

Chapter 2

OCPI Training Wheels

Summary This chapter introduces fundamental OCPI interface configuration and signaling concepts using simplistic examples with accompanying explanations. Later chapters assume knowledge of concepts developed here. All timing diagram examples in this chapter and the entire book examine signals at strategic interest points to highlight specific concepts. Hence they do not provide full explanations. For detailed information on any topic, refer to the latest OCPI Specification, available from OCPI-IP at www.OCPIP.org.

2.1 Simplistic OCPI Write

We begin studying OCPI interfaces with Fig. 2.1. This diagram depicts the most simplistic OCPI transfer possible – one type of an OCPI write operation – using a traditional signal timing diagram.

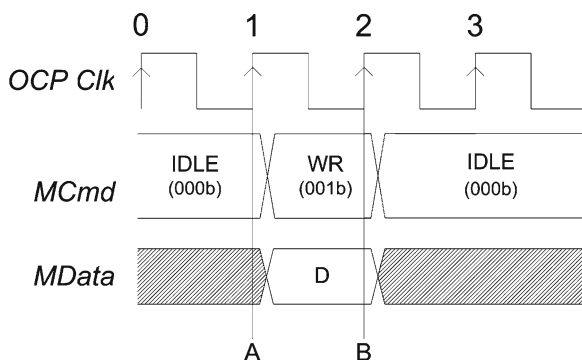
Examining Fig. 2.1, note it depicts a traditional clock signal with rising and falling edges labeled *OCPI Clk*. This is the *OCPI Clock* signal. All OCPI signaling, other than reset signaling, references the OCPI Clock and samples signals on the OCPI Clock's rising edge.

During clock cycle 0, the three-wire *MCmd* signal asserts an IDLE value (000b) to the slave core. This indicates to the slave the master core sharing the interface is not presently presenting a request (requesting transfer activity). The hatched area in the *MData* signal during OCPI Clock cycle 0 therefore indicates the *MData* signal is in a *don't care* state during OCPI Clock cycle 0.

At time point A, the OCPI clock presents a rising edge, beginning OCPI Clock cycle 1. Point A in this example therefore demarks the end of OCPI Clock cycle 0 and beginning of OCPI clock cycle 1 across all OCPI signals.

The master senses the beginning of this new OCPI clock cycle via the rising OCPI Clock edge and transitions the three-wire *MCmd* signal to a *posted write* (*WR* mnemonic, 001b value). Note that posted writes, by definition, do not expect

Fig. 2.1 Simplistic OCP posted write



acknowledgements. In practice, this is one of two possible non-posted write OCP models OCP. In the other model, both posted and non-posted writes have responses – useful with many interconnect designs. This allows designs to depend on eventual response appearances and removes some special cases in the circuitry. Here, the posted writes remain semantically different from non-posted writes; “early” responses to posted writes are allowed, for example from a write buffer. Non-posted write response receipt guarantees data visibility but posted write responses merely complete transactions on the local point-to-point OCP interface. Simultaneously, the master encodes the Mdata signal with the data values it wants the slave to write.

The master continues asserting these values for OCP Clock cycle 1’s full duration. At time point B, the OCP Clock presents another rising edge, signifying the end of OCP Clock cycle 1 beginning of OCP Clock cycle 2. The master detects this, knows clock cycle 1 is over, and transitions the MCmd signal to an IDLE value. This simultaneously causes the MData signal values to revert to the *don’t care* state.

That’s it – all of it. You have just experienced a complete OCP transfer. What could be simpler?

At this point, you are likely sensing a profound sense of engineering incredulity. And, you would be correct because the simplistic example begs a number of obvious questions. Here are just a few of them:

1. What is this *OCP Clock*?
2. How much data transfers in one transfer?
3. How does the design ensure the slave actually performed, or was even ready for, the transfer?
4. Why does the transfer mysteriously end at time point B?
5. Shouldn’t there be some way for the master to receive a response indicating a successful commit following the transfer operation?

These, and many others, are all appropriate questions because the above example deliberately omitted essential information by construction. Not only that, it intentionally contained a convenient, but somewhat misleading, picture error. Let’s resolve these example shortcomings now.

2.2 The OCP RTL Configuration File

What was not mentioned is that both sides of an OCP interface, one at the master and the other at the slave, have an associated RTL configuration file. For the two OCP sides to interoperate correctly, it is important for these two RTL configuration files to be *compatible*.

A full blown OCP interface can have dozens of signals and only three appear in the above example. Some OCP signals have fixed widths while others have variable, configurable widths. An OCP configuration file specifies what optional signals are present and, when they have a variable width, what their configured width is. Signals not explicitly specified may be assumed present and are assigned default *tie-down* values, resulting in what may seem to be phantom signals mysteriously affecting transfers.

2.3 Deriving the OCP Clock

With respect to the OCP Clock question, the OCP Clock signal is actually a *derived* signal, derived from two input signals to both the master and slave:

1. The *Clk* signal (the main clock signal)
2. The *EnableClk* signal – a signal a third entity provides as both a master and the slave input

In the above example, the OCP Clock signal shape suggests the configuration files for both the master and slave either:

- Omitted configuring the EnableClk signal using the configuration file enableclk parameter – causing it to default to a value of ‘1’ (constantly asserted)
- Configured the presence of the EnableClk signal using the configuration file enableclk parameter and the EnableClk signal is always asserted
- Designed the EnableClk signal with a constant tie-off value of ‘1’. (constantly asserted)

In any instance, the EnableClk signal in this simplistic example is *constantly asserted* as a result, causing the Clk signal to become the OCP Clock. Hence, the OCP Clock in this example has the standard square wave form of a traditional clock. However, from the OCP specification:

The rising edge of the OCP clock is defined as a rising edge of Clk that samples the asserted EnableClk. Falling edges of Clk and any rising edge of Clk that does not sample EnableClk asserted do not constitute rising edges of the OCP clock.

In the general instance then, the OCP clock really does not have falling edges, only rising edges. Figure 2.2 illustrates this with only time points A, B, C, and D presenting OCP Clock rising edges.

Figure 2.3 provides an illustration of the previous simplistic posted-write transfer depicting these signals. Here, the logic uses every other Clk cycle, generating an OCP Clk signal with half the frequency of the input Clk signal. For the remainder of this book, examples depict the OCP Clock with rising and falling edges for simplicity.

Fig. 2.2 OCP clock derivation

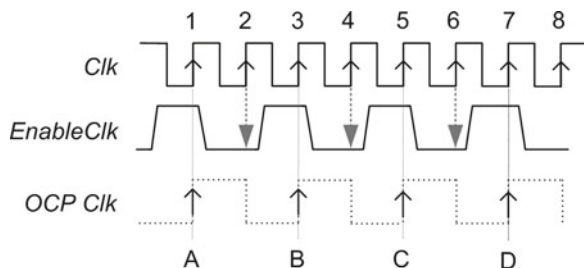
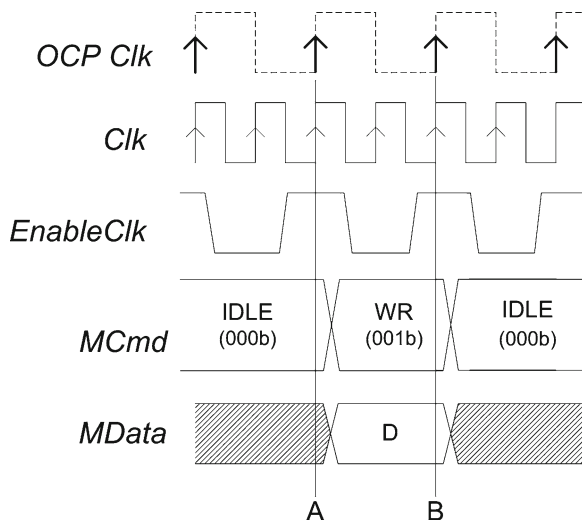


Fig. 2.3 Simplistic OCP posted write with EnableClk active



2.4 Derived OCP Clock Advantages

Deriving the OCP Clock allows OCP to provide flexible multi-rate systems support. By driving appropriate EnableClk waveforms, systems can control the effective clocking rate of OCP interfaces, and frequently, of the associated cores. This can eliminate introducing extra PLL outputs or requiring delay-matching logic spanning multiple clock distribution networks.

When EnableClk is constantly asserted, interfaces behave as if the EnableClk signal is not present. All rising Clk edges are therefore considered rising OCP clock edges, allowing the OCP to operate at the Clk signal frequency. If EnableClk is negated, no rising OCP clock edges can appear to the interface, effectively stopping the OCP clock. This can reduce dynamic power by idling attached cores while leaving the Clk signal active.

Alternately, the EnableClk signal can be periodic. Asserting EnableClk, say, every third Clk cycle causes the OCP interface to operate at one third the Clk's frequency and systems can modify the frequency by changing EnableClk's repeating pattern.

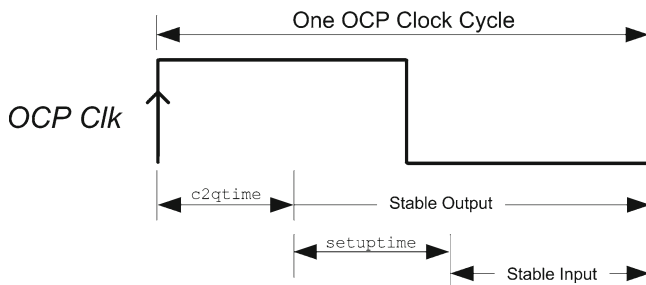


Fig. 2.4 OCP clock cycle signal timing

2.5 Clock Cycle Signal Timing

With OCP Clock derivation now understood, it is also useful to examine two other timing configuration parameters:

- For *output* signals: *c2qtime* – the amount of time required to guarantee an *output* signal is stable after the OCP clock rising edge.
- For *input* signals: *setuptime* – the amount of time an input signal is allowed to change before the OCP clock rising edge.

Figure 2.4 illustrates this relationship and depicts how determine how much time is available to an OCP core to sample stable input signals.

Finally, for a master and slave core to interconnect and function, they must have compatible timing behaviors. OCP defines three timing categories:

1. *Level 0* identifies core interfaces designed without observing any specific timing guidelines.
2. *Level 1* indicates conservative interface timing.
3. *Level 2* represents high performance interface timing.

Any category is not necessarily better than another. The timing categories are an indication of the timing characteristics of the core that allow core designers to communicate at a notional level about the core's interface timing. Table 2.1 describes possible inter-operability of two OCP interfaces.

Timing guidelines apply to dataflow and sideband signals only; there are no timing guidelines for scan and test-related signals.

Now, let's address the remaining questions on the above transfer example.

How much data transfers?

The OCP configuration file allows designers to specify both the presence of any optional MData signal and its width if it is variable. The *mdata* configuration parameter configures the signal into the OCP interface and the *data_width* configuration parameter configures its width. The MData signal width is not restricted to multiples

Table 2.1 Core interface compatibility

	Level 0	Level 1	Level 2
Level 0	X	X	X
Level 1	X	V	V
Level 2	X	V	V*

X – no guarantee
V – guaranteed inter-operability with possible performance loss (extra latency)
V* – high performance inter-operability with minor changes possibly required

of 8. Transfers can transfer data on all MData signal bits or use OCP *byte-enable* methods to effect partial-width transfers.

How does the design ensure the slave actually performed, or was even ready for, the transfer?

The depicted transfer example is conceivably plausible for slaves controlling high performance synchronous SRAMs or register banks. More typically, however, OCP masters require *explicit* command acceptance indicators. In this example, an invisible, defaulted phantom signal provided this indicator.

The signal that provides command acceptance is the optional, one-bit *SCmdAccept* signal that the master only references when the master is using MCmd to assert a command (a non-IDLE value.) When a slave can accept a command, it asserts this signal. Alternately, it de-asserts (negates) the signal, indicating that it cannot accept a command. When this happens, the master must continue asserting its signals until the slave provides command acceptance via the *SCmdAccept* signal.

Designers configure *SCmdAccept* signals into OCP interfaces using the *cmdaccept* configuration parameter. Figure 2.5 illustrates the timing and effect of the *SCmdAccept* signal. Following time point A, the slave detects the MCmd signal transition from IDLE to non-IDLE. In this simplistic example, it immediately indicates it accepts the transfer by asserting *SCmdAccept*.

Why does the transfer mysteriously end at time point B?

The transfer ends in this simplistic example when the master detects the *SCmdAccept* signal is asserted, either explicitly or implicitly (perhaps as a configuration default). This occurs at time point B.

Shouldn't there be some way for the master to receive a response indicating a successful commit following the transfer operation?

Yes, and there is, but not with the posted-write command. By definition, posted write operations do not expect a response. When responses are important, designers should use a different command – the non-posted write that requires an explicit response. The non-posted write has the mnemonic *WRNP* and the MCmd value 101b.

Fig. 2.5 Simplistic OCP posted write with SCmdAccept

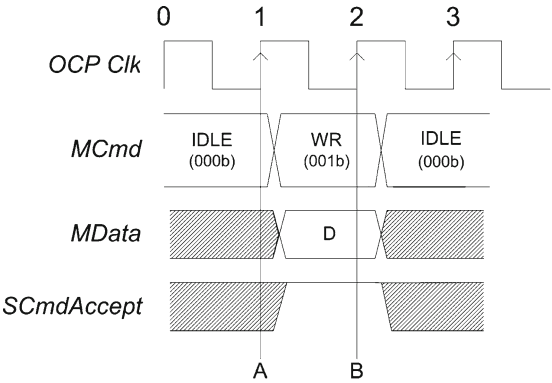


Table 2.2 MCmd command encoding

Command	Mnemonic	Request type	MCmd[2:0]		
Idle	IDLE	(None)	0	0	0
Write	WR	Write	0	0	1
Read	RD	Read	0	1	0
ReadEx	RDEX	Read	0	1	1
ReadLinked	RDL	Read	1	0	0
WriteNonPost	WRNP	Write	1	0	1
WriteConditional	WRC	Write	1	1	0
Broadcast	BCST	Write	1	1	1

2.6 OCP Commands

OCP transfers are all forms of read and write operations the master presents on the three-wire MCmd signal. Each type command has a unique mnemonic and MCmd value. Table 2.2 summarizes the seven possible commands and IDLE.

The next chapter continues examining write operations and their relationships to configurable OCP signals.



<http://www.springer.com/978-1-4614-0102-5>

Introduction to Open Core Protocol

Fastpath to System-on-Chip Design

Schwaderer, W.D.

2012, XVI, 164 p., Hardcover

ISBN: 978-1-4614-0102-5