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

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).

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

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

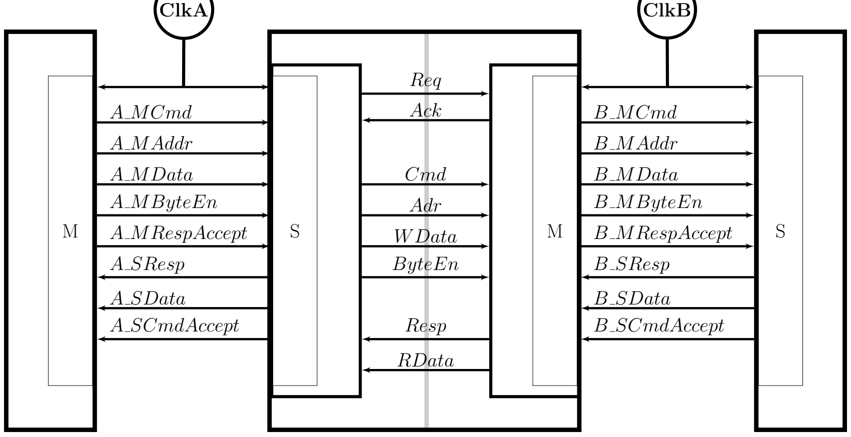

Figure 1: Block diagram of the OCP-to-OCP clock domain crossing module.

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 Accellerata 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.

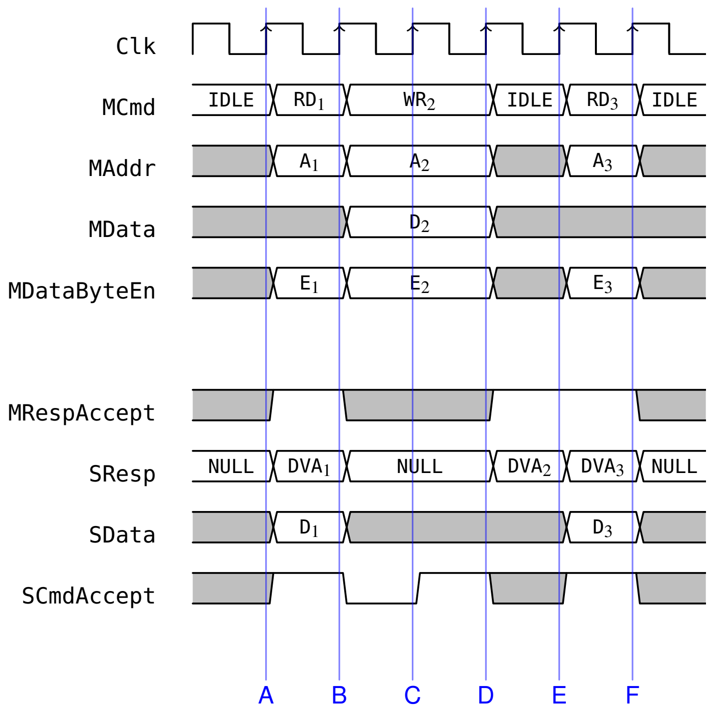

Figure 2: Timing diagram for the `OCPio` interface. (Reprinted from [9] with kind permission of its authors).

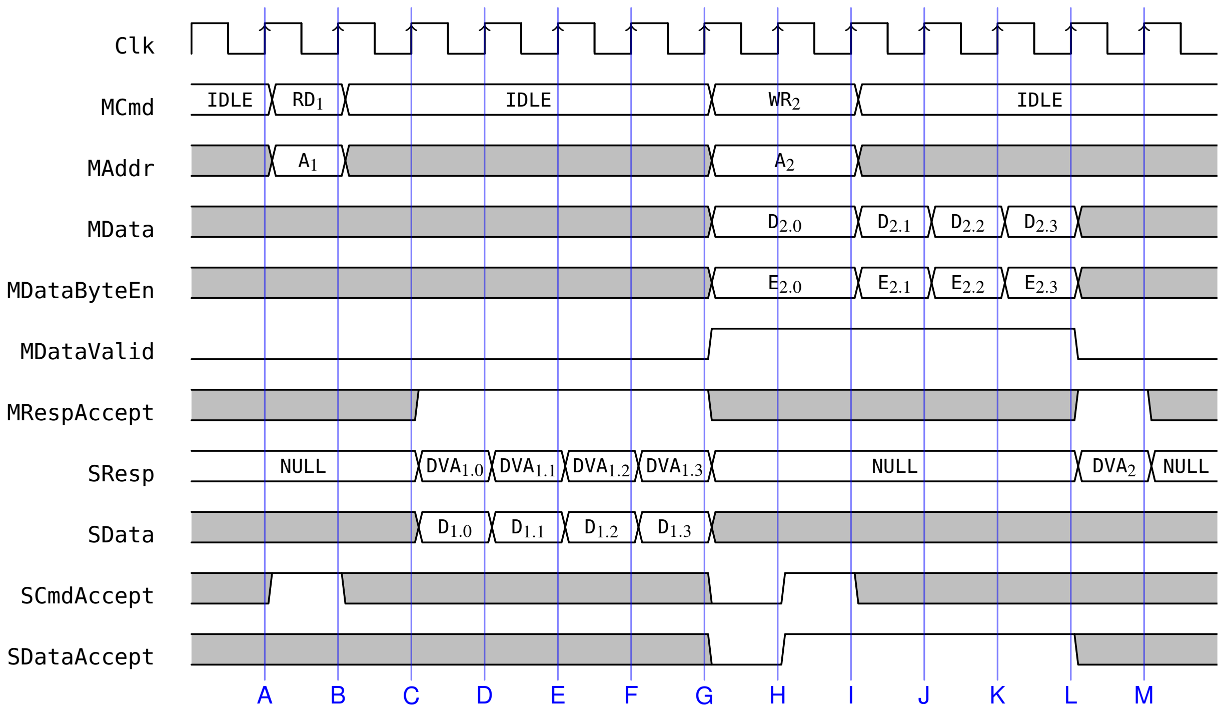

Figure 3: Timing diagram for the `OCPburst` interface. (Reprinted from [9] with kind permission of its authors).

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

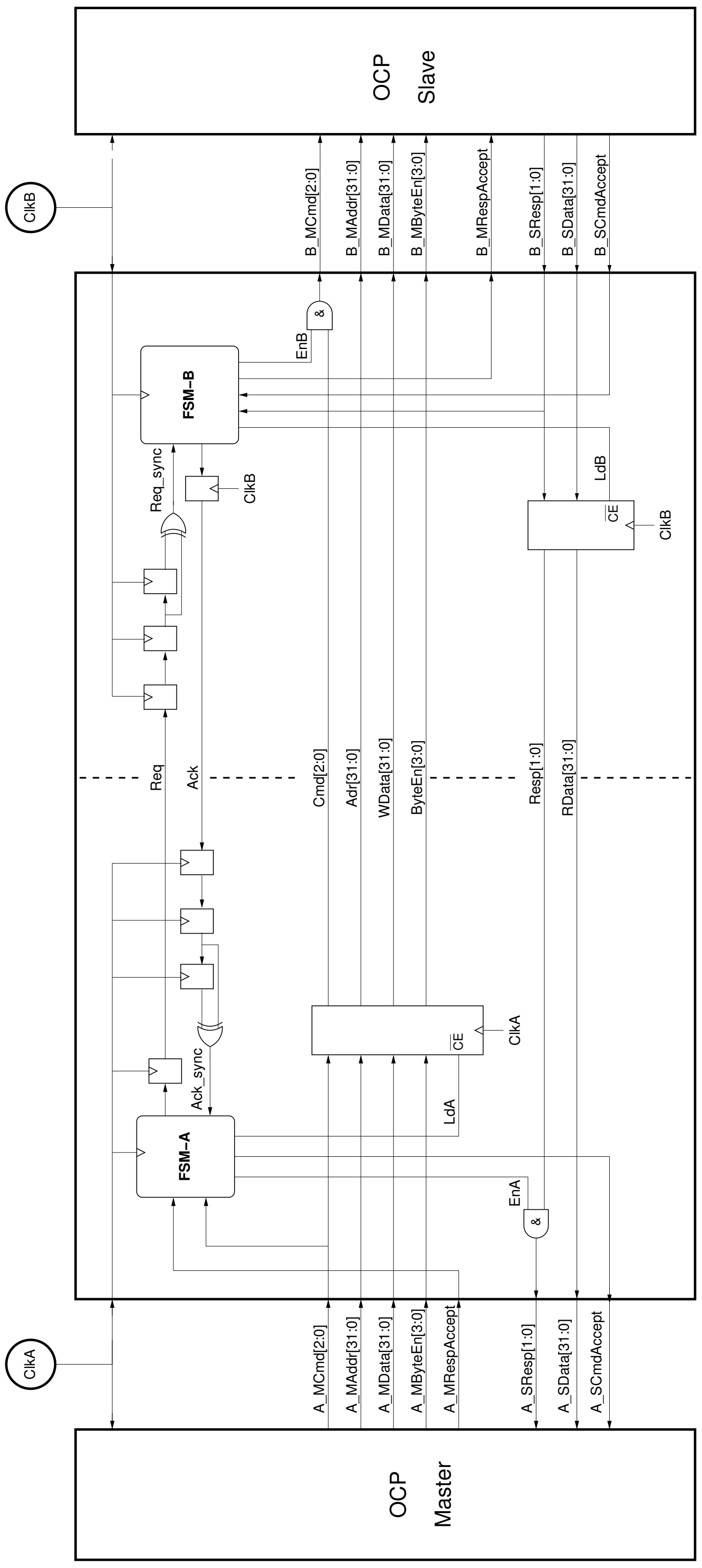

Figure 4: Diagram showing the implementation of the OCPio CDC-module.

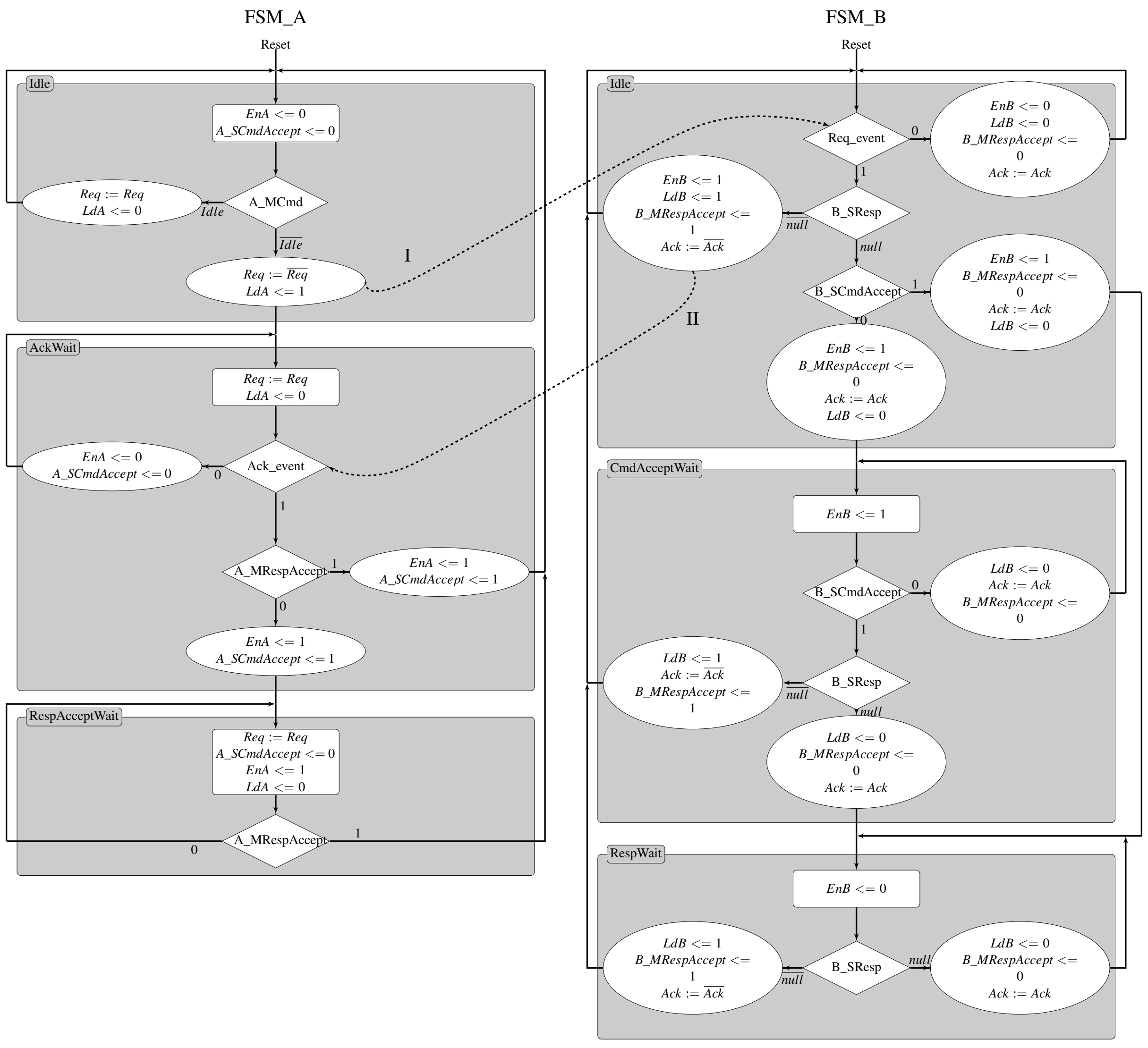

Figure 5: OCPio ASM Chart. The dashed arrows show a request (I) and an acknowledge (II) handshake.

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

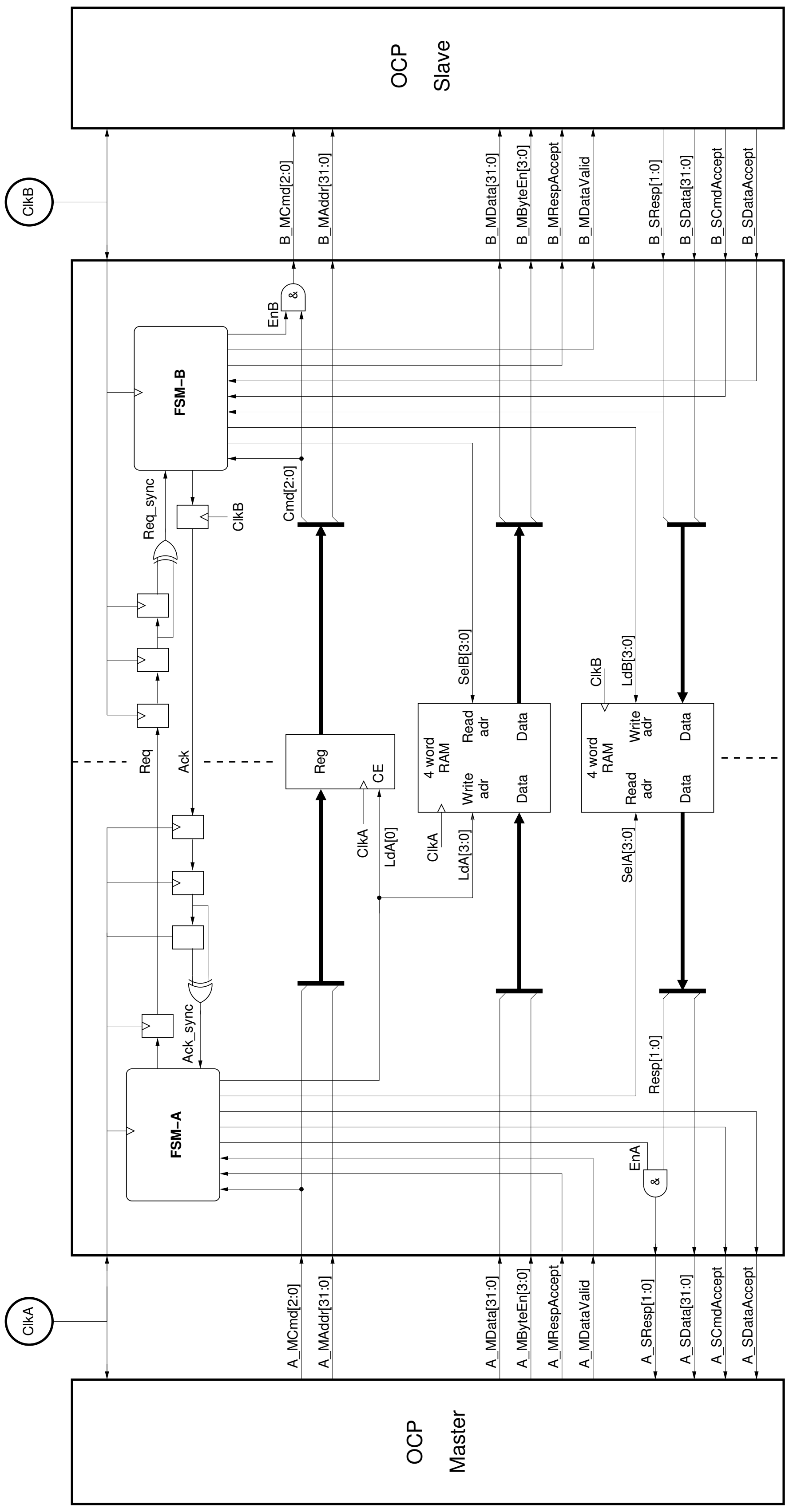

Figure 6: Diagram showing the implementation of the OCPburst CDC-module.

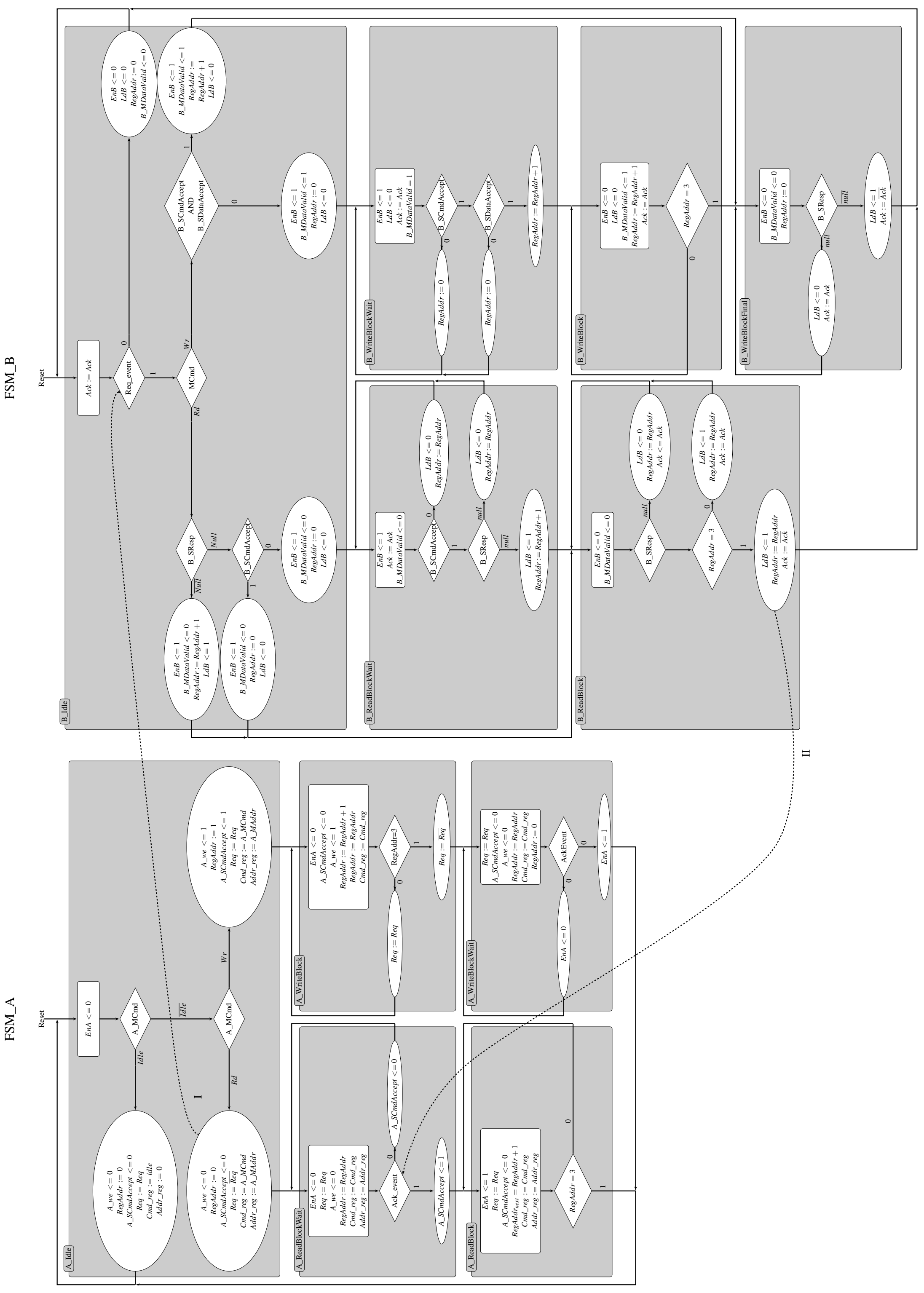

Figure 7: OCPburst ASM Chart. The dashed arrows shows a request (I) and acknowledge (II) handshake

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 `A_MCmd` 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` $\uparrow$

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` $\updownarrow$

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	@`ClkB`
	Rd:	$1 + n_S$ cycles	@`ClkB`
OCPburst	Wr:	$1 + N + n_S$ cycles	@`ClkB`
	Rd:	$1 + N + n_S$ cycles	@`ClkB`

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` $\updownarrow \rightarrow$ `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 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.

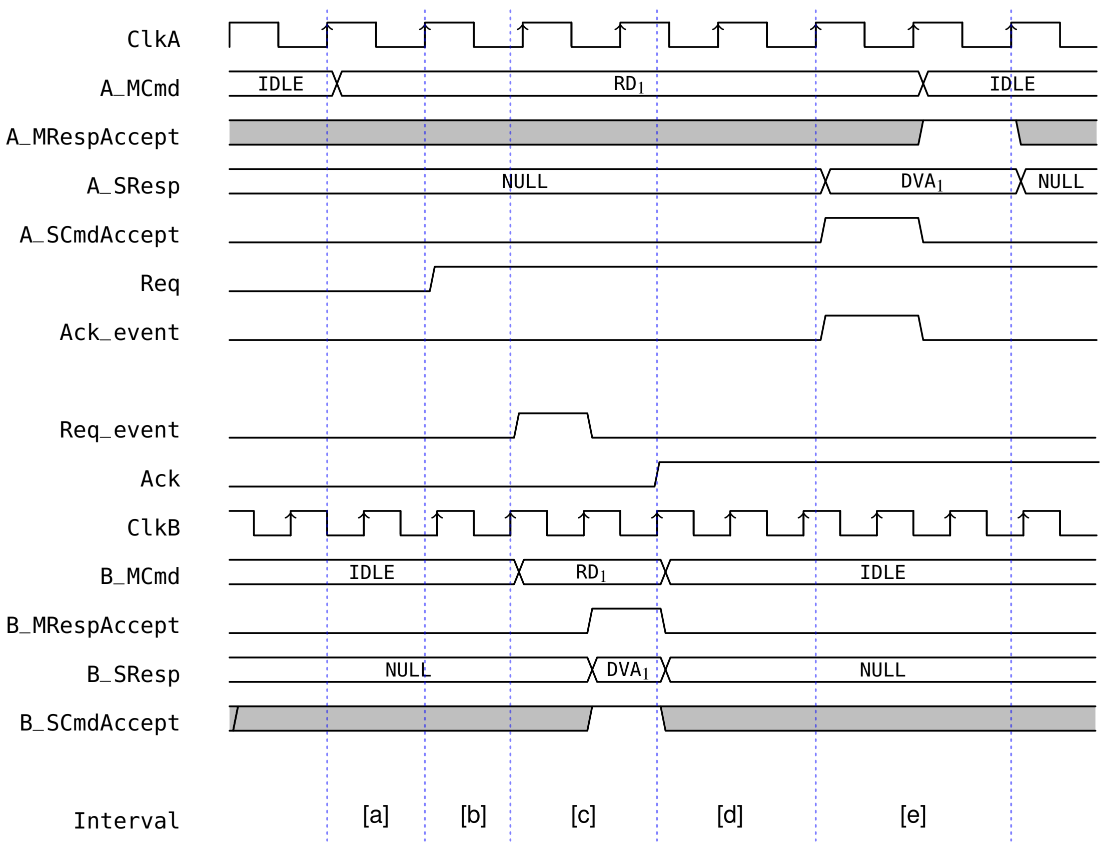

Figure 8: An example read-transaction propagated across the OCPio CDC-module.

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.

$$\begin{aligned} T_{Worst\_OCPio,Rd} &= 1 \cdot T_A + 2 \cdot T_B + (1 + n_S) \cdot T_B + 2 \cdot T_A + (1 + n_M) \cdot T_A && (5.1a) \\ &= (4 + n_M) \cdot T_A + (3 + n_S) \cdot T_B && (5.1b) \end{aligned}$$

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:

$$\begin{aligned} T_{Worst\_OCPio,Rd} &= \left\{ (4 + n_M) + \left\lfloor (3 + n_S) \cdot \frac{T_B}{T_A} \right\rfloor \right\} T_A && (5.2a) \\ &= \left\{ (4 + n_M) + \left\lfloor (3 + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} && (5.2b) \end{aligned}$$

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.

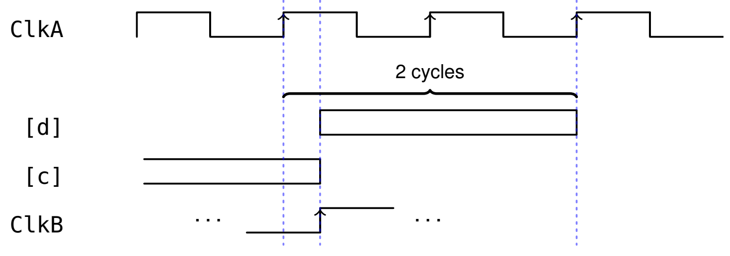

Figure 9: Relationship between intervals [c] and [d].

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

$$\begin{aligned} T_{Worst\_OCPio,Wr} &= \left\{ (4 + n_M) + \left\lfloor (3 + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} && (5.3) \\ T_{Worst\_OCPburst,Wr} &= \left\{ (3 + N) + \left\lfloor (3 + N + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} && (5.4) \\ T_{Worst\_OCPburst,Rd} &= \left\{ 3 + \left\lfloor (3 + N + n_S) \cdot \frac{f_A}{f_b} \right\rfloor \right\} \frac{1}{f_A} && (5.5) \end{aligned}$$

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

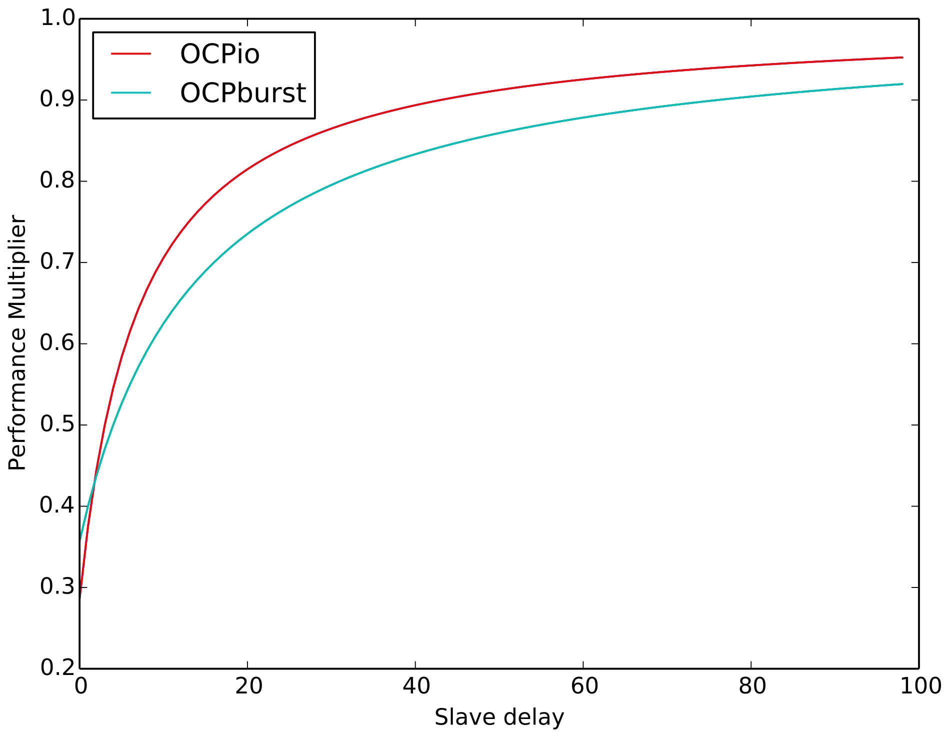

Figure 10: Performance of the two OCP CDC-modules normalized to a situation where the master and slave operate synchronously and are connected directly (without a CDC-module). For a memory the slave delay is the access time of the memory.

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/`, 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`, 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`.

[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

```
--------------------------------------------------------------------------------
-- Copyright (c) 2016, Mathias Herlev
-- All rights reserved.
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions are met:
--
-- 1. Redistributions of source code must retain the above copyright notice,
-- this list of conditions and the following disclaimer.
-- 2. Redistributions in binary form must reproduce the above copyright notice,
-- this list of conditions and the following disclaimer in the documentation
-- and/or other materials provided with the distribution.
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.
--------------------------------------------------------------------------------
-- Title        : OCPIO Clock Crossing Interface Slave
-- Type         : Entity
-- Description  : Slave Interface for the OCP clock crossing. Connects to a
--              : master
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY work;
USE work.OCPIOCDC_types.all;
USE work.ocp.all;

ENTITY OCPIOCDC_A IS
    GENERIC(IOSize : INTEGER := 1);
    PORT(   clk         : IN    std_logic;
            rst         : IN    std_logic;
            syncIn      : IN    ocp_io_m;
            syncOut     : OUT   ocp_io_s;
            asyncOut    : OUT   asyncIO_A_r;
            asyncIn     : IN    asyncIO_B_r
    );
END ENTITY OCPIOCDC_A;
--------------------------------------------------------------------------------
-- Buffered Architecture
--------------------------------------------------------------------------------
ARCHITECTURE Buffered OF OCPIOCDC_A IS
    ----------------------------------------------------------------------------
    -- FSM signals
    ----------------------------------------------------------------------------
    TYPE fsm_states_t IS   (IDLE_state, AckWait_state, RespAcceptWait_state);
    SIGNAL state, state_next     :    fsm_states_t;

    ----------------------------------------------------------------------------
    -- Async signals
    ----------------------------------------------------------------------------
    SIGNAL ack_prev, ack, ack_next  : std_logic := '0';
    SIGNAL req, req_next            : std_logic := '0';

    ----------------------------------------------------------------------------
    -- Registers
    ----------------------------------------------------------------------------
    SIGNAL masterData, masterData_next  : ocp_io_m;
    SIGNAL writeEnable  : std_logic := '0';

BEGIN

    asyncOut.req <= req;
    asyncOut.data   <= masterData;

    ----------------------------------------------------------------------------
    -- FSM
    ----------------------------------------------------------------------------
    FSM : PROCESS(state,syncIn,asyncIn,ack,ack_prev,req)
    BEGIN
        state_next      <= state;
        syncOut         <= OCPIOSlaveIdle_c;
        writeEnable     <= '0';
        req_next        <= req;

        CASE state IS
            WHEN IDLE_state =>
                --If command is different from idle
                IF syncIn.MCmd /= OCP_CMD_IDLE THEN
                    --Signal the B side
                    req_next <= NOT(req);
                    state_next <= AckWait_state;
                    --Buffer the command, address, and data
                    writeEnable <= '1';
                END IF;
            WHEN AckWait_state =>
                --If the slave has acknowledged
                IF ack = NOT (ack_prev) THEN
                    -- Then signal the master
                    state_next <= RespAcceptWait_state;
                    syncOut <= asyncIn.data;
                    syncOut.SCmdAccept <= '1';
                    IF syncIn.MRespAccept = '1'THEN
                        --If the master accepts, go to idle
                        state_next <= IDLE_state;
                    END IF;
                -- Else go to WriteWordFinal
                END IF;
            WHEN RespAcceptWait_state =>
                -- When OCP master accepts, go to idle
                syncOut <= asyncIn.data;
                IF syncIn.MRespAccept = '1'THEN
                    state_next <= IDLE_state;
                END IF;
            WHEN OTHERS =>
                state_next  <= IDLE_state;
        END CASE;
    END PROCESS FSM;

    --------------------------------------------------------------------------------
    -- Registers
    --------------------------------------------------------------------------------
    DataRegMux : PROCESS(writeEnable,syncIn)
    BEGIN
        masterData_next <= masterData;
        IF writeEnable = '1' THEN
            masterData_next <= syncIn;
        END IF;
    END PROCESS DataRegMux;

    Registers : PROCESS(clk,rst)
    BEGIN
        IF rst = '1' THEN
            state       <= IDLE_state;
            req         <= '0';
            ack_prev    <= '0';
            ack         <= '0';
            ack_next    <= '0';
            masterData  <= OCPIOmasteridle_c;

        ELSIF rising_edge(clk) THEN
            state       <= state_next;
            req         <= req_next;
            ack_prev    <= ack;
            ack         <= ack_next;
            ack_next    <= asyncIn.ack;
            masterData  <= masterData_next;
        END IF;
    END PROCESS Registers;

END ARCHITECTURE Buffered;

ARCHITECTURE NonBuffered OF OCPIOCDC_A IS
    TYPE fsm_states_t IS    (IDLE_state, AckWait_state, RespAcceptWait_state,
                             HandshakeFinal_state);
    SIGNAL state, state_next    :    fsm_states_t;

    SIGNAL ack, ack_next    : std_logic := '0';
    SIGNAL req              : std_logic := '0';

BEGIN

    asyncOut.req        <= req;
    asyncOut.data <= syncIn;

    FSM : PROCESS(state,syncIn,asyncIn,ack)
    BEGIN
        state_next  <= state;
        syncOut     <= OCPIOSlaveIdle_c;
        CASE state IS
            WHEN IDLE_state =>
                req <= '0';
                IF ack = '0' THEN
                    IF syncIn.MCmd /= OCP_CMD_IDLE THEN
                        req <= '1';
                        state_next <= AckWait_state;
                    END IF;
                END IF;
            WHEN AckWait_state =>
                req <= '1';
                IF ack = '1' THEN
                    state_next <= RespAcceptWait_state;
                    syncOut <= asyncIn.data;
                    syncOut.SCmdAccept <= '1';
                    IF syncIn.MRespAccept = '1' THEN
--                      req <= '0';
                        state_next <= Idle_state;
                    END IF;
                END IF;
            WHEN RespAcceptWait_state =>
                req <= '1';
                syncOut <= asyncIn.data;
                syncOut.SCmdAccept <= '0';
                IF syncIn.MRespAccept = '1' THEN
                    state_next <= Idle_state;
                END IF;
            WHEN HandshakeFinal_state =>
                req <= '0';
                IF ack = '0' THEN
                    state_next <= Idle_state;
                END IF;
            WHEN OTHERS =>
                state_next  <= IDLE_state;
        END CASE;
    END PROCESS FSM;


    Registers : PROCESS(clk, rst)
    BEGIN
        IF rst = '1' THEN
            state       <= IDLE_state;
            ack         <= '0';
            ack_next    <= '0';
        ELSIF rising_edge(clk) THEN
            state       <= state_next;
            ack         <= ack_next;
            ack_next    <= asyncIn.ack;
        END IF;
    END PROCESS Registers;

END ARCHITECTURE NonBuffered;
```

A.1.2 OCPio B side

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

```vhdl
LIBRARY work;
USE work.OCPIOCDC_types.all;
USE work.ocp.all;

ENTITY OCPIOCDC_B IS
    GENERIC(IOSize : INTEGER := 1);
    PORT(   clk         : IN    std_logic;
            rst         : IN    std_logic;
            syncIn      : IN    ocp_io_s;
            syncOut     : OUT   ocp_io_m;
            asyncOut    : OUT   asyncIO_B_r;
            asyncIn     : IN    asyncIO_A_r
    );
END ENTITY OCPIOCDC_B;

--------------------------------------------------------------------------------
-- Buffered Architecture
--------------------------------------------------------------------------------
ARCHITECTURE Buffered OF OCPIOCDC_B IS
    ----------------------------------------------------------------------------
    -- FSM signals
    ----------------------------------------------------------------------------
    TYPE fsm_states_t IS    (IDLE_state, CmdAcceptWait_state, RespWait_state);
    SIGNAL state, state_next    :   fsm_states_t;
    ----------------------------------------------------------------------------
    -- Async signals
    ----------------------------------------------------------------------------
    SIGNAL req_prev, req, req_next  : std_logic := '0';
    SIGNAL ack, ack_next            : std_logic := '0';

    ----------------------------------------------------------------------------
    -- Register signals
    ----------------------------------------------------------------------------
    SIGNAL slaveData, slaveData_next  : ocp_io_s;
    SIGNAL loadEnable    : std_logic;

BEGIN

    asyncOut.data <= slaveData WHEN loadEnable = '0' ELSE syncIn;
    asyncOut.ack    <= ack;

    ----------------------------------------------------------------------------
    -- FSM
    ----------------------------------------------------------------------------
    FSM : PROCESS(state, syncIn, asyncIn, req, req_prev,ack)
    BEGIN
        state_next  <= state;
        loadEnable  <= '0';
        syncOut     <= OCPIOMasterIdle_c;
        ack_next    <= ack;

        CASE state IS
            WHEN IDLE_state =>
                --If a new request is available
                IF req = NOT (req_prev) THEN
```

```
                    IF (asyncIn.data.MCmd /= OCP_CMD_IDLE) THEN
                        --Relay the command to the OCP slave
                        syncOut <= asyncIn.data;
                        state_next <= CmdAcceptWait_state;
                        -- If the command is accepted
                        IF syncIn.SCmdAccept = '1' THEN
                            state_next <= RespWait_state;
                            --And a response is ready
                            IF syncIn.SResp /= OCP_RESP_NULL THEN
                                state_next <= IDLE_state;
                                loadEnable <= '1';
                                -- Signal the A side
                                ack_next <= NOT (ack);
                                syncOut.MRespAccept <= '1';
                            END IF;
                        END IF;
                    END IF;
                END IF;
            WHEN CmdAcceptWait_state =>
                -- If command has not been accepted, Command+data has not been
                -- Registered in OCP Slave. Continue aserting command.
                syncOut <= asyncIn.data;
                IF syncIn.SCmdAccept = '1' THEN
                    state_next <= RespWait_state;
                    IF syncIn.SResp /= OCP_RESP_NULL THEN
                        state_next <= IDLE_state;
                        ack_next <= NOT (ack);
                        loadEnable <= '1';
                        syncOut.MRespAccept <= '1';
                    END IF;
                END IF;
            WHEN RespWait_state =>
                IF syncIn.SResp /= OCP_RESP_NULL THEN
                    state_next <= IDLE_state;
                    ack_next <= NOT (ack);
                    loadEnable <= '1';
                    syncOut.MRespAccept <= '1';
                END IF;
            WHEN OTHERS =>
                state_next <= IDLE_state;
        END CASE;
    END PROCESS FSM;


    DataRegMux   : PROCESS(loadEnable, syncIn)
    BEGIN
        slaveData_next <= slaveData;
        IF loadEnable = '1' THEN
            slaveData_next <= syncIn;
        END IF;
    END PROCESS DataRegMux;


    Registers   : PROCESS(clk, rst)
    BEGIN
```

```vhdl
        IF rst = '1' THEN
            state <= IDLE_state;
            req_prev    <= '0';
            req         <= '0';
            req_next    <= '0';
            ack         <= '0';
            slaveData   <= ocpioslaveidle_c;
        ELSIF rising_edge(clk) THEN
            state       <= state_next;
            req_prev    <= req;
            req         <= req_next;
            req_next    <= asyncIn.req;
            ack         <= ack_next;
            slaveData   <= slaveData_next;
        END IF;
    END PROCESS Registers;

END ARCHITECTURE Buffered;

------------------------------------------------------------------------
-- Unbuffered architecture
------------------------------------------------------------------------
ARCHITECTURE NonBuffered OF OCPIOCDC_B IS
    TYPE fsm_states_t IS    (Idle_state, CmdAcceptWait_state, RespWait_state,
                             ReqWait_state);
    SIGNAL state, state_next    :    fsm_states_t := Idle_state;

    SIGNAL req, req_next    : std_logic := '0';
    SIGNAL ack  : std_logic := '0';


BEGIN
    -- Async Data Signals

     asyncOut.data       <= syncIn;
     asyncOut.ack        <= ack;

    FSM : PROCESS(state, syncIn, asyncIn, req)
    BEGIN
        state_next  <= state;
        syncOut     <= OCPIOMasterIdle_c;

        CASE state IS
            WHEN Idle_state =>
                ack <= '0';
                IF req = '1' THEN
                    IF (asyncIn.data.MCmd /= OCP_CMD_IDLE) THEN
                        syncOut <= asyncIn.data;
                        syncOut.MRespAccept <= '0';
                        state_next <= CmdAcceptWait_state;
                        IF syncIn.SCmdAccept = '1' THEN
                            state_next <= RespWait_state;
                            IF syncIn.SResp /= OCP_RESP_NULL THEN
                                state_next <= ReqWait_state;
--                              ack <= '1';
```

```
                            END IF;
                        END IF;
                    END IF;
                END IF;
            WHEN CmdAcceptWait_state =>
                ack <= '0';
                syncOut <= asyncIn.data;
                syncOut.MRespAccept <= '0';
                IF syncIn.SCmdAccept = '1' THEN
                    state_next <= RespWait_state;
                    IF syncIn.SResp /= OCP_RESP_NULL THEN
                        state_next <= ReqWait_state;
--                      ack <= '1';
                    END IF;
                END IF;
            WHEN RespWait_state =>
                ack <= '0';
                IF syncIn.SResp /= OCP_RESP_NULL THEN
                    state_next <= ReqWait_state;
                    ack <= '1';
                END IF;
            WHEN ReqWait_state =>
                ack <= '1';
                IF req = '0' THEN
                    ack <= '0';
                    syncOut.MRespAccept <= '1';
                    state_next <= Idle_state;
                END IF;
            WHEN OTHERS =>
                state_next <= IDLE_state;
        END CASE;
    END PROCESS FSM;


    Registers    : PROCESS(clk, rst)
    BEGIN
        IF rst = '1' THEN
            state <= IDLE_state;
            req <= '0';
            req_next <= '0';
        ELSIF rising_edge(clk) THEN
            state       <= state_next;
            req         <= req_next;
            req_next    <= asyncIn.req;
        END IF;
    END PROCESS Registers;

END ARCHITECTURE NonBuffered;
```

A.2 OCPburst

A.2.1 OCPburst A side

```
--------------------------------------------------------------------------------
-- Copyright (c) 2016, Mathias Herlev
-- All rights reserved.
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions are met:
--
-- 1. Redistributions of source code must retain the above copyright notice,
-- this list of conditions and the following disclaimer.
-- 2. Redistributions in binary form must reproduce the above copyright notice,
-- this list of conditions and the following disclaimer in the documentation
-- and/or other materials provided with the distribution.
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.
--------------------------------------------------------------------------------
-- Title        : OCPburst Clock Crossing Interface A Side
-- Type         : Entity
-- Description  : Slave Interface for the OCPburst clock crossing. Connects to a
--              : master
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY work;
USe work.ocp_config.all;
USE work.ocp.all;
USE work.OCPBurstCDC_types.all;

ENTITY OCPBurstCDC_A IS
    GENERIC(burstSize : INTEGER := 4);
    PORT(   clk         : IN    std_logic;
            rst         : IN    std_logic;
            syncIn      : IN    ocp_burst_m;
            syncOut     : OUT   ocp_burst_s;
            asyncOut    : OUT   AsyncBurst_A_r;
            asyncIn     : IN    AsyncBurst_B_r
    );
END ENTITY OCPBurstCDC_A;

ARCHITECTURE behaviour OF OCPBurstCDC_A IS
    ----------------------------------------------------------------------------
    -- Constants
    ----------------------------------------------------------------------------
CONSTANT OCPBurstSlaveIdle_c : ocp_burst_s := (OCP_RESP_NULL,
                                               (OTHERS => '0'),
                                               '0',
                                               '0');
    ----------------------------------------------------------------------------
    -- FSM Signal Declarations
    ----------------------------------------------------------------------------
    TYPE fsm_states_t IS (  IDLE_state, ReadBlock, ReadBlockWait,
                            WriteBlockLoad, WriteBlockWait);
    SIGNAL state, state_next     :    fsm_states_t;
    ----------------------------------------------------------------------------
    -- Data Registers
    ----------------------------------------------------------------------------
    SIGNAL cmd, cmd_next      : std_logic_vector(OCP_CMD_WIDTH-1 downto 0)
                              := OCP_CMD_IDLE;
    SIGNAL addr, addr_next    : std_logic_vector(OCP_BURST_ADDR_WIDTH-1 downto 0)
                              := (others => '0');

    TYPE DataArray_t     IS
        ARRAY (burstSize-1 downto 0) OF
        std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
    SIGNAL data_arr : DataArray_t;

    TYPE ByteEn_Array_t IS
        ARRAY (burstSize downto 0) OF
        std_logic_vector(OCP_BYTE_WIDTH-1 downto 0);
    SIGNAL byteEn_arr    : ByteEn_Array_t;

    SIGNAL writeEnable                    : std_logic := '0';

    SIGNAL RegAddr, RegAddr_next     : unsigned(1 downto 0) := (others => '0');
    ----------------------------------------------------------------------------
    -- Asynchronous signals
    ----------------------------------------------------------------------------
    ALIAS o_async IS asyncOut;
    ALIAS i_async IS asyncIn;

    SIGNAL ack_prev, ack, ack_next   : std_logic := '0';
    SIGNAL req, req_next             : std_logic := '0';

BEGIN


    FSM : PROCESS(state,syncIn,asyncIn,ack,ack_prev,RegAddr, req, cmd, addr)
    BEGIN
        ------------------------------------------------------------------------
        -- Default Assignments
        ------------------------------------------------------------------------
        state_next       <= state;
        syncOut          <= OCPBurstSlaveIdle_c;
        writeEnable      <= '0';
        req_next <= req;
        cmd_next <= cmd;
        addr_next <= addr;
        RegAddr_next <= RegAddr;
        asyncOut.RegAddr     <= (others => '0');
        asyncOut.data.MDataValid <= '0';
        syncOut.SCmdAccept <= '0';
        syncOut.SDataAccept <= '0';
```

```vhdl

        CASE state IS
            WHEN IDLE_state =>
                --If command is read
                IF syncIn.Mcmd = OCP_CMD_RD THEN
                    -- Register Command and addres (MData not valid)
                    cmd_next <= syncIn.MCmd;
                    addr_next <= syncIn.MAddr;
                    -- Assert a request
                    req_next        <= NOT (req);
                    -- And go to ReadBlockWait, to await an acknowledge
                    state_next      <= ReadBlockWait;
                --If command is write
                ELSIF syncIn.Mcmd = OCP_CMD_WR AND syncIn.MDataValid = '1' THEN
                    --Start buffering MData + MCmd + MAddr
                    cmd_next <= syncIn.MCmd;
                    addr_next <= syncIn.MAddr;
                    syncOut.SCmdAccept  <= '1';
                    syncOut.SDataAccept <= '1';
                    writeEnable     <= '1';
                    RegAddr_next    <= RegAddr + to_unsigned(1,RegAddr'LENGTH);
                    state_next      <= WriteBlockLoad;
                END IF;
            ---------------------------------------------------------------------
            -- READ BLOCK
            ---------------------------------------------------------------------
            WHEN ReadBlockWait =>
                -- Wait until acknowledge
                IF ack = NOT(ack_prev) THEN
                    state_next          <= ReadBlock;
                    syncOut.SCmdAccept  <= '1';
                END IF;
            WHEN ReadBlock =>
                -- Write each word in buffer back to OCP Master
                asyncOut.RegAddr    <= std_logic_vector(RegAddr);
                syncOut         <= asyncIn.data;
                RegAddr_next    <= RegAddr + to_unsigned(1,RegAddr'LENGTH);
                IF RegAddr = to_unsigned(burstSize-1, RegAddr'LENGTH) THEN
                    state_next      <= IDLE_state;
                END IF;
            ---------------------------------------------------------------------
            -- WRITE BLOCK
            ---------------------------------------------------------------------
            WHEN WriteBlockLoad =>
                -- Continue buffering MData
                syncOut.SDataAccept  <= '1';
                writeEnable         <= '1';
                RegAddr_next        <= RegAddr + to_unsigned(1,RegAddr'LENGTH);
                IF RegAddr = to_unsigned(burstSize-1, RegAddr'LENGTH) THEN
                    -- And assert request once all words are buffered
                    req_next                    <= NOT (req);
                    asyncOut.data.MDataValid    <= '1';
                    state_next                  <= WriteBlockWait;
                END IF;
            WHEN WriteBlockWait =>
```

```
                -- Wait until B side has acknowledged finishing transaction
                IF ack = NOT(ack_prev) THEN
                    --And relay response to OCP Master
                    syncOut.Sresp    <= asyncIn.data.Sresp;
                    state_next       <= IDLE_state;
                    cmd_next <= OCP_CMD_IDLE;
                    addr_next <= (others => '0');
                END IF;
            WHEN OTHERS =>
                state_next  <= IDLE_state;
        END CASE;
    END PROCESS FSM;

    ----------------------------------------------------------------------------
    -- Output Map
    ----------------------------------------------------------------------------
    asyncOut.data.MCmd  <= cmd;
    asyncOut.data.MData <= data_arr(to_integer(unsigned(i_async.RegAddr)));
    asyncOut.data.MAddr <= addr;
    asyncOut.data.MDataByteEn <=
                        byteEn_arr(to_integer(unsigned(i_async.RegAddr)));
    asyncOut.req        <= req;

    ----------------------------------------------------------------------------
    -- Register Processes
    ----------------------------------------------------------------------------
    Registers : PROCESS(clk, rst)
    BEGIN
        IF rst = '1' THEN
            state       <= IDLE_state;
            req         <= '0';
            ack_prev    <= '0';
            ack         <= '0';
            ack_next    <= '0';
            RegAddr     <= (others=>'0');
            cmd         <= OCP_CMD_IDLE;
            addr        <= (others => '0');
    ELSIF rising_edge(clk) THEN
            state       <= state_next;
            req         <= req_next;
            ack_prev    <= ack;
            ack         <= ack_next;
            ack_next    <= asyncIn.ack;
            RegAddr     <= RegAddr_next;
            cmd         <= cmd_next;
            addr        <= addr_next;
        END IF;
    END PROCESS Registers;

    DataRam : PROCESS(clk)
    BEGIN
        IF rising_edge(clk) THEN
            IF writeEnable = '1' THEN
                data_arr(to_integer(RegAddr)) <= syncIn.MData;
            END IF;
            END IF;
        END PROCESS DataRam;

        ByteEnRam : PROCESS(clk)
        BEGIN
            IF rising_edge(clk) THEN
                IF writeEnable = '1' THEN
                    byteEn_arr(to_integer(RegAddr)) <= syncIn.MDataByteEn;
                END IF;
            END IF;
        END PROCESS ByteEnRam;

END ARCHITECTURE behaviour;
```

A.2.2 OCPburst B side

```
--------------------------------------------------------------------------------
-- Copyright (c) 2016, Mathias Herlev
-- All rights reserved.
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions are met:
--
-- 1. Redistributions of source code must retain the above copyright notice,
-- this list of conditions and the following disclaimer.
-- 2. Redistributions in binary form must reproduce the above copyright notice,
-- this list of conditions and the following disclaimer in the documentation
-- and/or other materials provided with the distribution.
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.
--------------------------------------------------------------------------------
-- Title        : OCPBurst Clock Crossing Interface Slave
-- Type         : Entity
-- Description  : Master Interface for the OCP clock crossing. Connects to a
--              : Slave
--------------------------------------------------------------------------------

LIBRARY ieee;
USE ieee.std_logic_1164.all;
USE ieee.numeric_std.all;
LIBRARY work;
USE work.ocp.all;
USE work.OCPBurstCDC_types.all;

ENTITY OCPBurstCDC_B IS
    GENERIC(burstSize : INTEGER := 4);
    PORT(   clk         : IN    std_logic;
            rst         : IN    std_logic;
            syncIn      : IN    ocp_burst_s;
            syncOut     : OUT   ocp_burst_m;
            asyncOut    : OUT   AsyncBurst_B_r;
            asyncIn     : IN    AsyncBurst_A_r
    );
END ENTITY OCPBurstCDC_B;

ARCHITECTURE behaviour OF OCPBurstCDC_B IS
    ------------------------------------------------------------------------
    -- FSM signals
    ------------------------------------------------------------------------
    TYPE fsm_states_t IS (  IDLE_state, ReadBlock, ReadBlockWait,
                            WriteBlock, WriteBlockWait, WriteBlockFinal);
    SIGNAL state, state_next    :   fsm_states_t;
    ------------------------------------------------------------------------
    -- Register signals
    ------------------------------------------------------------------------
    SIGNAL RegAddr, RegAddr_next    : unsigned(1 downto 0) := (others => '0');

    TYPE DataArray_t IS
        ARRAY (burstSize-1 downto 0) OF
         std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
    TYPE RespArray_t IS
        ARRAY (burstSize-1 downto 0) OF
         std_logic_vector(OCP_RESP_WIDTH-1 downto 0);

    SIGNAL data_arr : DataArray_t;
    SIGNAL resp_arr : RespArray_t;

    SIGNAL loadEnable   : std_logic;
    ------------------------------------------------------------------------
    -- Async signals
    ------------------------------------------------------------------------

    SIGNAL req_prev, req, req_next  : std_logic := '0';
    SIGNAL ack, ack_next            : std_logic := '0';

BEGIN
    asyncOut.ack    <= ack;
    asyncOut.data.SResp <= resp_arr(to_integer(unsigned(asyncIn.RegAddr)));
    asyncOut.data.SData <= data_arr(to_integer(unsigned(asyncIn.RegAddr)));

    ------------------------------------------------------------------------
    -- FSM
    ------------------------------------------------------------------------
    FSM : PROCESS(state, syncIn, asyncIn, req, req_prev,RegAddr, ack)
    BEGIN
        state_next  <= state;
        loadEnable  <= '0';
        RegAddr_next <= RegAddr;
        syncOut.MCmd <= OCP_CMD_IDLE;
        syncOut.MAddr <= (others => '0');
        syncOut.MData <= (others => '0');
        syncOut.MDataByteEn <= (others => '0');
        asyncOut.RegAddr     <= (others => '0');
        syncOut.MDataValid <= '0';
        ack_next <= ack;
        CASE state IS
            WHEN IDLE_state =>
                --Wait for request
                IF req = NOT(req_prev) THEN
                    --If read command
                    IF asyncIn.data.MCmd = OCP_CMD_RD THEN
                        --Relay command to OCP slave and either go to wait state
                        --or commence buffering read data
                        state_next       <= ReadBlockWait;
                        syncOut.MCmd     <= OCP_CMD_RD;
                        IF syncIn.SCmdAccept = '1' THEN
                            state_next  <= ReadBlock;
                        END IF;
                    --If write command
                    ElSIF asyncIn.data.MCmd = OCP_CMD_WR THEN
                        state_next            <= WriteBlockWait;
                        syncOut.MCmd           <= OCP_CMD_WR;
                        syncOut.MDataValid     <= '1';
                        syncOut.MDataByteEn <= asyncIn.data.MDataByteEn;
                        syncOut.MAddr         <= asyncIn.data.MAddr;
                        syncOut.MData         <= asyncIn.data.MData;
                        IF syncIn.SCmdAccept = '1' AND syncIn.SDataAccept = '1'
                        THEN
                            RegAddr_next <= RegAddr +
                                            to_unsigned(1,RegAddr'LENGTH);
                            state_next  <= WriteBlock;
                        END IF;
                    END IF;
                END IF;
            ----------------------------------------------------------------
            -- READ BLOCK
            ----------------------------------------------------------------
            WHEN ReadBlockWait =>
                syncOut.MCmd <= OCP_CMD_RD;
                IF syncIn.SCmdAccept = '1' THEN
                    state_next <= ReadBlock;
                END IF;
            WHEN ReadBlock =>
                IF syncIn.SResp /= OCP_RESP_NULL THEN
                    loadEnable <= '1';
                    RegAddr_next <= RegAddr + to_unsigned(1, RegAddr'LENGTH);
                    IF RegAddr = to_unsigned(burstSize-1,RegAddr'LENGTH) THEN
                        state_next <= IDLE_state;
                        ack_next <= NOT (ack);
                    END IF;
                END IF;
            -- WRITE BLOCK
            WHEN WriteBlockWait =>
                syncOut.MCmd          <= OCP_CMD_WR;
                syncOut.MDataValid <= '1';
                syncOut.MAddr          <= asyncIn.data.MAddr;
                syncOut.MDataByteEn <= asyncIn.data.MDataByteEn;
                syncOut.MData       <= asyncIn.data.MData;
                asyncOut.RegAddr    <= std_logic_vector(RegAddr);

                IF syncIn.SCmdAccept = '1' AND syncIn.SDataAccept = '1' THEN
                    RegAddr_next <= RegAddr + to_unsigned(1,RegAddr'LENGTH);
                    state_next   <= WriteBlock;
                END IF;
            WHEN WriteBlock =>
                -- Sync Data Signals
                syncOut.MDataValid  <= '1';
                syncOut.MDataByteEn <= asyncIn.data.MDataByteEn;
                syncOut.MAddr       <= asyncIn.data.MAddr;
                syncOut.MData       <= asyncIn.data.MData;
                RegAddr_next        <= RegAddr + to_unsigned(1,RegAddr'LENGTH);
                asyncOut.RegAddr    <= std_logic_vector(RegAddr);
                IF RegAddr = to_unsigned(burstSize-1, RegAddr'LENGTH) THEN
                    state_next <= WriteBlockFinal;
                    --ack_next <= NOT (ack);
                END IF;
            WHEN WriteBlockFinal =>
                IF syncIn.SResp /= OCP_RESP_NULL THEN
                    ack_next <= NOT(ack);
                    loadEnable <= '1';
                    state_next <= IDLE_state;
                END IF;
            WHEN OTHERS =>
                state_next <= IDLE_state;
        END CASE;
    END PROCESS FSM;

    ------------------------------------------------------------------------
    -- Registers
    ------------------------------------------------------------------------
    Registers    : PROCESS(clk, rst)
    BEGIN
        IF rst = '1' THEN
            state <= IDLE_state;
            req_prev    <= '0';
            req         <= '0';
            req_next    <= '0';
            ack         <= '0';
            RegAddr     <= (others=>'0');
        ELSIF rising_edge(clk) THEN
            state       <= state_next;
            req_prev    <= req;
            req         <= req_next;
            req_next    <= asyncIn.req;
            ack         <= ack_next;
            RegAddr     <= RegAddr_next;
        END IF;
    END PROCESS Registers;

    DataRam  : PROCESS(clk)
    BEGIN
```

```
            IF rising_edge(clk) THEN
                IF loadEnable = '1' THEN
                    data_arr(to_integer(RegAddr)) <= syncIn.SData;
                END IF;
            END IF;
        END PROCESS DataRam;

        RespRam : PROCESS(clk)
        BEGIN
            IF rising_edge(clk) THEN
                IF loadEnable = '1' THEN
                    resp_arr(to_integer(RegAddr)) <= syncIn.SResp;
                END IF;
            END IF;
        END PROCESS RespRam;

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

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


```

```
--------------------------------------------------------------------------------
-- Definitions package
--
-- Author: Evangelia Kasapaki
-- Author: Rasmus Bo Soerensen
--------------------------------------------------------------------------------

library ieee;
use ieee.std_logic_1164.all;

use work.ocp_config.all;

package ocp is

        -- OCP
    constant OCP_CMD_WIDTH  : integer := 3;     -- 8 possible cmds --> 2
    constant OCP_ADDR_WIDTH : integer := 32;    --32
    constant OCP_BURST_ADDR_WIDTH : integer := BURST_ADDR_WIDTH;    --32
    constant OCP_DATA_WIDTH : integer := 32;
    constant OCP_BYTE_WIDTH : integer := OCP_DATA_WIDTH/8;
    constant OCP_RESP_WIDTH : integer := 2;

    constant OCP_CMD_IDLE : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "000";
    constant OCP_CMD_WR   : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "001";
    constant OCP_CMD_RD   : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "010";
    --constant OCP_CMD_RDEX : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "011";
    --constant OCP_CMD_RDL  : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "100";
    --constant OCP_CMD_WRNP : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "101";
    --constant OCP_CMD_WRC  : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "110";
    --constant OCP_CMD_BCST : std_logic_vector(OCP_CMD_WIDTH-1 downto 0) := "111";

    constant OCP_RESP_NULL : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "00";
    constant OCP_RESP_DVA  : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "01";
    constant OCP_RESP_FAIL : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "10";
    constant OCP_RESP_ERR  : std_logic_vector(OCP_RESP_WIDTH-1 downto 0) := "11";

    type ocp_core_m is record
        MCmd        : std_logic_vector(OCP_CMD_WIDTH-1 downto 0);
        MAddr       : std_logic_vector(OCP_ADDR_WIDTH-1 downto 0);
        MData       : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
        MByteEn     : std_logic_vector(OCP_BYTE_WIDTH-1 downto 0);
    end record;

    type ocp_core_s is record
        SResp       : std_logic_vector(OCP_RESP_WIDTH-1 downto 0);
        SData       : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
    end record;

    type ocp_io_m is record
        MCmd        : std_logic_vector(OCP_CMD_WIDTH-1 downto 0);
        MAddr       : std_logic_vector(OCP_ADDR_WIDTH-1 downto 0);
        MData       : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
        MByteEn     : std_logic_vector(OCP_BYTE_WIDTH-1 downto 0);
        MRespAccept : std_logic;
    end record;

    type ocp_io_s is record
        SResp       : std_logic_vector(OCP_RESP_WIDTH-1 downto 0);
        SData       : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
        SCmdAccept  : std_logic;
    end record;

    type ocp_burst_m is record
        MCmd        : std_logic_vector(OCP_CMD_WIDTH-1 downto 0);
        MAddr       : std_logic_vector(OCP_BURST_ADDR_WIDTH-1 downto 0);
        MData       : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
        MDataByteEn : std_logic_vector(OCP_BYTE_WIDTH-1 downto 0);
        MDataValid  : std_logic;
    end record;

    type ocp_burst_s is record
        SResp       : std_logic_vector(OCP_RESP_WIDTH-1 downto 0);
        SData       : std_logic_vector(OCP_DATA_WIDTH-1 downto 0);
        SCmdAccept  : std_logic;
        SDataAccept : std_logic;
    end record;

end package ; -- ocp
```

A.3.2 OCPburst CDC Types

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

```vhdl
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.ocp.all;

PACKAGE OCPBurstCDC_types IS

    TYPE OCPBurstCDCIn_r IS
    RECORD
        clk_A        : std_logic;
        rst_A        : std_logic;
        clk_B        : std_logic;
        rst_B        : std_logic;
        OCPB_slave   : ocp_burst_s;
        OCPB_master  : ocp_burst_m;
    END RECORD;

    TYPE OCPBurstCDCOut_r IS
    RECORD
        OCPB_A   : ocp_burst_s;
        OCPB_B   : ocp_burst_m;
    END RECORD;

    TYPE AsyncBurst_A_r IS
    RECORD
        req      : std_logic;
        Data     : ocp_burst_m;
        RegAddr  : std_logic_vector(1 downto 0);
    END RECORD;

    TYPE AsyncBurst_B_r IS
    RECORD
        ack      : std_logic;
        Data     : ocp_burst_s;
        RegAddr  : std_logic_vector(1 downto 0);
    END RECORD;
END OCPBurstCDC_types;
```

A.3.3 OCPio CDC Types

```vhdl
--------------------------------------------------------------------------------
-- Copyright (c) 2016, Mathias Herlev
-- All rights reserved.
--
-- Redistribution and use in source and binary forms, with or without
-- modification, are permitted provided that the following conditions are met:
--
-- 1. Redistributions of source code must retain the above copyright notice,
-- this list of conditions and the following disclaimer.
-- 2. Redistributions in binary form must reproduce the above copyright notice,
-- this list of conditions and the following disclaimer in the documentation
-- and/or other materials provided with the distribution.
--
-- THIS SOFTWARE IS PROVIDED BY THE COPYRIGHT HOLDERS AND CONTRIBUTORS "AS IS"
-- AND ANY EXPRESS OR IMPLIED WARRANTIES, INCLUDING, BUT NOT LIMITED TO, THE
-- IMPLIED WARRANTIES OF MERCHANTABILITY AND FITNESS FOR A PARTICULAR PURPOSE
-- ARE DISCLAIMED. IN NO EVENT SHALL THE COPYRIGHT OWNER OR CONTRIBUTORS BE
-- LIABLE FOR ANY DIRECT, INDIRECT, INCIDENTAL, SPECIAL, EXEMPLARY, OR
-- CONSEQUENTIAL DAMAGES (INCLUDING, BUT NOT LIMITED TO, PROCUREMENT OF
-- SUBSTITUTE GOODS OR SERVICES; LOSS OF USE, DATA, OR PROFITS; OR BUSINESS
-- INTERRUPTION) HOWEVER CAUSED AND ON ANY THEORY OF LIABILITY, WHETHER IN
-- CONTRACT, STRICT LIABILITY, OR TORT (INCLUDING NEGLIGENCE OR OTHERWISE)
-- ARISING IN ANY WAY OUT OF THE USE OF THIS SOFTWARE, EVEN IF ADVISED OF THE
-- POSSIBILITY OF SUCH DAMAGE.
--------------------------------------------------------------------------------
-- Title        : OCPBurst Interface Types
-- Type         : Type Package
-- Description  : Record types for OCPio CDC interface
--------------------------------------------------------------------------------
LIBRARY ieee;
USE ieee.std_logic_1164.all;
LIBRARY work;
USE work.ocp.all;
PACKAGE OCPIOCDC_types IS

    TYPE OCPIOCDCIn_r IS
    RECORD
        clk_A   : std_logic;
        rst_A   : std_logic;
        clk_B   : std_logic;
        rst_B   : std_logic;
        ocpio_B : ocp_io_s;
        ocpio_A : ocp_io_m;
    END RECORD;

    TYPE OCPIOCDCOut_r IS
    RECORD
        ocpio_A : ocp_io_s;
        ocpio_B : ocp_io_m;
    END RECORD;

    TYPE asyncIO_A_r IS
    RECORD
        req     : std_logic;
        data    : ocp_io_m;
    END RECORD;

    TYPE asyncIO_B_r IS
    RECORD
        ack     : std_logic;
        data    : ocp_io_s;
    END RECORD;


END OCPIOCDC_types;
```