



## Clock domain crossing modules for OCP-style read/write interfaces

Herlev, Mathias; Sparsø, Jens

*Publication date:*  
2016

*Document Version*  
Publisher's PDF, also known as Version of record

[Link back to DTU Orbit](#)

*Citation (APA):*  
Herlev, M., & Sparsø, J. (2016). Clock domain crossing modules for OCP-style read/write interfaces. Kgs. Lyngby: Technical University of Denmark (DTU). DTU Compute-Technical Report-2016, No. 4

---

### General rights

Copyright and moral rights for the publications made accessible in the public portal are retained by the authors and/or other copyright owners and it is a condition of accessing publications that users recognise and abide by the legal requirements associated with these rights.

- Users may download and print one copy of any publication from the public portal for the purpose of private study or research.
- You may not further distribute the material or use it for any profit-making activity or commercial gain
- You may freely distribute the URL identifying the publication in the public portal

If you believe that this document breaches copyright please contact us providing details, and we will remove access to the work immediately and investigate your claim.

# Clock domain crossing modules for OCP-style read/write interfaces

Mathias Herlev and Jens Sparsø

Department of Applied Mathematics and Computer Science  
Technical University of Denmark

Email: s103006@student.dtu.dk, jspa@dtu.dk

April 29, 2016

DTU Compute Technical Report-2016-4.

ISSN: 1601-2321

## Abstract

The open core protocol (OCP) is an openly licensed, configurable, and scalable interface protocol for on-chip subsystem communications. The protocol defines read and write transactions from a master towards a slave across a point-to-point connection and the protocol assumes a single common clock.

This paper presents the design of two OCP clock domain crossing interface modules, that can be used to construct systems with multiple clock domains. One module (called OCPio) supports a single word read-write interface and the other module (called OCPburst) supports a four word burst read-write interface.

The modules has been developed for the T-CREST multi-core platform [8, 9, 13] but they can easily be adopted and used in other designs implementing variants of the OCP interface standard. The OCPio module is used to connect a Patmos processor to a message passing network-on-chip and the OCPburst is used to connect the Processor and its cache controllers to a shared off-chip memory.

While the problem of synchronizing a simple streaming interface is well described in the literature and often solved using bi-synchronous FIFOs we found surprisingly little published material addressing synchronization of bus-style read-write transaction interfaces. An OCP interface typically has control signals related to both the master issuing a read or write request and the slave producing a response. If all these control signals are passed across the clock domain boundary and synchronized it may add significant latency to the duration of a transaction. Our interface designs avoid this and synchronize only a single signal transition in each direction during a read or a write transaction. The designs are available as open source, and the modules have been tested in a complete multi-core platform implemented on an FPGA board.

# Contents

<b>Preface</b>	<b>ii</b>
<b>Acknowledgements</b>	<b>iii</b>
<b>1 Introduction</b>	<b>1</b>
<b>2 Background and related work</b>	<b>3</b>
<b>3 OCP transactions in T-CREST</b>	<b>5</b>
<b>4 Designs</b>	<b>8</b>
4.1 The OCPio CDC-module . . . . .	8
4.2 The OCPburst CDC-module . . . . .	11
<b>5 Latency of a transaction</b>	<b>15</b>
5.1 Analysis . . . . .	15
5.2 Worst-case and best-case bounds . . . . .	18
<b>6 Implementation</b>	<b>19</b>
<b>7 Conclusion</b>	<b>21</b>
<b>References</b>	<b>23</b>
<b>A Code</b>	<b>24</b>
A.1 OCPio . . . . .	24
A.1.1 OCPio A side . . . . .	24
A.1.2 OCPio B side . . . . .	28
A.2 OCPburst . . . . .	32
A.2.1 OCPburst A side . . . . .	32
A.2.2 OCPburst B side . . . . .	37
A.3 Packages . . . . .	41
A.3.1 OCP Types . . . . .	41
A.3.2 OCPburst CDC Types . . . . .	43
A.3.3 OCPio CDC Types . . . . .	44

# Preface

The work presented in this report started as a small project in course 02204 Design of Asynchronous Circuits in the spring 2014. The work was continued in a special topics course during the summer and this resulted in a paper presented at the Norchip 2014 conference in Helsinki [5]. This paper presented two alternative designs of a module implement clock domain crossing of a single-word read-write transaction – a version that buffers address and data and a version that avoids such buffering.

Following the publication of [5], we continued the work and expanded with a module implementing implement clock domain crossing burst read-write transaction. In addition abandoned the unbuffered single-word design and eliminated a potential timing glitch/bug in the buffered design. In this report, we present the final designs of the single-word transaction and the burst transaction clock domain crossing modules, and we provide an analysis of the latency of the read and write transactions as seen from the master. Both designs have been extensively tested in the T-CREST multicore platform, and the source code is available as open source.

# Acknowledgements

This work was partially funded by the Danish Council for Independent Research | Technology and Production Sciences, Project no. 12-127600: Hard Real-Time Embedded Multiprocessor Platform (RTEMP) project.

# Chapter 1

## Introduction

The design of Systems-on-Chip (SoCs) where billions of transistors are integrated in a single chip puts emphasis on modularity and reuse of components. Components like processors, cache memories, IO-units, and HW-accelerators are typically developed by different vendors and are collectively known as intellectual property cores (IP-cores). To support reuse and modular composition of such IP-cores, interfaces are of paramount importance and a number of interface standards have emerged. Three typical and widely used interfaces are Wishbone, Open Core Protocol (OCP), and AMBA-AXI, introduced in more detail in the next chapter. They all specify a point-to-point connection that offers read and write transactions from a master (M) towards a slave (S), and they all assume synchronous operation of the master and the slave.

Another important aspect of designing SoCs is the timing organization. IP-cores may be designed for different clock frequencies and, to save power, scaling of the clock frequency, and the supply voltage, is often employed at the level of individual IP-cores. In addition the timing uncertainty in today's sub-micron CMOS technologies make clock-distribution and globally synchronous operation practically impossible [12]. As a result SoCs typically use some form of globally-asynchronous locally-synchronous operation [2].

In this report we address the design of two clock domain crossing interface modules (in the following denoted CDC-modules), that each implement a distinct subset of the

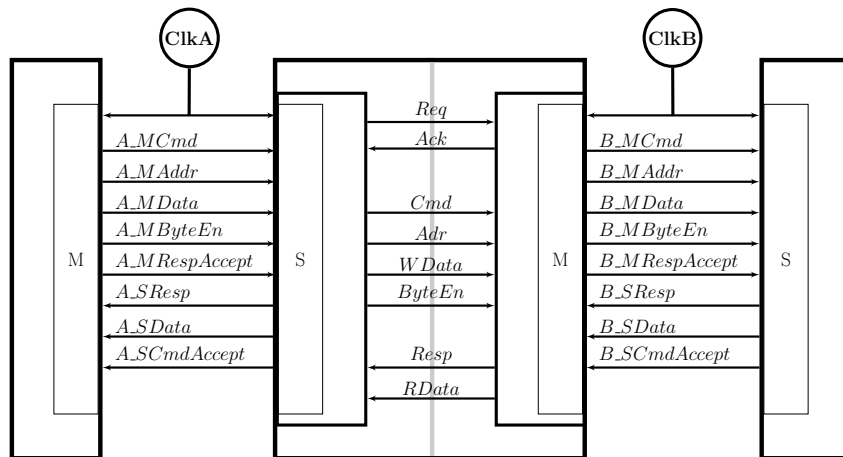


Figure 1: Block diagram of the OCP-to-OCP clock domain crossing module.

OCP, see Figure 1. The CDC-modules have been developed for the T-CREST multicore platform [13]. One CDC-module supports single-word transactions and the other CDC-module supports four-word burst transaction. Our designs synchronize only a single signal in each direction for a complete transaction. In this way, the performance impact of synchronization is reduced to the minimum possible.

In [5] we presented two designs for the single-word transaction CDC-module; one design using buffer-registers for all signals and another slightly slower but minimum hardware solution that avoids buffering of address and data signals. It turned out that the former design suffered from a timing glitch problem. Fixing this problem increased the latency by one cycle in each direction. As the use of buffer registers simplify timing analysis by breaking signal paths directly from one clock domain to the other, our preferred designs use buffer registers. In this report, we extend the work with a CDC-module that supports four-word burst transactions. In addition, we made a minor improvement of the single-word CDC-module. The aim of this report is to document the final designs of the two CDC-modules developed for and used in the T-CREST platform. The report is largely based on and reusing material from [5]. The designs have been tested in platform with four processor nodes implemented on an FPGA-board. The code is open sourced under a simplified (2-Clause) BSD license, and is available on Github [11], as well as being listed in Appendix A.

The report is organized as follows. Chapter 2 presents background material and related work. Chapter 3 presents the specific OCP interfaces that we use. Chapter 4 presents the design of the clock domain crossing module. Chapter 6 presents results of Field Programmable Gate Array (FPGA) realization. Finally, Chapter 7 concludes the report.



# Chapter 2

## Background and related work

The Open Core Protocol (OCP) is an openly licensed interface originally developed by the OCP International Partnership organization and now maintained by the Accelerata Systems Initiative [1]. The “Wishbone System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores” – in short the Wishbone bus – is an open source hardware interface that is used by many designs in the OpenCores project [6]. Both Wishbone and OCP specify signals and protocols for point-to-point connections between a master and a slave, allowing a master to perform read or write transactions into the address space of the slave. Wishbone specifies a single bus standard while OCP is highly configurable and scalable. Typical instances of OCP are very similar to the wishbone protocol. The Advanced Microcontroller Bus Architecture, Advanced eXtensible Interface (AMBA-AXI) is a bus developed by ARM Inc. It uses separate channels for address, write data and read data while OCP and Wishbone are more conventional bus-style interfaces.

Common to the three interface standards mentioned above is that they all assume a single common clock. In multi-clock systems there is a need for clock domain crossing modules that implement the same interface on both sides except for the different clocks.

There is a rich body of literature addressing communication between different clock domains and the problems related to synchronization and metastability are well understood [3, Ch. 10] [4]. Common to all forms of clock domain crossing is that synchronization of a signal (by passing it through two or more flip-flops) incurs latency.

Most published solutions consider a simple streaming interface between a producer and a consumer. A commonly used solution when connecting a producer and a consumer is to use a bi-synchronous FIFO that provide full and empty signals that are synchronized to the producer and receiver clocks respectively [3, Ch. 10] [4] [7] [15].

A CDC-module for bus-style transactions like OCP is a more challenging design than for a simple streaming interface. Read or write transactions are typically atomic and blocking and use flow control signals related to the transmission of both: (a) command and address, (b) write-data and, (c) read-data. The blocking behavior means that transactions cannot be pipelined and the accumulated latency of synchronizing several flow control signals may be significant.

We have only been able to find few publications that address clock domain crossing between two (bus-style) read/write-transaction interfaces. Closest to our work is [14] [16] [10]. Common to these designs are the use of several bi-synchronous FIFO’s, for example for address and data, for write data and for read data. Furthermore both [14] and [10]

consider the interfacing between a synchronous domain and an asynchronous domain.

Our designs connect two identical clocked interfaces driven by independent clocks, and our designs avoid the use of FIFO-based synchronizers resulting in very small and efficient hardware implementations.

# Chapter 3

## OCP transactions in T-CREST

The OCP standard [1] allows for a large variety of specific point-to-point master-to-slave protocol instances. The context for the work presented in this paper is T-CREST multi-core processor [8, 13]. In this design two instances of the OCP protocol using the set of signals shown in Table 1 are used [9]:

1. **OCPio**; a single-word read-write interface used to access memories and IO-devices connected to a processor node. Transactions on this interface are generated by load and store instructions executed by the processor. Some example transactions are shown in Figure 2.
2. **OCPburst**; a burst read-write interface used to access a shared memory. Transactions on this interface are initiated by the cache controller in the processor node. The burst size is fixed to blocks of four words. A burst is identified by an aligned address and transferred as an immediate succession of words from incrementing addresses. Some example transactions are shown in Figure 3.

Table 1: Signals in the **OCPio** and **OCPburst** interfaces.

Signal	Description
MCmd[2:0]	Command (idle, read or write)
MAddr[31:0]	Address, byte-based, MAdr[1:0] always 00
MData[31:0]	Data for writes
MDataValid *)	Signal that write data is valid.
MDataByteEn[3:0]	Byte-level mask for sub-word writes
MRespAccept	Signal that read data is accepted
SCmdAccept	Signal that command is accepted
SDataAccept *)	Signal that write data is accepted.
SData[31:0]	Data for reads
SResp[1:0]	Slave response: NULL - No response DVA - Data valid/accept FAIL - Request failed ERR - Response error

\*) only implemented in the **OCPburst** interface.

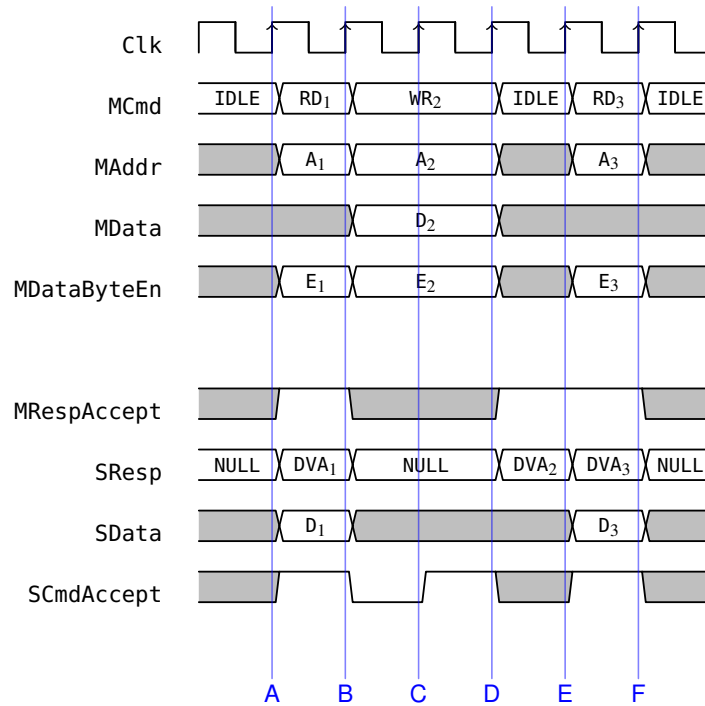


Figure 2: Timing diagram for the OCPio interface. (Reprinted from [9] with kind permission of its authors).

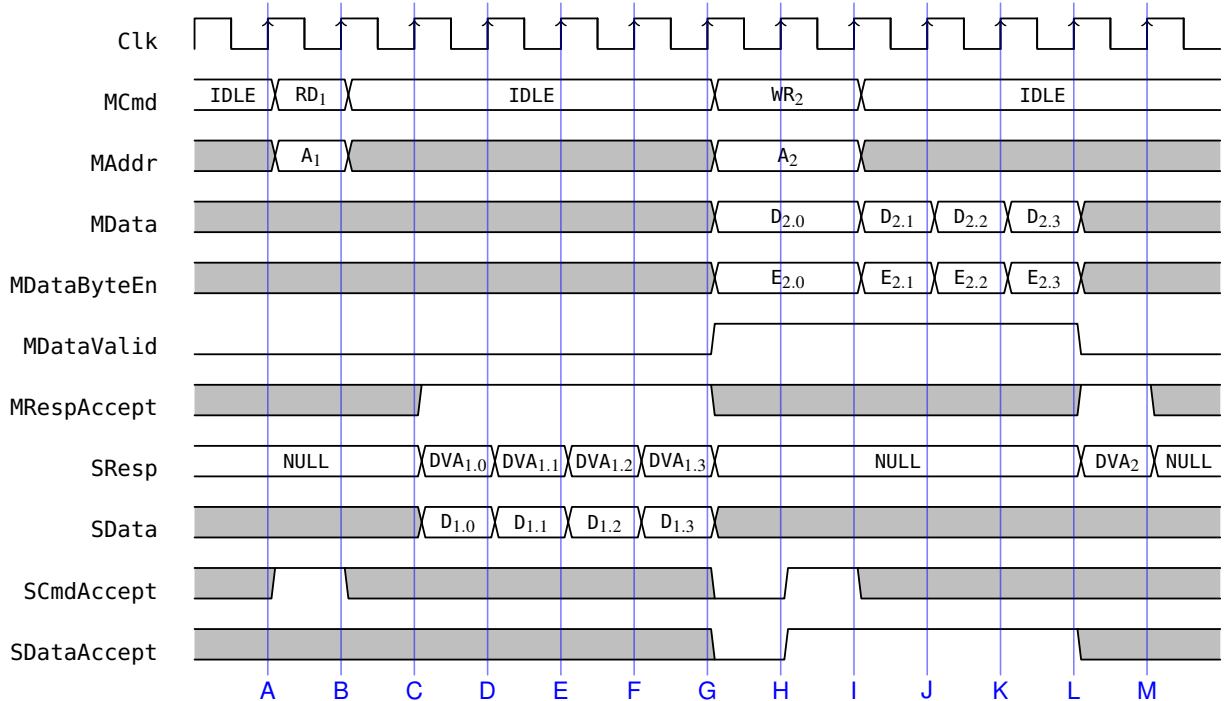


Figure 3: Timing diagram for the OCPburst interface. (Reprinted from [9] with kind permission of its authors).

Both interfaces provide elasticity at the clock-cycle level and include several flow-control signal pairs: **MCmd** and **SCmdAccept** that control the transfer of the command and also write data for the **OCPio** interface; **MDataValid** and **SDataAccept** that control the transfer of write data for the **OCPburst** interface; **SResp** and **MRespAccept** that control the transfer of read data. **SResp** also serves the purpose of indicating the correct or faulty conclusion of a complete (read or write) transaction. Thus, for both **OCPio** and **OCPburst**, a write is terminated by a response from the Slave. More details can be found in the handbook for the PATMOS processor [9].

# Chapter 4

## Designs

The designs for both the OCPio and the OCPburst CDC-modules are based the same key principles, and the OCPburst CDC-module can be seen as an extension of the OCPio design. In this chapter, we first introduce the design of the OCPio CDC-module and the underlying principles and we then we present the design of the OCPburst CDC-module.

### 4.1 The OCPio CDC-module

The design of the OCPio CDC-module is a lightly modified version of the buffered design we presented in [5]. The design can be seen in Figure 4. As depicted in figure 4 the design consist of two parts each controlled by a finite state machine, denoted **FSM\_A** and **FSM\_B**. These FSMs are responsible for the signaling on the interfaces towards the OCP master and the OCP slave. The FSMs interact and coordinate their operation using two signals **Req** and **Ack** that are synchronized using a pair of flip-flops in the destination clock-domain. To minimize the latency of a transaction (as seen from the master) it is important to minimize the number of signal events that needs to be synchronized. Our design involves only one event (signal transition) on the **Ack** signal and one event on the **Req** signal per OCP transaction. In this way, the **Req** and **Ack** signals implement a non-return-to-zero (NRZ) or two-phase handshake per transaction. The third flip flop and the exclusive-or gate involved in the **Req** and **Ack** handshaking converts the (synchronized) signal transitions into pulses with a duration of one clock period.

OCP-signals are buffered in the source side and loading of a buffer registers is controlled by the FSM. The and-gates driving signals **B\_MCmd[2:0]** allow **FSM\_B** to drive these signals low until the synchronized signal **Req\_event** indicate that a transaction is in progress. In the same manner, **FSM\_B** can drive signal **A\_SResp[1:0]** low until the synchronized signal **Ack\_sync** indicate that a response is ready.

The detailed operation of the design is determined by the two FSMs. Figure 5 shows ASM-charts for **FSM\_A** and **FSM\_B**.

When a command is issued on the **A\_MCmd** signal, the control FSM is in the Idle state. Upon receiving the command, it stores the signals **A\_MCmd**, **A\_MAddr**, **A\_MData**, and **A\_MByteEn** in registers, and asserts the **req** on the next rising clock edge of **clkA**. Upon this, the **FSM\_A** controller will transition to state **AckWait**. On a **req\_event** the **FSM\_B** controller asserts **EnB**, and waits for the Slave to accept and respond. If the slave does not accept immediately, the **FSM\_B** controller moves into state **CmdAcceptWait**. If, when in

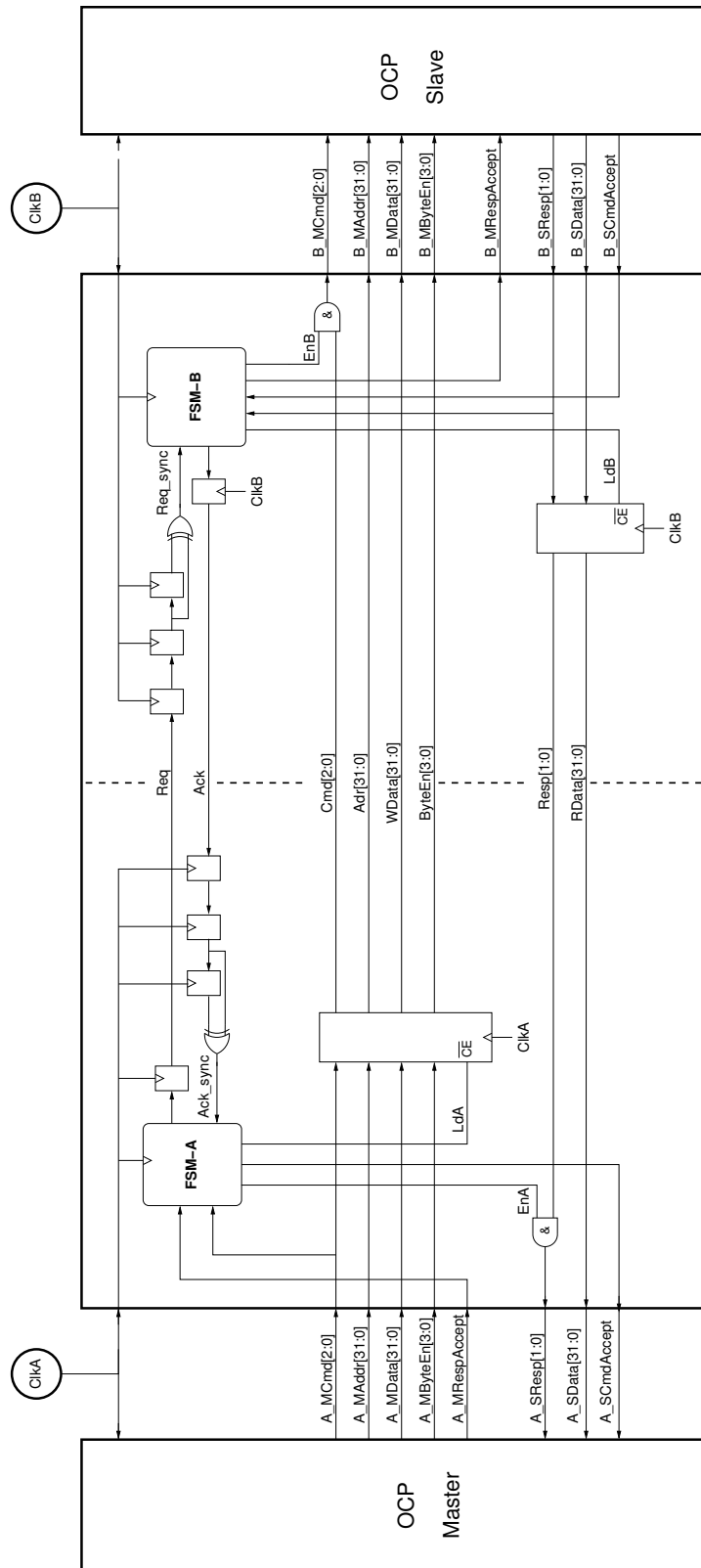


Figure 4: Diagram showing the implementation of the OCPio CDC-module.

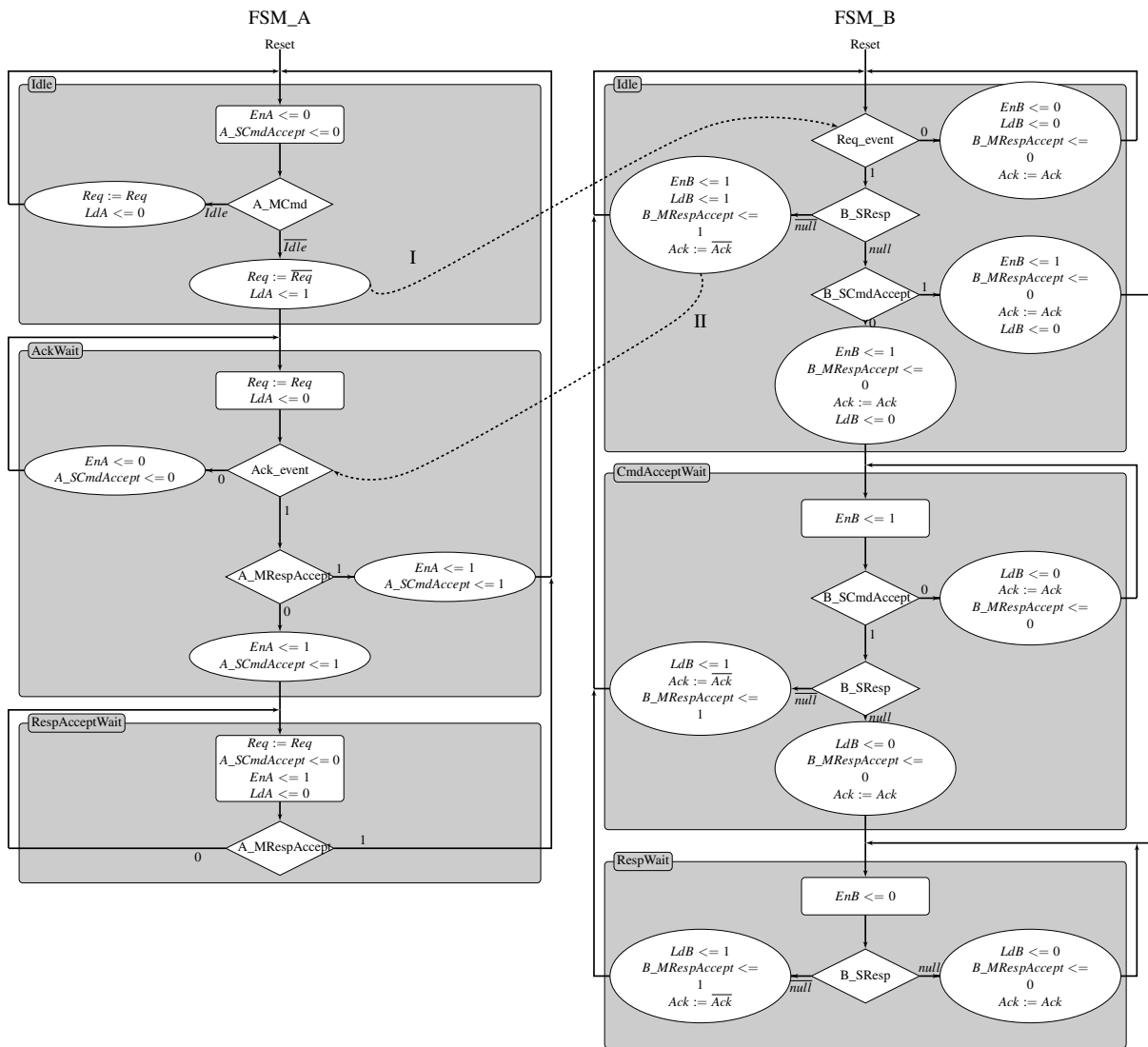


Figure 5: OCPio ASM Chart. The dashed arrows show a request (I) and an acknowledge (II) handshake.



**CmdAcceptWait**, the slave accepts but does not provide a response, the **FSM\_B** controller will transition into **RespWait**. At any point, when the slave does provide a response, said response will be stored, and any accompanying data, in the register, by asserting **LdB**. When the response has been stored, the controller will assert **Ack**. When the **Ack** has been synchronized, the **FSM\_A** controller will assert **EnA**, and await a response accept. If the OCP Master does not accept immediately, the controller transitions to **RespAcceptWait**. When the response has been accepted, the **FSM\_A** controller transitions back to **Idle**, ready to receive a new command.

## 4.2 The OCPburst CDC-module

The design of the OCPburst CDC-module is an extension of the OCPio design. The design can be seen in Figure 4. The concept of treating all multibit signals as data still stands for this design. The values of the **A\_MCmd** and **A\_MAddr** signals are the same for an entire transaction, as opposed to for example the **A\_MData** signal, and thus only require only a single register. For the signals **A\_MData** and **A\_MDataByteEn** (on the A-side) and **B\_SResp** and **B\_SData** (on the B-side) the design uses a register-file of the same size as the burst. In addition, each side has a register to keep track of how many words have been read or written.

The registers on the A-side are clocked using **clkA**, while the registers on the B-side are clocked using **clkB**. Reading from the other side of the interface is a combinatorial and asynchronous operation.

The state machines for the controllers in the OCPburst CDC-module can be seen in Figure 7. Unlike the OCPio design where the behavior is independent of the command type (read or write), both A-side and B side in the OCP burst design is dependent on whether it is a read or a write command.

**Read** Upon a read command the **FSM\_A** controller, will assert the **LdA** signal to register the "data", and on the rising clock edge drive a signal transition of the **req** signal, and transition to the state **A\_ReadBlockWait**. Once the **req** signal has been synchronized to B side, generating a **req\_event** the **FSM\_B** controller asserts **EnB** to allow the signals to pass through to the OCP Slave. If the Slave accepts immediately, the **FSM\_B** controller transitions to the **B\_ReadBlock** state. In case it does not accept immediately, it transitions to **B\_ReadBlockWait**, until the command has been accepted. If the command has been accepted, the **FSM\_B** controller will also check if the first response is provided, and increment the **RegAddr**. Every time a response is available the **RegAddr** is incremented. When the third response has been stored (and **RegAddr** is reset) **ack** is asserted. Once **ack** has been synchronized, generating an **ack\_event**, the **FSM\_A** controller asserts **A\_SCmdAccept**, and transitions into **A\_ReadBlock**, where, one by one, the responses stored on the B side are transmitted to the OCP Master.

**Write** If instead the master issues a write command, the **FSM\_A** controller immediately asserts **A\_SCmdAccept**, and begin buffering the data in the register files by transitioning into **A\_WriteBlock**. Once **RegAddr** reaches 3, the **FSM\_A** controller asserts a request, and transitions into **A\_WriteBlockWait**. Once the **req** has been synchronized, generating a

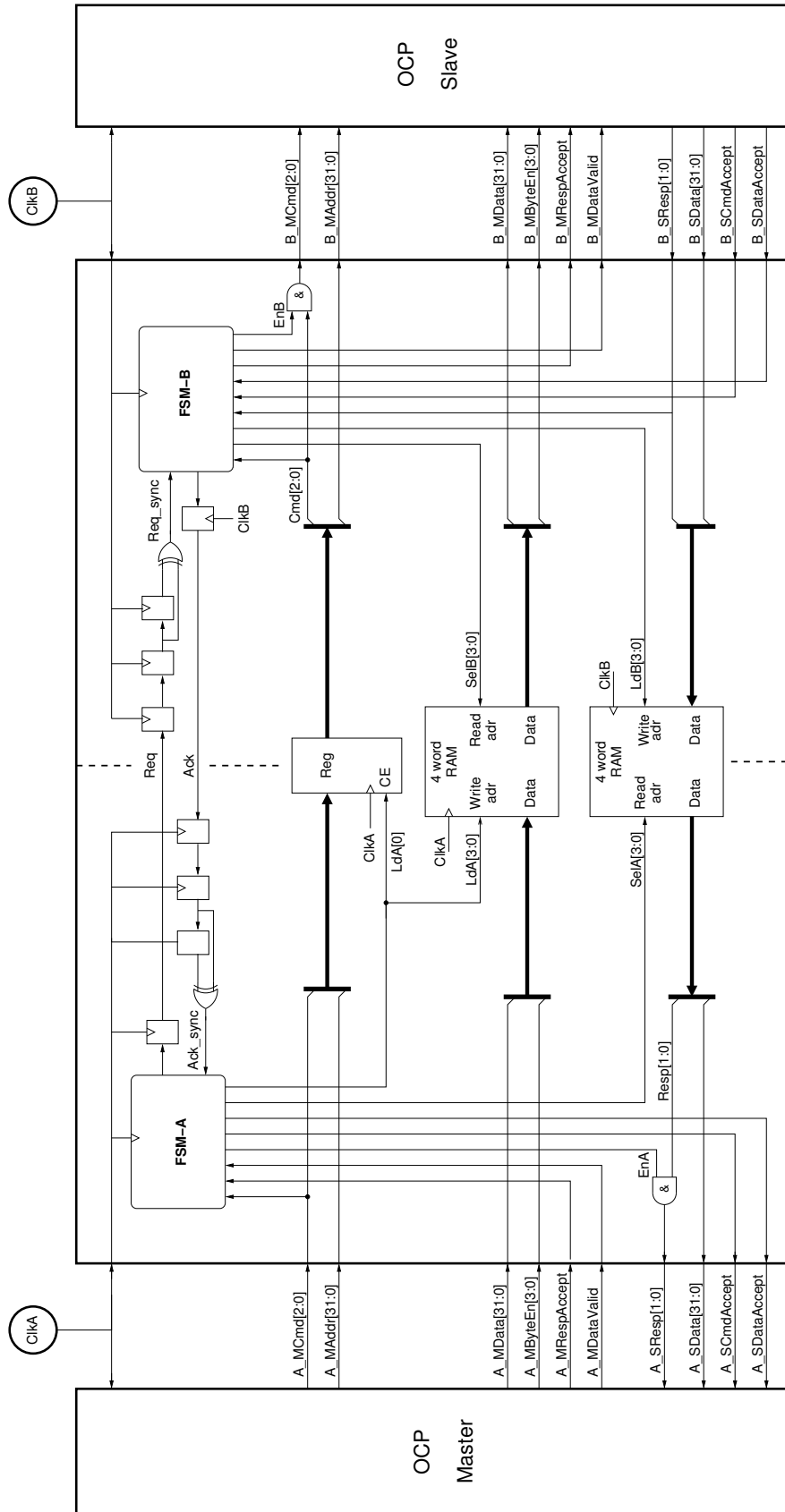


Figure 6: Diagram showing the implementation of the OCPburst CDC-module.

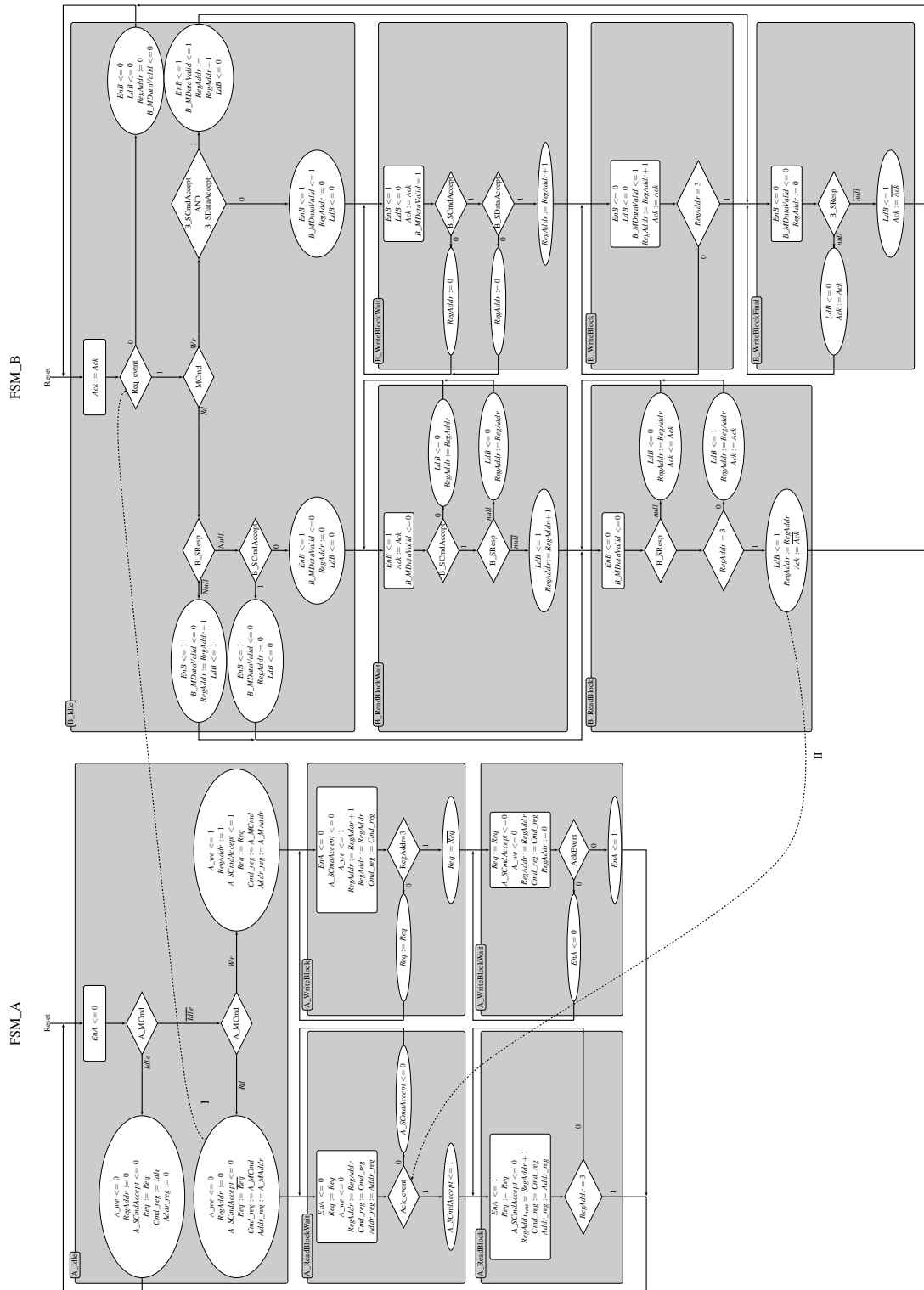


Figure 7: OCPburst ASM Chart. The dashed arrows shows a request (I) and acknowledge (II) handshake

`req_event`, the `FSM_B` controller asserts `EnB` allowing the command to pass through to the OCP Slave. If the slave accepts, it transitions directly to `B_WriteBlock` and transmits each buffered word. Otherwise, it waits in the state `B_WriteBlockWait`. Once all words have been transmitted it transitions into `B_WriteBlockFinal`. In `B_WriteBlockFinal` the `FSM_B` controller waits until the Slave responds, upon which it saves the response, in the response register file, and asserts `ack`. When `ack` has been synchronized, the `FSM_A` controller asserts `EnA`, for a single cycle and transitions back to `A_Idle`.

# Chapter 5

## Latency of a transaction

In this section we analyze latency of the OCP transactions as seen by a master that performs a read or write towards a slave that sits on the other side of a clock domain crossing module. We first analyze the factors that contribute to this latency and then use this analysis to provide expressions for the worst-case latency of the different OCP-transactions.

### 5.1 Analysis

The latency depends on the type of interface (OCPburst or OCPio), the type of transaction (read or write), the ratio between the two clock signals (**ClkA** and **ClkB**) and the phase between the two clock signals when a signal from one clock domain is synchronized to the other domain. Furthermore, it should be kept in mind (c.f. chapter 3) that the OCPburst and OCPio protocols allow a slave to idle between a command and the associated response and allow a master to idle between a response and the acknowledgement of the response.

The latency of a transaction can be subdivided into 5 intervals. Since the different transactions ( OCPburst read, OCPburst write, OCPio read and OCPio write) are relatively similar the following analysis is structured according to the 5 intervals. The reader is encouraged to refer to Figure 4 and Figure 6 while reading the descriptions.

**Interval [a]:**  $MCmd \neq idle \rightarrow Req \updownarrow$

The time from **AMCmd** changes from **Idle** to a valid command until a transition is produced on signal **Req**. Everything happens in the **ClkA** domain and the latencies for the different transactions are:

OCPio	Wr:	1 cycle	@ClkA
	Rd:	1 cycle	@ClkA
OCPburst	Wr:	$N$ cycles	@ClkA
	Rd:	1 cycle	@ClkA

where  $N$  is the number of words in a burst transfer.

**Interval [b]:**  $Req \updownarrow \rightarrow Req\_event \up$

The time from an event (an up or down-going transition) on **Req** until the beginning

of the synchronized one-cycle wide pulse that is produced on signal `Req_event`. The time is related to `ClkB` and also depends on the phase difference between `ClkB` the `Req` signal that is produced in the `ClkA` domain. If the transition of `Req` happens slightly before the rising edge of `ClkB` the duration of interval [b] is slightly more than 1 cycle @`ClkB` and if the transition happens slightly after the rising edge of the duration of interval [b] is slightly less than 2 cycles @`ClkB`. If the transition coincide with the rising edge of `ClkB`, the first flip-flop in the two flop synchronizer goes metastable, and the resulting duration of interval [b] is one or two cycles corresponding to the two previously mentioned scenarios.

In summary, the duration of interval [b] is ]1;2[ cycles @`ClkB`.

**Interval [c]:** `Req_event`  $\uparrow \rightarrow$  `Ack`  $\downarrow$

The time from a synchronized `req_event` until an event (an up or down-going transition) on signal `Ack`. This includes the time it takes for the OCP slave to respond to the transaction, and the time required to buffer the response. Everything relates to `ClkB` and for each of the 4 possible transactions the duration of interval [c] and is as follows:

OCPio	Wr:	$1 + n_S$ cycles	@ <code>ClkB</code>
	Rd:	$1 + n_S$ cycles	@ <code>ClkB</code>
OCPburst	Wr:	$1 + N + n_S$ cycles	@ <code>ClkB</code>
	Rd:	$1 + N + n_S$ cycles	@ <code>ClkB</code>

where  $N$  is the number of words in a burst transfer and where  $n_S$  is the accept/response time of the slave. For the OCPio write command illustrated in Figure 2  $n_S = 1$ ; the cycle from B to C. For the OCPburst write command illustrated in Figure 3  $n_S = 1$ ; the cycle from G to H. For the OCPio write command illustrated in Figure 3  $n_S = 1$ ; the cycle from B to C. In our current implementation  $N = 4$  and  $n_S = 0$ .

**Interval [d]:** `Ack`  $\downarrow \rightarrow$  `Ack_event`  $\uparrow$

The time from an event (an up or down-going transition) on `Ack` until the beginning of the synchronized one-cycle wide pulse that is produced on signal `Ack_event`. The analysis is similar to interval [b] and the duration of interval [d] is ]1;2[ cycles @`ClkA`.

**Interval [e]:** `Ack_event`  $\uparrow \rightarrow$  Transaction completed (FSM\_A enters state Idle)

The time from a synchronized `req_event` until the A-side of the CDC-module is ready to receive a new command. This includes the time it takes to write buffered responses (including data) back to the OCP Master, the time it takes the master to accept a response, and any latency added by the `FSM_A`. For the OCPio transactions it takes a minimum of 1 cycle for the master to accept the response. However, it can delay for an unspecified number of cycles  $n_M$ . For the OCPburst transactions such delaying is not allowed. Everything relates to `ClkA` and for each of the 4 possible transactions the duration of interval [e] and is as follows:

OCPio	Wr:	$1 + n_M$ cycles	@ClkA
	Rd:	$1 + n_M$ cycles	@ClkA
OCPburst	Wr:	1 cycle	@ClkA
	Rd:	$N$ cycles	@ClkA

where  $N$  is the number of words in a burst transfer and where  $n_M$  is the delay between the response and the acknowledgement of the response that is allowed for the OCPio transactions. In our current implementation  $N = 4$  and  $n_M = 0$ .

Figure 8 shows an example OCPio Read transaction propagated across the OCPio CDC-module. The set of signals shown is reduced to improve readability.

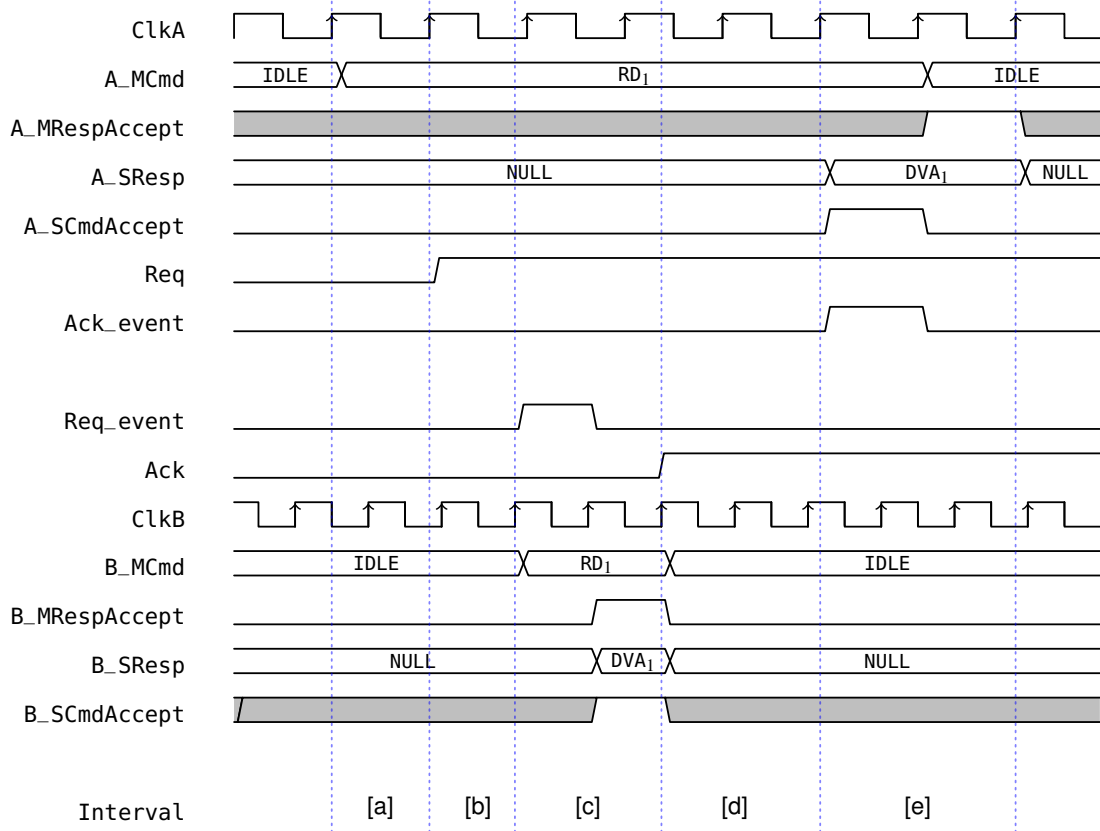


Figure 8: An example read-transaction propagated across the OCPio CDC-module.

## 5.2 Worst-case and best-case bounds

In the following  $T_A$ ,  $f_A$ ,  $T_B$  and  $f_B$  denote the period and frequency of **ClkA** and **ClkB** respectively. The latency of a transaction as seen from the master (operating at **ClkA**) is the sum of intervals [a] through [e]. Assuming for intervals [b] and [d] the maximum synchronization latency of 2 cycles we get the following sum for the OCPio write transaction.

$$T_{Worst\_OCPio,Rd} = 1 \cdot T_A + 2 \cdot T_B + (1 + n_S) \cdot T_B + 2 \cdot T_A + (1 + n_M) \cdot T_A \quad (5.1a)$$

$$= (4 + n_M) \cdot T_A + (3 + n_S) \cdot T_B \quad (5.1b)$$

Keeping in mind that the latency is seen as an integer number of cycles of **ClkA** we get the following worst-case latency of an OCPio Read transaction:

$$T_{Worst\_OCPio,Rd} = \left\{ (4 + n_M) + \left\lfloor (3 + n_S) \cdot \frac{T_B}{T_A} \right\rfloor \right\} T_A \quad (5.2a)$$

$$= \left\{ (4 + n_M) + \left\lfloor (3 + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} \quad (5.2b)$$

The reason we use the floor-operator rather than the ceiling-operator in an expression for a worst-case latency is a bit subtle and is illustrated in Figure 9. In the above expression we calculate interval [d] to be two 2 cycles @**ClkA**, but as shown in Figure 9 the beginning of interval [d] is related to **ClkB**. When interval [d] is taken as 2 cycles @ **ClkA** it overlaps with interval [c], and this explains the use of the floor-operator. With this explanation the reader is encouraged to refer back to Figure 8.

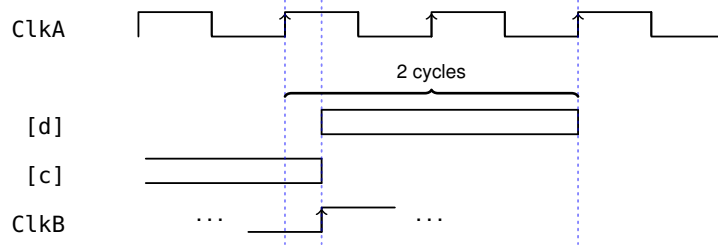


Figure 9: Relationship between intervals [c] and [d].

In a similar way we we obtain the expressions for the remaining three transactions:

$$T_{Worst\_OCPio,Wr} = \left\{ (4 + n_M) + \left\lfloor (3 + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} \quad (5.3)$$

$$T_{Worst\_OCPburst,Wr} = \left\{ (3 + N) + \left\lfloor (3 + N + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} \quad (5.4)$$

$$T_{Worst\_OCPburst,Rd} = \left\{ 3 + \left\lfloor (3 + N + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} \quad (5.5)$$



# Chapter 6

## Implementation

Both designs have been implemented in RTL VHDL and verified using ModelSim. In addition to this, both interfaces have been implemented in a 4 core T-CREST platform synthesized for an Altera DE2-115, with a Master core (Patmos 0) running at a different clock from the rest of the platform. To ensure independent clocks, the master core is running using the onboard 50MHz clock, while an external clock generator running at a variable frequency (up to 50MHz) is driving the rest of the platform.

Altera Quartus 15.0 puts the MTBF for the synchronization chain at greater than 1 billion years. Additionally, the physical setup was tested over a week with no failures.

As both interfaces are fairly simple, their resource and speed measurements are less interesting than the performance impact. As we noted in Chapter 5 the simplest way to compare performance impact is to assume that two independent clock domains each running at equal frequencies, with a constant phase difference always resulting in Worst-Case Execution Time (WCET), and compare this to an interface without clock crossing.

**OCPio** For OCPio two best-case performances exist, ie. 1 cycle per word for reads, and 2 cycles per word for writes. Using worst-case timings with clock domain crossings, this becomes 7 cycles per word for both. This leaves us with a 1/7 bandwidth for reads and 2/7 bandwidth for writes. Of course we note that if the slave introduces a read or write delay  $n_S$  then we will achieve  $\frac{1+n_S}{7+n_S}$  for reads and  $\frac{2+n_S}{7+n_S}$ .

**OCPburst** For OCPburst one best-case performance exist; 5 cycles per 4 words. Using worst-case timings with clock domain crossings, this becomes 14 cycles per word for both. This leaves us with a 5/14 bandwidth. Of course we note that if the slave introduces a read or write delay  $n_S$  then we will achieve  $\frac{5+n_S}{14+n_S}$ . For an external memory, such as a DRAM,  $n_S$  is relatively high, meaning that the performance of the domain crossing interface will trend towards the synchronous interface. Figure 10 shows the performance of the two interfaces normalized to a situation where the master and the slave operate

Table 2: Synthesis Results

	LC Combinatorial	LC Register
Burst	309	320
IO	86	93

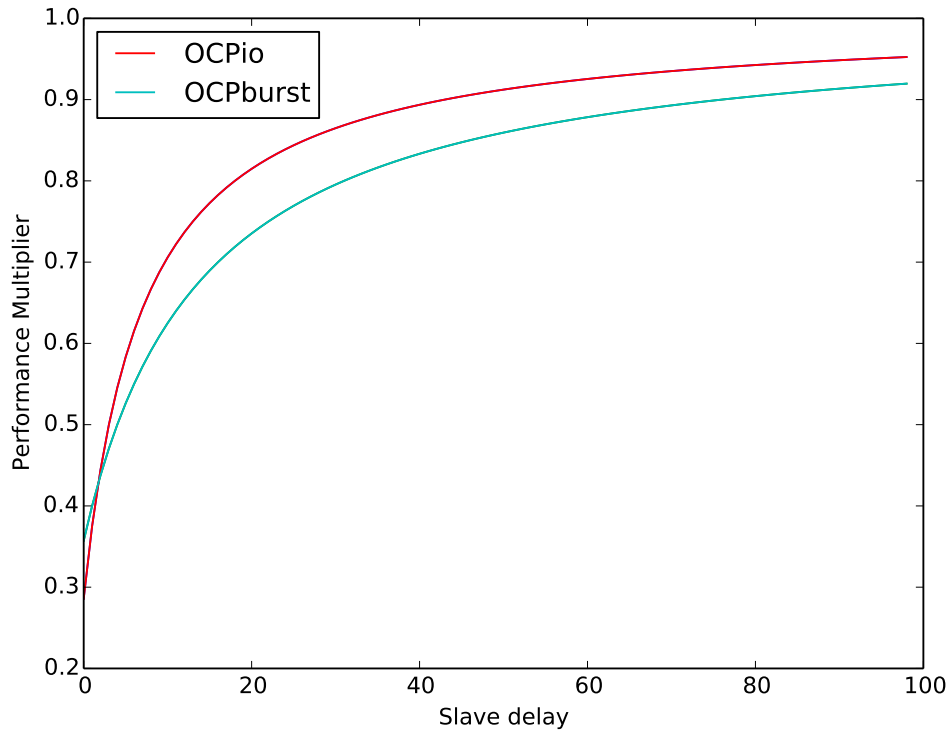


Figure 10: Performance of the two OCP CDC-modules normalized to a situation where the master and slave operate synchronously and are connected directly (without a CDC-module). For a memory the slave delay is the access time of the memory.

synchronously and without a clock domain crossing module. As seen, the performance approaches that of a plain synchronous interface as the access time of the slave increases. The latter is a likely situation when several processors share and access an of chip DRAM-based memory.

# Chapter 7

## Conclusion

This report has presented two different designs of a clock domain crossing module for two distinct OCP-style interfaces supporting either single word read/write transactions, or burst read/write transactions, respectively. Both designs use Non-Return-to-Zero (NRZ) synchronization protocols, and pass the command, address, write data and read data signals through buffer registers clocked by the source clock.

The performance of the two designs can be quantified by how many cycles a transaction takes when a clock domain crossing module is added between a synchronous master and slave pair, assuming no accept/response delays. For the OCPio design, this is 7 cycles. For the OCPburst design, it is 14 cycles. We note however that for OCPburst, whose primary application is interfacing towards a shared main memory, the latency can be masked by the relatively high access time of the memory.

Furthermore the basic structure and the underlying ideas can be used in the design of clock domain crossing modules for other bus-style read/write-transaction interfaces such as Wishbone.

# Bibliography

- [1] Accellera Systems Initiative. Open Core Protocol specification, release 3.0. [http://www.accellera.org/downloads/standards/ocp/ocp\\_3.0/](http://www.accellera.org/downloads/standards/ocp/ocp_3.0/), 2013.
- [2] Daniel M. Chapiro. *Globally-Asynchronous Locally-Synchronous Systems*. PhD thesis, Stanford University, October 1984. Report No. STAN-CS-84-1026.
- [3] W. J. Dally and J. W. Poulton. *Digital Systems Engineering*. Cambridge University Press, 1998.
- [4] R. Ginosar. Metastability and synchronizers: A tutorial. *IEEE Design & Test of Computers*, 28(5):23–35, 2011.
- [5] Mathias Herlev, Christian Keis Poulsen, and Jens Sparsø. Open Core Protocol (OCP) Clock Domain Crossing Interfaces. In *Proc. 32th IEEE Norchip Conference*, pages 1–6, 2014.
- [6] OpenCores. Wishbone B4: The WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores. <http://www.opencores.org>, 2010.
- [7] I. Miro Panades and A. Greiner. Bi-synchronous FIFO for synchronous circuit communication well suited for network-on-chip in gals architectures. In *Proc. IEEE/ACM Intl. Symposium on Networks-on-Chip (NOCS)*, pages 83–92, 2007.
- [8] Martin Schoeberl, Sahar Abbaspourseyedi, Alexander Jordan, Evangelia Kasapaki, Wolfgang Puffitsch, Jens Sparsø, Benny Akesson, Neil Audsley, Jamie Garside, Raffaele Capasso, Alessandro Tocchi, Kees Goossens, Sven Goossens, Yonghui Li, Scott Hansen, Reinhold Heckmann, Stefan Hepp, Benedikt Huber, Jens Knoop, Daniel Prokesch, Peter Puschner, Andr Rocha, and Cludio Silva. T-crest: Time-predictable multi-core architecture for embedded systems. *Journal of Systems Architecture*, 61(9):449–471, 2015.
- [9] Martin Schoeberl, Florian Brandner, Stefan Hepp, Wolfgang Puffitsch, and Daniel Prokesch. Patmos Reference Handbook. Technical Report, Technical University of Denmark. [patmos.compute.dtu.dk/patmos\\_handbook.pdf](http://patmos.compute.dtu.dk/patmos_handbook.pdf), 2016.
- [10] Ahmed H. M. Soliman, E. M. Saad, M. El-Bably, and Hesham M. A. M. Keshk. Designing a WISHBONE Protocol Network Adapter for an Asynchronous Network-on-Chip. *International Journal of Computer Science Issues*, 8(2):261–267, 2012.
- [11] T-CREST. Github - Argo source code repository. <https://github.com/t-crest/argo.git>, 2016.

- [12] The International Technology Roadmap for Semiconductors. <http://www.itrs2.net/>.
- [13] Time-predictable Multi-Core Architecture for Embedded Systems (T-CREST). [www.t-crest.org](http://www.t-crest.org).
- [14] Vikas S. Vij, Raghu Prasad Gudla, and Kenneth S. Stevens. Interfacing synchronous and asynchronous domains for open core protocol. In *27th International Conference on VLSI Design and 13th International Conference on Embedded Systems*, pages 282–287, 2014.
- [15] P. Wielage, J.E. Marinissen, M. Altheimer, and C. Wouters. Design and DfT of a high-speed area-efficient embedded asynchronous FIFO. In *Proc. Design, Automation and Test in Europe (DATE)*, pages 853–858, 2007.
- [16] Po Han Wu, Chi Guang Lin, Chia Sheng Wen, and Shen Fu Hsiao. Asynchronous AHB bus interface designs in a multiple-clock-domain graphics system. *IEEE Asia-Pacific Conference on Circuits and Systems, Proceedings, APCCAS*, pages 408–411, 2012.

# Appendix A

## Code

### A.1 OCPio

#### A.1.1 OCPio A side

```
1 

---


2 — Copyright (c) 2016, Mathias Herlev
3 — All rights reserved.
4 —
5 — Redistribution and use in source and binary forms, with or without
6 — modification, are permitted provided that the following conditions are met:
7 —
8 — 1. Redistributions of source code must retain the above copyright notice,
9 — this list of conditions and the following disclaimer.
10 — 2. Redistributions in binary form must reproduce the above copyright notice,
11 — this list of conditions and the following disclaimer in the documentation
12 — and/or other materials provided with the distribution.
13 —
14 — THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 — AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16 — IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17 — ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
18 — LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
19 — CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
20 — SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
21 — INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
22 — CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
23 — ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
24 — POSSIBILITY OF SUCH DAMAGE.
25 

---


26 — Title : OCPIO Clock Crossing Interface Slave
27 — Type : Entity
28 — Description : Slave Interface for the OCP clock crossing. Connects to a
29 — : master
30 

---


31 LIBRARY ieee;
32 USE ieee.std_logic_1164.all;
33 USE ieee.numeric_std.all;
34 LIBRARY work;
35 USE work.OCPIOCDC.types.all;
```

```

36 USE work.ocp.all;
37
38 ENTITY OCPIOCDCA IS
39     GENERIC( IOWidth : INTEGER := 1 );
40     PORT(
41         clk          : IN    std_logic;
42         rst          : IN    std_logic;
43         syncIn       : IN    ocp_io_m;
44         syncOut      : OUT   ocp_io_s;
45         asyncOut     : OUT   asyncIO_A_r;
46         asyncIn      : IN    asyncIO_B_r
47     );
48 END ENTITY OCPIOCDCA;
49
50 

---


51 -- Buffered Architecture
52 

---


53 ARCHITECTURE Buffered OF OCPIOCDCA IS
54
55     -- FSM signals
56     TYPE fsm_states_t IS (IDLE_state, AckWait_state, RespAcceptWait_state);
57     SIGNAL state, state_next : fsm_states_t;
58
59     -- Async signals
60
61     SIGNAL ack_prev, ack, ack_next : std_logic := '0';
62     SIGNAL req, req_next           : std_logic := '0';
63
64     -- Registers
65
66     SIGNAL masterData, masterData_next : ocp_io_m;
67     SIGNAL writeEnable : std_logic := '0';
68
69 BEGIN
70
71     asyncOut.req <= req;
72     asyncOut.data <= masterData;
73
74
75     -- FSM
76
77     FSM : PROCESS( state, syncIn, asyncIn, ack, ack_prev, req )
78     BEGIN
79         state_next <= state;
80         syncOut <= OCPIOSlaveIdle_c;
81         writeEnable <= '0';
82         req_next <= req;
83
84         CASE state IS
85             WHEN IDLE_state =>
86                 -- If command is different from idle
87                 IF syncIn.MCmd /= OCP_CMD_IDLE THEN
88                     -- Signal the B side
89                     req_next <= NOT(req);
90

```

```

91         state_next <= AckWait_state;
92         — Buffer the command, address, and data
93         writeEnable <= '1';
94     END IF;
95     WHEN AckWait_state =>
96         — If the slave has acknowledged
97         IF ack = NOT (ack_prev) THEN
98             — Then signal the master
99             state_next <= RespAcceptWait_state;
100            syncOut <= asyncIn.data;
101            syncOut.SCmdAccept <= '1';
102            IF syncIn.MRespAccept = '1' THEN
103                — If the master accepts, go to idle
104                state_next <= IDLE_state;
105            END IF;
106            — Else go to WriteWordFinal
107        END IF;
108        WHEN RespAcceptWait_state =>
109            — When OCP master accepts, go to idle
110            syncOut <= asyncIn.data;
111            IF syncIn.MRespAccept = '1' THEN
112                state_next <= IDLE_state;
113            END IF;
114        WHEN OTHERS =>
115            state_next <= IDLE_state;
116    END CASE;
117 END PROCESS FSM;
118
119
120 — Registers
121
122 DataRegMux : PROCESS( writeEnable , syncIn)
123 BEGIN
124     masterData_next <= masterData;
125     IF writeEnable = '1' THEN
126         masterData_next <= syncIn;
127     END IF;
128 END PROCESS DataRegMux;
129
130 Registers : PROCESS( clk , rst)
131 BEGIN
132     IF rst = '1' THEN
133         state          <= IDLE_state;
134         req            <= '0';
135         ack_prev      <= '0';
136         ack           <= '0';
137         ack_next      <= '0';
138         masterData    <= OCPIOmasteridle_c;
139
140     ELSIF rising_edge( clk) THEN
141         state          <= state_next;
142         req            <= req_next;
143         ack_prev      <= ack;
144         ack           <= ack_next;
145         ack_next      <= asyncIn.ack;

```



```

146         masterData <= masterData_next;
147     END IF;
148 END PROCESS Registers;
149
150 END ARCHITECTURE Buffered;
151
152 ARCHITECTURE NonBuffered OF OCPIOCDCA IS
153     TYPE fsm_states_t IS (IDLE_state, AckWait_state, RespAcceptWait_state,
154                          HandshakeFinal_state);
155     SIGNAL state, state_next : fsm_states_t;
156
157     SIGNAL ack, ack_next : std_logic := '0';
158     SIGNAL req : std_logic := '0';
159
160 BEGIN
161
162     asyncOut.req <= req;
163     asyncOut.data <= syncIn;
164
165     FSM : PROCESS(state, syncIn, asyncIn, ack)
166     BEGIN
167         state_next <= state;
168         syncOut <= OCPIOSlaveIdle_c;
169         CASE state IS
170             WHEN IDLE_state =>
171                 req <= '0';
172                 IF ack = '0' THEN
173                     IF syncIn.MCmd /= OCP_CMD_IDLE THEN
174                         req <= '1';
175                         state_next <= AckWait_state;
176                     END IF;
177                 END IF;
178             WHEN AckWait_state =>
179                 req <= '1';
180                 IF ack = '1' THEN
181                     state_next <= RespAcceptWait_state;
182                     syncOut <= asyncIn.data;
183                     syncOut.SCcmdAccept <= '1';
184                     IF syncIn.MRespAccept = '1' THEN
185 —                         req <= '0';
186                         state_next <= Idle_state;
187                     END IF;
188                 END IF;
189             WHEN RespAcceptWait_state =>
190                 req <= '1';
191                 syncOut <= asyncIn.data;
192                 syncOut.SCcmdAccept <= '0';
193                 IF syncIn.MRespAccept = '1' THEN
194                     state_next <= Idle_state;
195                 END IF;
196             WHEN HandshakeFinal_state =>
197                 req <= '0';
198                 IF ack = '0' THEN
199                     state_next <= Idle_state;
200                 END IF;

```

```

201         WHEN OTHERS =>
202             state_next <= IDLE_state;
203     END CASE;
204 END PROCESS FSM;
205
206
207 Registers : PROCESS(clk , rst )
208 BEGIN
209     IF rst = '1' THEN
210         state      <= IDLE_state;
211         ack        <= '0';
212         ack_next   <= '0';
213     ELSIF rising_edge(clk) THEN
214         state      <= state_next;
215         ack        <= ack_next;
216         ack_next   <= asyncIn.ack;
217     END IF;
218 END PROCESS Registers;
219
220 END ARCHITECTURE NonBuffered;

```

## A.1.2 OCPio B side

---

```

1
2 — Copyright (c) 2016, Mathias Herlev
3 — All rights reserved.
4 —
5 — Redistribution and use in source and binary forms, with or without
6 — modification, are permitted provided that the following conditions are met:
7 —
8 — 1. Redistributions of source code must retain the above copyright notice,
9 — this list of conditions and the following disclaimer.
10 — 2. Redistributions in binary form must reproduce the above copyright notice,
11 — this list of conditions and the following disclaimer in the documentation
12 — and/or other materials provided with the distribution.
13 —
14 — THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 — AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16 — IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17 — ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
18 — LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
19 — CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
20 — SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
21 — INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
22 — CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
23 — ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
24 — POSSIBILITY OF SUCH DAMAGE.
25
26 — Title : OCPIO Clock Crossing Interface Slave
27 — Type : Entity
28 — Description : Master Interface for the OCP clock crossing. Connects to a
29 — Slave
30
31 LIBRARY ieee;
32 USE ieee.std_logic_1164.all;
33 USE ieee.numeric_std.all;

```

```

34 LIBRARY work;
35 USE work.OCPIOCDC_types.all;
36 USE work.ocp.all;
37
38 ENTITY OCPIOCDC.B IS
39     GENERIC(IOSize : INTEGER := 1);
40     PORT(    clk          : IN    std_logic;
41           rst          : IN    std_logic;
42           syncIn       : IN    ocp_io_s;
43           syncOut      : OUT   ocp_io_m;
44           asyncOut     : OUT   asyncIO_B_r;
45           asyncIn      : IN    asyncIO_A_r
46     );
47 END ENTITY OCPIOCDC.B;
48
49 -----
50 --- Buffered Architecture
51 -----
52 ARCHITECTURE Buffered OF OCPIOCDC.B IS
53     -----
54     --- FSM signals
55     -----
56     TYPE fsm_states_t IS    (IDLE_state , CmdAcceptWait_state , RespWait_state);
57     SIGNAL state , state_next    :    fsm_states_t;
58     -----
59     --- Async signals
60     -----
61     SIGNAL req_prev , req , req_next    : std_logic := '0';
62     SIGNAL ack , ack_next              : std_logic := '0';
63     -----
64     --- Register signals
65     -----
66     SIGNAL slaveData , slaveData_next  : ocp_io_s;
67     SIGNAL loadEnable    : std_logic;
68
69 BEGIN
70
71     asyncOut.data <= slaveData WHEN loadEnable = '0' ELSE syncIn;
72     asyncOut.ack  <= ack;
73
74     -----
75     --- FSM
76     -----
77
78     FSM : PROCESS(state , syncIn , asyncIn , req , req_prev , ack)
79     BEGIN
80         state_next <= state;
81         loadEnable <= '0';
82         syncOut    <= OCPIOMasterIdle.c;
83         ack_next   <= ack;
84
85         CASE state IS
86             WHEN IDLE_state =>
87                 --- If a new request is available
88                 IF req = NOT (req_prev) THEN

```

```

89         IF (asyncIn.data.MCmd /= OCP_CMD_IDLE) THEN
90             —Relay the command to the OCP slave
91             syncOut <= asyncIn.data;
92             state_next <= CmdAcceptWait_state;
93             — If the command is accepted
94             IF syncIn.SCcmdAccept = '1' THEN
95                 state_next <= RespWait_state;
96                 —And a response is ready
97                 IF syncIn.SResp /= OCP_RESP_NULL THEN
98                     state_next <= IDLE_state;
99                     loadEnable <= '1';
100                    — Signal the A side
101                    ack_next <= NOT (ack);
102                    syncOut.MRespAccept <= '1';
103                END IF;
104            END IF;
105        END IF;
106    END IF;
107    WHEN CmdAcceptWait_state =>
108        — If command has not been accepted, Command+data has not been
109        — Registered in OCP Slave. Continue asserting command.
110        syncOut <= asyncIn.data;
111        IF syncIn.SCcmdAccept = '1' THEN
112            state_next <= RespWait_state;
113            IF syncIn.SResp /= OCP_RESP_NULL THEN
114                state_next <= IDLE_state;
115                ack_next <= NOT (ack);
116                loadEnable <= '1';
117                syncOut.MRespAccept <= '1';
118            END IF;
119        END IF;
120    WHEN RespWait_state =>
121        IF syncIn.SResp /= OCP_RESP_NULL THEN
122            state_next <= IDLE_state;
123            ack_next <= NOT (ack);
124            loadEnable <= '1';
125            syncOut.MRespAccept <= '1';
126        END IF;
127    WHEN OTHERS =>
128        state_next <= IDLE_state;
129    END CASE;
130 END PROCESS FSM;
131
132
133 DataRegMux      : PROCESS(loadEnable , syncIn)
134 BEGIN
135     slaveData_next <= slaveData;
136     IF loadEnable = '1' THEN
137         slaveData_next <= syncIn;
138     END IF;
139 END PROCESS DataRegMux;
140
141
142 Registers      : PROCESS(clk , rst)
143 BEGIN

```

```

144     IF rst = '1' THEN
145         state <= IDLE_state;
146         req_prev    <= '0';
147         req         <= '0';
148         req_next   <= '0';
149         ack        <= '0';
150         slaveData  <= ocpioslaveidle_c;
151     ELSIF rising_edge(clk) THEN
152         state      <= state_next;
153         req_prev   <= req;
154         req        <= req_next;
155         req_next  <= asyncIn.req;
156         ack       <= ack_next;
157         slaveData <= slaveData_next;
158     END IF;
159 END PROCESS Registers;
160
161 END ARCHITECTURE Buffered;
162
163 -----
164 -- Unbuffered architecture
165 -----
166 ARCHITECTURE NonBuffered OF OCPIOCDC.B IS
167     TYPE fsm_states_t IS (Idle_state , CmdAcceptWait_state , RespWait_state ,
168                          ReqWait_state);
169     SIGNAL state , state_next      : fsm_states_t := Idle_state;
170
171     SIGNAL req , req_next          : std_logic := '0';
172     SIGNAL ack  : std_logic := '0';
173
174
175 BEGIN
176     -- Async Data Signals
177
178     asyncOut.data    <= syncIn;
179     asyncOut.ack     <= ack;
180
181     FSM : PROCESS(state , syncIn , asyncIn , req)
182     BEGIN
183         state_next <= state;
184         syncOut    <= OCPIOMasterIdle_c;
185
186         CASE state IS
187             WHEN Idle_state =>
188                 ack <= '0';
189                 IF req = '1' THEN
190                     IF (asyncIn.data.MCmd /= OCP_CMD_IDLE) THEN
191                         syncOut <= asyncIn.data;
192                         syncOut.MRespAccept <= '0';
193                         state_next <= CmdAcceptWait_state;
194                         IF syncIn.SCmdAccept = '1' THEN
195                             state_next <= RespWait_state;
196                             IF syncIn.SResp /= OCP_RESP_NULL THEN
197                                 state_next <= ReqWait_state;
198                                 --
199                                 ack <= '1';

```

```

199             END IF;
200         END IF;
201     END IF;
202 END IF;
203 WHEN CmdAcceptWait_state =>
204     ack <= '0';
205     syncOut <= asyncIn.data;
206     syncOut.MRespAccept <= '0';
207     IF syncIn.SCmdAccept = '1' THEN
208         state_next <= RespWait_state;
209         IF syncIn.SResp /= OCP_RESP_NULL THEN
210             state_next <= ReqWait_state;
211         --         ack <= '1';
212     END IF;
213 END IF;
214 WHEN RespWait_state =>
215     ack <= '0';
216     IF syncIn.SResp /= OCP_RESP_NULL THEN
217         state_next <= ReqWait_state;
218         ack <= '1';
219     END IF;
220 WHEN ReqWait_state =>
221     ack <= '1';
222     IF req = '0' THEN
223         ack <= '0';
224         syncOut.MRespAccept <= '1';
225         state_next <= Idle_state;
226     END IF;
227 WHEN OTHERS =>
228     state_next <= IDLE_state;
229 END CASE;
230 END PROCESS FSM;
231
232
233 Registers    : PROCESS(clk , rst)
234 BEGIN
235     IF rst = '1' THEN
236         state <= IDLE_state;
237         req <= '0';
238         req_next <= '0';
239     ELSIF rising_edge(clk) THEN
240         state      <= state_next;
241         req        <= req_next;
242         req_next   <= asyncIn.req;
243     END IF;
244 END PROCESS Registers;
245
246 END ARCHITECTURE NonBuffered;

```

## A.2 OCPburst

### A.2.1 OCPburst A side

```

3  — All rights reserved.
4  —
5  — Redistribution and use in source and binary forms, with or without
6  — modification, are permitted provided that the following conditions are met:
7  —
8  — 1. Redistributions of source code must retain the above copyright notice,
9  — this list of conditions and the following disclaimer.
10 — 2. Redistributions in binary form must reproduce the above copyright notice,
11 — this list of conditions and the following disclaimer in the documentation
12 — and/or other materials provided with the distribution.
13 —
14 — THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 — AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16 — IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17 — ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
18 — LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
19 — CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
20 — SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
21 — INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
22 — CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
23 — ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
24 — POSSIBILITY OF SUCH DAMAGE.
25 —
26 — Title      : OCPburst Clock Crossing Interface A Side
27 — Type       : Entity
28 — Description : Slave Interface for the OCPburst clock crossing. Connects to a
29 —           : master
30 —
31 LIBRARY ieee;
32 USE ieee.std_logic_1164.all;
33 USE ieee.numeric_std.all;
34 LIBRARY work;
35 USE work.ocp_config.all;
36 USE work.ocp.all;
37 USE work.OCPBurstCDC_types.all;
38
39 ENTITY OCPBurstCDC_A IS
40     GENERIC(burstSize : INTEGER := 4);
41     PORT(
42         clk      : IN    std_logic;
43         rst      : IN    std_logic;
44         syncIn   : IN    ocp_burst_m;
45         syncOut  : OUT   ocp_burst_s;
46         asyncOut : OUT   AsyncBurst_A_r;
47         asyncIn  : IN    AsyncBurst_B_r
48     );
49 END ENTITY OCPBurstCDC_A;
50
51 ARCHITECTURE behaviour OF OCPBurstCDC_A IS
52     — Constants
53
54     CONSTANT OCPBurstSlaveIdle_c : ocp_burst_s := (OCP_RESP_NULL,
55                                                     (OTHERS => '0'),
56                                                     '0',
57                                                     '0');

```

```

58
59  — FSM Signal Declarations
60
61  TYPE fsm_states_t IS ( IDLE_state , ReadBlock , ReadBlockWait ,
62                          WriteBlockLoad , WriteBlockWait );
63  SIGNAL state , state_next      :    fsm_states_t ;
64
65  — Data Registers
66
67  SIGNAL cmd , cmd_next      : std_logic_vector( OCP_CMD_WIDTH-1 downto 0)
68                          := OCP_CMD_IDLE;
69  SIGNAL addr , addr_next    : std_logic_vector( OCP_BURST_ADDR_WIDTH-1 downto 0)
70                          := (others => '0');
71
72  TYPE dataArray_t      IS
73      ARRAY (burstSize-1 downto 0) OF
74      std_logic_vector( OCP_DATA_WIDTH-1 downto 0);
75  SIGNAL data_arr : dataArray_t;
76
77  TYPE ByteEn_Array_t IS
78      ARRAY (burstSize downto 0) OF
79      std_logic_vector( OCP_BYTE_WIDTH-1 downto 0);
80  SIGNAL byteEn_arr    : ByteEn_Array_t;
81
82  SIGNAL writeEnable          : std_logic := '0';
83
84  SIGNAL RegAddr , RegAddr_next    : unsigned(1 downto 0) := (others => '0');
85
86  — Asynchronous signals
87
88  ALIAS o_async IS asyncOut;
89  ALIAS i_async IS asyncIn;
90
91  SIGNAL ack_prev , ack , ack_next  : std_logic := '0';
92  SIGNAL req , req_next             : std_logic := '0';
93
94  BEGIN
95
96
97  FSM : PROCESS(state , syncIn , asyncIn , ack , ack_prev , RegAddr , req , cmd , addr)
98  BEGIN
99
100  — Default Assignments
101
102  state_next      <= state;
103  syncOut         <= OCPBurstSlaveIdle_c;
104  writeEnable     <= '0';
105  req_next <= req;
106  cmd_next <= cmd;
107  addr_next <= addr;
108  RegAddr_next <= RegAddr;
109  asyncOut.RegAddr <= (others => '0');
110  asyncOut.data.MDataValid <= '0';
111  syncOut.SCmdAccept <= '0';
112  syncOut.SDataAccept <= '0';

```



```

113
114     CASE state IS
115         WHEN IDLE_state =>
116             — If command is read
117             IF syncIn.Mcmd = OCP_CMD_RD THEN
118                 — Register Command and address (MData not valid)
119                 cmd_next <= syncIn.MCmd;
120                 addr_next <= syncIn.MAddr;
121                 — Assert a request
122                 req_next      <= NOT (req);
123                 — And go to ReadBlockWait, to await an acknowledge
124                 state_next    <= ReadBlockWait;
125             — If command is write
126             ELSIF syncIn.Mcmd = OCP_CMD_WR AND syncIn.MDataValid = '1' THEN
127                 — Start buffering MData + MCmd + MAddr
128                 cmd_next <= syncIn.MCmd;
129                 addr_next <= syncIn.MAddr;
130                 syncOut.SCcmdAccept <= '1';
131                 syncOut.SDataAccept <= '1';
132                 writeEnable      <= '1';
133                 RegAddr_next     <= RegAddr + to_unsigned(1, RegAddr'LENGTH);
134                 state_next       <= WriteBlockLoad;
135             END IF;
136
137         — READ BLOCK
138
139         WHEN ReadBlockWait =>
140             — Wait until acknowledge
141             IF ack = NOT(ack_prev) THEN
142                 state_next    <= ReadBlock;
143                 syncOut.SCcmdAccept <= '1';
144             END IF;
145         WHEN ReadBlock =>
146             — Write each word in buffer back to OCP Master
147             asyncOut.RegAddr    <= std_logic_vector(RegAddr);
148             syncOut             <= asyncIn.data;
149             RegAddr_next       <= RegAddr + to_unsigned(1, RegAddr'LENGTH);
150             IF RegAddr = to_unsigned(burstSize - 1, RegAddr'LENGTH) THEN
151                 state_next     <= IDLE_state;
152             END IF;
153
154         — WRITE BLOCK
155
156         WHEN WriteBlockLoad =>
157             — Continue buffering MData
158             syncOut.SDataAccept <= '1';
159             writeEnable        <= '1';
160             RegAddr_next       <= RegAddr + to_unsigned(1, RegAddr'LENGTH);
161             IF RegAddr = to_unsigned(burstSize - 1, RegAddr'LENGTH) THEN
162                 — And assert request once all words are buffered
163                 req_next      <= NOT (req);
164                 asyncOut.data.MDataValid <= '1';
165                 state_next     <= WriteBlockWait;
166             END IF;
167         WHEN WriteBlockWait =>

```

```

168         — Wait until B side has acknowledged finishing transaction
169         IF ack = NOT(ack_prev) THEN
170             —And relay response to OCP Master
171             syncOut.Sresp <= asyncIn.data.Sresp;
172             state_next <= IDLE_state;
173             cmd_next <= OCP_CMD_IDLE;
174             addr_next <= (others => '0');
175         END IF;
176     WHEN OTHERS =>
177         state_next <= IDLE_state;
178 END CASE;
179 END PROCESS FSM;
180
181
182 — Output Map
183
184 asyncOut.data.MCmd <= cmd;
185 asyncOut.data.MData <= data_arr(to_integer(unsigned(i_async.RegAddr)));
186 asyncOut.data.MAddr <= addr;
187 asyncOut.data.MDataByteEn <=
188     byteEn_arr(to_integer(unsigned(i_async.RegAddr)));
189 asyncOut.req <= req;
190
191
192 — Register Processes
193
194 Registers : PROCESS(clk,rst)
195 BEGIN
196     IF rst = '1' THEN
197         state <= IDLE_state;
198         req <= '0';
199         ack_prev <= '0';
200         ack <= '0';
201         ack_next <= '0';
202         RegAddr <= (others=>'0');
203         cmd <= OCP_CMD_IDLE;
204         addr <= (others => '0');
205     ELSIF rising_edge(clk) THEN
206         state <= state_next;
207         req <= req_next;
208         ack_prev <= ack;
209         ack <= ack_next;
210         ack_next <= asyncIn.ack;
211         RegAddr <= RegAddr_next;
212         cmd <= cmd_next;
213         addr <= addr_next;
214     END IF;
215 END PROCESS Registers;
216
217 DataRam : PROCESS(clk)
218 BEGIN
219     IF rising_edge(clk) THEN
220         IF writeEnable = '1' THEN
221             data_arr(to_integer(RegAddr)) <= syncIn.MData;
222         END IF;

```

```

223     END IF ;
224 END PROCESS DataRam ;
225
226 ByteEnRam : PROCESS (clk)
227 BEGIN
228     IF rising_edge (clk) THEN
229         IF writeEnable = '1' THEN
230             byteEn_arr (to_integer (RegAddr)) <= syncIn.MDataByteEn ;
231         END IF ;
232     END IF ;
233 END PROCESS ByteEnRam ;
234
235 END ARCHITECTURE behaviour ;

```

## A.2.2 OCPburst B side

```

1  -----
2  — Copyright (c) 2016, Mathias Herlev
3  — All rights reserved.
4  —
5  — Redistribution and use in source and binary forms, with or without
6  — modification, are permitted provided that the following conditions are met:
7  —
8  — 1. Redistributions of source code must retain the above copyright notice,
9  — this list of conditions and the following disclaimer.
10 — 2. Redistributions in binary form must reproduce the above copyright notice,
11 — this list of conditions and the following disclaimer in the documentation
12 — and/or other materials provided with the distribution.
13 —
14 — THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 — AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16 — IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17 — ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
18 — LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
19 — CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
20 — SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
21 — INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
22 — CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
23 — ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
24 — POSSIBILITY OF SUCH DAMAGE.
25  -----
26 — Title : OCPBurst Clock Crossing Interface Slave
27 — Type : Entity
28 — Description : Master Interface for the OCP clock crossing. Connects to a
29 — : Slave
30  -----
31
32 LIBRARY ieee ;
33 USE ieee.std_logic_1164.all ;
34 USE ieee.numeric_std.all ;
35 LIBRARY work ;
36 USE work.ocp.all ;
37 USE work.OCPBurstCDC_types.all ;
38
39 ENTITY OCPBurstCDC_B IS
40     GENERIC (burstSize : INTEGER := 4);

```

```

41     PORT(    clk          : IN    std_logic;
42            rst          : IN    std_logic;
43            syncIn       : IN    ocp_burst_s;
44            syncOut      : OUT   ocp_burst_m;
45            asyncOut     : OUT   AsyncBurst_B_r;
46            asyncIn      : IN    AsyncBurst_A_r
47        );
48 END ENTITY OCPBurstCDC_B;
49
50 ARCHITECTURE behaviour OF OCPBurstCDC_B IS
51     

---


52     -- FSM signals
53     

---


54     TYPE fsm_states_t IS ( IDLE_state , ReadBlock , ReadBlockWait ,
55                          WriteBlock , WriteBlockWait , WriteBlockFinal);
56     SIGNAL state , state_next      : fsm_states_t;
57     

---


58     -- Register signals
59     

---


60     SIGNAL RegAddr , RegAddr_next  : unsigned(1 downto 0) := (others => '0');
61
62     TYPE DataArray_t IS
63         ARRAY (burstSize-1 downto 0) OF
64             std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
65     TYPE RespArray_t IS
66         ARRAY (burstSize-1 downto 0) OF
67             std_logic_vector(OCP_RESP_WIDTH-1 downto 0);
68
69     SIGNAL data_arr : DataArray_t;
70     SIGNAL resp_arr : RespArray_t;
71
72     SIGNAL loadEnable : std_logic;
73     

---


74     -- Async signals
75     

---


76
77     SIGNAL req_prev , req , req_next : std_logic := '0';
78     SIGNAL ack , ack_next           : std_logic := '0';
79
80 BEGIN
81     asyncOut.ack      <= ack;
82     asyncOut.data.SResp <= resp_arr(to_integer(unsigned(asyncIn.RegAddr)));
83     asyncOut.data.SData <= data_arr(to_integer(unsigned(asyncIn.RegAddr)));
84
85     

---


86     -- FSM
87     

---


88     FSM : PROCESS(state , syncIn , asyncIn , req , req_prev , RegAddr , ack)
89     BEGIN
90         state_next <= state;
91         loadEnable <= '0';
92         RegAddr_next <= RegAddr;
93         syncOut.MCmd <= OCP_CMD_IDLE;
94         syncOut.MAddr <= (others => '0');
95         syncOut.MData <= (others => '0');

```

```

96     syncOut.MDataByteEn <= (others => '0');
97     asyncOut.RegAddr    <= (others => '0');
98     syncOut.MDataValid <= '0';
99     ack_next <= ack;
100    CASE state IS
101        WHEN IDLE_state =>
102            — Wait for request
103            IF req = NOT(req_prev) THEN
104                — If read command
105                IF asyncIn.data.MCmd = OCP_CMD_RD THEN
106                    — Relay command to OCP slave and either go to wait state
107                    — or commence buffering read data
108                    state_next    <= ReadBlockWait;
109                    syncOut.MCmd   <= OCP_CMD_RD;
110                    IF syncIn.SCmdAccept = '1' THEN
111                        state_next <= ReadBlock;
112                    END IF;
113                — If write command
114                ELSIF asyncIn.data.MCmd = OCP_CMD_WR THEN
115                    state_next    <= WriteBlockWait;
116                    syncOut.MCmd   <= OCP_CMD_WR;
117                    syncOut.MDataValid <= '1';
118                    syncOut.MDataByteEn <= asyncIn.data.MDataByteEn;
119                    syncOut.MAddr    <= asyncIn.data.MAddr;
120                    syncOut.MData    <= asyncIn.data.MData;
121                    IF syncIn.SCmdAccept = '1' AND syncIn.SDataAccept = '1'
122                    THEN
123                        RegAddr_next <= RegAddr +
124                            to_unsigned(1, RegAddr'LENGTH);
125                        state_next <= WriteBlock;
126                    END IF;
127                END IF;
128            END IF;
129
130    — READ BLOCK
131
132    WHEN ReadBlockWait =>
133        syncOut.MCmd <= OCP_CMD_RD;
134        IF syncIn.SCmdAccept = '1' THEN
135            state_next <= ReadBlock;
136        END IF;
137    WHEN ReadBlock =>
138        IF syncIn.SResp /= OCP_RESP_NULL THEN
139            loadEnable <= '1';
140            RegAddr_next <= RegAddr + to_unsigned(1, RegAddr'LENGTH);
141            IF RegAddr = to_unsigned(burstSize - 1, RegAddr'LENGTH) THEN
142                state_next <= IDLE_state;
143                ack_next <= NOT (ack);
144            END IF;
145        END IF;
146    — WRITE BLOCK
147    WHEN WriteBlockWait =>
148        syncOut.MCmd    <= OCP_CMD_WR;
149        syncOut.MDataValid <= '1';
150        syncOut.MAddr    <= asyncIn.data.MAddr;

```

```

151         syncOut.MDataByteEn <= asyncIn.data.MDataByteEn;
152         syncOut.MData         <= asyncIn.data.MData;
153         asyncOut.RegAddr      <= std_logic_vector(RegAddr);
154
155         IF syncIn.SCmdAccept = '1' AND syncIn.SDataAccept = '1' THEN
156             RegAddr_next <= RegAddr + to_unsigned(1,RegAddr'LENGTH);
157             state_next   <= WriteBlock;
158         END IF;
159     WHEN WriteBlock =>
160         — Sync Data Signals
161         syncOut.MDataValid <= '1';
162         syncOut.MDataByteEn <= asyncIn.data.MDataByteEn;
163         syncOut.MAddr      <= asyncIn.data.MAddr;
164         syncOut.MData      <= asyncIn.data.MData;
165         RegAddr_next      <= RegAddr + to_unsigned(1,RegAddr'LENGTH);
166         asyncOut.RegAddr   <= std_logic_vector(RegAddr);
167         IF RegAddr = to_unsigned(burstSize-1, RegAddr'LENGTH) THEN
168             state_next <= WriteBlockFinal;
169             —ack_next <= NOT(ack);
170         END IF;
171     WHEN WriteBlockFinal =>
172         IF syncIn.SResp /= OCP_RESP_NULL THEN
173             ack_next <= NOT(ack);
174             loadEnable <= '1';
175             state_next <= IDLE_state;
176         END IF;
177     WHEN OTHERS =>
178         state_next <= IDLE_state;
179     END CASE;
180 END PROCESS FSM;
181
182
183 — Registers
184
185 Registers : PROCESS(clk , rst)
186 BEGIN
187     IF rst = '1' THEN
188         state <= IDLE_state;
189         req_prev <= '0';
190         req <= '0';
191         req_next <= '0';
192         ack <= '0';
193         RegAddr <= (others=>'0');
194     ELSIF rising_edge(clk) THEN
195         state <= state_next;
196         req_prev <= req;
197         req <= req_next;
198         req_next <= asyncIn.req;
199         ack <= ack_next;
200         RegAddr <= RegAddr_next;
201     END IF;
202 END PROCESS Registers;
203
204 DataRam : PROCESS(clk)
205 BEGIN

```

```

206     IF rising_edge(clk) THEN
207         IF loadEnable = '1' THEN
208             data_arr(to_integer(RegAddr)) <= syncIn.SData;
209         END IF;
210     END IF;
211 END PROCESS DataRam;
212
213 RespRam : PROCESS(clk)
214 BEGIN
215     IF rising_edge(clk) THEN
216         IF loadEnable = '1' THEN
217             resp_arr(to_integer(RegAddr)) <= syncIn.SResp;
218         END IF;
219     END IF;
220 END PROCESS RespRam;
221
222 END ARCHITECTURE behaviour;

```

## A.3 Packages

### A.3.1 OCP Types

Following package is redistributed from the T-CREST project under a Simplified (2-Clause) BSD License

```

1  — Copyright Technical University of Denmark. All rights reserved.
2  — This file is part of the T-CREST project.
3  —
4  — Redistribution and use in source and binary forms, with or without
5  — modification, are permitted provided that the following conditions are met:
6  —
7  —     1. Redistributions of source code must retain the above copyright notice,
8  —     this list of conditions and the following disclaimer.
9  —
10 —     2. Redistributions in binary form must reproduce the above copyright
11 —     notice, this list of conditions and the following disclaimer in the
12 —     documentation and/or other materials provided with the distribution.
13 —
14 — THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDER ‘AS IS’ AND ANY EXPRESS
15 — OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE IMPLIED WARRANTIES
16 — OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE ARE DISCLAIMED. IN
17 — NO EVENT SHALL THE COPYRIGHT HOLDER OR CONTRIBUTORS BE LIABLE FOR ANY
18 — DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR CONSEQUENTIAL DAMAGES
19 — (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF SUBSTITUTE GOODS OR SERVICES;
20 — LOSS OF USE, DATA, OR PROFITS; OR BUSINESS INTERRUPTION) HOWEVER CAUSED AND
21 — ON ANY THEORY OF LIABILITY, WHETHER IN CONTRACT, STRICT LIABILITY, OR TORT
22 — (INCLUDING NEGLIGENCE OR OTHERWISE) ARISING IN ANY WAY OUT OF THE USE OF
23 — THIS SOFTWARE, EVEN IF ADVISED OF THE POSSIBILITY OF SUCH DAMAGE.
24 —
25 — The views and conclusions contained in the software and documentation are
26 — those of the authors and should not be interpreted as representing official
27 — policies, either expressed or implied, of the copyright holder.
28 —
29
30

```

```

31 -----
32 -- Definitions package
33 --
34 -- Author: Evangelia Kasapaki
35 -- Author: Rasmus Bo Soerensen
36 -----
37
38 library ieee;
39 use ieee.std_logic_1164.all;
40
41 use work.ocp_config.all;
42
43 package ocp is
44
45     -- OCP
46     constant OCP_CMD_WIDTH   : integer := 3;      -- 8 possible cmds --> 2
47     constant OCP_ADDR_WIDTH  : integer := 32;     --32
48     constant OCP_BURST_ADDR_WIDTH : integer := BURST_ADDR_WIDTH;  --32
49     constant OCP_DATA_WIDTH  : integer := 32;
50     constant OCP_BYTE_WIDTH  : integer := OCP_DATA_WIDTH/8;
51     constant OCP_RESP_WIDTH  : integer := 2;
52
53     constant OCP_CMD_IDLE    : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "000";
54     constant OCP_CMD_WR     : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "001";
55     constant OCP_CMD_RD     : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "010";
56     -- constant OCP_CMD_RDEX : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "011";
57     -- constant OCP_CMD_RDL : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "100";
58     -- constant OCP_CMD_WRNP : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "101";
59     -- constant OCP_CMD_WRC : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "110";
60     -- constant OCP_CMD_BCST : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "111";
61
62     constant OCP_RESP_NULL  : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "00";
63     constant OCP_RESP_DVA   : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "01";
64     constant OCP_RESP_FAIL  : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "10";
65     constant OCP_RESP_ERR   : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "11";
66
67     type ocp_core_m is record
68         MCmd      : std_logic_vector(OCP_CMD_WIDTH-1 downto 0);
69         MAddr     : std_logic_vector(OCP_ADDR_WIDTH-1 downto 0);
70         MData     : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
71         MByteEn   : std_logic_vector(OCP_BYTE_WIDTH-1 downto 0);
72     end record;
73
74     type ocp_core_s is record
75         SResp     : std_logic_vector(OCP_RESP_WIDTH-1 downto 0);
76         SData     : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
77     end record;
78
79     type ocp_io_m is record
80         MCmd      : std_logic_vector(OCP_CMD_WIDTH-1 downto 0);
81         MAddr     : std_logic_vector(OCP_ADDR_WIDTH-1 downto 0);
82         MData     : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
83         MByteEn   : std_logic_vector(OCP_BYTE_WIDTH-1 downto 0);
84         MRespAccept : std_logic;
85     end record;

```



```

86
87  type ocp_io_s is record
88      SResp      : std_logic_vector(OCP_RESP_WIDTH-1 downto 0);
89      SData      : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
90      SCmdAccept : std_logic;
91  end record;
92
93  type ocp_burst_m is record
94      MCmd      : std_logic_vector(OCP_CMD_WIDTH-1 downto 0);
95      MAddr     : std_logic_vector(OCP_BURST_ADDR_WIDTH-1 downto 0);
96      MData     : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
97      MDataByteEn : std_logic_vector(OCP_BYTE_WIDTH-1 downto 0);
98      MDataValid : std_logic;
99  end record;
100
101  type ocp_burst_s is record
102      SResp      : std_logic_vector(OCP_RESP_WIDTH-1 downto 0);
103      SData      : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
104      SCmdAccept : std_logic;
105      SDataAccept : std_logic;
106  end record;
107
108 end package ; — ocp

```

### A.3.2 OCPburst CDC Types

---

```

1
2 — Copyright (c) 2016, Mathias Herlev
3 — All rights reserved.
4 —
5 — Redistribution and use in source and binary forms, with or without
6 — modification, are permitted provided that the following conditions are met:
7 —
8 — 1. Redistributions of source code must retain the above copyright notice,
9 — this list of conditions and the following disclaimer.
10 — 2. Redistributions in binary form must reproduce the above copyright notice,
11 — this list of conditions and the following disclaimer in the documentation
12 — and/or other materials provided with the distribution.
13 —
14 — THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 — AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
16 — IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
17 — ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
18 — LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
19 — CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
20 — SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
21 — INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
22 — CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
23 — ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
24 — POSSIBILITY OF SUCH DAMAGE.
25
26 — Title : OCPBurst Interface Types
27 — Type : Type Package
28 — Description : Record types for OCPburst CDC interface

```

---

```

29
30 LIBRARY ieee;
31 USE ieee.std_logic_1164.all;
32 LIBRARY work;
33 USE work.ocp.all;
34
35 PACKAGE OCPBurstCDC_types IS
36
37     TYPE OCPBurstCDCIn_r IS
38     RECORD
39         clk_A      : std_logic;
40         rst_A      : std_logic;
41         clk_B      : std_logic;
42         rst_B      : std_logic;
43         OCPB_slave : ocp_burst_s;
44         OCPB_master : ocp_burst_m;
45     END RECORD;
46
47     TYPE OCPBurstCDCOut_r IS
48     RECORD
49         OCPB_A : ocp_burst_s;
50         OCPB_B : ocp_burst_m;
51     END RECORD;
52
53     TYPE AsyncBurst_A_r IS
54     RECORD
55         req      : std_logic;
56         Data     : ocp_burst_m;
57         RegAddr : std_logic_vector(1 downto 0);
58     END RECORD;
59
60     TYPE AsyncBurst_B_r IS
61     RECORD
62         ack      : std_logic;
63         Data     : ocp_burst_s;
64         RegAddr : std_logic_vector(1 downto 0);
65     END RECORD;
66 END OCPBurstCDC_types;

```

### A.3.3 OCPio CDC Types

---

```

1
2 — Copyright (c) 2016, Mathias Herlev
3 — All rights reserved.
4 —
5 — Redistribution and use in source and binary forms, with or without
6 — modification, are permitted provided that the following conditions are met:
7 —
8 — 1. Redistributions of source code must retain the above copyright notice,
9 — this list of conditions and the following disclaimer.
10 — 2. Redistributions in binary form must reproduce the above copyright notice,
11 — this list of conditions and the following disclaimer in the documentation
12 — and/or other materials provided with the distribution.
13 —
14 — THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
15 — AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE

```

16 — *IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE*  
17 — *ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE*  
18 — *LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR*  
19 — *CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF*  
20 — *SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS*  
21 — *INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN*  
22 — *CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)*  
23 — *ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE*  
24 — *POSSIBILITY OF SUCH DAMAGE.*

---

25  
26 — *Title* : *OCPBurst Interface Types*  
27 — *Type* : *Type Package*  
28 — *Description* : *Record types for OCPio CDC interface*

---

```
30 LIBRARY ieee;  
31 USE ieee.std_logic_1164.all;  
32 LIBRARY work;  
33 USE work.ocp.all;  
34 PACKAGE OCPIOCDC_types IS  
35  
36     TYPE OCPIOCDCIn_r IS  
37     RECORD  
38         clk_A    : std_logic;  
39         rst_A    : std_logic;  
40         clk_B    : std_logic;  
41         rst_B    : std_logic;  
42         ocpio_B  : ocp_io_s;  
43         ocpio_A  : ocp_io_m;  
44     END RECORD;  
45  
46     TYPE OCPIOCDCOut_r IS  
47     RECORD  
48         ocpio_A  : ocp_io_s;  
49         ocpio_B  : ocp_io_m;  
50     END RECORD;  
51  
52     TYPE asyncIO_A_r IS  
53     RECORD  
54         req      : std_logic;  
55         data     : ocp_io_m;  
56     END RECORD;  
57  
58     TYPE asyncIO_B_r IS  
59     RECORD  
60         ack      : std_logic;  
61         data     : ocp_io_s;  
62     END RECORD;  
63  
64  
65 END OCPIOCDC_types;
```