

# Memories and I/O — DDR Signaling

ENEE 245: Digital Circuits and Systems Laboratory, Fall 2014 Lab 8

# Objectives

The objectives of this laboratory are the following:

- To learn how to instantiate the block RAMs offered in FPGAs
- To start learning about high-speed techniques of DDR signaling using source-synchronous clocking of data

In this lab you will extend your simple controller from the previous lab to send out three different commands: Reset, Read, and Write. The important things will be keeping track of where in the sequence of operations your controller is, and reading data on and off the data bus, into and out of an internal data array. This array will be implemented by the built-in memory blocks provided in modern FPGAs: the block RAMs.

### Flash Command Interface

ONFI-compliant NAND flash chips have the following interface and internals:



The ONFI 4.0 NAND flash interface uses these pins in the following way:



This is the latest DDR interface, NV-DDR2/3. The CLK signal at the top is not used in the interface but is shown so you understand that the *controller* might very well use a clock that is much faster than the interface between the controller and the flash device. In fact, this example shows a clock that is four times faster than most of the signals, and twice as fast as the DDR input/output data. This is how you should build your controller simply because it is the easiest way to do it.

In this lab, we will implement a subset of these signals. First, we will redefine the 8-bit DQ bus in flash to use only 2 bits, to reduce the complexity of your implementation and to simplify your testing process when hooking the DLA to your board.

Your design will export the following I/O pins to the world:

DQ bus	2-bit I/O — SDR commands & addresses, DDR data
CE#	Chip enable, active low
CLE	Command-latch enable
ALE	Address-latch enable
WE#	Write enable, active low (used as command/address strobe)
DQS	Read enable, active low

These commands take the following forms. In this lab, you will be emulating their *timing*, but you will be using different *command-code numbers*, because you will use a 2-bit data bus for simplicity. Implement the following, where the 00b. 01b, 10b, and 11b numbers represent the binary command values that you are to put out onto the bus:





**Page Program** — Note that *Page Program* has two command cycles, during which the controller sends the values 0b01 (indicating to prepare for a *Program* command) and 0b10 (indicating *Do Command*). The CE, CLE, ALE, and WE signals need to be operated appropriately; they are simply not shown for the sake of simplicity. There will be 1024 data transfers from the controller to the device: 1K pulses (D0 .. D1023), so a total of 2Kbits.



C1-C2 Column address of the starting buffer location to write data to. C1 is the least significant byte.

R1-R3 Row address of the page being programmed. R1 is the least significant byte.

D0-Dn Data bytes/words to be written to the addressed page.

**Page Read** — Note that *Page Read* has two command cycles, during which the controller sends the values 0b00 (indicating to prepare for a *Read* command) and 0b10 (indicating *Do Command*). The CE, CLE, ALE, and WE signals need to be operated appropriately; they are simply not shown for the sake of simplicity. There will be 1024 data transfers from the device to the controller: 1K pulses (D0 .. D1023), so a total of 2Kbits.



On the FPGA board there are four buttons and 8 switches. You will use these as follows:

- You will use the 8 switches to identify address values: your controller will read the address from the switches, interpreting switches 0–3 to indicate the column address (the "C1" and "C2" bytes in the various commands), and switches 4–7 to indicate the row address (the "R1," "R2," and "R3" bytes in the various commands). The smaller numbers represent less significant bits. For the top two most significant bits of the *Row Address*, i.e. R3, just use zeroes (0b00).
- You will use the buttons to identify commands: your controller will respond to the press of a button by reading the address from the switches and issue the appropriate command signals.
  - → Button 1: 0b11 *Reset* command (command only, no address)
  - → Button 2: 0b01 *Page Program* command (use switches 0–3 to indicate the column address and switches 4–7 to indicate the row address)
  - → Button 3: 0b00 *Page Read* command (use switches 0–3 to indicate the column address and switches 4–7 to indicate the row address)

You will use a 2-bit data bus (as opposed to 8-bit), which means that you will use the 2-bit command codes provided above instead of the real ones in ONFI flash devices, and you will break the row and column addresses into 2-bit chunks and send those chunks over multiple cycles. Otherwise, it will resemble fairly closely a real flash interface. You need to implement the following six signals:

DQ bus	2-bit bidirectional I/O, including commands, addresses, and data Commands & addresses are single data rate (SDR), timed by the WE# strobe; the data is double data rate (DDR), timed by the DQS (DQ Strobe).
DQS	DQ Strobe, bidirectional
CE#	Chip Enable, active low
CLE	Command-Latch Enable
ALE	Address-Latch Enable
WE#	Write Enable, active low (is effectively the command/address timing strobe)

The I/O and DQS signals are bidirectional ("inout" in Verilog). The rest of the signals are all output only; the controller drives them, and the device receives them.

You should map these six signals (seven wires) as follows in your User Constraints File (the first seven lines in the FX2 connector set, which implements general I/O):

```
      NET "ceb"
      LOC = "B4"; # Bank = 0, Pin name = IO_L24N_0, Type = I/0, Sch name = R-IO1

      NET "cle"
      LOC = "A4"; # Bank = 0, Pin name = IO_L24P_0, Type = I/0, Sch name = R-IO2

      NET "ale"
      LOC = "C3"; # Bank = 0, Pin name = IO_L25P_0, Type = I/0, Sch name = R-IO3

      NET "web"
      LOC = "C4"; # Bank = 0, Pin name = IO_L20P_0, Type = I/0, Sch name = R-IO4

      NET "dq[0]"
      LOC = "B6"; # Bank = 0, Pin name = IO_L20P_0, Type = I/0, Sch name = R-IO5

      NET "dq[1]"
      LOC = "D5"; # Bank = 0, Pin name = IO_L23N_0/VREF_0, Type = VREF, Sch name = R-IO6

      #NET "dqs"
      LOC = "C5"; # Bank = 0, Pin name = IO_L23P_0, Type = I/0, Sch name = R-IO7
```

#### Block RAM in the FPGA

Your FPGA has quite a few libraries that provide already-created circuits for you to use in your designs. The library documentation is on the course website. For this lab, you will use the Block RAM facility, which is extremely powerful. Because we only want to get the main idea and not necessarily talk to a real flash device, we will just use a narrow bus and a small burst length. Nonetheless, the general facility that you use and get familiar with as part of this lab is extremely useful and will become an important tool in your future designs.

We will instantiate the RAMB16\_S2 block, which is a 16Kbit RAM that is 2 bits wide, running off the positive edge of the clock. In general, your FPGA has numerous block RAMs, each up to 2KB in size, so this really is just a taste of what the FPGA can do. It is instantiated this way:



If you put this into your code, giving it a unique instance name (i.e., replace "instance\_name" with whatever you want to call it), then the synthesis tools will grab one of these from the Xilinx library and instantiate it onto your FPGA. The format is that the top parenthetical (preceded by a # sign) is

optional and, if present, represents the initialization values; the bottom parenthetical is not optional and represents the wire connections to the module.

**Remember:** The data I/O between controller and memory device will be in 1024-pule increments, i.e., 1024 x 2 bits in total. Each read/write operation will be this large, no smaller. So your data must be stored and read out from the large Block RAM, which has enough space to hold all of that data. So you will be using a 13-bit address input to the Block RAM, in which the top 3 bits will be 0s: an effective 10-bit address bus, specifying 1024 different data locations.

#### Should You Desire a Challenge

If you design your chip to use a clock that cycles twice as fast as the DDR input/output, and therefore four times as fast as the commands and addresses, then it should be relatively straightforward to make sure all of the signals are synchronized. However, your design will be fairly complex because each atomic operation (such as sending a single command or address word on the 2-bit bus) will span multiple cycles of the clock. This will require a state machine to keep track of where you are in the sequence.

You could make your state machine simpler by using a 2x clock instead of a 4x clock, and using special DDR input/output buffers for the I/O. In other words, you could move data around the interior of the controller in 4-bit chunks, using 4-bit data busses and the RAMB16\_S4 block, which has 4-bit data ports, instead of the RAMB16\_S2 block, which has 2-bit data ports. And you could use the Xilinx library's IFDDRRSE and OFDDRRSE registers that can automatically handle the DDR input/output for you. They are instantiated as follows:



IFDDRRSE instance\_name (

.Q0(Q0) .Q1(Q1) .C0(C0) .C1(C1) .CE(CE) .D(D) .D(D)	<pre>// Posedge data output // Negedge data output // 0 degree clock input // 180 degree clock input // Clock enable input // Data input (connect directly to top-level port) // Complements input</pre>
.D(D)	// Data input (connect directly to top-level port)
.R(R)	// Synchronous reset input
.S(S)	// Synchronous preset input

);



# **Pre-Lab Preparation**

Design your controller. Your goal is to create timing diagrams like the ones above. Use the Xilinx design facility to generate waveforms for each command, and bring these as your pre-lab write-up.

On the course website will be found a module for you to instantiate in your test bench and use as a memory device for testing, so that it can respond to your read request and return data with correct timing.

# **In-Lab Procedure**

Bring flash drives to store your data.

Ask the TA questions regarding any procedures about which you are uncertain.

Complete the following tasks:

- Program your system onto the FPGA board.
- Connect the FPGA to the break-out board.
- Run the system and use the DLA to observe the 6 bits of the controller output. Save the DLA's output for your post-lab report.
- Look at the detailed RTL schematics produced by the software; save it for your post-lab report.
- Look at the timing report that gives the pin-to-pin delays for input/output combinations of every pin. Save these reports and tables for your post-lab report.

# **Post-Lab Report**

Write up your code, schematics, and lab procedures. Demonstrate the correctness of your designs through your DLA output and note any differences between what you simulated and how the circuits behaved in the lab.

Regarding the RTL schematics produced by the software—how did the design software synthesize your code? Where did it choose poorly, and how could it have done better? Could you have better specified your design to get more efficient results?

Regarding the timing report that gives the pin-to-pin delays for input/output combinations of every pin—what does the design software say for the timing? How fast is each component? How fast could you, in theory, run your design?