Journal 2, Creation of FPGA designs

1 Objectives

The main objective is to create an FPGA design that supports the PCI Express interface, and features a usable, generic interface for connecting this to other hardware modules inside the FPGA. The design should also be created with the requirements regarding transfer speed and storage possibilities specified in journal 1 in mind.
Additionally, it could be interesting to demonstrate some of the possibilities of the Spartan3 PCI Express Starter kit, such as usage of the onboard DDR RAM memory and the VGA output port.

2 Problem analysis

The Spartan3 PCI Express Starter kit includes the IOControl sample demo design[1]. This is based on the LogiCore Endpoint PIPE for PCI Express IPcore, and includes an implementation of a programmable input/output system (PIO), and some functionality for accessing the leds and buttons on the board through BAR0. It is written in Verilog. The demo is made for Xilinx 8.1 though, and uses version 1.3.2 of the PIPE IPcore. Therefore it is not directly usable with Xilinx 9.2 which comes with version 1.7 of the PIPE IPcore. It should be possible to reuse the functionality for accessing the leds and buttons though.
The obtainable transfer speeds depend heavily on the host system, so the evaluation of these will need to wait until some drivers have been created.
Another requirement is the possibility to store 100 MB of data on the board. The easiest way to store data in a Spartan3 FPGA is usually to use the integrated blockram. However, as the used FPGA only has 54 kB available, the onboard 128 MB of DDR SDRAM could be a possibility instead.
So the following needs to be done:

- A new design and structure based on version 1.7 of the PIPE IPcore needs to be created.
- A BAR interface has to be specified and implemented.
- A few demo designs, exploring the possibilities of the board, including the DDR SDRAM, need to be created.

3 The PIPE core

Xilinx provides the Endpoint PIPE for PCI Express IPcore, which implements all three layers of the PCI Express protocol (physical, datalink and transaction). The core exposes a PXPIPE interface to an external physical layer chip mounted on the Spartan3 board, and transaction (trn) and configuration (cfg) interfaces to modules in the FPGA. The trn interface is used for communicating with the host system over the PCI Express interface, and accepts and delivers TLPs. The cfg interface provides access to the configuration space of the PIPE core.
The core has been tested and verified by Xilinx to comply with the PCI Express v1.1 standard, and thus developers can concentrate on designing the actual functionality of the device, without worrying too much about the PCI Express interface.
To use the core, a license is needed. For this project, a “Full System Hardware Evaluation” license is used, which will work for 8 hours after loading the design into the FPGA[2].

3.1 Generating the PIPE core

The described PIPE core can be generated directly from Xilinx ISE, by adding a new IP source and selecting the *Endpoint PIPE for PCI Express 1.7* core (under *Standard Bus Interfaces -> PCI Express*). *CoreGen* will then popup a menu through which the PIPE core can be configured. The

[1] This can be found in the ”*Tools\Spartan3 PCI Express Starter kit Demo*” folder on CD1.
[2] More info: http://www.xilinx.com/ipcenter/ipevaluation/pcie_pipe_evaluation.htm

most important parameters here are the vendor and device IDs (page 2), as these are used by the host system to match the device to the correct device driver, and the BAR setup (page 3 and 4). The following is used throughout the project for IDs and class codes:

- Vendor ID / Subsystem Vendor ID: 1597
- Device ID / Subsystem ID: 0301
- Revision ID: 00
- Base class: 0B (Processors)
- Sub-Class: 40 (Co-processors)
- Interface: 00
- Cardbus CIS Pointer: 00000000

The class codes[1] can be set to more appropriate values if desired, but the ones listed work just fine. The configuration of the BAR registers varies with the application. Other more advanced parameters, like power saving, can be configured too, but here they are just kept at their default values.

4 The overall structure

When generating the PIPE core with *CoreGen*, a sample design, including a programmed I/O (PIO) module, is automatically generated also. The PIO_EP module hooks up to the trn interface, and exposes a simple memory address-/data-bus (with a few extra control signals) to be used by the BAR modules. The structure of this sample design can be seen in figure 4.a.

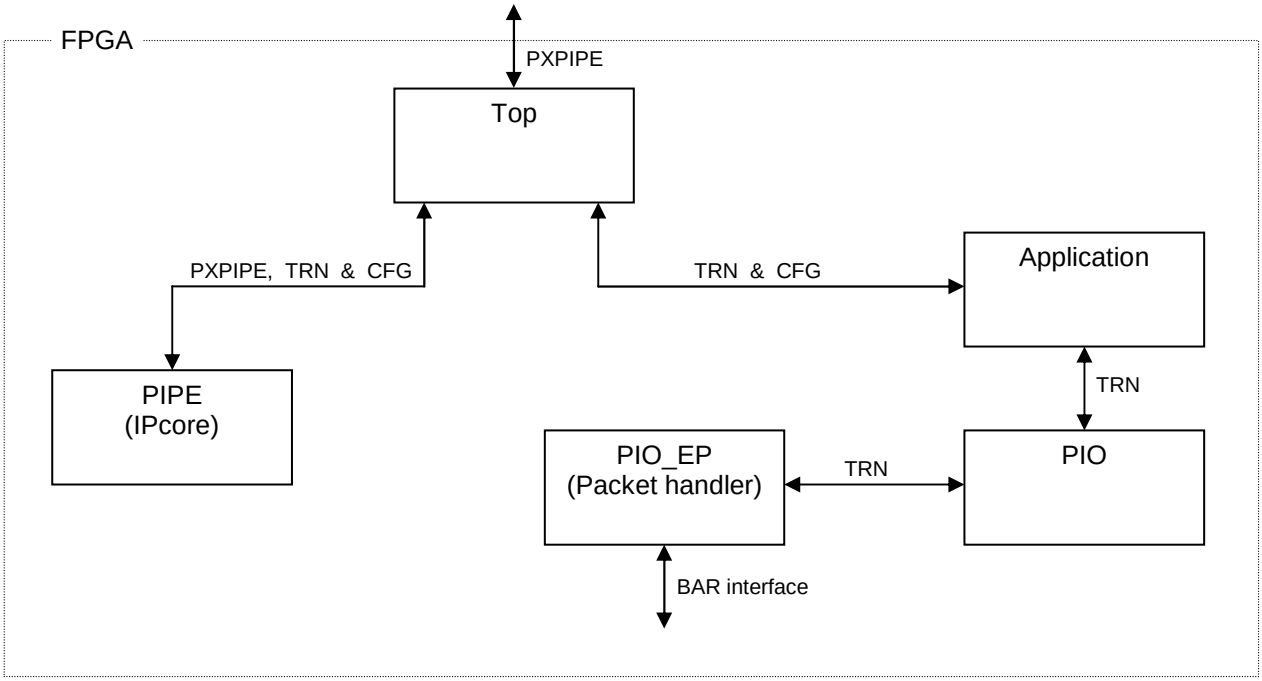

Figure 4.a: The sample design structure (simplified).

The modules are:

- *Top:* The *Top* module instantiates the PIPE core and application module, and provides buffering for both the physical in- and outputs and the signals for the transaction interface exposed by the PIPE core.
- *PIPE:* The PIPE IPcore handles the lower layers of the PCI Express protocol. It connects (through the *Top* module) to a separate PHY chip through an 8-bit 250 MHz PXPIPE interface

[1] A list of class codes can be found in appendix D of *PCI Express System Architecture* (for PCI v.2.3) (see the literature list for more info), or at: http://www.acm.uiuc.edu/sigops/roll_your_own/7.c.1.html (for PCI v.2.0)

on one side, and exposes a configuration and a transaction interface (also through the *Top* module) on the other side. It also provides the 62.5 MHz transmission clock signal used to clock the rest of the structure.
- *Application:* The *Application* module instantiates the *PIO* module.
- *PIO:* This module instantiates the *PIO_EP* module, and a module for handling PIO turn-off requests from the host systems (not shown on figures).
- *PIO_EP:* This module instantiates receive and transmit modules which connect to the transaction interface of the PIPE core. These handle decoding of received packets, and encoding of packets to be returned. A simple read/write memory interface (the BAR interface) is exposed.

The structure is not optimal due to two reasons – the BAR modules do not have access to the cfg interface, and if they need to be connected to physical I/O pins on the FPGA, four modules will need to be modified (*Top*, *Application*, *PIO* and *PIO_EP*). Instead, the BAR hardware interface is run through the *PIO* module to the *Application* module. The BAR hardware will then need to be instantiated from here, as can be seen in figure 4.b.

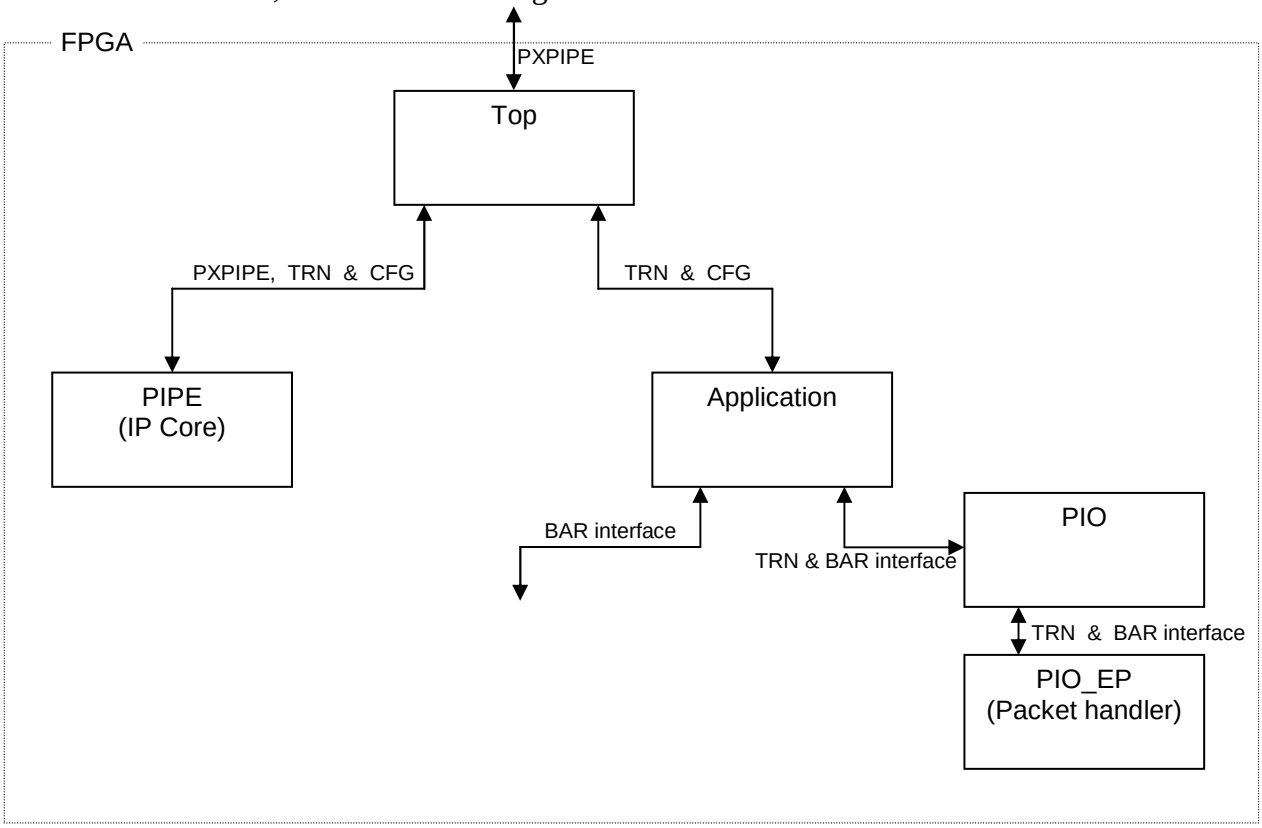

Figure 4.b: The used structure (simplified).

It should be noted that the PIO functionality used does not support some of the more advanced features that are possible with PCI Express. It only supports single DWORD memory read and write transactions to a 32 bit memory space. Also, delayed responses are not supported, so only one transaction can be running at a time.
As pointed out, all modules are generated using the Verilog language. This is not too practical though, as VHDL is currently the only used HDL language at SDU. For this reason, the *Application* module, which is the module expected to be modified the most, is rewritten using VHDL. The only other module that would need modifications (except if for instance more functionality is needed from the *PIO* module) would be the *Top* module, when external I/O pins are needed. As Verilog and VHDL are rather similar, this is not seen as a problem though, as it should be easy to add the few needed lines of code, without having to do a larger study on the syntax of Verilog.

5 The BAR hardware interface

The existing BAR interface in the example design consists of a 10 bit address bus, a 32 bit data bus, byte enable signals, and write-enable and -busy signals. Before deciding on the specifications of the interface to implement, some information was gathered from the project groups from journal 1. It was determined that most projects at SDU do not use standardized interfaces like Wishbone[1], in spite of these making reuse of existing modules possible (or in any case, easier). Instead, most projects simply use whatever interface is at hand, or the easiest to implement. It was therefore decided to make the BAR interface as simple as possible, while still retaining enough functionality to be usable with a large number of different BAR hardware modules.
The created BAR interface is based on the interface exposed by the sample *PIO* module. A few control-signals have been added, and some signals multiplexed to allow the interface to work with up to six BAR modules in one design. It is a master-slave setup, with the BAR modules as slaves, and the *PIO* module as master. It consists of the following signals:
(Inputs are signals *to* the BAR modules, outputs are signals *from* the BAR modules. *Shared* implies that all BAR modules are connected to the same signal, *multiplexed* implies that only the currently selected BAR module is connected to the signal.)

- *trn_clk:* Clock signal, *input*, *shared*. This is the 62.5 MHz clock signal from the PIPE core, which the interface is clocked in relation to.
- *trn_reset_n:* Reset signal, *input*, *shared*. This is the active-low reset signal of the transaction interface, and is asserted if the DCM generating *trn_clk* looses its input clock.
- *rd_addr:* Read address bus (30 bit), *input*, *shared*. This provides the address for read operations, and addresses DWORDs. It is therefore automatically DWORD aligned.
- *rd_be*: Read byte enable (4 bit), *input*, *shared*. This is a byte enable used when transmitting data of 3 bytes or less, since *rd_data* will always be 32 bit. A logic '1' signals that the corresponding byte in read data is enabled.
- *rd_data:* Read data (32 bit), *output*, *multiplexed*. This is the data read from the BAR module.
- *rd_en:* Read enable (1 bit), *input*, *multiplexed*. This active high bit is used to signal a read operation to the targeted BAR module.
- *rd_busy:* Read busy (1 bit), *output*, *multiplexed*. This active high bit is used by the BAR modules to signal that a read is in progress.
- *wr_addr:* Write address bus (30 bit), *input*, *shared*. This provides the address for write operations, and addresses DWORDs. It is therefore automatically DWORD aligned.
- *wr_be:* Write byte enable (8 bit), *input*, *shared*. This is a byte enable used when transmitting data of 3 bytes or less, or when transmitting data that is not fully DWORD aligned. A logic '1' signals that the corresponding byte in write data is enabled.
- *wr_data:* Write data (32 bit), *input*, *shared*. This is the data to write to the BAR module.
- *wr_en:* Write enable (1 bit), *input*, *multiplexed*. This active high bit is used to signal a write operation to the targeted BAR module.
- *wr_busy:* Write busy (1 bit), *output*, *multiplexed*. This active high bit is used by the BAR modules to signal that a write is in progress.
- *bar_select:* BAR select register (7 bits). This register contains the currently active BAR (one-hot encoded, active high), and can thus be used as a select/enable signal for the BAR modules if necessary.

[1] Wishbone is an open-source hardware bus. The specification can be found in "*Datasheets\Wishbone_b3.pdf*" on CD1.

5.1 Read operation

An example read operation that reads from a negative edge clocked blockram can be seen in figure 5.1.a.

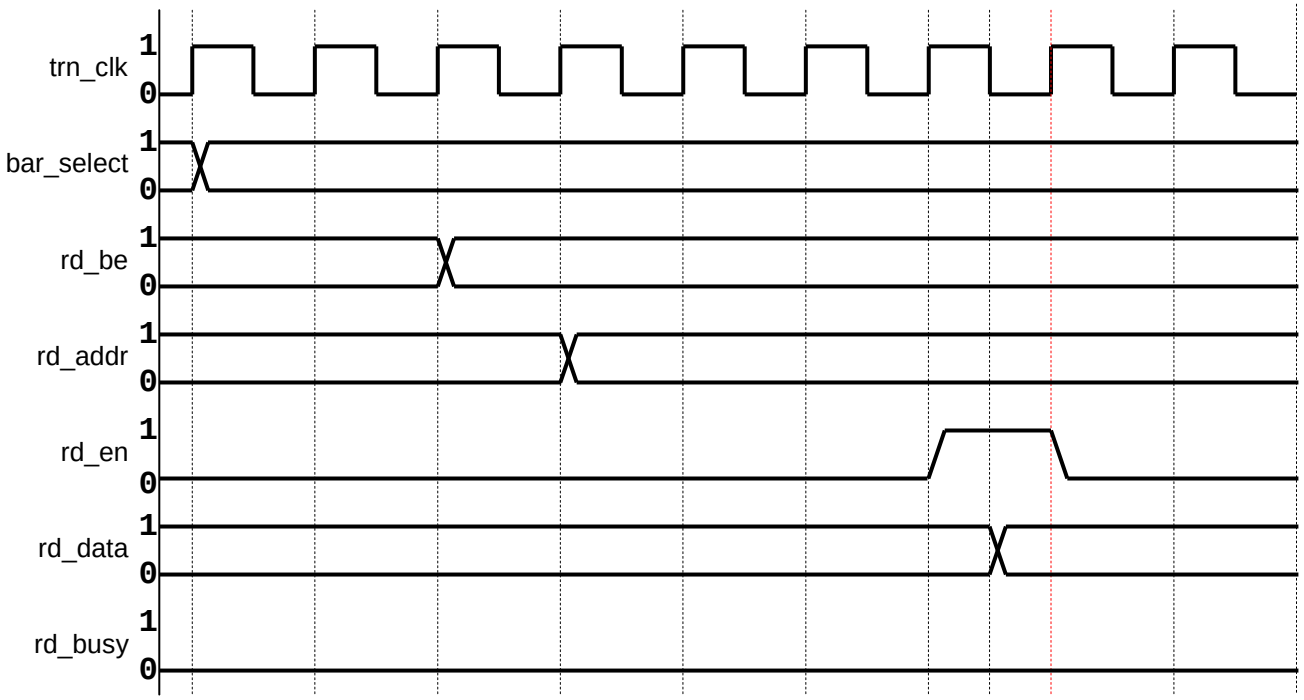

Figure 5.1.a: A read operation.

Data from *rd_data* is read into the *PIO* module on the first rising edge of the clock after *rd_en* goes high where *rd_busy* is not set (the red line). If the BAR module is not able to drive *rd_data* with valid data before the next rising edge of the clock after *rd_en* goes high, it will need to drive *rd_busy* high until valid data is present on *rd_data*. This can be seen in figure 5.1.b.

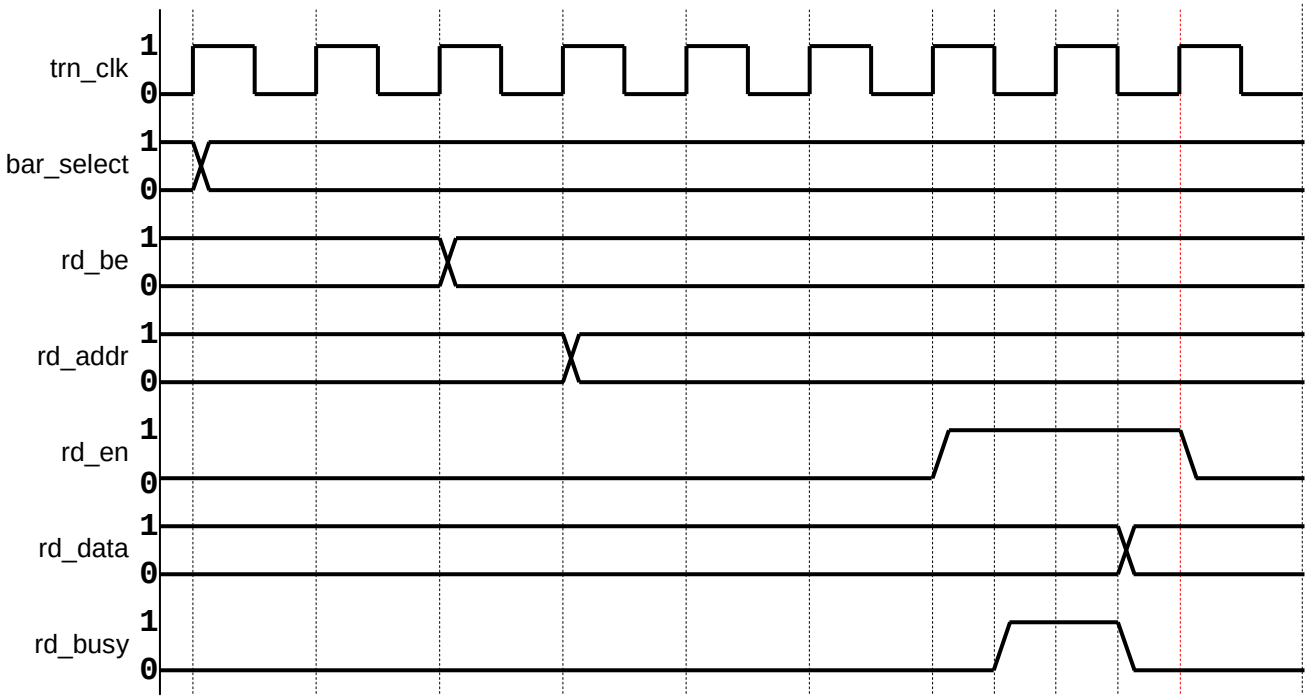

Figure 5.1.b: A read operation using *rd_busy*.

Here the BAR module is not able to produce valid data within one clock cycle, and therefore needs to set *rd_busy* high until valid data is present on *rd_data*.

5.2 Write operation

An example write operation that writes to a negative edge clocked blockram can be seen in figure 5.2.a.

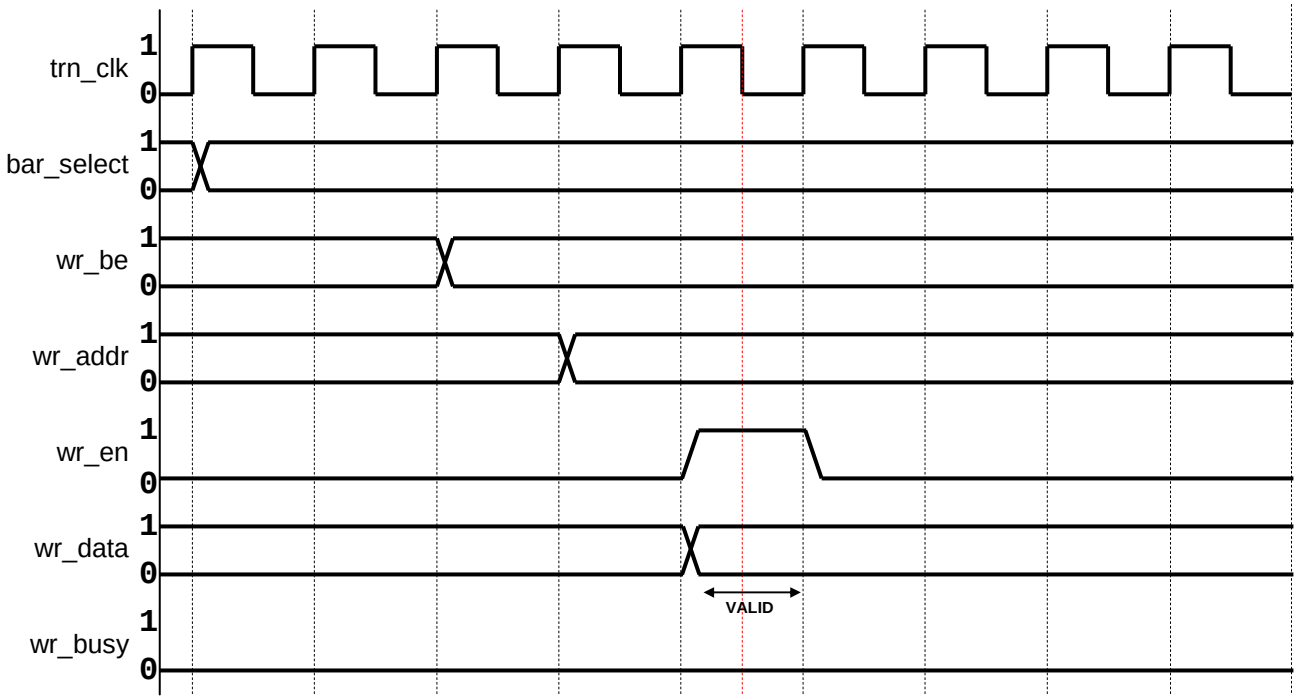

Figure 5.2.a: A write operation.

The red line indicates the point at which data from *wr_data* is read into the blockram. The data on *wr_data* is only guaranteed to be valid as long as *wr_en* is high. *wr_data* remains high until the next rising edge of the clock where *wr_busy* is not set. Therefore, if the BAR module is not able to read in the data within one clock cycle, it will need to set *wr_busy* high until the write has been completed, to signal to the *PIO* module that it needs *wr_data* to be kept valid for a longer period. An example of this can be seen in figure 5.2.b.

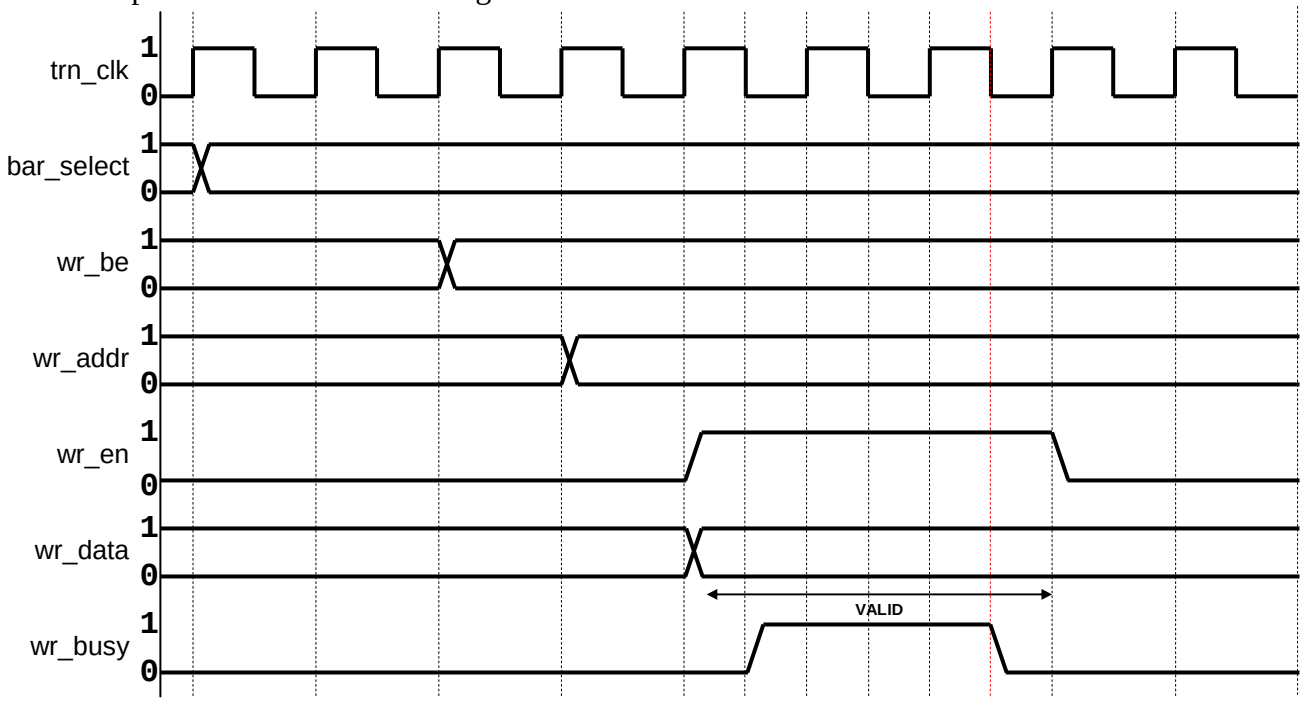

Figure 5.2.b: A write operation using *wr_busy*.

6 The designs

Four FPGA designs have been created for the project. These can be found as Xilinx 9.2 ISE projects in the "*Source\FPGA_Design*" folder on CD1. The designs are further described in their respective applicationguides. The projects are:

- *Empty:* The *Empty* project contains the described structure (including the PIPE and *PIO* modules), but no BAR modules. It is meant to be used as a starting point for new designs using the Spartan3 PCI Express Starter kit.
- *IOControlDemo:* This design builds on the *Empty* design. It includes the BAR module from the original demo design, which can access the buttons and leds on the Spartan3 PCI Express Starter kit board as BAR0, an 8 kB blockram as BAR1 and a PIPE core setup to match this.
- *VGADemo:* A design built on the *Empty* design. This uses a 24 kB blockram at BAR0 as a framebuffer. An additional module reads the framebuffer and outputs it on the VGA port.
- *DDRControllerTest:* This design does not build on the *Empty* design. It consists of a DDR memory controller connected to the DDR memory on the Spartan3 PCI Express Starter kit. A testbench is included to verify that it is operational.

7 Conclusion

An FPGA design structure with a working PCI Express interface and an interface for hardware modules has now been created. Up to six independent hardware modules are supported at a time, through a multiplexed bus. The Xilinx Endpoint PIPE for PCI Express IPcore is used, which implements a complete PCI Express protocol.
Using this structure, three designs have been created – a base design, *Empty*, without hardware modules, to be used as a starting point for new designs, the *IOControlDemo* with modules implemented to control the onboard leds and buttons and an 8 kB blockram, and the *VGADemo* design demonstrating the use of the VGA port. The possibilities of using the onboard DDR RAM memory for storage have also been investigated, and a test design, *DDRControllerTest*, has been created. This has been brought up and running, and can successfully access the onboard DDR memory. It is not implemented together with the PCI Express interface though. The designs are further described in the applicationguides.
The performance will be evaluated when a driver has been created.

.