Memories and I/O — Bidirectional Data

ENEE 245: Digital Circuits and Systems Laboratory
Lab 11

Objectives

The objectives of this laboratory are the following:

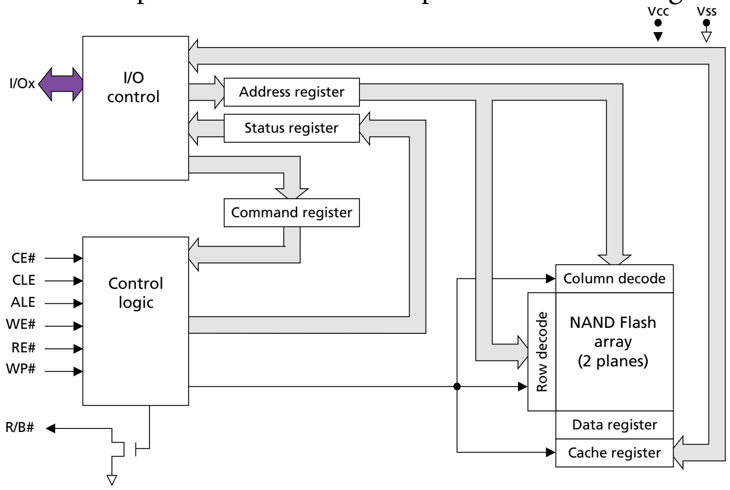
- To learn how to read and write external data on a bi-directional bus
- To learn how to use a source-synchronous data strobe

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.

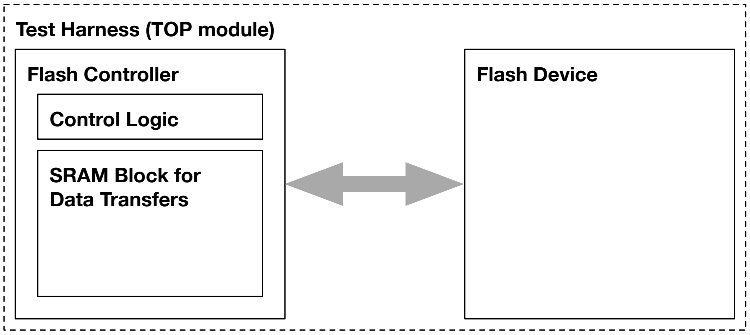
For simplicity and completeness, we have repeated the material from the last lab, which this extends.

Flash Command Interface

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



And the following figure shows how the NAND flash device fits into a system. The figure shows a flash controller, which contains an SRAM used for transferring data in and out of the memory, some



control logic, and the interface (the bus lines between the controller and the flash device). Over the course of the next two labs, you will build the controller; the flash device is given to you as **MEM_block.v** on the course website.

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

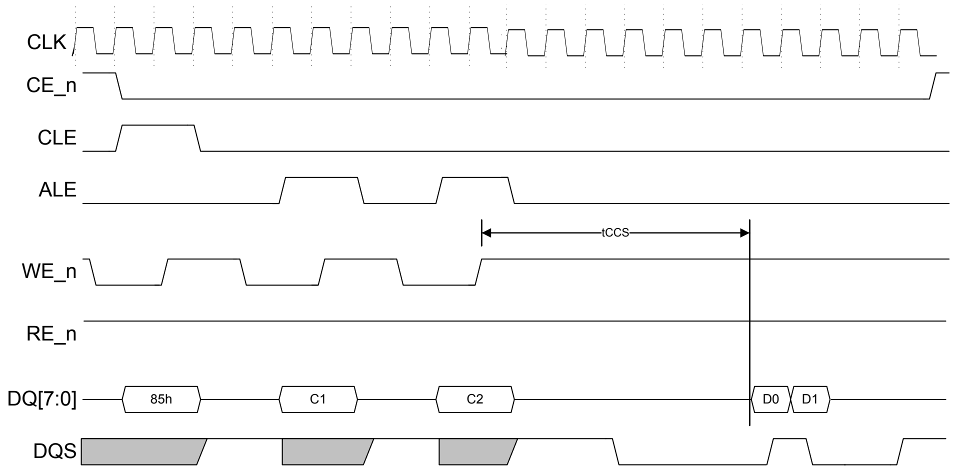


Figure 76 NV-DDR2 and NV-DDR3 data interface command description

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, and we will start with single data rate (SDR) instead of double data rate (DDR). 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.

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” and “R2” bytes in the various commands). The smaller numbers represent less significant bits.
- 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)
  - ➡ Button 4: **Initialize the device**: set all internal registers to the correct starting values and drive the bi-directional (inout) DQ bus with high-Z.

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. Your data will be SDR, not DDR. Other than these points, your implementation will resemble fairly closely a real flash interface. You need to implement the following six signals:

DQ bus 2-bit unidirectional I/O, including commands, addresses, and data Commands, addresses, and data are all single data rate (SDR); all but the data pulses are timed by the WE# strobe; the data pulses are timed by the DQS (DQ Strobe).

DQS DQ Strobe, bidirectional

CE# Chip Enable, active low

CLE Command-Latch Enable

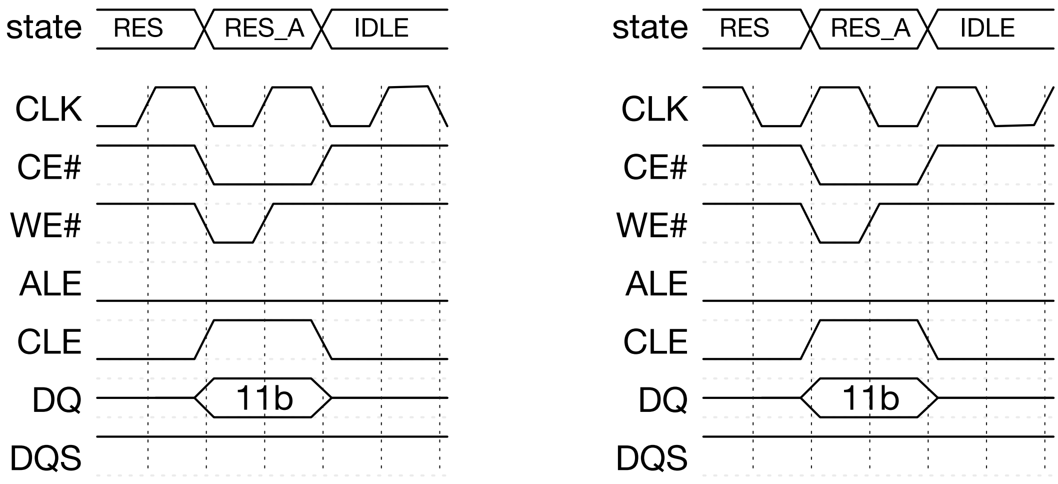
ALE Address-Latch Enable

WE# Write Enable, active low (is effectively the command/address timing strobe)

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. The internal CLK signal and example **state** values are shown just for your convenience; you need not implement your controller this way—only the timing of the external signals matters. For each command, there are two variations given, in which the only difference is whether the internal state machine is run off the rising or falling edge of the clock.

Reset Command

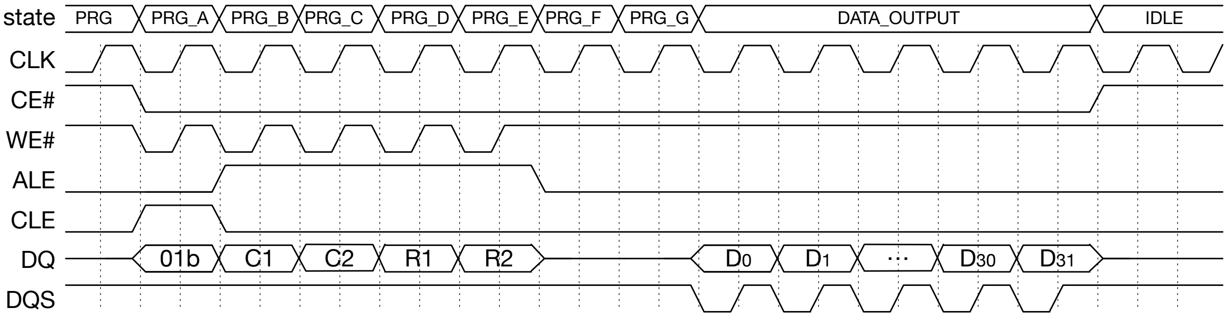
The timing for the *Reset* command is given below, in two variations, depending on whether you want to use the rising or falling edge of the clock to drive your state machine.

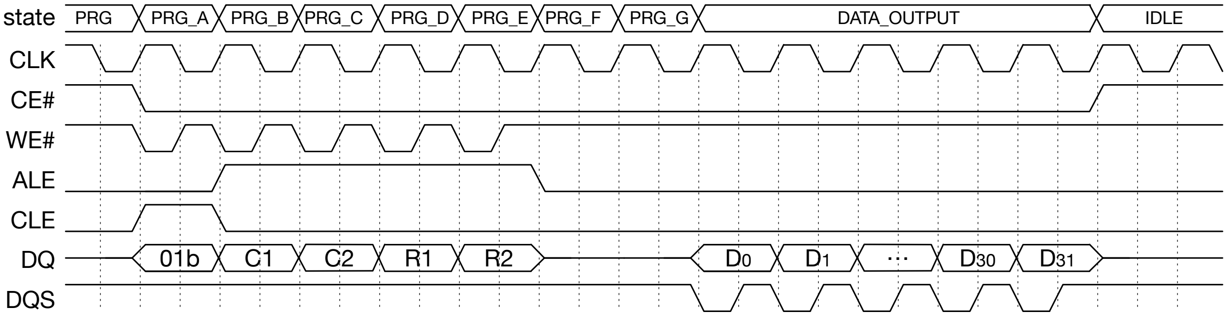


Page Program Command

The timing for the *Page Program* command is given below, in two variations, depending on whether you want to use the rising or falling edge of the clock to drive your state machine.

Note that *Page Program* has a lengthy command sequence, as you must send not only the command (01b) but also four address pulses on the bus, giving the eight total bits, two at a time, that were read from the 8-bit switch bus. The CE, CLE, ALE, and WE signals need to

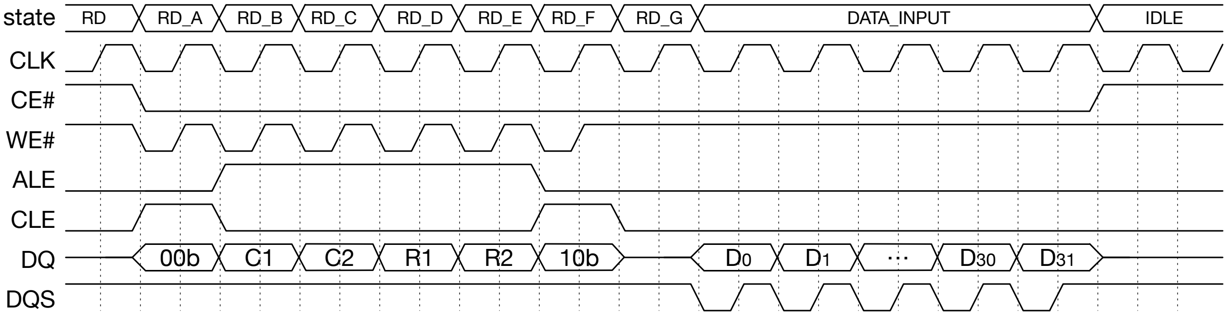


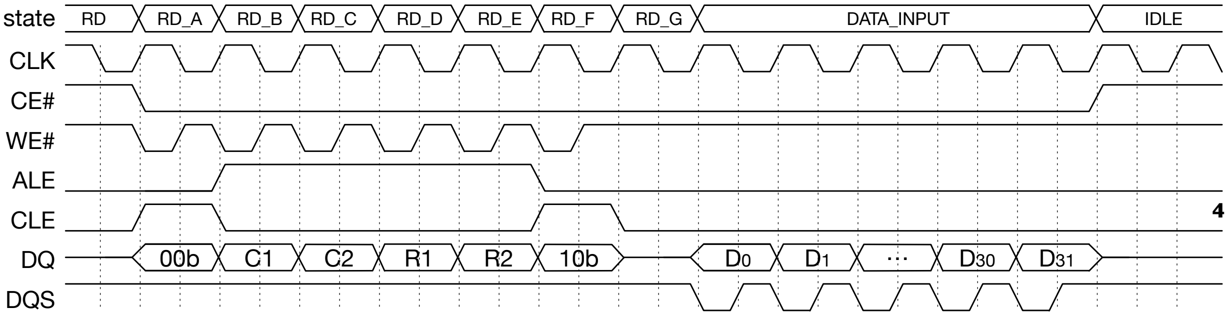


be operated appropriately and are given. This command adds the component of data, which was not part of the *Reset* command. There will be 32 data transfers from the controller to the memory device: 32 2-bit pulses (D0 .. D31), so a total of 64 bits. It is recommended that you have a separate 5-bit counter that gets initialized to 0x1F when you enter the DATA_OUTPUT state, and you simply count down to zero; when done, stop transmitting data, drive the DQ bus and DQS wire with high-Z, and go into the IDLE state. The data is clocked with a different clocking signal than the command and address pulses. The DQS signal (*DQ Strobe*) is sent along with the data. For this lab, we will not do DDR data transfers but will instead do simple single-data-rate transmission of data.

Page Read Command

The timing for the *Page Read* command is given below, in two variations, depending on whether you want to use the rising or falling edge of the clock to drive your state machine.





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 *Start Sending Data*). As with the *Page Program* command, there will be 32 data transfers from the memory device to the controller: 32 two-bit data pulses (D0 .. D31), so a total of 64 bits. It is recommended that you have a separate 5-bit counter that gets initialized when you enter the DATA_INPUT state, and you simply count down to zero; when done, stop receiving data, drive the DQ bus and DQS wire with high-Z, and go into the IDLE state. The data is clocked with a different clocking signal than the command and address pulses and on a *Page Read* command, the controller receives the clock, as opposed to sending it. The DQS signal (*DQ Strobe*) is received along with the data.

The two-bit I/O bus (the “DQ” bus) and the DQS timing signal are both bidirectional (they are to be defined as “inout” in your Verilog). This means that the device responsible for driving the signal changes depending on whether it is a read or write operation (the controller drives the data and timing signal onto the bus for write operations; the memory device drives the data and timing signal onto the bus for read operations). **Note:** in Verilog, when you are not actively driving an “inout” signal you must explicitly assign it high impedance (‘Z’). The rest of the signals are all output only; the controller drives them, and the memory 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/O, Sch name = R-IO1

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

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

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

NET “dq[0]"   LOC = “B6"; # Bank = 0, Pin name = IO_L20P_0, Type = I/O, 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/O, 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 Static Synchronous 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 RAM32X2S block, which is a 32-deep SRAM that is 2 bits wide, running off the positive edge of the clock. In general, your FPGA has numerous block RAMs, up to 2KB in size, so this really is just a taste of what the FPGA can do. It is instantiated this way:

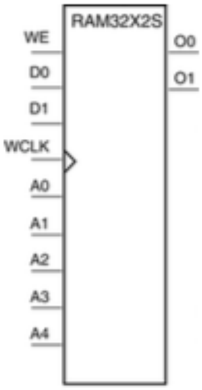
```
RAM32X2S #(

      .INIT_00(32’hCAFEF00D), // INIT for bit 0 of RAM
      .INIT_01(32’h005EABED)  // INIT for bit 1 of RAM

) RAM32X2S_inst (

      .O0(O0),                // RAM data[0] output
      .O1(O1),                // RAM data[1] output
      .A0(A0),                // RAM address[0] input
```

```
         .A1(A1),                // RAM address[1] input
         .A2(A2),                // RAM address[2] input
         .A3(A3),                // RAM address[3] input
         .A4(A4),                // RAM address[4] input
         .D0(D0),                // RAM data[0] input
         .D1(D1),                // RAM data[1] input
         .WCLK(WCLK),            // Write clock input
         .WE(WE)                 // Write enable input
    );
```



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. Use the given initialization values, so that checking your output will be easier.

Note that the instantiated test memory device will check your values to see that you write the same values written.

Remember: The data I/O between controller and memory device will be in 32-pulse increments, i.e., 32 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 Static RAM, which has enough space to hold the data. So you will be using a 5-bit address input to the Block RAM, specifying 32 different data locations.

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.
- **Test your system by running a *Page Read* command followed by a *Page Program* command, so that the memory device can validate whether your controller correctly received the data.**
- 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?