can again fail, but only if the receive EP is invalid or full. If credits are not used and the receive EP is full, the state has not changed. Otherwise, the failure indicates a broken communication channel and we consider it acceptable to leave the send EP in a broken state, too (one credit is lost and we never get it back).

Algorithm 3: The TCU's SEND command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.type != SEND then
3  |  unpriv_stop(NO_SEP)
4  if ep.reply != 0 then
5  |  unpriv_stop(SEND_REPLY_EP)
6  if ep.act != CUR_ACT.id then
7  |  unpriv_stop(FOREIGN_EP)
8  if ep.msg_sz > 11 then
9  |  unpriv_stop(SEND_INV_MSG_SZ)
10 if DATA_SIZE + sizeof(header) > (1 << ep.msg_sz) then
11 |  unpriv_stop(OUT_OF_BOUNDS)
12 if (DATA_ADDR & 0xF) != 0 then
13 |  unpriv_stop(MSG_UNALIGNED)
14 if (DATA_ADDR & 0xFFF) + DATA_SIZE > 0x1000 then
15 |  unpriv_stop(PAGE_BOUNDARY)
16 phys ← DATA_ADDR;
17 (phys, err) ← lookup_tlb( CUR_ACT.id,  DATA_ADDR, READ);
18 if err != 0 then
19 |  unpriv_stop(TRANSLATION_FAULT)
20 if COMMAND.arg_0 != 0xFFFF then
21 |  rep ← read_ep(COMMAND.arg_0);
22 |  if rep.type != RECEIVE then
23 |  |  unpriv_stop(NO_REP)
24 |  repid ← COMMAND.arg_0;
25 |  rsize ← rep.slot_size;
26 else
27 |  repid ← 0xFFFF;
28 |  rsize ← log2(sizeof(header));
29 (payload, err) ← read_local(phys, DATA_SIZE);
30 if err != 0 then
31 |  unpriv_stop(ABORT)
32 if ep.cur_crd != 0x3F then
33 |  if ep.cur_crd == 0 then
34 |  |  unpriv_stop(NO_CREDITS)
35 |  sepid ← COMMAND.ep;
36 else
37 |  sepid ← 0xFFFF;
```

Algorithm 4: The TCU's SEND command (continued).

```
38 header ← { flags ← 0;
39             label ← ep.label;
40             length ← DATA_SIZE;
41             rsize ← rsize;
42             rlabel ← ARG1;
43             schip ← ownChip;
44             stile ← ownTile;
45             sep ← sepid;
46             rep ← repid };
47 send_msg(header | payload, ep.tgt_chip, ep.tgt_tile, ep.tgt_ep);
48 err ← wait_for_ack();
49 if err != 0 then
50 |   unpriv_stop(err)
51 if ep.cur_crd != 0x3F then
52 |   ep.cur_crd -= 1;
53 |   write_ep(COMMAND.ep, ep);
54 COMMAND ← 0;
```

3.4.2 RECEIVE

Note that RECEIVE is not a command, but just the logic at the receiver side that accepts and stores messages. Nevertheless, its behavior is described by the following algorithm:

Algorithm 5: If 'header | payload' is received via EP 'rep'.

```
1  ep ← read_ep(rep);
2  if ep.type != RECEIVE then
3  |  send_ack(RECV_GONE) and drop message
4  if sizeof(header) + header.length > (1 << ep.slot_size) then
5  |  send_ack(RECV_OUT_OF_BOUNDS) and drop message
6  if ep.rpl_eps != 0xFFFF and ep.rpl_eps + (1 << ep.slots) > EP_COUNT then
7  |  send_ack(RECV_INV_RPL_EPS) and drop message
8  idx ← find_slot(ep.occupied, ep.wpos, ep.slots, 0);
9  if idx == -1 then
10 |  send_ack(RECV_NO_SPACE) and drop message
11 ep.occupied.set_bit(idx);
12 ep.wpos ← idx + 1;
13 dest ← ep.buffer + (idx << ep.slot_size);
14 write_local(header | payload, dest, sizeof(header) + header.length);
15 ep.unread.set_bit(idx);
16 write_ep(rep, ep);
17 if (header.flags & REPLY) == 0 and ep.rpl_eps != 0xFFFF and header.rep !=
     0xFFFF then
18 |  sep ← { type ← SEND;
19 |           act ← ep.act;
20 |           reply ← 1;
21 |           tgt_chip ← header.schip;
22 |           tgt_tile ← header.stile;
23 |           tgt_ep ← header.rep;
24 |           label ← header.rlabel;
25 |           msg_sz ← header.rsize;
26 |           max_crd ← 1;
27 |           cur_crd ← 1;
28 |           crd_ep ← header.sep };
29 |  write_ep(ep.rpl_eps + idx, sep);
30 if (header.flags & REPLY) != 0 and header.rep != 0xFFFF then
31 |  sep ← read_ep(header.rep);
32 |  sep.cur_crd += 1;
33 |  write_ep(header.rep, sep);
34 send_ack(NONE);
35 if ep.act == CUR_ACT.id then
36 |  CUR_ACT.msgs += 1;
37 else
38 |  queue_foreign_msg_req(rep, ep.act);
```

3.4.3 REPLY

Note that DATA_SIZE == 0 is allowed and sends only the message header to the receiver without payload. Like for SEND, note also the placement of read_local and places where state is changed.

Algorithm 6: The TCU's REPLY command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.type != RECEIVE then
3  |  unpriv_stop(NO_REP)
4  if ep.rpl_eps == 0xFFFF then
5  |  unpriv_stop(REPLIES_DISABLED)
6  if ep.rpl_eps + (1 << ep.slots) > EP_COUNT then
7  |  unpriv_stop(RECV_INV_RPL_EPS)
8  if ep.act != CUR_ACT.id then
9  |  unpriv_stop(FOREIGN_EP)
10 idx ← COMMAND.arg_0 >> ep.slot_size;
11 if idx >= 1 << ep.slots then
12 |  unpriv_stop(INV_MSG_OFF)
13 sep = read_ep(ep.rpl_eps + idx);
14 if sep.type != SEND then
15 |  unpriv_stop(NO_SEP)
16 if sep.reply == 0 then
17 |  unpriv_stop(SEND_REPLY_EP)
18 if sep.crd_ep != 0xFFFF and sep.crd_ep >= EP_COUNT then
19 |  unpriv_stop(SEND_INV_CRD_EP)
20 if sep.msg_sz > 11 then
21 |  unpriv_stop(SEND_INV_MSG_SZ)
22 if DATA_SIZE + sizeof(header) > (1 << sep.msg_sz) then
23 |  unpriv_stop(OUT_OF_BOUNDS)
24 if (DATA_ADDR & 0xF) != 0 then
25 |  unpriv_stop(MSG_UNALIGNED)
26 if (DATA_ADDR & 0xFFF) + DATA_SIZE > 0x1000 then
27 |  unpriv_stop(PAGE_BOUNDARY)
```

Algorithm 7: The TCU's REPLY command (continued).

```
28 phys ← DATA_ADDR;
29 (phys, err) ← lookup_tlb( CUR_ACT.id, DATA_ADDR, READ);
30 if err != 0 then
31 |   unpriv_stop(TRANSLATION_FAULT)
32 (payload, err) ← read_local(phys, DATA_SIZE);
33 if err != 0 then
34 |   unpriv_stop(ABORT)
35 header ← { flags ← REPLY;
36              label ← sep.label;
37              length ← DATA_SIZE;
38              rsize ← 0;
39              rlabel ← 0;
40              schip ← ownChip;
41              stile ← ownTile;
42              sep ← COMMAND.ep;
43              rep ← sep.crd_ep };
44 send_msg(header | payload, sep.tgt_chip, sep.tgt_tile, sep.tgt_ep);
45 err ← wait_for_ack();
46 if err != 0 then
47 |   unpriv_stop(err)
48 sep ← { type ← INVALID };
49 write_ep(ep.rpl_eps + idx, sep);
50 if ep.unread.is_set(idx) then
51 |   CUR_ACT.msgs -= 1;
52 ep.occupied.clear_bit(idx);
53 ep.unread.clear_bit(idx);
54 write_ep(COMMAND.ep, ep);
55 COMMAND ← 0;
```

3.4.4 FETCH

Algorithm 8: The TCU's FETCH command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.type != RECEIVE then
3  |   unpriv_stop(NO_REP)
4  if ep.act != CUR_ACT.id then
5  |   unpriv_stop(FOREIGN_EP)
6  if ep.unread == 0 or CUR_ACT.msgs == 0 then
7  |   ARG1 ← −1;
8  |   unpriv_stop(NONE);
9  idx ← find_slot(ep.unread, ep.rpos, ep.slots, 1);
10 ep.unread.clear_bit(idx);
11 ep.rpos ← idx + 1;
12 write_ep(COMMAND.ep, ep);
13 CUR_ACT.msgs -= 1;
14 ARG1 ← idx << ep.slot_size;
15 COMMAND ← 0;
```

3.4.5 ACK_MSG

Algorithm 9: The TCU's ACK_MSG command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.type != RECEIVE then
3  |  unpriv_stop(NO_REP)
4  if ep.rpl_eps != 0xFFFF and ep.rpl_eps + (1 << ep.slots) > EP_COUNT then
5  |  unpriv_stop(RECV_INV_RPL_EPS)
6  if ep.act != CUR_ACT.id then
7  |  unpriv_stop(FOREIGN_EP)
8  idx ← COMMAND.arg_0 >> ep.slot_size;
9  if idx >= 1 << ep.slots then
10 |  unpriv_stop(INV_MSG_OFF)
11 if ep.unread.is_set(idx) then
12 |  CUR_ACT.msgs -= 1;
13 ep.occupied.clear_bit(idx);
14 ep.unread.clear_bit(idx);
15 write_ep(COMMAND.ep, ep);
16 if ep.rpl_eps != 0xFFFF then
17 |  sep ← { type ← INVALID };
18 |  write_ep(ep.rpl_eps + idx, sep);
19 COMMAND ← 0;
```

3.5 Debug Prints

The register PRINT can be used to perform debug prints with the TCU. The software has to first write the string to print into the print registers (PR0, ...), starting with the first byte of the string at the least significant byte of PR0. Afterwards, the number of bytes to print has to be written to PRINT. As soon as the print was carried out, the TCU writes 0 to PRINT to acknowledge the print. Note that the effect of the debug prints is implementation defined. For example, a simulator might write the string into a log file, whereas a hardware implementation could output the string via a serial port.

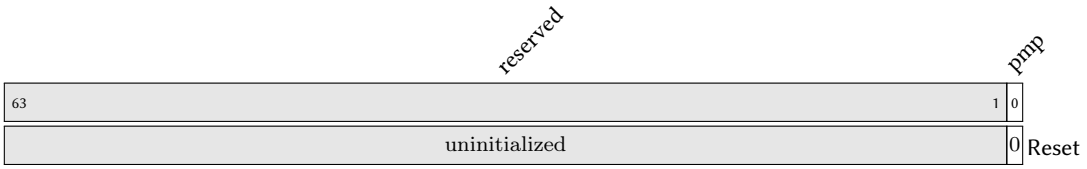
Chapter 4

Privileged Interface

The privileged interface of the TCU adds support for virtual memory and tile sharing to the TCU and is thus only required if either of these features are desired. Thus, all non-highlighted functionality is required if either extension is used, whereas the highlighted functionality is only required for one specific extension.

The interface consists of multiple privileged registers, privileged commands, and CU requests, which are raised by the TCU and answered by the privileged software. The privileged interface is configured via the PRIV_CTRL register, shown below:

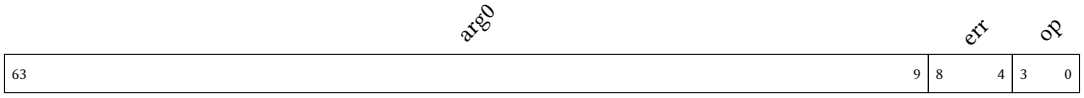
Register 4.1: PRIV_CTRL (0xF000_2008)



pmp if set, PMP failures are reported via CU request

To support the privileged software in the management of multiple activities or virtual memory, privileged commands are used, which can be performed by writing to PRIV_CMD. The register is defined as follows:

Register 4.2: PRIV_CMD (0xF000_2010)



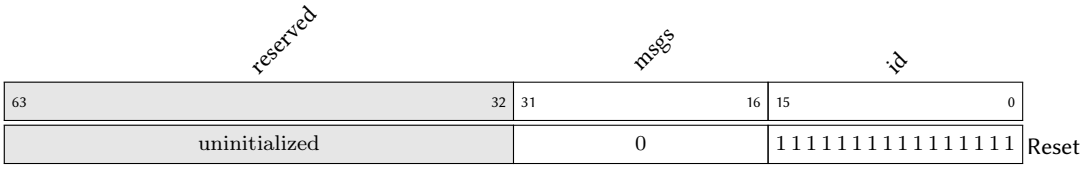
arg0 the first argument for the privileged command (PRIV_CMD_ARG1 is the second)

err the result of the operation (output field)

op the opcode of the privileged command

Note that none of the privileged commands can currently fail. Thus, only unknown command opcodes lead to err being set to an a non-zero value.
The TCU stores the id of the current activity including the number of unread messages in all its receive EPs in the CUR_ACT register:

Register 4.3: CUR_ACT (0xF000_2020)



msgs the sum of all unread messages in the receive EPs for this activity

id the activity id

4.1 Command List

The following privileged commands are supported with the opcodes in parentheses. Other opcodes lead to an UNKNOWN_CMD error.

- IDLE (0): don't do anything,
- INV_PAGE (1): invalidate a page in the TLB,
- INV_TLB (2): invalidate the complete TLB,
- INS_TLB (3): insert an entry into the TLB,
- XCHG_ACT (4): change the activity,
- SET_TIMER (5): start/stop the timer,

- ABORT_CMD (6): abort the unprivileged command.

4.2 Pseudo Code Building Blocks

The following sections use pseudo code to describe the behavior of the privileged commands, based on several building blocks:

- evict_tlb_entry() -> Entry:
  searches for a TLB entry that can be evicted and returns it. If all entries are fixed and therefore cannot be evicted, null is returned.

- set_timer(timeout):
  starts/restarts the timer to fire an interrupt after timeout nanoseconds

- unset_timer():
  stops the timer

- waiting_for_resp() -> bool:
  returns whether the TCU is currently waiting for a remote TCU's response (to SEND, REPLY, READ, or WRITE)

- wait_for_resp():
  waits until a remote TCU responds (to SEND, REPLY, READ, or WRITE)

- abort_remote_xfer():
  lets the current remote transfer fail, so that read_remote or write_remote return the ABORT error (see Section 3.2)

- abort_local_xfer() -> bool:
  checks if there is a local transfer (for the current unprivileged command) and if so, aborts it, so that read_local() or write_local() return the ABORT error (see Section 3.2). Returns true if it was aborted.

- resume_local_xfer():
  resumes a previously paused local transfer (due to an address translation).

- priv_stop(error):
  stop the execution of the privileged command by setting the opcode in PRIV_CMD to 0 and the error code to the given error.

4.3 Command Description

4.3.1 INV_PAGE

Algorithm 10: The TCU's INV_PAGE command.

```
1 act ← PRIV_CMD.arg0 ;
2 virt ← PRIV_CMD_ARG1 & 0xFFFF_FFFF_FFFF_F000;
3 forall entries e do
4 |   if e.act == act and e.virt == virt then
5 |   |   e.flags ← 0;
6 end
7 PRIV_CMD.op ← 0;
```

Note that INV_PAGE invalidates the entry regardless of whether the flag FIXED is set or not.

4.3.2 INV_TLB

Algorithm 11: The TCU's INV_TLB command.

```
1 forall entries e do
2 |   if (e.flags & FIXED) == 0 then
3 |   |   e.flags = 0;
4 end
5 PRIV_CMD.op ← 0;
```

Note that INV_TLB does not invalidate the entries with set FIXED flag.

4.3.3 INS_TLB

Algorithm 12: The TCU's INS_TLB command.

```
1  act ← PRIV_CMD.arg0 >> 32;
2  virt ← PRIV_CMD_ARG1 & 0xFFFF_FFFF_FFFF_F000;
3  phys ← PRIV_CMD.arg0 & 0xFFFF_F000;
4  flags ← PRIV_CMD.arg0 & 0x1F;
5  entry ← null;
6  forall entries e do
7  |  if e.act == act and e.virt == virt then
8  |  |  entry ← e;
9  end
10 if entry is null then
11 |  entry ← evict_tlb_entry();
12 |  if entry is null then
13 |  |  priv_stop(TLB_FULL);
14 entry.act = act;
15 entry.virt = virt;
16 entry.phys = phys;
17 entry.flags = flags;
18 PRIV_CMD.op ← 0;
```

4.3.4 XCHG_ACT

Algorithm 13: The TCU's XCHG_ACT command.

```
1  PRIV_CMD_ARG1 ← CUR_ACT;
2  CUR_ACT ← PRIV_CMD.arg0 & 0xFFFF_FFFF;
3  PRIV_CMD.op ← 0;
```

4.3.5 SET_TIMER

Algorithm 14: The TCU's SET_TIMER command.

```
1  nanos ← PRIV_CMD.arg0;
2  if nanos == 0 then
3  |  unset_timer();
4  else
5  |  set_timer(nanos);
6  PRIV_CMD.op ← 0;
```

4.3.6 ABORT_CMD

Algorithm 15: The TCU's ABORT_CMD command.

```
1 if COMMAND.op != 0 then
2 |  if waiting_for_resp() then
3 |  |  if COMMAND.op == READ or COMMAND.op == WRITE then
4 |  |  |  PRIV_CMD.arg0 ← 1;
5 |  |  |  abort_remote_xfer();
6 |  |  wait_for_resp();
7 |  else if abort_local_xfer() then
8 |  |  PRIV_CMD.arg0 ← 1;
9 PRIV_CMD.op ← 0;
```

4.4 CU Requests

In some cases, the TCU needs assistance of the CU or wants to notify the CU of an event. For example, if a message for another activity than the currently running activity (CUR_ACT) is received, the assistance of the CU is needed.

Since multiple CU requests might occur simultaneously, the TCU keeps a queue of pending CU requests and handles them in FIFO order. To handle the first in the queue, the TCU writes to CU_REQ and injects an interrupt to inform the CU about the request. After the handling of the CU request, the CU is expected to write to CU_REQ again to signal the completion of the request. Upon receiving this signal, as described in algorithm 17, the TCU starts the next CU request, if there is any. New CU requests, as described in algorithm 16, are appended to the queue and started, if possible.

4.4.1 Pseudo Code

The following pseudo code describes the TCU's behavior regarding CU requests in more detail.

Algorithm 16: Enqueuing and starting of CU requests.

```
1  Function queue_foreign_msg_req(act, ep):
2  |   queue.enqueue(act | ep | 2);
3  |   try_start_req();
4  End Function
5  Function queue_pmp_failure_req(phys, write, error):
6  |   queue.enqueue(phys | write | error | 3);
7  |   try_start_req();
8  End Function
9  Function try_start_req():
10 |   if CU_REQ.type == 0 and not queue.empty() then
11 |   |   CU_REQ ← queue.front();
12 End Function
```

Algorithm 17: Dequeuing and finishing of CU requests.

```
1  if CU_REQ.type == 1 and not queue.empty() then
2  |   cur = queue.dequeue();
3  |   CU_REQ ← 0;
4  |   try_start_req();
```

4.4.2 Registers

Register 4.4: CU_REQ (0xF000_2000)

type the type (0 = idle, 1 = response, 2 = foreign message, 3 = PMP failure)

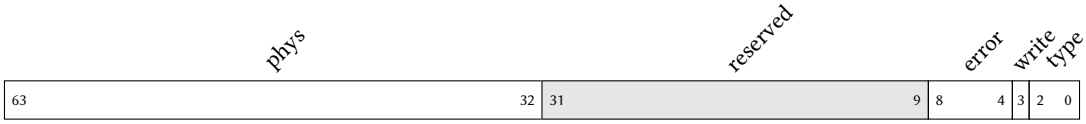
As the CU_REQ register is used for different kinds of requests and responses, most of the bits depend on the type of request/response, defined in the following.

Register 4.5: CU_REQ for foreign message requests (0xF000_2000)

ep the receive EP which received a message

act the activity that received the message

Register 4.6: CU_REQ for PMP failure requests (0xF000_2000)



phys the physical access for which the access failed

error the error code:

- NO_PMP_EP: the EP is greater or equal to the number of PMP EPs
- NO_MEP: the EP is no memory endpoint
- OUT_OF_BOUNDS: the access refers outside of the memory EP's region
- NO_PERM: the memory EP does not have the required permission

write 1 for write accesses and 0 for read accesses

Register 4.7: CU_REQ for responses (0xF000_2000)

4.5 Translation Look-aside Buffer

With virtual memory support, the TCU maintains a TLB to cache recent address translations. These translations are exclusively used for local transfers and thus always explicitly triggered by the software on the CU via TCU commands. The privileged software is responsible to remove entries from the TLB when pages get unmapped from the application's

virtual address space or if page permissions were downgraded. Additionally, the TCU can assume that TLB entries stay valid during the complete execution of a command. In other words, the privileged software is responsible to abort a potentially running command when removing entries from the TLB to ensure that it is not used anymore.

The TLB contains a 52-bit virtual page number (virtual address shifted right by 12 bits), 16-bit activity id, 20-bit physical page number (physical address shifted right by 12 bits), and 3 bits for flags. The virtual and physical address are always page-size aligned. The flags are defined as follows:

- READ (1): read permission,
- WRITE (2): write permission,
- FIXED (4): fixed entry, will not be evicted.

If no flag bit is set, the TLB entry is invalid and will be ignored during lookups.

4.6 Physical Memory Protection

With virtual memory support, tiles get access to a shared physical memory. The mapping from virtual to physical memory is performed by privileged software on each tile. To prevent that this privileged software and the CU within the tile is part of the trusted computing base (as it can access all physical memory without further measures), the TCU supports physical memory protection.

The physical memory of each tile is split into multiple fixed-sized regions, whereas each region is protected by a memory EP. The current implementation uses four regions of 1 GiB each and thus uses the upper two bits of the physical address to determine the memory EP. In other words, the first four EPs define to which parts of external memories the tile has access and also defines the access permissions (read and/or write).

Instead of connecting the last-level cache (LLC) directly to the NoC, the LLC is connected to the TCU's physical memory protection, which validates and translates the address before passing it to the NoC. The following pseudo code describes this process:

Algorithm 18: The validation and translation of physical addresses.

```
1  Function llc_miss(phys, size, access):
2      addr ← phys − 0x10000000;
3      epid ← addr >> 30;
4      off ← addr & 0x3FFFFFFF;
5      if epid >= 4 then
6          return and queue_pmp_failure_req(phys, access == WRITE,
             NO_PMP_EP)
7      ep ← read_ep(epid);
8      if ep.type != MEMORY then
9          return and queue_pmp_failure_req(phys, access == WRITE, NO_MEP)
10     if ep.rw & access == 0 then
11         return and queue_pmp_failure_req(phys, access == WRITE,
             NO_PERM)
12     if off + size > ep.size then
13         return and queue_pmp_failure_req(phys, access == WRITE,
             OUT_OF_BOUNDS)
14     if access == READ then
15         read_remote(ep.tile, ep.addr + off, size);
16     else
17         write_remote(data, ep.tile, ep.addr + off, size);
18 End Function
```

The function llc_miss above is called for each LLC miss and validates whether the memory access is allowed before performing the memory access. The first parameter specifies the physical address, the second parameter specifies the cache-line size, and the third parameter specifies whether the access reads from memory or writes to memory. Note that the current implementation substracts 0x10000000 from the physical access, because the Rocket Core maps the DRAM to that address.

If the access fails, a PMP-failure CU request is enqueued and the PMP access is ignored. This is because PMP accesses are triggered by last-level cache misses, but the code causing the access executed potentially a long time ago. Thus, the software has no chance to handle this case except for reporting an error.

Chapter 5

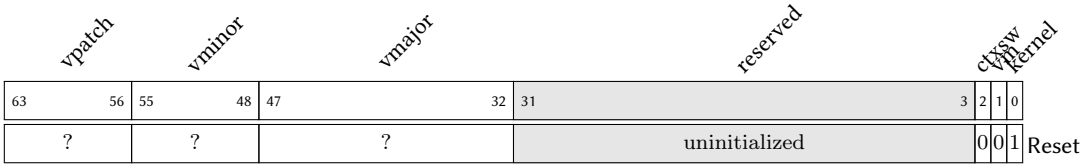
External Interface

The external interface of the TCU allows remote tiles to retrieve meta information about the tile and TCU and manipulate its state.

5.1 Meta Information

The first part of the external interface are meta information about the TCU or the tile it belongs to that can be retrieved and partially also be changed. The available TCU features can be configured through the FEATURES register, which has the following format:

Register 5.1: FEATURES (0xF000_0000)



vmajor the TCU major version number

vminor the TCU minor version number

vpatch the TCU patch version number

ctxsw whether the context-switching extension is available

vm whether the virtual-memory extension is available

kernel whether the TCU belongs to a kernel tile

At reset, FEATURES.kernel is set to 1, so that all tiles are kernel tiles. The software starting on one tile can afterwards downgrade the other tiles to user tiles. If FEATURES.kernel is 1, the CU can write to endpoint registers. This can be used to create a communication

channel to the TCU's MMIO region in another tile, which allows to establish other communication channels by writing to the endpoint registers within the MMIO region.

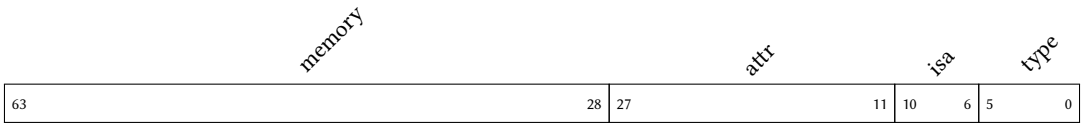
The bits FEATURES.ctxsw and FEATURES.vm control whether the context-switching and virtual-memory extension is enabled, respectively. If the former is disabled, foreign message receptions do not raise a CU request. If the latter is disabled, no address translation is performed and thus no translation CU request is raised. The behavior of privileged commands is undefined if the associated extension is disabled.

The TCU uses semantic versioning[1] to keep track of changes. Thus, incompatible changes result in a new major version, compatible changes result in a new minor version, and bugfixes result in a new patch version. All three version numbers are implementation defined and can be used to verify whether the software is compatible to the TCU implementation at hand. The version cannot be changed at runtime. The major and minor version of the implementation should correspond to the major and minor version of the specification it implements, which is shown on the title page of this document.

Besides FEATURES, the TCU provides the register TILE_DESC that contains a description of the tile. The description is set by the hardware and cannot be changed.

[1] https://semver.org

Register 5.2: TILE_DESC (0xF000_0008)



memory the internal memory size in 4 KiB pages

attr additional attributes with the following bits:

- BOOM (1): the tile contains a BOOM core
- ROCKET (2): the tile contains a Rocket core
- NIC (4): a network interface card attached to the core
- SERIAL (8): a serial interface attached to the core
- IMEM (16): the tile has an internal memory

isa the instruction set architecture:

- NONE (0): dummy ISA for memory tiles
- RISCV (1)

type the type of tile:

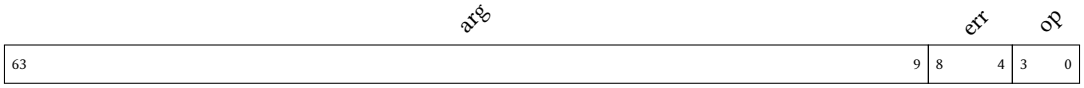
- COMP (0): contains a processor for computation
- MEM (1): contains memory

Note that other values and bits are currently not defined. The memory size is 0 if the attribute IMEM is not set and the size of the internal memory in 4 KiB pages otherwise.

5.2 External Commands

The TCU supports external commands, which are triggered by writing to the EXT_CMD register and feedback is given via this register as well. The EXT_CMD register has the following format:

Register 5.3: EXT_CMD (0xF000_0010)



arg an argument for the operation

err the result of the operation (output field)

op the operation to execute

The TCU supports the following external commands with the opcodes in parentheses. Other opcodes lead to an UNKNOWN_CMD error.

- IDLE (0): don't do anything,

- INV_EP (1): invalidate an endpoint.

5.3 Pseudo Code Building Blocks

The following sections use pseudo code to describe the behavior of the external TCU commands, based on the following building blocks:

- ext_stop(error):
  stop the execution of the external command by setting the opcode in EXT_CMD.op to 0 and EXT_CMD.err to the given error.

5.4 Command Description

5.4.1 INV_EP

Algorithm 19: The TCU's INV_EP command.

```
1  epid ← EXT_CMD.arg & 0xFFFF;
2  force ← EXT_CMD.arg >> 16;
3  EXT_CMD.arg ← 0;
4  ep ← read_ep(epid);
5  if force == 0 and ep.type == SEND then
6  |  if ep.cur_crd != ep.max_crd then
7  |  |  ext_stop(NO_CREDITS)
8  if force == 0 and ep.type == RECEIVE then
9  |  EXT_CMD.arg ← ep.unread;
10 ep ← { type ← INVALID };
11 write_ep(epid, ep);
12 EXT_CMD.err ← NONE;
13 EXT_CMD.op ← 0;
```