DTU Specification

Nils Asmussen

July 31, 2019

1 Overview

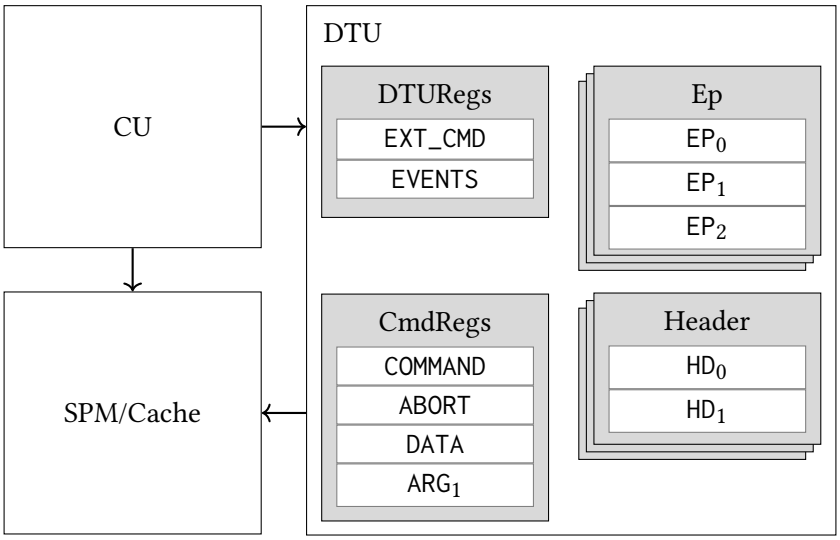


Figure 1: Overview of the DTU's registers and its connections to other components.

As shown in Figure 1, the compute unit (CU) is connected to the data transfer unit (DTU) and can access the DTU's registers via memory mapped input/output (MMIO). Additionally, the CU is connected to the local memory. The DTU is also connected to the local memory to, for example, access messages. These components are not necessarily arranged in this way. For example, the DTU might interpose itself between the CU and local memory.

2 MMIO region

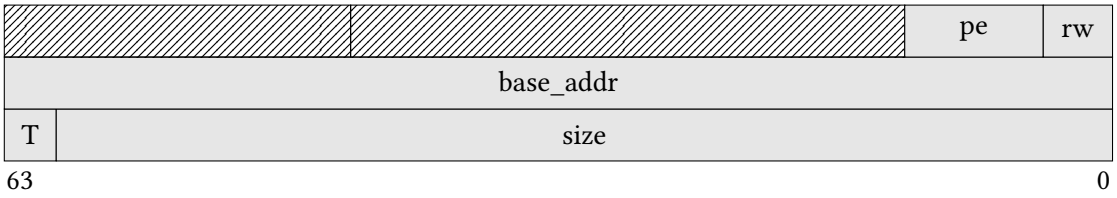
The memory interface from CU to DTU is expected to be 64-bit wide. The MMIO region of the DTU is defined as follows:

Address	Register	Group	Description
0xF000_0000	EXT_CMD	DtuRegs	Triggers external commands
0xF000_0008	EVENTS	DtuRegs	Contains outstanding events
0xF000_0010	COMMAND	CmdRegs	Triggers internal commands
0xF000_0018	ABORT	CmdRegs	Aborts internal commands
0xF000_0020	DATA	CmdRegs	Specifies the data for commands
0xF000_0028	ARG_1	CmdRegs	Additional argument for commands
0xF000_0030	EP_{00}	EpRegs	First register of EP_0
0xF000_0038	EP_{01}	EpRegs	Second register of EP_0
0xF000_0040	EP_{02}	EpRegs	Third register of EP_0
0xF000_0048	EP_{10}	EpRegs	First register of EP_1
0xF000_0050	EP_{11}	EpRegs	Second register of EP_1
0xF000_0058	EP_{12}	EpRegs	Third register of EP_1
		...	
0xF000_0100	EP_{n0}	EpRegs	First register of EP_n
0xF000_0108	EP_{n1}	EpRegs	Second register of EP_n
0xF000_0110	EP_{n2}	EpRegs	Third register of EP_n
0xF000_0118	HD_{00}	Header	First register of HD_0
0xF000_0120	HD_{01}	Header	Second register of HD_0
0xF000_0128	HD_{02}	Header	Third register of HD_0
0xF000_0130	HD_{10}	Header	First register of HD_1
0xF000_0138	HD_{11}	Header	Second register of HD_1
0xF000_0140	HD_{12}	Header	Third register of HD_1
		...	
0xF000_0200	HD_{n0}	Header	First register of HD_n
0xF000_0208	HD_{n1}	Header	Second register of HD_n
0xF000_0210	HD_{n2}	Header	Third register of HD_n

3 Endpoints

The DTU 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 DTU 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) and 189 bits, whose meaning depends on the EP type. T is either INVALID (0), SEND (1), RECEIVE (2), or MEMORY (3). The endpoints are defined as follows:

3.1 Memory EP



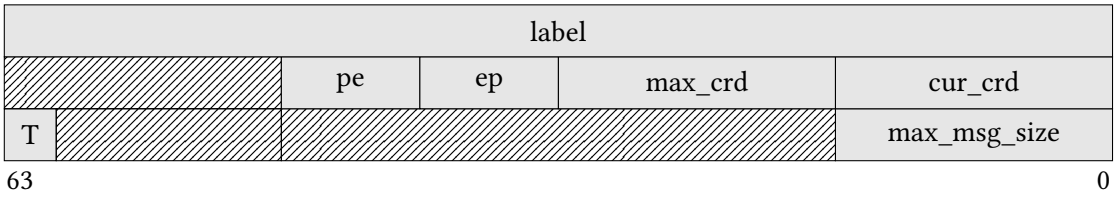
pe[11:4]: the destination PE ID

rw[3:0]: the permission bits (read = 1, write = 2)

base_addr[63:0]: the base address of the region at the destination

size[60:0]: the size of the region at the destination

3.2 Send EP



label[63:0]: the label the DTU puts into the header of each sent message

pe[47:40]: the ID of the destination PE

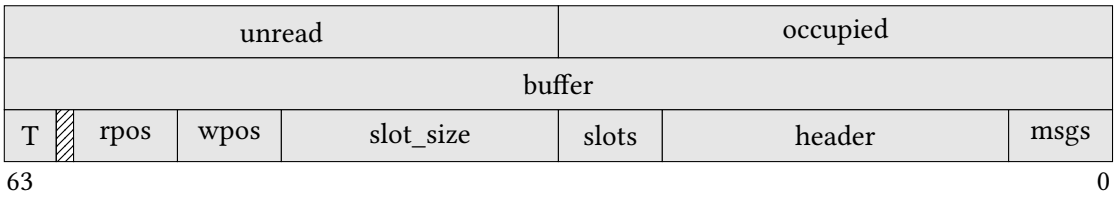
ep[39:32]: the ID of the receive EP

max_crd[31:16]: the initially received (=max) credits

cur_crd[15:0]: the currently owned credits

max_msg_size[15:0]: the maximum message size supported by the receiver

3.3 Receive EP



unread[63:32]: a bitmask with the unread (not yet fetched) messages in the buffer

occupied[31:0]: a bitmask with the occupied slots in the buffer

buffer[63:0]: the address of the receive buffer in local memory

rpos[59:54]: the read position (for message fetches) within the receive buffer

wpos[53:48]: the write position (for message receptions) within the receive buffer

slot_size[47:32]: the size of one slot as a power of 2

slots[31:26]: the number of slots in the receive buffer as a power of 2

header[25:6]: the offset of the headers for this receive buffer in the header table

msgs[5:0]: the number of unread messages

4 Commands

The CU can use the DTU's endpoints via *internal commands*. The command registers are used to pass input arguments for a command to the DTU, start a command, and wait until the command is finished. The following command registers are used:

arg0[63:16]: the first argument for the command

err[15:13]: the error code (0 = no error)

ep[11:4]: the endpoint to use for the command

op[3:0]: the command (opcode)

size[63:48]: the size of the data in local memory

address[47:0]: the address of the data in local memory

A write to the `COMMAND` register starts the command with opcode `COMMAND.op`. The meaning of the three argument registers depends on the opcode.

4.1 Memory Access

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

4.1.1 READ

Algorithm 1: The DTU's READ command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.T != MEMORY then
3  |   exit(COMMAND.err ← INV_EP)
4  if ep.rw & READ == 0 then
5  |   exit(COMMAND.err ← NO_PERM)
6  if ep.base_addr + COMMAND.arg_0 > ep.size then
7  |   exit(COMMAND.err ← INV_ARGS)
8  data ← read_remote(ep.PE, ep.base_addr + COMMAND.arg_0);
9  write_mem(data, DATA.address, DATA.size);
10 COMMAND ← 0;
```

4.1.2 WRITE

Algorithm 2: The DTU's WRITE command.

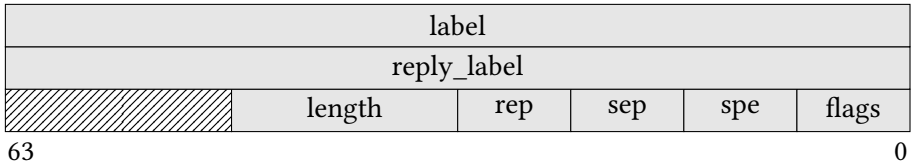
```
1  ep ← read_ep(COMMAND.ep);
2  if ep.T != MEMORY then
3  |   exit(COMMAND.err ← INV_EP)
4  if ep.rw & WRITE == 0 then
5  |   exit(COMMAND.err ← NO_PERM)
6  if ep.base_addr + COMMAND.arg_0 > ep.size then
7  |   exit(COMMAND.err ← INV_ARGS)
8  data ← read_mem(DATA.address, DATA.size);
9  write_remote(data, ep.PE, ep.base_addr + COMMAND.arg_0);
10 COMMAND ← 0;
```

4.2 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.

Each message consists of a header and a payload. The header is built by the DTU and the payload is given by the application. The header is defined as:

Header:



label[63:0]: the label of the sender

reply_label[63:0]: the label the receiver should use for the reply

length[47:32]: the payload size in bytes

rep[31:24]: the receive endpoint ID for the reply at the sender side

sep[23:16]: the sender endpoint ID

spe[15:8]: the sender PE ID

flags[7:0]: contains the following flags:

- `REPLY` (1): the message is a reply
- `GRANT_CREDITS` (2): the receiver of the message should receive credits
- `REPLY_ENABLED` (4): the receiver of the message may reply to the message

The commands and the message reception behave as follows:

4.2.1 SEND

Algorithm 3: The DTU's `SEND` command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.T != SEND then
3  |   exit(COMMAND.err ← INV_EP)
4  if ep.cur_crd != UNLIMITED then
5  |   if ep.cur_crd < ep.max_msg_size then
6  |   |   exit(COMMAND.err ← NO_CREDITS)
7  |   ep.cur_crd -= ep.max_msg_size;
8  |   write_ep(COMMAND.ep, ep);
9  header ← { flags ← REPLY_ENABLED;
10            label ← ep.label;
11            length ← DATA.size;
12            reply_label ← ARG_1;
13            spe ← ownPE;
14            sep ← COMMAND.ep;
15            rep ← COMMAND.arg_0 };
16 payload ← read_mem(DATA.address, DATA.size);
17 send_msg(header | payload, ep.pe, ep.ep);
18 wait_for_ack();
19 COMMAND ← 0;
```

4.2.2 RECEIVE

Algorithm 4: If 'header | payload' is received via EP 'rep'.

```
1  ep ⟵ read_ep(rep);
2  if ep.T != RECEIVE then
3  |   exit(send_ack() and drop message)
4  idx ⟵ find unoccupied slot starting at ep.wpos;
5  if idx not found then
6  |   exit(send_ack() and drop message)
7  ep.occupied.set_bit(idx);
8  ep.wpos ⟵ idx + 1;
9  dest ⟵ ep.buffer + (idx << ep.msg_size);
10 write_mem(header | payload, dest, sizeof(header) + header.length);
11 ep.msgs += 1;
12 ep.unread.set_bit(idx);
13 write_ep(rep, ep);
14 write_header(ep.header + idx, header);
15 if header.flags & GRANT_CREDITS then
16 |   sep ⟵ read_ep(header.rep);
17 |   sep.cur_crd += sep.max_msg_size;
18 |   write_ep(header.rep, sep);
19 send_ack();
```

4.2.3 REPLY

Algorithm 5: The DTU's REPLY command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.T != RECEIVE then
3  |   exit(COMMAND.err ← INV_EP)
4  idx ← (COMMAND.arg_0 − ep.buffer) >> ep.msg_size;
5  hd ← read_header(ep.header + idx);
6  if hd.flags & REPLY_ENABLED == 0 then
7  |   exit(COMMAND.err ← INV_ARGS)
8  hd.flags ← 0;
9  write_header(ep.header + idx, hd);
10 header ← { flags ← REPLY | GRANT_CREDITS;
11            label ← hd.reply_label;
12            length ← DATA.size;
13            reply_label ← 0;
14            spe ← ownPE;
15            sep ← COMMAND.ep;
16            rep ← hd.sep };
17 payload ← read_mem(DATA.address, DATA.size);
18 send_msg(header | payload, hd.spe, hd.rep);
19 wait_for_ack();
20 COMMAND ← 0;
```

4.2.4 FETCH

Algorithm 6: The DTU's FETCH command.

```
1  ep ← read_ep(COMMAND.ep);
2  if ep.T != RECEIVE then
3  |   exit(COMMAND.err ← INV_EP)
4  if ep.msgs == 0 then
5  |   exit(OFFSET ← 0 and COMMAND ← 0)
6  idx ← find unread message starting at ep.rpos;
7  ep.unread.clear_bit(idx);
8  ep.msgs -= 1;
9  ep.rpos ← idx + 1;
10 write_ep(COMMAND.ep, ep);
11 OFFSET ← ep.buffer + (idx << ep.msg_size);
12 COMMAND ← 0;
```

4.2.5 ACK_MSG

Algorithm 7: The DTU's ACK_MSG command.

```
1  ep ⟵ read_ep(COMMAND.ep);
2  if ep.T != RECEIVE then
3  |   exit(COMMAND.err ⟵ INV_EP)
4  idx ⟵ (COMMAND.arg_0 − ep.buffer) >> ep.msg_size;
5  ep.occupied.clear_bit(idx);
6  if ep.unread.bit_set(idx) then
7  |   ep.unread.clear_bit(idx);
8  |   ep.msgs -= 1;
9  write_ep(COMMAND.ep, ep);
10 hd ⟵ read_header(ep.header + idx);
11 hd.flags ⟵ 0;
12 write_header(ep.header + idx, hd);
13 COMMAND ⟵ 0;
```