A Custom Serial Peripheral Interface (SPI) Bus Slave Controller with Read/Write Register Banks

Vladimir Milovanović, *Senior Member, IEEE,* and Darko Tasovac, *Member, IEEE*

***Abstract*—Interchip communications often exploit the so-called serial peripheral interface (SPI) bus for lower-level and medium-speed exchange of data. Typical use cases include application-specific integrated circuit (ASIC) setup and testing via scan chain. A lack of formal standardization and ubiquitous presence inevitably lead to abundance of SPI protocol definitions. This paper presents a custom SPI bus slave controller which includes single and block addressable bank of read-only status, as well as read/write configuration registers. The controller employs a rather efficient in-frame addressing scheme which is supported by low-propagation register bank design. Thanks to these two, the FPGA bus implementation managed to be clocked at 100 MHz, thus effectively achieving a 100 Mbit/s burst data throughput.**

***Index Terms*—SPI, serial peripheral interface bus, serial link, serial communication, master and slave controller, register bank.**

I. Introduction

SERIAL communication can be described as the process of sequentially sending single bit data at a time, over a bus or communication channel in general. In telecommunication, this contrasts the parallel communication, in which several bits are sent as a whole, on a link with several parallel channels.

Serial type of communication is used for literally every long-haul communication and most computer networks, where the cost of cable and synchronization difficulties make parallel communication impractical. The advantages of parallel data transmission (i.e., no need for SerDes or serializer and deserializer) are outweighed by an improved signal integrity and transmission speeds in newer serial technologies [1] making serial computer buses more common even at shorter distances.

Although serial links might, at first, seem inferior to parallel ones, since they can transmit less data per single clock cycle, it is often the case that serial links can be clocked considerably faster thus achieving higher data rates. Factors like clock skew and crosstalk between lines, which are far more pronounced in parallel links, disallow those to be clocked at a higher rates.

Furthermore, in many cases, serial links are simply cheaper to implement than parallel. The vast majority of integrated circuits (ICs) have serial interfaces [2], as opposed to parallel ones, so that they can have fewer pads/pins and hence shrink die area and reduce package size and cost. This is always done when data transfer speeds are not of paramount importance.

Some examples of such low-cost serial buses for integrated circuits include SPI, I^2C, SMBus, UNI/O, 1-Wire, etc. Generally speaking, many of these communication systems were originally designed to connect two or more integrated circuits.

V. M. Milovanović is with the Faculty of Engineering, University of Kragujevac, Sestre Janjić 6, 34000 Kragujevac, Serbia (e-mail: vlada@kg.ac.rs).

Darko Tasovac is with NovelIC Microsystems, Veljka Dugoševića 54/A3, 11060 Belgrade, Republic of Serbia (e-mail: darko.tasovac@novelic.com).

The Serial Peripheral Interface bus (SPI), developed by [2] Motorola in the late seventies, is a synchronous serial communication interface specification used for short distance communication, primarily in connection with integrated circuits and embedded systems. Since then, the interface has become a *de facto* standard with typical applications which include, but are not limited to, communication with all sorts of sensors, data converters, (flash) memories, real-time clocks (RTCs), various types of displays such as LCDs, and custom integrated circuits.

SPI devices communicate between each other in full duplex mode using a master-slave configuration with a single master and one or more slaves. Multiple slave devices are supported through selection with individual slave select ($\overline{SS}$) lines but the master device originates the frame for reading and writing.

Sometimes SPI is called a four-wire serial bus, contrasting with three-wire, two-wire, and one-wire serial buses. Although the SPI is accurately described as a synchronous serial interface, it should nevertheless be distinguished from the so-called Synchronous Serial Interface (SSI) protocol, which is a completely different variant of a four-wire synchronous serial communication protocol that employs differential signaling. Even though there are some similarities between the SPI bus and the JTAG protocol, they are not interchangeable. The SPI bus is intended for much higher speeds, on board initialization of device peripherals, while the JTAG protocol is utilized to provide reliable test access to the I/O pins from an off board controller with less precise signal delay and skew parameters.

Advantages of the SPI over the competing serial communication protocols are mostly connected to higher achievable throughput. It is usually categorized [1] as relatively short-distance and medium-speed interface (clock is not limited to any maximum speed). Moreover, since the default version of the SPI protocol defines all signals as unidirectional, both Galvanic isolation and software implementation are feasible.

Among the main disadvantages of the SPI bus are absence of multiple master device support, no formal standard definition, and finally, a lack of hardware slave acknowledgment allowing the master to transmit basically nowhere and not knowing it.

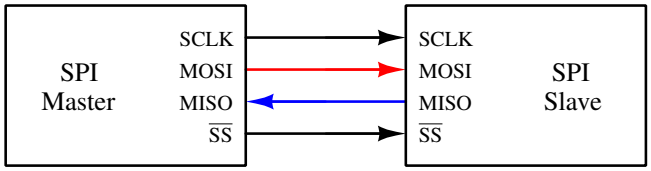

Fig. 1. A basic SPI bus example of Single SPI Master to Single SPI Slave.

II. The Serial Peripheral Interface (SPI) Bus

The standard four-wire Serial Peripheral Interface (SPI) bus, as depicted in Fig. 1, specifies the following four logic signals:

- SCLK or SCK: Serial Clock (generated by the master);
- MOSI: Master Output Slave Input (data out from master);
- MISO: Master Input Slave Output (data out from slave);
- $\overline{\text{SS}}$: Slave Select (often active low, output from master).

The above pin names are currently the most popular. In the past, alternative pin naming conventions were sometimes used, and so the SPI port pin names for older IC products may differ. Slave Select ($\overline{\text{SS}}$) fundamentally has the same functionality as Chip Select ($\overline{\text{CS}}$) and is used instead of an addressing concept. If a single slave device is used, the $\overline{\text{SS}}$ pin may be fixed to logic low if the slave permits it. However, some slaves require a falling edge of the chip select signal to initiate an action.

All slave devices intended for operation in multi-slave SPI environments posses tri-state outputs so that their MISO signal becomes high impedance (logically disconnected) when the device is not selected. Devices without tri-state outputs cannot share SPI bus segments with other slaves devices, instead as in Fig. 1, only one such slave device could talk to the master.

To begin communication, the bus master configures the clock, using a frequency supported by the slave device, typically in the order of dozens of megahertz. In addition to setting the clock frequency, the master must also configure the clock polarity and phase with respect to the data. Originally Motorola names [3], [4] these two options as CPOL and CPHA respectively, and most vendors have adopted that convention.

The timing diagram, shown in Fig. 2 applies to both the SPI master and the slave device. For CPOL=0 the SCLK clock active is high, while for CPOL=1 it is low. Likewise, CPHA=0 means sampling on the first clock edge, while CPHA=1 means sampling on the second clock edge, regardless of the actual CPOL value and whether that clock edge is rising or falling.

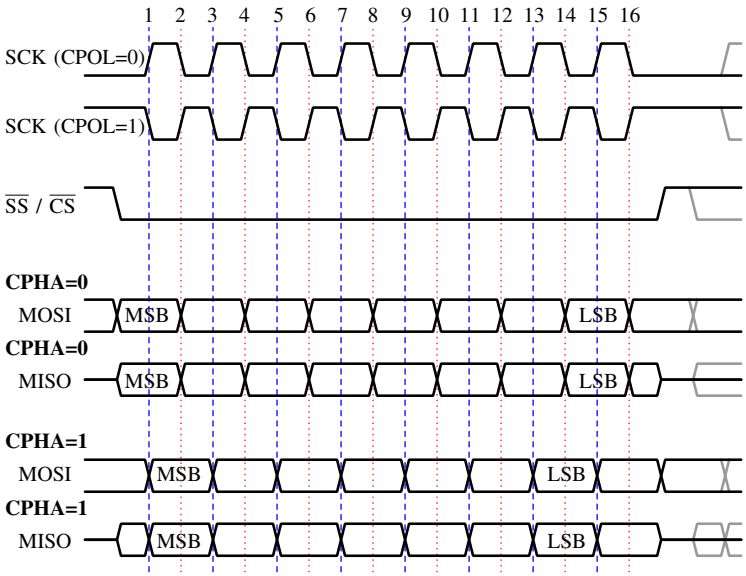

Fig. 2. A timing diagram showing SPI's clock polarity and phase. The first two diagrams are of SCLK for CPOL=0 and CPOL=1 while the dashed and dotted vertical lines represent sampling time instances for CPHA=0 and CPHA=1.

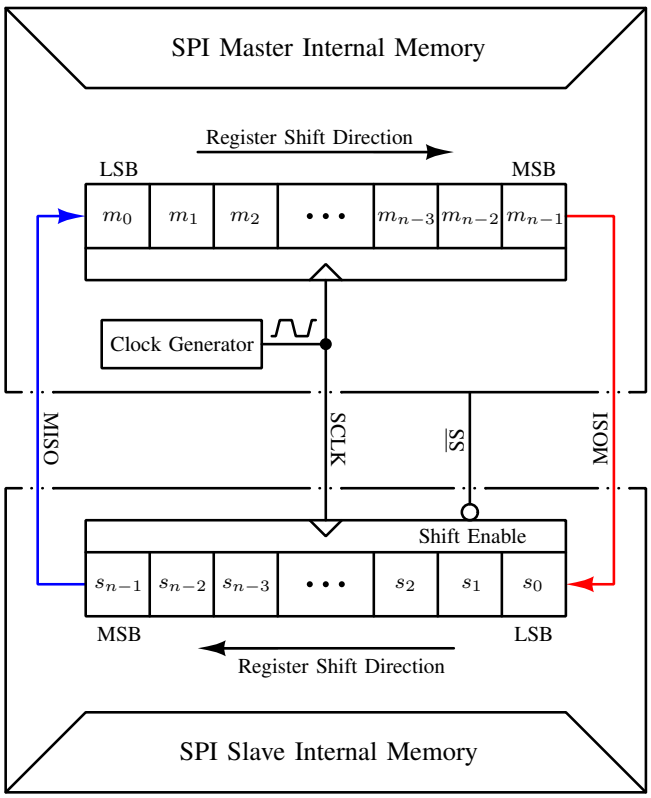

Fig. 3. A typical SPI Master and SPI Slave hardware setup using two same sized shift registers to form an inter-chip circular buffer for data transmission.

During each four-wire SPI bus clock cycle, a full duplex data transmission occurs. The master sends a bit on the MOSI line and the slave reads it, while the slave sends a bit on the MISO line and the master reads it. This sequence is maintained even when only unidirectional data [3] transfer is intended.

Transmissions normally involve two shift registers of some given word size, such as, e.g., 8-, 12-, 16-, 24-, or 32-bits, one on the SPI master and the other on the SPI slave side. Just as provided in Fig. 3, these shift registers are connected in a virtual ring topology which is actually a circular shift register buffer of $2n$-bits in length, with n-bits present in each device as master and slave vectors, $m_{n-1:0}$ and $s_{n-1:0}$, respectively.

Data is usually shifted out with the most-significant bit (MSB) first, while shifting a new least-significant bit into the same register. At the same time, data from the counterpart is shifted into the least-significant bit (LSB) register. After the register bits have been shifted out and in, the master and slave have exchanged register values. If more data needs to be exchanged, the shift registers are reloaded to and from some other means of the internal memory (FIFO, register bank, etc.) and the process repeats. Transmission may continue for any number of clock cycles. When complete, the master stops toggling the clock signal, and typically deselects the slave.

It is not seldom for some SPI devices to use different bit-length communications, as, for example, when SPI is used to access the scan chain of a digital integrated circuit by issuing a command word of one size and then getting a response of a different size (one bit for each pin in that scan chain).

III. The Custom SPI Slave System and Protocol

As already mentioned in the introduction, the SPI bus is a *de facto* standard. However, the lack of a formal standard is reflected in a wide variety of protocol options. Namely, different word sizes are common. Practically every device defines its own protocol, including whether it supports commands at all. Some devices are transmit-only while others are receive-only. Chip selects are sometimes active-high rather than active-low. Some SPI protocols [4] even send the least significant bit first.

There are also hardware-level differences. Some chips may combine MOSI and MISO into a single bidirectional SISO data line. This is sometimes referred to as the "three-wire" signaling (in contrast to normal "four-wire" SPI bus). Another variation of SPI bus removes the chip select line altogether, managing protocol state machine entry/exit using other methods. Anyone needing an external connector for SPI defines their own. Signal levels depend entirely on the chips involved.

Some SPI devices even have minor variances from the CPOL/CPHA modes described in the previous section. Sending data from slave to master may use the opposite clock edge as master to slave. Devices often require extra clock idle time before the first clock or after the last one, or between a command and its response. Besides, there exist two possible logical frame dependencies within SPI logical layers. One is called in-frame communication since the data of the slave response is within the same time slot as the master's request. The other, named out-of-frame communication, is when the slave's logical response is within the master's next data frame.

A custom SPI bus slave controller proposed by this paper will use the "most standard" four-signal physical layer with the most significant bit (MSB) sent first and the least significant bit (LSB) sent last, just as described in Section II. Further, the active-low chip/slave select line will be used for managing the protocol's state machine. Additionally it will bring the full support for all CPOL/CPHA combinations and modes through appropriate parameters. Instead of both in-frame and out-of-frame communication, which is supported by, for example, the automotive industry standard named SafeSPI [5], only the former query-response frame dependency will be implemented.

In spite of a somewhat more complicated implementation of the in-frame with respect to the out-of-frame communication when higher data throughputs are considered, it fundamentally does allow higher serial communication speeds since masters' requests are directly followed by slave's response within the same time slot. Even though the use of longer word sizes are advantageous for communication speeds, a single byte (8-bit) memory addressing granularity is targeted, and hence a single byte word sizes for master and slave shift registers are chosen.

It is assumed that all the SPI devices communicate synchronously using a master/slave relationship over a serial data link that operates in full duplex mode, i.e., signals carry data in both directions simultaneously. The master device initiates the communication by generating a clock signal and selecting a slave device. This action is followed by the data transfer in either one or both directions simultaneously. In fact, as far as the SPI bus protocol is concerned, the data frame is always transferred in both directions. It is up to the master and slave devices to know whether a received byte is meaningful or not.

To establish the communication, the SPI master sets the wanted slave select pin to a low state. This $\overline{SS}$ pin has to remain low during the complete frame transfer. As soon as the transfer is complete, the $\overline{SS}$ pin is set back to a high state.

The SPI master and slave components exchange two frames whose structure differentiates in the so-called single and block or burst operation modes. The frame constitution and MISO/MOSI protocol architecture during the SPI slave read and write operations are sketched in Fig. 4 and Fig. 5, respectively. The frame data transfers through the SPI bus occur at the same time in both directions starting with the most significant bits (MSB) and ending with the least significant bits (LSB) of address and data. In the frame sent from the master to the slave the first seven bits transmitted represent a register address. The eighth bit is a read/write bit and it tells the slave if a register content (at the address previously sent) has to be read (0) or written (1). The next byte (8-bits) is dependent on the eighth bit and in slave write mode it represents the data that has to be written in the register while in slave read mode it is simply a dummy data (0x00) that is discarded by the slave.

On the other hand, in the frame sent from the SPI slave to the SPI master, the first seven bits are always dummy data (0x00). The eighth bit, denoted by C, is a check bit and it takes value of 1 if the address sent by the SPI master is valid or 0 if the address sent by the SPI master is invalid. The next byte depends on the SPI slave read/write mode and the actual check bit value. Namely, in the slave read mode, if the check bit is high, the last 8-bits represent the register content whose address was sent by the SPI master, otherwise, in the SPI slave write mode or if the address is invalid, a dummy data is sent.

Normally, the SPI master and slave communicate for 16 clock cycles during which a particular register is accessed for writing or reading. If other registers need to be accessed, new 16 clock cycle SPI frames are issued, each of those containing the address and data fields as explained by Fig. 4 and Fig. 5.

However, if the complete register bank, or larger successive portions of it need to be read or written, the SPI master is able to issue much more convenient block or burst commands, which address multiple registers within a single transaction.

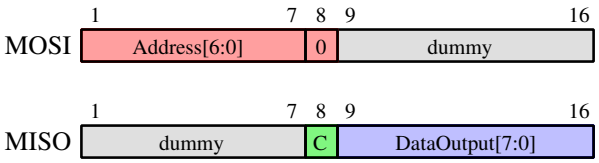

Fig. 4. A custom SPI protocol architecture showing MISO and MOSI data exchange between master and slave devices during the SPI bus read operation.

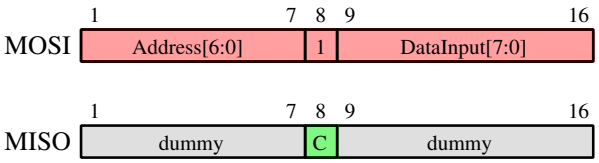

Fig. 5. A custom SPI protocol architecture showing MISO and MOSI data exchange between master and slave devices during the SPI bus write operation.

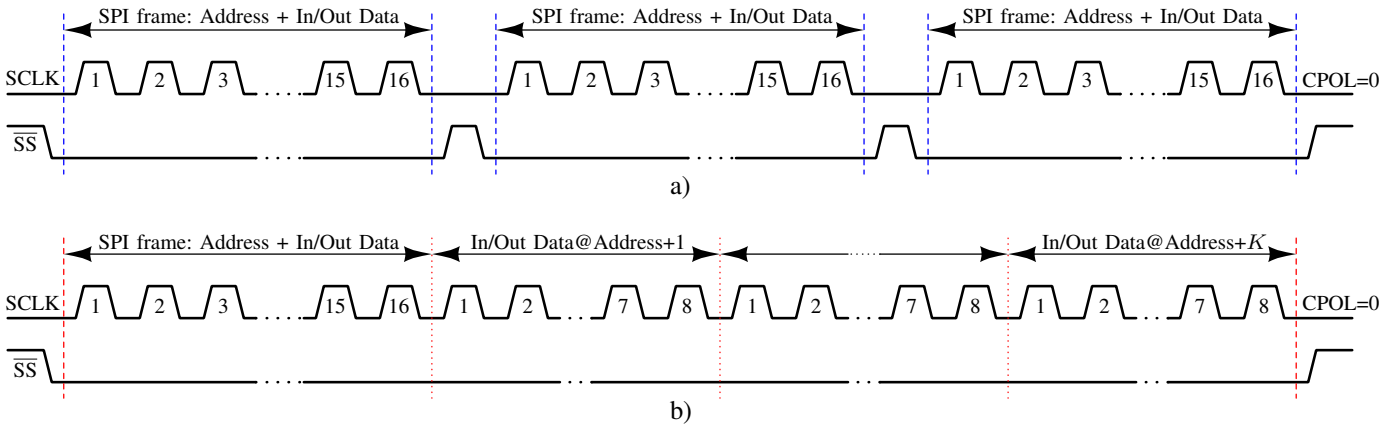

Fig. 6. Principle comparison diagrams of the two SPI master and slave controller modes showing only SCLK and $\overline{\text{SS}}$ for a) single, and b) burst operations.

IV. Custom SPI Slave Controller Implementation

The motivation to implement the so-called block or burst SPI read and write functions is to enable a quick set and save of a complete register bank configuration. This proved to be especially useful during test or operational phase to set back the integrated circuit of interest to a known configuration.

Normally, after the transfer of the 16-bit frame, the $\overline{\text{SS}}$ pin is put back in a high logic state to disable the SPI slave, just as given by Fig. 6a. To initiate a new transfer, another 16-bit SPI frame is issued. Nonetheless, if after the transfer $\overline{\text{SS}}$ is still low and the SCLK is still running, the SPI bus will operate in the burst register read or write regime, as shown in Fig. 6b.

Namely, if the SPI write burst operation was called by the SPI master, the master will send the data that has to be set on the succeeding register in the register map. Similarly, if the SPI slave is asked by the master to send data within the SPI block read operation, the slave will send the register content of the next register in the register map. No explicit addressing is necessary for any of the following read/write data transactions. Burst-mode operation finishes when the $\overline{\text{SS}}$ pin goes back high.

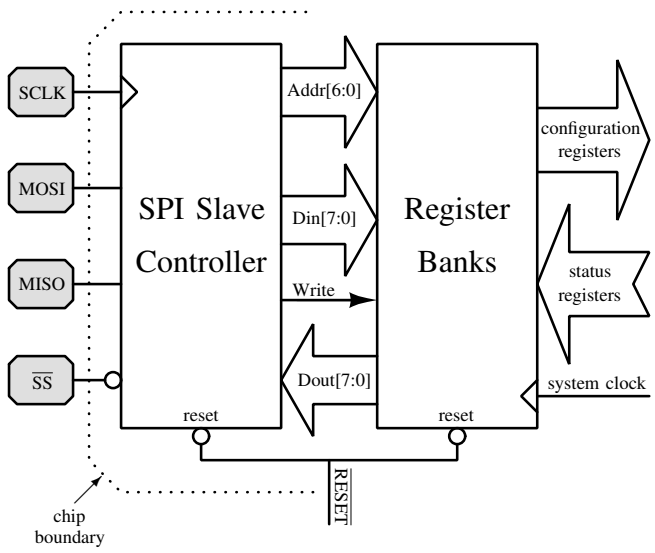

Fig. 7. A simplified SPI slave controller and its register bank interconnection.

A custom SPI bus slave controller with read/write register banks has been designed to support all previously described features. A simplified architecture is depicted in Fig. 7 which shows logical interconnect of two main parts: the SPI slave controller itself, and the supporting addressable register banks.

The SPI compliant slave controller decodes the SPI bus signals and deserializes them into a series of 8-bit bytes [6]. Communication with the SPI slave controller is established by means of a simple data structure: a single 7-bit address register and a single bit control register. The control register defines whether the transfer is a read or write while the address register provides an index into the register bank. Both configuration (write) and status (read) registers are directly connected to the external ports of the controller. The configuration registers provide general purpose read/write bits for the control of an external device. The status registers are read only and allow the state of external pins to be monitored via the SPI interface. The configuration/status register space is uniquely addressable.

All inputs to the slave interface controller are driven off-chip by the SPI bus master with the exception of MISO which is a three-state output. The MISO signal is normally in the high-impedance state unless a read operation is in active progress.

The slave controller core is a state-machine that continually monitors the state of the SPI signals. An SPI transfer begins with the high-to-low transition of the slave select signal $\overline{\text{SS}}$. Once $\overline{\text{SS}}$ is driven low, the controller will sample the next 16 bits from the master at the MOSI input. Bits are sampled on either the rising or falling edge of SCLK depending on the clock's CPOL and CPHA configuration settings. The first 7 bits in the transfer are written to the internal address register while the eighth bit is written to the internal control register.

The address register contains the index of the (first) register to be accessed in the register bank. Once the address and control registers have been written, the next eight serial clocks are used to synchronize a write to a configuration register or a read from any register. In the burst mode regime, after each 8-bit read or write, the internal address register is incremented and the master may read or write a further byte. This means that successive back-to-back reads or writes will be performed on the next register in the register bank. Once the maximum address has been reached the address pointer will wrap around.

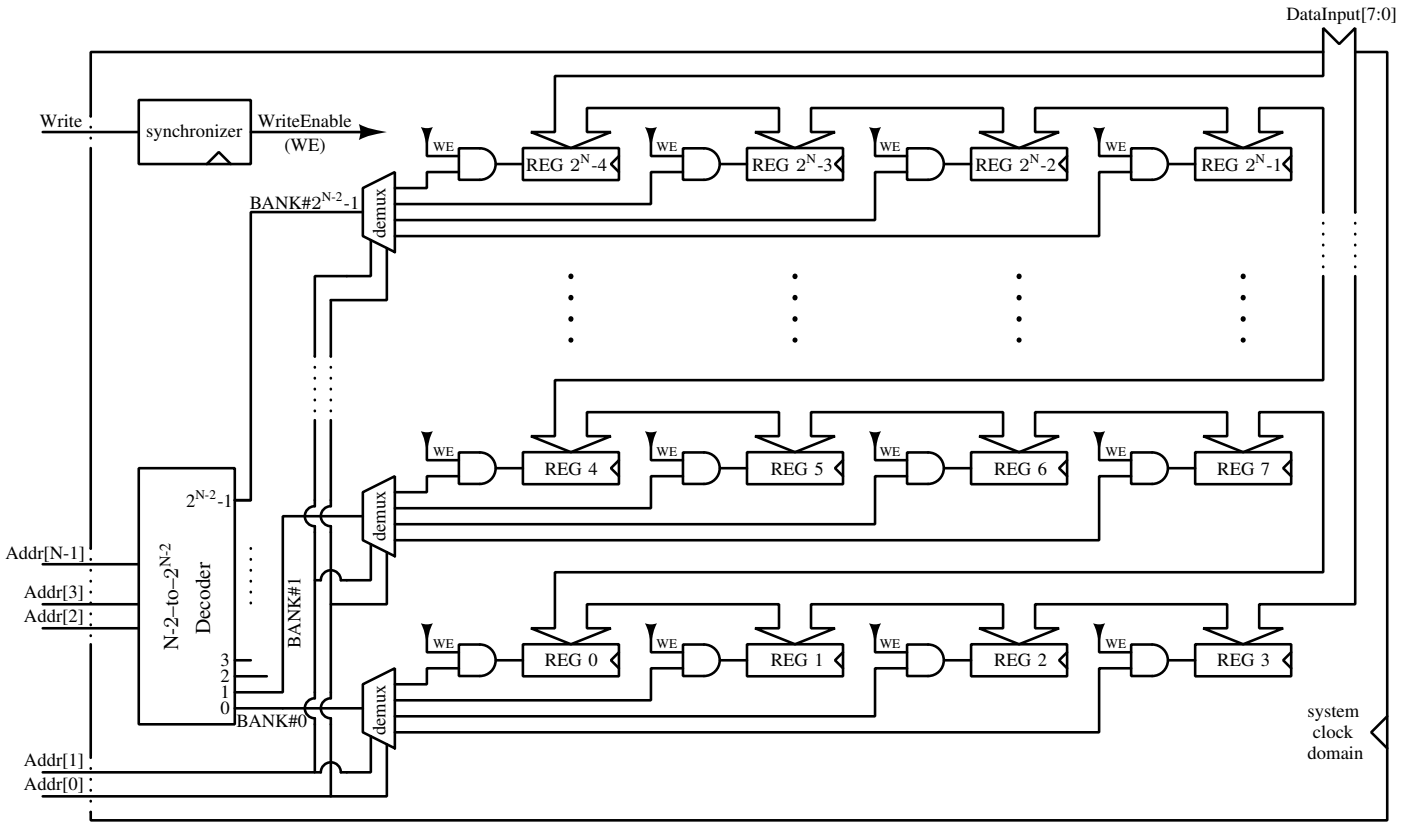

Fig. 8. A simplified write operation data path within the SPI register bank. Write synchronization exists due to difference between the SPI and system clock.

An internal bit-counter is responsible for the current SPI bus frame monitoring thus supporting the controller state machine.

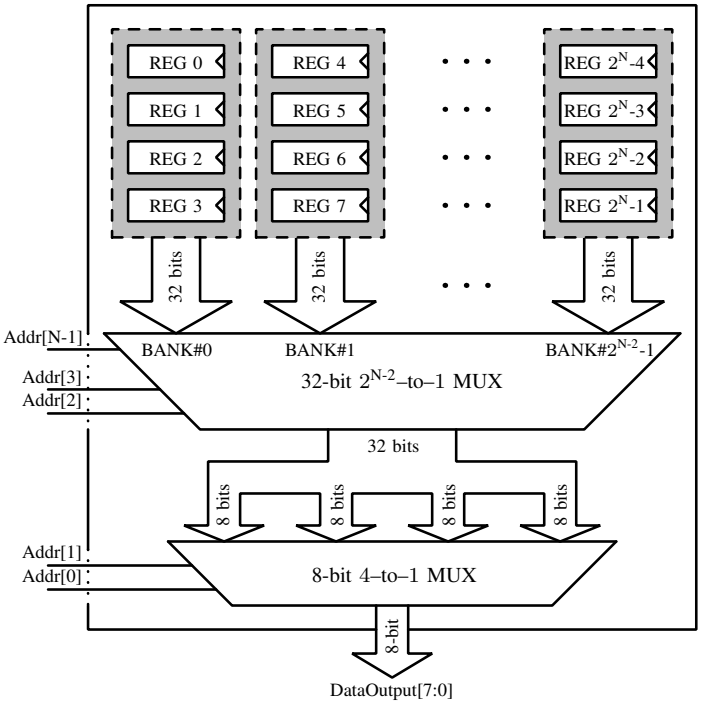

Fig. 9. A simplified SPI register bank's read operation data path "pipeline".

Instead of a single circular buffer, as sketched in Fig. 3, the actual SPI slave controller core consists of two 8-bit shift registers, one for transmitting (TX) and the other for receiving (RX) the bus messages. The LSB of the RX register is directly connected to the MOSI line, while the MSB of the TX register is connected (through a three-state buffer) to the MISO pin.

A simple 8-bit data buffering is performed. Also, a separate `Write` signal between the SPI slave controller core and the register bank is necessary to synchronize the write operation, as the SPI clock and the system clock generally differ both in phase and frequency (with both possible relations in-between).

In contrast to data lines, which are buffered at the end of corresponding SPI transaction, a two-way address buffering with an intermediate point is performed. The 7-bit address is divided/buffered into 5-bit pre-address and 2-bit post-address. The reason for this separation is explained in the next section.

Generation of the so-called check bit C is accomplished via simple minimized combinational logic network. The address validation is hard-coded with no real insight into the register bank. In this way the check bit information is readily available.

The burst regime entry point matches the internal bit-counter overrun (value of sixteen) which is used by the state machine.

Finally, it is worth of noting that any number of sequential register read or register write operations may be performed within the "extended" burst frame. The SPI bus transfer will terminate immediately as soon as $\overline{SS}$ is driven high. If the user wishes to switch read and write mode, the current SPI transfer must be terminated before the read or write mode is changed.

V. A Read & Write Register Bank Architecture

The configuration registers are organized as a bank of 8-bit general purpose read and write registers that may be accessed via the SPI bus slave controller. The configuration registers are designed to be used for the general configuration of devices external to the controller. The contents of these registers are made available at the controller output port which is composed of all the configuration register bits [6] concatenated together.

The status registers follow exactly the same structure as the configuration registers. The difference is that these registers are read only. Any attempt to write these registers will have no effect other than to perform a dummy transfer on the SPI bus. The status registers are designed to be used for snooping the state of control signals in an external device. Equivalently, the input port contains the contents of all the status register bits concatenated together. The total number of 8-bit registers in this implementation is 128 (defined by seven address bits).

The register bank architecture uses a system clock, therefore at least a single clock synchronization from the SPI clock (SCLK) on the write strobe signal is necessary to avoid metastability when crossing between these different clock domains.

The critical point of the proposed custom slave controller architecture is the SPI read operation. It is obvious from Fig. 4, that there is only a single SPI clock period between the last address bit that comes on the MOSI line and the first output data bit that should appear on the MISO line. The complete controller and the register bank should react within this SPI clock cycle and provide the requested data to the bus output.

To support this key feature, an innovative register bank has been introduced, whose separated write and read data paths are provided in Fig. 8 and Fig. 9, respectively. Both of these two data paths are in a certain sense "pipelined" in order to decrease propagation delays. Namely, five MSBs of the seven-bit address come first and are immediately applied to the larger 32-to-1 multiplexer array and 5-to-32 binary decoder in case of read and write data paths, respectively. By the time point in which SPI slave controller accepts the two address LSBs, the four-register bank data has already propagated through major combinational components and only 4-to-1 multiplexer array and 1-to-4 demultiplexer are remaining in the read/write data path. Propagation through the elementary networks is feasible even for higher clock speeds. The timing requirements are in this manner substantially relaxed. Now it becomes apparent why addressing was divided in pre- and post-address buffering.

Although other divisions and even deeper decoder and multiplexer granularities are certainly possible, register organization into groups of four with final two-bit (post-address) decoding, seemed as sort of a sweet spot for our design needs.

VI. Functional Simulation & Measurement Tests

The described custom SPI slave controller together with the register banks was implemented in both Verilog and VHDL Register-Transfer Level (RTL) behavioral modeling code. The slave controller was implemented with a maximum bank size of $2^7 = 128$ single byte wide (8-bit) registers. After the functional verification, the code was synthesized and placed on the ZedBoard's [7] Zynq™-7000 All Programmable (AP) SoC with external SPI wires lead to the PL-side Pmod connectors.

For practical testing purposes, an appropriate SPI bus master controller also had to be implemented. Both, the hard-coded FPGA [8], [9] and kernel-based [10] masters were realized on another ZedBoard. Correct operation was achieved with serial SPI clock frequencies of 100 MHz which roughly correspond to data transfer rates of 100 Mbit/s in the burst mode. These numbers are in agreement with, but still somewhat higher than other pure FPGA implementations of the SPI master/slave transceivers [11] found in the technical literature. Comparable data throughputs are expected in the pending ASIC realization.

VII. Conclusion

A custom serial peripheral interface (SPI) bus slave controller together with configuration and status register banks is presented in this paper. The SPI slave controller is based upon a standard four-wire interface but it defines its own in-frame communication protocol. This SPI protocol supports both single register addressing as well as burst transfer modes, and is tailored for obtaining the maximum data rate performance.

In such constellation, bus read operation presented a bottleneck as only a single SPI clock period was allocated for the complete address decoding and data delivery. Innovative hardware implementation which organized the bank registers in groups of four allowed a separate pre-address to decode the register group and the post-address to reach the explicit single-byte register in the final decoding step. This "combinational pipelining" to a large extent relaxed the timing requirements.

This design was tested on a commercially available FPGA development kit. Using hardware- and software-based SPI master implementations, the bus was clocked up to 100 MHz which translates to nearly 100 Mbit/s, a state of the art figure of merit among rather simple serial communication protocols.

Acknowledgements

The authors would like to thank the colleagues from the University of Kragujevac and NovelIC on helpful discussions.

References

[1] Louis E. Frenzel, "Handbook of Serial Communications Interfaces: A Comprehensive Compendium of Serial Digital Input/Output (I/O) Standards", 1st Edition, *Newnes Books – Elsevier*, 2015

[2] A. K. Oudjida, M. L. Berrandjia, R. Tiar, A. Liacha and K. Tahraoui, "FPGA Implementation of I²C & SPI Protocols: A Comparative Study", *Proceedings of the 16th IEEE International Conference on Electronics, Circuits and Systems (ICECS)*, pp. 507-510, 13-16 Dec. 2009, Tunisia

[3] Motorola/Freescale/NXP, "Serial Peripheral Interface (SPI) Block User Guide", version 3.06, document number: S12SPIV3/D, 4 February 2003

[4] The Linux Kernel Archives, "Overview of Linux Kernel SPI Support", www.kernel.org/doc/Documentation/spi/spi-summary, 2 February 2012

[5] SafeSPI Open Standard Specification, "SafeSPI – Serial Peripheral Interface for Automotive Safety", version 1.0, www.safespi.org, 3 June 2016

[6] ZIPcores Data Sheet, "SPI Slave Serial Interface Controller", version 1.1, www.zipcores.com/datasheets/spi_slave.pdf, 31 May 2012

[7] Avnet and Digilent, "ZedBoard (Zynq™Evaluation and Development Board) Hardware User's Guide", version 2.2, 27 January 2014

[8] Xilinx Data Sheet, "LogiCORE's IP AXI Serial Peripheral Interface (AXI SPI)", version 1.02a, 18 January 2012

[9] Xilinx Product Guide, "LogiCORE's IP AXI Quad Serial Peripheral Interface (AXI Quad SPI)", version 3.2, 5 April 2017

[10] The Linux Driver Implementers API Guide, "Serial Peripheral Interface (SPI)", www.kernel.org/doc/html/latest/driver-api/spi.html, 1 May 2017

[11] A. K. Oudjida, M. L. Berrandjia, A. Liacha, R. Tiar, K. Tahraoui and Y. N. Alhoumays, "Design and Test of General-Purpose SPI Master/Slave IPs on OPB bus", *Proceedings of the 7th International Multi-Conference on Systems, Signals and Devices (SSD)*, 27-30 June 2010, Amman, Jordan