Implementing an all-digital PHY and delay-locked loop for high-speed DDR2/3 memory interfaces

Luigi Ternullo - October 15, 2009

A high-speed DDR2, DDR2/3, or DDR3 DRAM interface for off-chip memory provides a powerful tool to meet the high-performance demands of new electronic products. However, with advancements come new challenges. The DDR DRAM high-speed interface between the system-on-chip (SoC) and off-chip memory requires specialty circuits. These circuits, often referred to as a physical layer (PHY), comprise high-speed I/Os; a high-resolution, high-precision delay lock loop (DLL); and specialty high-speed logic (PHY logic) to manage the data transfers between the SoC and off-chip DRAM.

Area, power, performance, and time to market are all critical design concerns in a competitive marketplace. A new, all-digital approach to implementing high-speed PHY logic and a DLL offers a path to addressing increasingly stringent market requirements. This article will outline the methodology behind an all-digital PHY+DLL and describe several key implementation techniques used when transitioning to a new standard cell library.

High-speed DDR2/3 memory interface solution

A typical DDR2/3 memory interface solution consists of four key functional blocks: the controller, the PHY, the I/Os, and the DDR memory devices. The controller, PHY, and I/Os reside on the target SoC, and the DDR memory devices reside off-chip, in a module or mounted directly to the printed circuit board (PCB). The controller provides all the decision-making and access scheduling required to convert a user memory access stream into data available for the user. The PHY provides the interface between the memory controller and the I/O pads. It implements all functions and timing required to control command launch, write-data launch, and read-data capture. The I/Os provide the signal driving, shaping, protection, and connection to the off-chip memory components. A simple block diagram of a typical DDR3 memory interface (Figure 1) illustrates how these key function blocks interconnect with each other.

Figure 1: Typical DDR memory interface block diagram

PHY and DLL structure for a high-speed DDR interface
The physical layer of a DDR interface solution on an SoC manages the information transfer between the SoC and the off-chip DDR DRAM. Figure 2 illustrates one representation of a high-speed DDR3 interface solution, which includes the high-speed PHY logic and the DLL. The tan-colored blocks in Figure 2 are all hardened GDS II digital macros; the remaining blocks comprise digital components that are synthesized during chip integration. All of the critical timing and path balancing should be contained within the hard macros.

Overview of PHY+DLL function

The data slices on the right side of Figure 2 show the data transfer between the DDR3 PHY and the off-chip DDR3 memory. The control block on the right side of Figure 2 generates the address and command signals required by the off-chip DDR3 memory. The data slice contains the storage registers and timing elements required to position the data within the acceptable timing window (data eye) needed for robust data transfer. The master DLL, on the left side of Figure 2, determines the timing relationship required for robust operation. Slave DLLs within each data slice pick up this timing data and implement a precise 90-degree phase shift. Read and write leveling operations required for DDR3 are controlled by the read and write leveling finite state machines (FSMs) on the left side of Figure 2. The rest of the interface to the DDR3 controller includes the FIFO and data enable generation logic that simplifies the transmission of data between the controller and the PHY.

DLL operation

A master and a slave DLL work together to implement a precise 90-degree shift in the clock for DDR3 operation. The master DLL precisely calculates the clock period and adjusts this calculation across voltage and temperature. The slave DLL performs the 90-degree clock shift. As can be seen in
Figure 2, three slave DLLs are used in each PHY_DATA macro. One DLL is required to align the DQ on the write across the byte, another is required to align the DQS during a write operation, and a third is required to align the DQS during a read operation. In this implementation, the DQ and DQS alignment is contained within an 8-bit data slice to minimize the amount of uncertainty and jitter introduced into the DQ and DQS signals.

PHY+DLL hard macros

Three unique hard macros are used as building blocks to construct a high-speed DDR3 interface: the PHY_DATA macro, the PHY_CONTROL macro, and the MASTER_DLL macro. We have just described the tasks of the PHY_DATA macro and MASTER_DLL macro. The third hard macro is PHY_CONTROL, which comprises all of the high-speed logic associated with the address and command interface.

During verification of the all-digital PHY and DLL, the PHY_DATA macro and the MASTER_DLL are the ones with the most stringent verification requirements, so we will explore those in more detail to better understand their particular requirements.

PHY_DATA macro

The PHY_DATA macro for a high-speed DDR3 interface comprises all the signals required to support a complete 8-bit data slice. The typical signals required for an 8-bit PHY_DATA macro include 8-DQ signals, positive and negative DQS, and data mask. The PHY_DATA macro controls both the write and read operations.

During the write launch, the memory write data is center-aligned with the data strobe (DQS) signal. The write clock is generated by phase shifting the core clock. The precise phase shift applied to the core clock is controlled by the ratio logic and is determined by the results of the write leveling function for DDR3 memories. Write leveling determines the phase necessary to adjust for the skew between the address and data in the signal path between the controller and the memory devices. The two write slave DLLs are used to set the timing for the DQ and DQS signals.

During the read operation, the PHY_DATA macro captures the read data from memory using the data strobe (DQS) signal as a clock and re-synchronizes the DQS-domain data to the controller's internal clock domain. Read data (DQ) signals are sent edge-aligned with the DQS signal from the memory, so the DQS needs to be phase shifted properly to position it at the center of the DQ data-valid window. This delay is determined during the read leveling operation, which eliminated the skew between the received DQ and the DQS signals. The read slave DLL determines the read DQS signal delay.

MASTER_DLL

The MASTER_DLL is responsible for the delay computation required to precisely position the controlled signal edge. As shown in Figure 3, the MASTER_DLL is composed of five primary building blocks. These blocks include the clock divider, phase detector, delay line, delay control state machine, and output filter.
The clock divider generates a half-speed clock to help with the 90-degree phase shift. The phase detector aligns the reference clock with the phase-shifted clock from the delay line. The delay line uses a series of digital delay elements—the taps—to delay the clock by a selected number of delays. The taps include both course- and fine-grain delay, enabling the high resolution required for DDR3 1600 operation. The delay-control state machine adjusts the tap settings as needed to find the optimal delay setting. The output filter monitors the changes to the tap settings to filter out any unnecessary cycling (a push-out followed right away by a pull-in, or the reverse) of the of the tap setting.

**All-digital PHY+DLL verification methodology**

Though the building blocks in the all-digital PHY+DLL comprise standard digital components in a standard cell library, the high-performance nature of the design and the identified critical timing components require a specialized methodology to develop a high-performance, all-digital PHY and DLL. A flow diagram for the overall process is given in Figure 4.
various critical components in each of the hard macros. The careful selection and evaluation of these components determines the quality of results across all operating conditions. Once cell selection is complete, the functional RTL is generated along with an initial macro layout strategy. These results are passed on to the layout team to create the initial layout exchange format (LEF) file and associated documentation. While the layout team is doing this, the design team works to close timing on the front-end portion of the design. During this step, a number of experience-based scripts are used to close front-end timing and update timing budgets. The layout team then works to close back-end timing and, again using scripts, places hard macros and extracts timing data to verify the timing has been met. Only a few manual design rule checks (DRCs) are left at this point. These are closed in the next phase by the design team, and the final DRCs are all reviewed and closed by the layout team in the final step.

**Specific examples**

The process of implementing the previously described methodology leverages hundreds of lessons learned during the creation of many designs over many processes and applications. In some cases, design teams have developed scripts to capture a lesson in the methodology and carry it forward. In other cases, experience guides the design team's use of scripts that need to be optimized or controlled to achieve the best results. The following examples show how some of these lessons have been added to the all-digital design methodology driving a DLL verification effort.

**Register selection for data and DQS in PHY data**

The key to ensure correct placement of the logic elements in the PHY_DATA macro is constructing an accurate floor plan that will allow the PHY_DATA macro to merge with the respective PHY_DATA I/O macro. In addition to the accuracy of the floor planning, one special logic element must be selected to ensure the interface timing is met during read operations. All other logic cells can be auto selected by the placement and timing tools. Specifically, designers must identify the register in the standard cell library that has the smallest combined setup and hold time. This register is required to capture data and DQS during a read operation. The smaller the minimum data capture window, the more margin the design has for system-level uncertainty in the DQ path. In this instance, a compromise can be made by selecting a register without a scan function so the combined setup and hold time is as small as possible. A good target would be 50 ps or less. When a non-scannable register is selected, test coverage can be regained by employing an at-speed loop-back test.

The placement of the registers and logic associated with other performance-sensitive signals connecting to the I/Os is critical. This can make timing closure a tedious and repetitive process, but the process is repeatable and very precise and is a good candidate for the use of scripts. Timing should be closed with positive slack in a minimum of five process, voltage, and temperature (PVT) corners. It is critical to monitor jitter and delays over these corners, but it is also important to monitor the variance in the modeled parameters.

**Register selection for phase detector**

The phase detector is an instrumental circuit in the MASTER_DLL, as shown in Figure 3. All of the logic cells used to construct the phase detector are automatically selected by industry-standard synthesis and placement tools with the exception of the phase detection register. The register used for the phase detector is the same register that is used for read data capture in the PHY_DATA macro. The minimum achievable worst-case combined setup and hold time for this register is critical to optimize the resolution and minimize the jitter of the clock period calculation. The faster the operating frequency of the reference clock, the more important the worst-case combined flop setup
and hold times become. A general first-order rule is that the variability of the clock period
calculation is only as good as the worst-case combined setup and hold time on the phase detection
flop. For example, if the worst-case combined setup and hold time is 50 ps, then the clock period
calculation accuracy will be about ±50 ps.

**Use of regional placement**

In order to reduce delays and minimize skews, it is possible to use regional placement, a common
digital design technique to constrain certain logic elements and functions to a specific location or a
specific relationship with other functions. For example, the layout tools are directed to place the
read-capture registers near the input pins, reducing the delay and skew between data signals to
improve and simplify timing closure. Using these regioning features allows scripts to automatically
assist with timing closure, to speed layout, and to simplify timing verification.

The design hierarchy can also be used to improve the use of "regioning" as a directive. For example,
the data portion of the PHY can be organized on a bit basis instead of on a byte basis. This allows
each bit slice to be better positioned with its adjacent I/O location and routing channel into the
controller. If regioning was done on a larger basis, the tool would have more difficulty in finding the
best placements and routing strategy to create an optimal result.

Regional placement typically does not require an initial placement and verification run. It is not
iterative. But the next two techniques are typically used after the first timing verification run and
can use these results to drive their implementation.

**Custom placement**

There are times when regional placement is not enough. Over the course of several designs the team
will learn to recognize these situations. In these cases, an experienced layout team can place cells in
specific locations to optimize the design. In this case a script can also be used, but the specific
locations are determined by the layout team and the coordinates can be easily "dialed" into the
script. Experience guides the location, and with just one or two iterations an optimal location can be
found. This is a common practice in high-speed design and is a typical part of an all-digital flow.

An example occurs with the location of the write slave DLLs within the PHY_DATA macro. The write
DLLs need to be located close to the data launch registers within the PHY_DATA block but do not
need to be as tightly coupled to the MASTER_DLL. The layout tool may not understand this
constraint as well as the designer, so custom placement of these DLLs within the PHY_DATA macro
can help to improve the timing results considerably. Central location of the DLL with respect to the
individual data slices is optimal and can be done effectively with custom placement.

**Buffer addition and removal; drive strength adjustments**

Another common high-speed digital design technique for achieving timing closure is to adjust drive
strength by adding or removing buffers or setting buffer drive strengths. This is especially useful on
control signals, like enables on registers or select lines on multiplexers, which can span several
function blocks and can have large fan-out requirements. Once an initial timing run has been
completed and specific signals for timing have been determined, improvement scripts can be used to
adjust fan-out by adding or removing buffers on selected signals. Improvement scripts can improve
delays or adjust skews as needed in the design. For finer adjustments, it is possible to simply raise
or lower the drive strength of existing buffers to bring signals into even tighter alignment. If
multiple signals require adjustments, it is possible to use the same script with just some simple edits
to make several changes. Creating scripts with parameterized capabilities (to set fan-out, buffer
strength, etc.) can take longer initially, but once they are placed in the script library they can be re-used over and over to save development time.