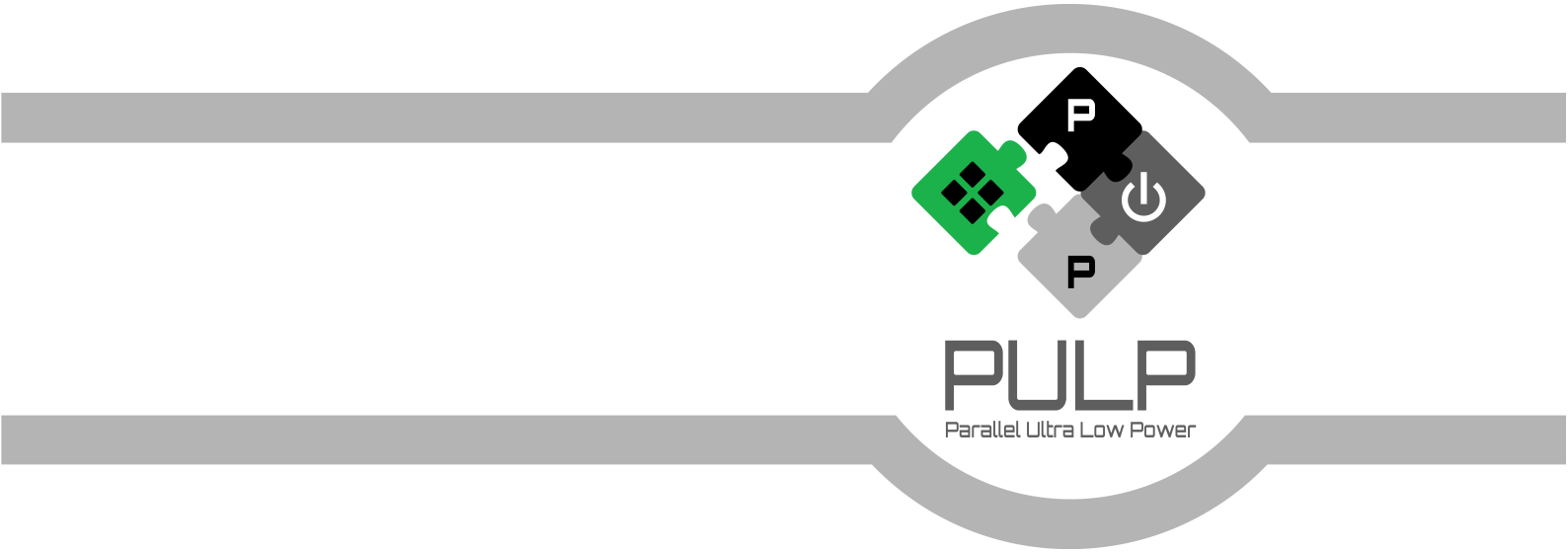

MCHAN: User Manual

March 2016
Revision 1.0

Davide Rossi (davide.rossi@unibo.it)

Micrel Lab and Multitherman Lab
University of Bologna, Italy

Integrated Systems Lab
ETH Zürich, Switzerland

Document Revisions

Rev.	Date	Author	Description
1.0	09/03/16	Davide Rossi	First Draft
1.1	14/06/16	Davide Rossi	Updated programming FSM

Table of Contents

1 Introduction

MCHAN is a Direct Memory Access (DMA) engine specifically developed for integration in the PULP platform. MCHAN is a generic component that can be multi instantiated depending on features and performance requirement. The main features of the MCHAN are the following:

- MCHAN allows buffer transfer between memories (accessible from EXT interface to TCDM interface and vice versa)
- Supports several 1D and 2D buffer transfer types.
- Ultra-low-latency programming interface (10 cycles for linear transfer).
- Up to 4GB/s on memory interfaces (500Mhz).
- Up to 16 outstanding request on memory interfaces to hide memory latency.
- Lightweight implementation without need to store internally big data buffers.

1.1 Scope and Purpose

This document aims at describing the functional specification of the PULP cluster DMA channel. It includes:

- Main features of the DMA
- IP Parameters
- Pinout of the IP
- Application Programming interface

1.1 Acronyms and Synonyms

PULP: Parallel processing Ultra-Low-Power platform

TCDM: Tightly Coupled Data Memory

DMA: Direct Memory Access

1.2 Deliveries

Library	Verilog File Name	Description
mchan_lib	include/mchan_defines.sv	MCHAN defines
mchan_lib	top/mchan.sv	MCHAN top
mchan_lib	ctrl_unit/ctrl_unit.sv	Control unit
mchan_lib	ctrl_unit/core_if.sv	Core interface
mchan_lib	ctrl_unit/address_decoder.sv	Address decoder

Library	Verilog File Name	Description
mchan_lib	ctrl_unit/synch_unit.sv	Synchronization unit
mchan_lib	ctrl_unit/trans_splitter.sv	Transactions splitter
mchan_lib	ctrl_unit/trans_unpack.sv	Transactions unpack
mchan_lib	ext_unit/ext_unit.sv	External unit
mchan_lib	ext_unit/ext_rx_if.sv	AXI RX interface
mchan_lib	ext_unit/ext_tx_if.sv	AXI TX interface
mchan_lib	ext_unit/ext_tid_gen.sv	TID Generator
mchan_lib	ext_unit/ext_opc_buf.sv	Opcodes buffer
mchan_lib	ext_unit/ext_buffer.sv	Generic buffer for external interface
mchan_lib	ext_unit/ext_ar_buffer.sv	AXI AR register slice
mchan_lib	ext_unit/ext_aw_buffer.sv	AXI AW register slice
mchan_lib	ext_unit/ext_b_buffer.sv	AXI B register slice
mchan_lib	ext_unit/ext_r_buffer.sv	AXI R register slice
mchan_lib	ext_unit/ext_w_buffer.sv	AXI W register slice
mchan_lib	tcdm_unit/tcdm_unit.sv	TCDM unit
mchan_lib	tcdm_unit/tcdm_cmd_unpack.sv	TCDM command unpack
mchan_lib	tcdm_unit/tcdm_rx_if.sv	TCDM RX interface
mchan_lib	tcdm_unit/tcdm_tx_if.sv	TCDM TX interface
mchan_lib	tcdm_unit/tcdm_synch.sv	TCDM synchronization unit
mchan_lib	trans_unit/trans_unit.sv	Transfer unit
mchan_lib	misc/mchan_arbiter.sv	Generic Arbiter
mchan_lib	misc/mchan_arb_primitive.sv	Generic Arbiter Primitive
mchan_lib	misc/mchan_fifo.sv	Generic FIFO
mchan_lib	misc/mchan_rr_flag_req.sv	Round robin flag generator

Table 1: MCHAN deliveries

1.3 MCHAN Architecture Overview

A block scheme of the MCHAN DMA channel is presented in Figure 3.1. It is consists of four modules. The control unit (CTRL UNIT) handles the transfer requests coming from the processors, arbitrates and forwards command requests to the global command queue, and generates the control signals for the other units to properly handle data transfers. The external unit is responsible for providing an interface toward the external bus, implementing a full-featured AXI 4 protocol interface. The TCDM unit implements the interface toward the cluster local memory. The transfer unit contains two FIFOs, one for TX transfers, one for RX transfers used to decouple data packets flowing through the two interfaces.

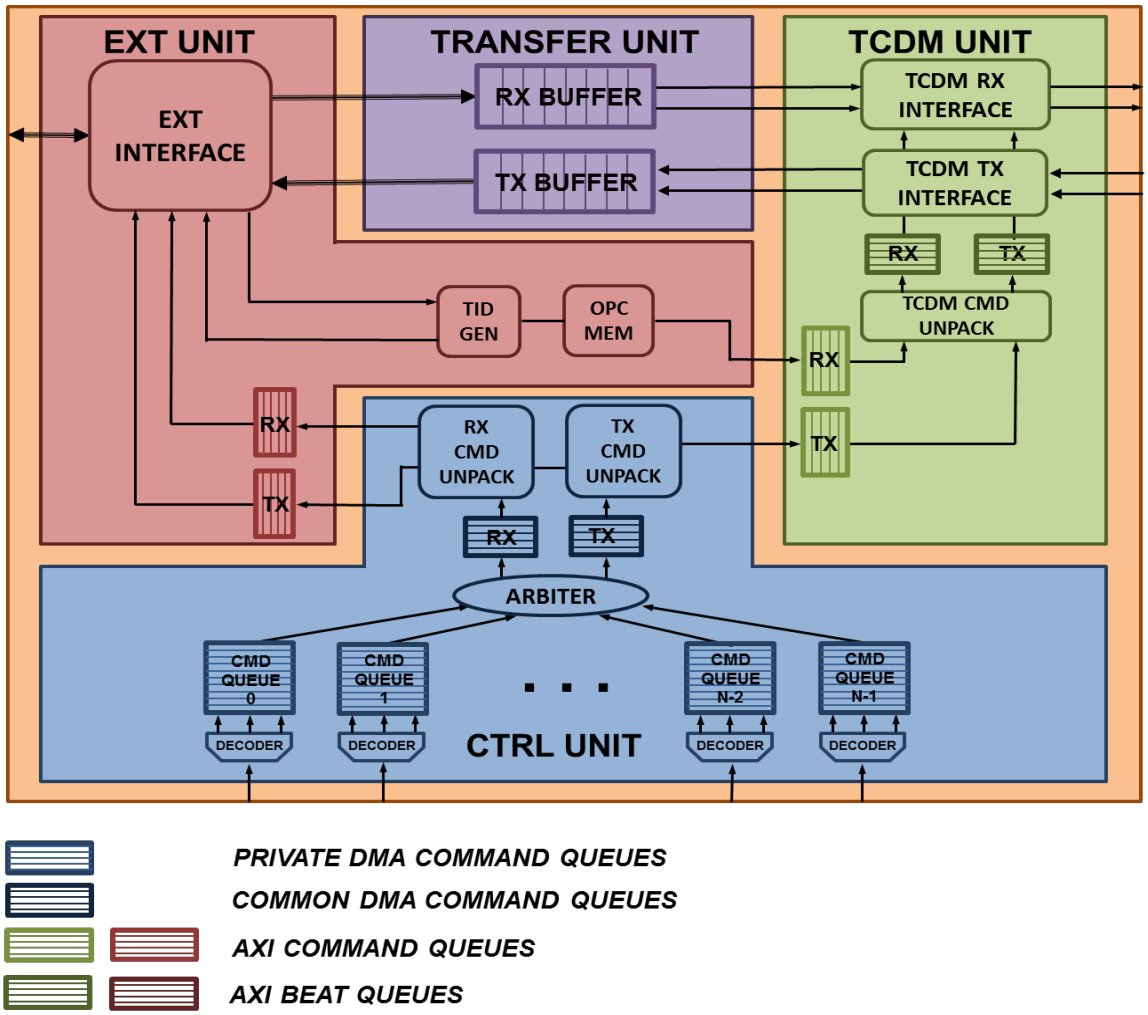

Figure 1: MCHAN architecture

1.3.1 Control Unit

The control unit is responsible for managing transfer requests coming from the processors, enqueue the requests to the global command queue, and generate the control and synchronization signals for the EXT unit, *TCDM unit* and *TRANS unit*.

The private command queues within the private per-core core interfaces include memory mapped registers implementing the three fields required to initialize one transfer. This entity, composed of TCDM address, external address and the operation required is called private command. Moreover, the core interfaces includes a mechanism to generate the termination events when a transfer issued by one core is being terminated.

The termination of a transfer is notified to the processor by an event generated by each core interface, or the core itself by polling on the status register can check it. To manage termination, each private DMA command queue features its own source identifier. When a DMA transfer is issued from a private command queue, the source ID associated to the private queue is forwarded to the common DMA command queue together with the DMA command. At the same time, a counter within each private command queue is incremented by the number of beats present in the operation submitted. Every time a transfer of the AXI command completes, both the EXT UNIT and TCDM UNIT notify the end of transfer to the command queue associated to the ID of the transfer, which decrement the value of the counter. When the value of the counter is equal to zero, the transfer is complete.

A round robin arbiter arbitrates commands incoming from the private command queues, and forwards them to a common global queue. An SID representing the source identification tag of the core is attached to the private command before being pushed to the global command queue. We call this entity *global command*. In order to decouple RX and TX commands, that can be completely overlapped on the AXI 4 interconnect due to its five-channel structure, the global command queue is implemented with 2 FIFOs, one for TX and one for RX transactions.

Two command unpack modules (one for TX operations and one for RX operations) splits the global commands incoming from the command queue into several commands and forward them to the interfaces, ensuring that they are fully compliant with AXI 4.0 protocol specifications. A command processed by the command unpack consists of a transfer implementing a subset of the operations supported by the DMA commands coupled with all the information necessary to perform the transfer on the AXI channels. According to AXI 4.0 specifications, these commands do not cross the 4K boundary, this means that if one command crosses the 4K boundary it is split into two smaller transfers, the latter aligned to 4K boundary. With the current configuration of the DMA (non-configurable 64-bit width AXI interface on the EXT unit), in accordance to the AXI 4.0 specifications, the maximum burst size is 2K (AXI max size = 8 bytes, AXI max len = 256).

1.3.2 Ext Unit

The EXT unit is responsible for managing the data transfers through the EXT interface through the 5 AXI channels: AW, AR, W, R, B. The ingoing commands from the CTRL unit are stored into TX and RX command queues. TX commands are processed by the EXT TX interface and sent to the cluster bus through the AW, W, and B AXI channels. RX commands are processed by the EXT RX interface and sent to the cluster bus through the AR and R AXI channels.

The EXT unit supports up to 16 out-of-order outstanding transactions on the AXI AW and AR channels. To implement the multiple out-of-order outstanding transaction mechanism, the EXT unit includes two tables one for RX transfers, and one for TX transfers (OPC buffers) that associate to each transaction request identified by a transaction identifier (TID) all the information required to handle the response packet. When a new request packet is issued a new TID is generated by two dedicated modules (TX and RX TID GEN) and all the information is stored in the OPC buffer. Once the response packet comes back from the interconnect, the control data associated to the response packet TID is fetched by the OPC buffer, and the TID is released to allow another request packet to be issued. The information stored in the OPC buffer required to forward the RX packets to the TCDM unit includes the address in TCDM, the SID and the opcode. The TX OPC MEM forwards the SID to the ctrl unit that back routes the termination event to the proper core interface (or global ctrl interface).

1.3.3 TCDM Unit

The TCDM unit is responsible for management of transfer toward the TCDM interface. The ingoing commands from the CTRL unit and EXT unit are stored into the TCDM TX and RX command queues, respectively.

The TCDM TX and RX CMD unpack blocks process the commands coming from the command queues in the required number of 32-bit beats, and forward the beats to two decoupled beat queues. During the execution of a burst transfer the cmd unpack blocks are responsible for the calculation of the address of the current beat depending on the required kind of monolithic transfer (e.g, linear, 2D, not incrementing transfer of arbitrary size). The main difference between packed and unpacked commands consists in the size of the transfer encoded into the opcodes, that can be arbitrary within the command queue, and it is forced to be one into the beat queues.

The TCDM interfaces collect the data to be processed from/to the transfer unit through 4 32-bit busses (two 32-bit busses for RX operations and 2 32-bit busses for TX operations) and the control information (address, SID) from the beat queue. All the interfaces are decoupled, thus stalls on the TCDM interface with index 0 do not affect the flow on beat queues with index 1, and vice versa.

The synchronization blocks present within the TCDM units are responsible for notifying to the control interface the termination of commands associated to an SID transported through the beat command queues. Depending on the ID the control unit back-routes the termination signal to the proper core interface (or global ctrl interface).

1.3.4 TRANS Unit

2 Integration Guide

As shown in Figure 1, MCHAN DMA channel targets the integration in a tightly coupled cluster of processors, featuring three interfaces toward the control and memory access busses of the cluster: CTRL interface, TCDM interface and EXT interface.

The control interface consist of a set of per-core slave ports that allows to access memory mapped register and FIFOs used to control the DMA. The control interface ports are compliant with TCDM interconnect protocol. The cores communicates to the DMA ctrl interface through a port mapped on the DEMUX, in order to guarantee low latency access to the DMA control registers. The TCDM interface consists of 4x32-bit master ports (2 for read accesses, 2 for write accesses), used to access the TCDM of the pulp cluster. The TCDM interface ports are compliant with the TCDM interconnect protocol. The EXT interface consists of one 64-bit AXI4 master port and connects the DMA to the PULP cluster bus. The EXT interface port is compliant with AXI 4.0 protocol specifications.

2.1 DMA Channel parameters

Table 1 summarizes the main configuration parameters of the DMA channel.

NB_CORES defines the number of cores attached to the DMA control interfaces. Hence, this parameter defines the number of control ports and private command queues instantiated.

RX_BUFFER_DEPTH and TX_BUFFER_DEPTH define the depth of the FIFOs present on the transfer unit. Although this parameter does not directly affect the performance of the DMA, it defines the number of

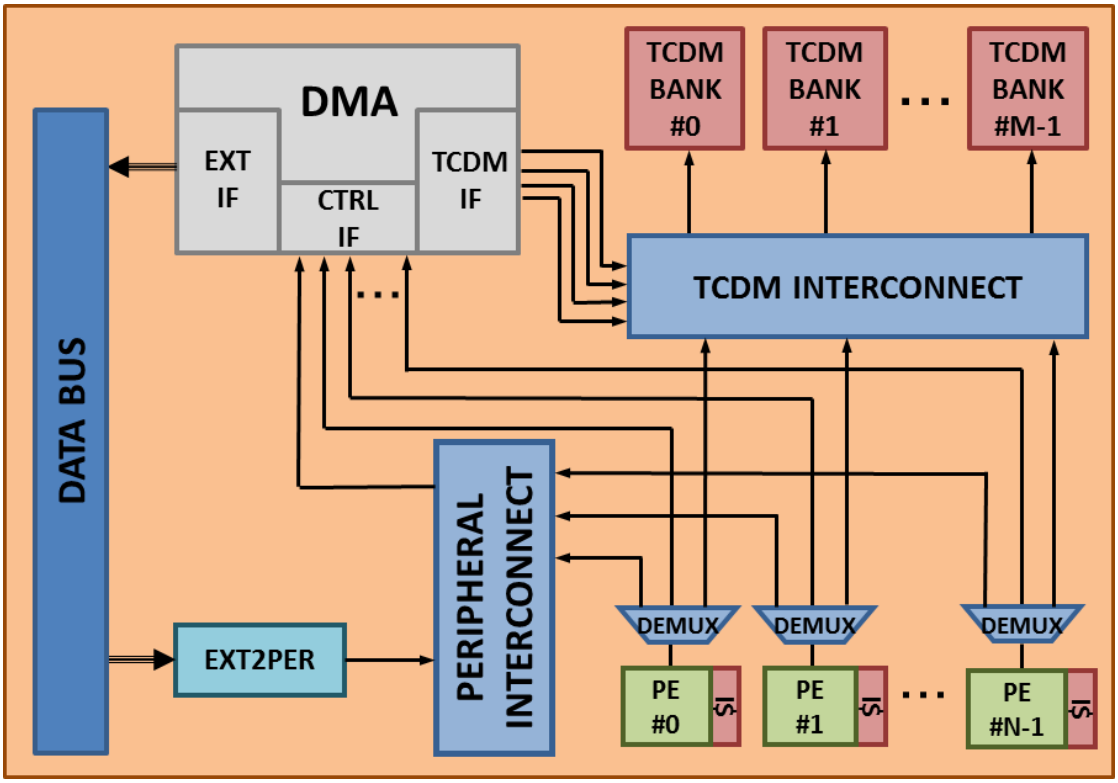

Figure 2: Integration of the MCHAN DMA on the PULP platform.

maximum number of stalls on the TCDM interface before the stalls propagation to the EXT interface and the maximum number of stalls on the EXT interface before the stalls propagation to the TCDM interface, respectively.

CORE_CMD_QUEUE DEPTH and GLOBAL_CMD_QUEUE_DEPTH define the depth of the private, per core command queue and of the global, shared command queue. These parameters defines the maximum number of commands that can be enqueued by a processor before being stalled from the control interface.

NB_OUTSTANDING_TRANS defines the maximum number of outstanding transaction on the EXT interface (on both AXI 4.0 AW and AR channels). This can be useful to hide the huge latencies of DDR memories.

TCDM_ADDR_WIDTH and EXT_ADDR_WIDTH defines the width of the TCDM and external address space. The size of the external address space is 32, as it is required to address 4 GB of global memory space, while TCDM_ADDR_WIDTH can be smaller, and depends on the actual size of the TCDM memory.

Parameter	Description	Valid values	Default value
NB_CORES	Number of DMA control ports	1 - 16	4
NB_TRANSFERS	Maximum number of transfer that can be enqueued on the core command queues	1 - 16	4
CORE_TRANS_QUEUE_DEPTH	Depth of core command queues	2 - 16	2
GLOBAL_TRANS_QUEUE_DEPTH	Depth of the global command queue	2 - 64	8
TCDM_ADDR_WIDTH	Width of TCDM interface address	1 - 32	16
EXT_ADDR_WIDTH	Width of external interface address	1 - 32	32
NB_OUTSND_TRANS	Number of outstanding transactions	1 - 16	16
MCHAN_BURST_LENGTH	Length of MCHAN burst [bytes]	64 - 2048	256
TWD_COUNT_WIDTH	Width of 2D transfers count	4-16	8
TWD_STRIDE_WIDTH	Width of 2D transfers stride	4-16	8

2.1 DMA Channel pinout

Port	Width	Direction	Description
Global Signals Bus			
clk_i	1	Input	Global Clock signal
rst_ni	1	Input	Global Reset signal
Control Slave Bus [NB_CORES-1:0]			
ctrl_targ_req_i	1	Input	Request
ctrl_targ_type_i	1	Input	Type: 0 = wr, 1 =rd
ctrl_targ_be_i	4	Input	Byte enable
ctrl_targ_add_i	32	Input	Address
ctrl_targ_data_i	32	Input	Input data
ctrl_targ_gnt_o	1	Output	Request granted
ctrl_targ_r_valid_o	1	Output	Output data valid
ctrl_targ_r_data_o	32	Output	Output data
TCDM Master data bus [3:0]			
tcdm_init_req_o	1	Output	Request
tcdm_init_type_o	1	Output	Type: 0 = wr, 1 =rd
tcdm_init_be_o	4	Output	Byte enable
tcdm_init_add_o	32	Output	Address
tcdm_init_data_o	32	Output	output data
tcdm_init_gnt_i	1	Input	Request granted
tcdm_init_r_valid_i	1	Input	Input data valid
tcdm_init_r_data_i	32	Input	Input data
AXI Master data BUS			
axi_master_aw_valid_o	1	Output	Write address valid
axi_master_aw_addr_o	32	Output	Write address
axi_master_aw_prot_o	3	Output	Write Protection type
axi_master_aw_region_o	4	Output	Write region
axi_master_aw_len_o	8	Output	Burst length
axi_master_aw_size_o	3	Output	Burst size
axi_master_aw_burst_o	2	Output	Burst type
axi_master_aw_lock_o	1	Output	Lock type
axi_master_aw_cache_o	4	Output	Cache type

Port	Width	Direction	Description
axi_master_aw_qos_o	4	Output	Quality of service value
axi_master_aw_id_o	4	Output	Write ID tag
axi_master_aw_user_o	6	Output	Write address channel user
axi_master_aw_ready_i	1	Input	Write address ready
axi_master_ar_valid_o	1	Output	Read address valid
axi_master_ar_addr_o	32	Output	Read address
axi_master_ar_prot_o	3	Output	Read Protection type
axi_master_ar_region_o	4	Output	Read region
axi_master_ar_len_o	8	Output	Burst length
axi_master_ar_size_o	3	Output	Burst size
axi_master_ar_burst_o	2	Output	Burst type
axi_master_ar_lock_o	1	Output	Lock type
axi_master_ar_cache_o	4	Output	Cache type
axi_master_ar_qos_o	4	Output	Quality of service value
axi_master_ar_id_o	4	Output	Read ID tag
axi_master_ar_user_o	6	Output	Read address channel user
axi_master_ar_ready_i	1	Input	Read address ready
w_valid_o	1	Output	Write valid
w_data_o	64	Output	Write data
w_strb_o	8	Output	Write strobes
w_user_o	6	Output	Write user signals
w_last_o	1	Output	Write last
w_ready_i	1	Input	Write ready
r_valid_i	1	Input	Read valid
r_data_i	64	Input	Read data
r_resp_i	1	Input	Read response
r_last_i	1	Input	Read last
r_id_i	4	Input	Read ID tag

Port	Width	Direction	Description
r_user_i	6	Input	Read user
r_ready_o	1	Output	Read ready
b_valid_i	1	Input	Write response valid
b_resp_i	1	Input	Write response
b_id_i	4	Input	Responde ID
b_user_i	6	Input	Write response user
b_ready_o	1	Output	Response ready
Termination Events Bus [NB_CORES-1:0]			
term_event_o	NB_CORES	Output	Termination Events

Table 2: MCHAN pinout description

3 Programming Manual

This chapter provides details of the sequences to program DMA transactions and get information about termination of transfers with MCHAN. MCHAN is programmed through a per-core private interface mapped on the processors DEMUX that allow to access two memory mapped locations described in Table 3. CONTROL FSM is a R/W memory location used to access the FSM that enables the access to MCHAN control registers, allowing to configure the DMA to implement the supported DMA transfers. STATUS REGISTER is a read-only register and provides information about the status of the enqueued transfers.

3.1 Programming FSM

The programming FSM, illustrated in Figure 3, consist of a 6 states finite state machine that manages the access to the control registers of MCHAN. Each state acknowledge the access to one of the MCHAN program registers:

- **IDLE**: MCHAN ctrl FSM is in an idle state. Either configuration of a new transfer or the query of the status register can be done. The status of the MCHAN transfers can be retrieved reading the STATUS REGISTER from the IDLE state (read operation at address 0x4). In the STATUS REGISTER the status information (1=pending, 0=done) of each transfer ID can be retrieved by reading the bit in the associated position of the register. The number of actual transfer IDs is implemented in hardware is bounded to the value of the parameter NB_TRANSFERS (Table 4). A

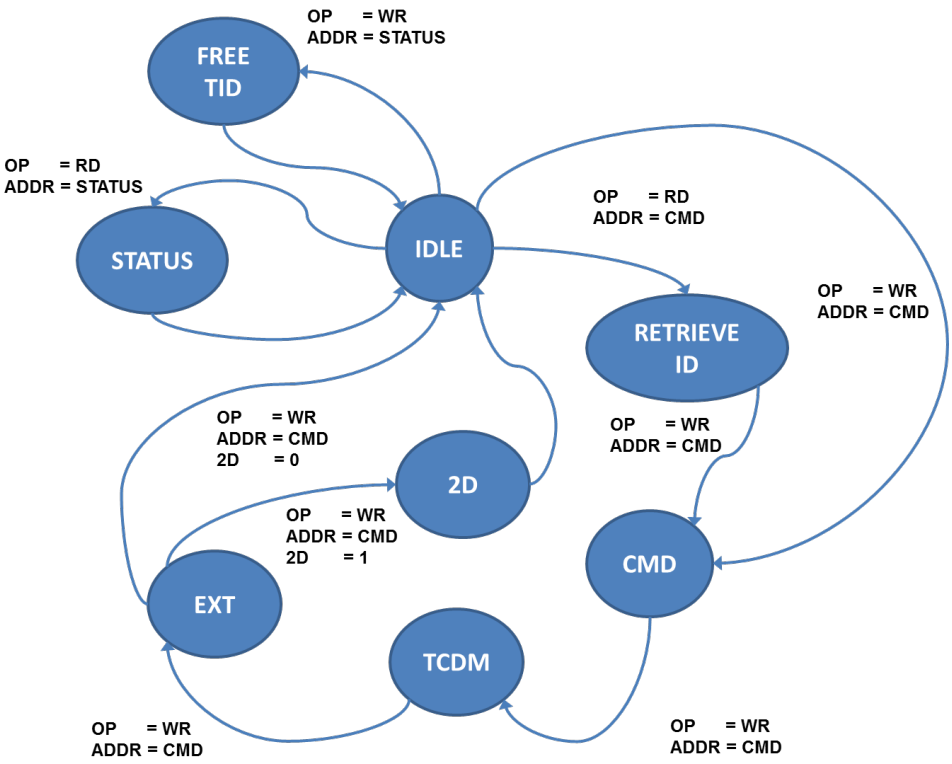

Figure 3: MCHAN CONTROL FSM

new transfer configuration can be initiated by retrieving the transfer id reading the CONTROL FSM at address 0x0 (Table 5).

- **ID ACK:** The cores have read a new transfer ID and initiated the configuration of a transfer with associated ID. In this state, the cores can push a new command in COMMAND REGISTER (Table 6) through a write operation at address 0x0.
- **CMD ACK:** The cores have pushed a new command to the COMMAND REGISTER. In this state, the cores can push the TCDM address of the transfer to the TCDM address register (Table 7) through a write operation at address 0x0.
- **TCDM ACK:** The cores have pushed the TCDM address of the transfer to the TCDM address register. In this state, the cores can push the EXR address of the transfer to the EXT address register (Table 8) through a write operation at address 0x0.
- **EXT ACK:** The cores have pushed the EXT address of the transfer to the EXT address register. If the 2D field of the COMMAND register was set to 1 (i.e. 2D transfer) during its configuration the cores can push the 2D information (i.e. count, stride) to the 2D register (Table 9) through a write operation at address 0x0. If the 2D field of the COMMAND register was set to 0 (i.e. linear transfer), the control FSM goes back to IDLE mode.
- **2D ACK:** The cores have pushed the 2D information of the transfer to the 2D register. The control FSM goes back to IDLE mode.

Register Name	Offset	Description
CONTROL FSM	0x00	Used to program DMA transfers
STATUS REGISTER	0x04	Contains status of DMA transfers

Table 3: MCHAN registers address map

Bit #	R/W	Description
31	R	Status of transfer allocator with TID 16 (1: reserved, 0: free)
30	R	Status of transfer allocator with TID 15 (1: reserved, 0: free)
…	R	
17	R	Status of transfer allocator with TID 1 (1: reserved, 0: free)
16	R	Status of transfer allocator with TID 0 (1: reserved, 0: free)
15	R	Status of transfer with TID 15 (1: active, 0: not active)
14	R	Status of transfer with TID 14 (1: active, 0: not active)
…	R	…
1	R	Status of transfer with TID 1 (1: active, 0: not active)
0	R	Status of transfer with TID 0 (1: active, 0: not active)

Table 4: Status register fields description (read operations)

Bit #	R/W	Description
31:16	W	**RESERVED**
15	W	Free transfer with TID 15
14	W	Free transfer with TID 14
…	W	…
1	W	Free transfer with TID 1
0	W	Free transfer with TID 0

Table 5: Status register fields description (write operations)

Bit #	R/W	Description
31:4	R	**RESERVED**
3:0	R	**TID:** Transfer Identifier

Table 6: TID register fields description

Bit #	R/W	Description
31:22	W	**RESERVED**
21	W	**BLE:** Broadcast lines enable 0= disabled. Events or interrupts are routed to the core who initiated the transfer. 1=enabled. Events or interrupts are broadcasted.
20	W	**ILE:** Interrupt lines enable (0= disabled, 1= enabled)
19	W	**ELE:** Event lines enable (0= disabled, 1= enabled)
18	W	**2D:** 2D transfer (0=linear transfer; 1=2D transfer)
17	W	**INC:** Incremental transfer (0=not incremental; 1=incremental)
16	W	**TYP:** Transfer type (0= from TCDM to L2; 1= from L2 to TCDM)
15:0	W	**LEN:** Length of transfer

Table 7: Command register fields description

Bit #	R/W	Description
31:0	W	**TCDM ADDR:** TCDM Address

Table 8: TCDM address register fields description

Bit #	R/W	Description
31:0	W	**EXT ADDR:** External Address

Table 9: External address register fields description

Bit #	R/W	Description
31:16	W	**STRIDE:** Stride value for 2D transfers

Bit #	R/W	Description
15:0	W	**COUNT:** Count value for 2D transfers

Table 10: 2D register fields description

3.2 Transaction IDs management

The transaction queue has a number of entries that are retrieved during phase 1 of the DMA programming (ID is by the hardware) and explicitly enqueued.

If we are in idle and there is no read operation the core will push another transfer to the same ID.

When the core finished the transfers to a specified ID the core will explicitly free the ID.

3.3 Supported transfers

…

3.3.1 Linear transfers

…

3.3.2 2D transfers

…

3.3.3 Non-incrementing transfers

…

3.4 Interrupts and events

…

```
#define MCHAN_COMMAND_QUEUE   ( MCHAN_BASE_ADDR + 0x0 )
#define MCHAN_STATUS_REGISTER ( MCHAN_BASE_ADDR + 0x4 )

#define MCHAN_LEN_WIDTH 16

#define TX 0x0
#define RX 0x1

#define FIX 0x0
#define INC 0x1

#define LIN 0x0
#define TWD 0x1
int mchan_transfer(unsigned int len, char type, char incr, char twd, unsigned int ext_addr, unsigned int tcdm_addr, unsigned short int count, unsigned short int
stride){
  int id = *(volatile int*) MCHAN_COMMAND_QUEUE;
  *(volatile int*) MCHAN_COMMAND_QUEUE = len | (type << MCHAN_LEN_WIDTH) | ( incr <<  (MCHAN_LEN_WIDTH + 1) ) | ( twd <<  (MCHAN_LEN_WIDTH
+ 2) );
  *(volatile int*) MCHAN_COMMAND_QUEUE = tcdm_addr;
  *(volatile int*) MCHAN_COMMAND_QUEUE = ext_addr;
  if (twd == 1)
```

```
    *(volatile int*) MCHAN_COMMAND_QUEUE = count | (stride << 16);
  return id;
}

#define mchan_barrier(id)                                                    \
  while(((*(volatile int*)(MCHAN_STATUS_REGISTER)) >> id ) & 0x1 );
```