Development of I²C Bus Driver

D. Ivošević and V. Sruk
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing, University of Zagreb
Complete Address: Unska 3, Zagreb, 10000, Croatia
Phone: (+385) 1 6129 926 Fax: (+385) 1 6129 653 E-mail: danko.ivosevic@fer.hr

Abstract - A typical embedded system encompasses logic design, processors, and access to the peripheral devices. In the embedded systems design usage of Field Programmable Gate Arrays (FPGAs) allows a system designer to customize both processor and digital logic to the specific needs of a target application. Usually, it also requires appropriate access to the peripheral devices. One of the approaches to access peripheral devices is the usage of the I²C bus which is currently the industry de-facto standard for low-speed IC-to-IC communication. The I²C bus is a simple two wire serial bi-directional bus supported by wide range of peripheral devices such as memories, A/D and D/A converters, real-time clocks, video drivers, and microprocessor.

In this paper, we present comparison of different I²C bus implementations approaches as IP-cores using FPGA: an open-source core written in VHDL, PicoBlaze soft microcontroller core, and proprietary IP core component. We have analyzed features and implementation issues such as area and timing performance on different FPGA architectures.

I. INTRODUCTION

Rapid embedded system design usually starts using test and development platforms that include Field Programmable Gate Array (FPGA) devices and numerous peripheral devices for accessing the outer world. The spectrum of available platforms is wide as it is the choice of FPGA's and peripheral devices. The most common peripherals are all types of memories, audio and video ports, Ethernet, USB, and peripheral busses. Including their functionalities into some design puts a demand of accessing them through various buses and protocols. The communication can be based on serial communication buses such that are RS-232, I²C, I²S, SPI or CAN.

There is no doubt that the FPGA technology has an enormous impact on embedded computing. The concept of a programmable logic device enables designer the flexibility to implement customized digital logic circuits from dedicated logical function to complex processors. In developing process designers have, depending of problem complexity, a variety of choices used to develop designs from using Hardware Description Languages (HDLs) to use of advanced methodologies based on extensive Intellectual Property component reuse. Minimizing design and development time and cost is the goal of an efficient design methodology.

Today FPGA based design is typically realized using appropriate Computer Aided Design tools. These tools support design entry using several different methods. As the design complexity grows, HDLs become leading choice since they handle low-level implementation details automatically. They are also suitable for the development of the different forms of IP cores which are used to build a larger or more complex system. IP cores are divided into three categories according to the flexibility of reuse: soft cores, firm cores, and hard cores.

It this paper, we present comparison of three I²C bus implementation approaches on two development platforms. Section II. talks on I²C bus standard specifications, and Section III. presents different implementation approaches. Section IV. deals with implementation details for two FPGA platforms and presents approaches comparisons. The last section, Section V., gives conclusions on presented aspects of undertaken implementation approaches.

II. I²C BUS STANDARD

An Inter IC bus widely known as I²C bus is a multi-master serial bus introduced in 1982 by Philips Semiconductors [1]. It was primary designed for efficient inter-IC control and nowadays is widespread in consumer electronics, telecommunications and industrial electronics. The bus protocol basis is a simple two wire bi-directional serial master/slave communication. The communication lines are connecting devices in the system and are denoted as SCL and SDA lines, as illustrated in Fig. 1.

The SCL line is a serial port clock line and it is driven by the master device. The SDA line is a bi-directional serial port data line. The master can demand reading and writing to a slave device, and the SDA line is used for the data interchange during the realization of those operations. The data transfer is 8-bit oriented. The protocol defines several data transfer rates from up to 100 kbits/s in the Standard mode to maximum of 3.4 Mbits/s in the High-speed mode.

The master device uses I²C bus protocol for reading and writing to a slave device internal registers. Fig. 2. shows patterns of read and write operations. As I²C bus protocol figures out, start of the communication is marked using START condition and it ends using STOP condition. When the bus is in the idle state, the SCL and SDA lines are kept high. The START condition is recognized when the SDA

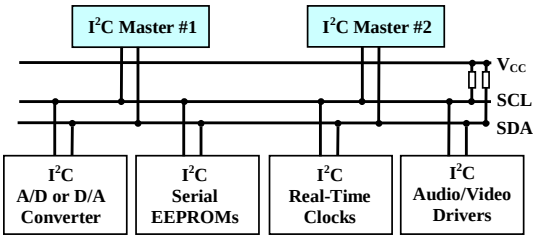


Fig. 1. Example of I²C bus applications

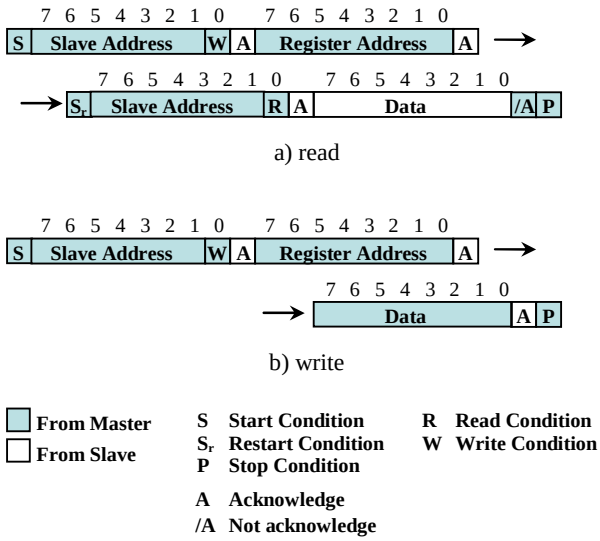


Fig. 2. I²C read (a) and write (b) patterns.

line is driven high-to-low while the SCL line is high. The STOP condition is recognized when the SDA line is driven low-to-high while the SCL line is high. Since I²C bus protocol allows multiple slave devices, a specific slave device is accessed by setting appropriate address on the SDA line. After this, the addressing of the device's internal register follows. The steps following the addressing phase depends of the type of the operation that master device wants to perform. The types of the operation are read or write. The I²C protocol uses 7-bit slave address and the eighth bit specifies whether master wants to read or write to slave. In a case of writing, next byte of data contains contents to be written to the slave device register, and the master finishes the transfer using the STOP condition, as in Fig. 2a. During the read operation, master repeats the START condition and the slave addressing sequence, but with a last bit in slave address byte set low to signify the read operation. After that the master expects appropriate data on the SDA line set by the slave and finishes the transfer with the STOP condition, as in Fig. 2b.

During initiation of communication and addressing phase the slave device is required to respond to every received byte with acknowledgment signal. Using low leveled signal the slave affirms its presence on the bus and its preparedness for the transfers. After receiving data from the slave, master device is also required to acknowledge every received byte except for the last received byte. By not acknowledging the last received byte it is announcing to slave not to send more bytes what concludes the read sequence.

III. IMPLEMENTATION APPROACHES

In this work, we examine different approaches for I²C master implementation using FPGA technologies. The first studied implementation is based on an open source I²C controller core from OpenCores [2]. The second is based on a software implementation using Xilinx PicoBlaze soft core processor [3, 4]. The third is based on a proprietary I2CM IP core from Altium Limited [5]. The features of the approaches are described in following subsections.

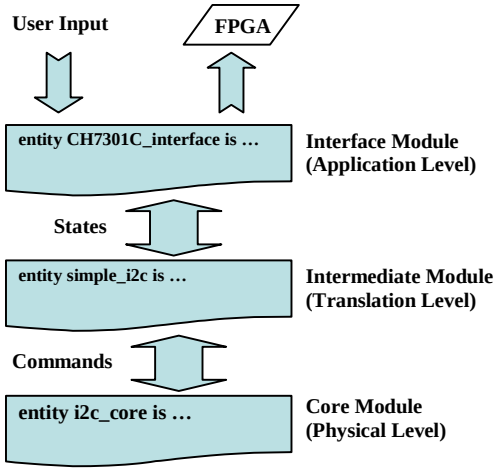


Fig. 3. The configuration of OpenCores.org based implementation.

A. OpenCores.org I²C Controller

The solution from OpenCores represents one of the open-source I²C controller cores. The advantage of using such an approach is in the full accessibility to low-level implementation of I²C bus protocol. The scheme of the implementation is presented in Fig. 3. In this case, typical code consists of several entities written in VHDL: the core module, the interface module and the intermediate module.

There are two purposes of the interface module: it defines access points to I²C pins on the target and enables the user to specify the device's and its internal registers' addresses. Also, this module implements the protocol command sequences as a Moore state machine, as shown in Fig 4. The states of the machine are defined at higher level easily understandable to user. The intermediate module is used to interpret these states and translate them to I²C commands. The core module interprets the commands into bit-level data for sending over SDA and SCL lines. The SDA and SCL signals are "wired" with other data through all three modules and in the interface module they are connected to target SDA and SCL pins.

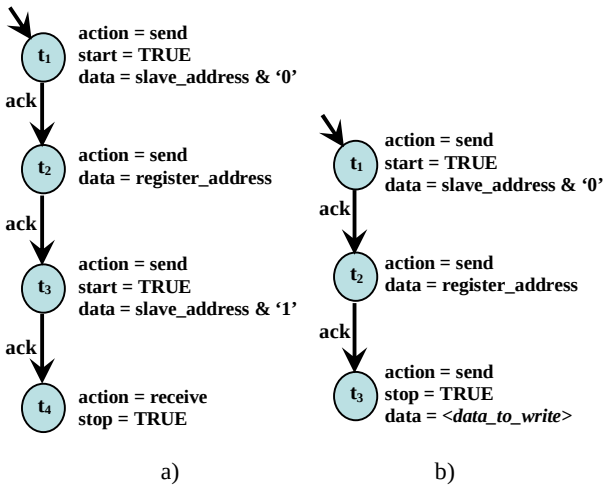


Fig. 4. OpenCores.org Interface module state machines for read (a) and write (b) operation.

B. Xilinx PicoBlaze I²C Controller

The PicoBlaze solution is one of the soft processors based solution. The soft processor core generally refers to an IP core that is every time synthesized through a logic synthesis process and can be implemented on different FPGA architectures. The PicoBlaze is a specific implementation of 8-bit RISC soft processor core which is optimized and embedded within Xilinx FPGA architectures. Inside the FPGA device it is interfaced to available peripheral logic through its input and output ports.

The PicoBlaze core is delivered as synthesizable HDL source code and accompanying development suite. The suite consists of assembler for writing an application and a template code for generating HDL source that represents the instruction memory.

The implementation scheme of the I²C master is presented in Fig. 5. It consists of top-level interface module, controller module, PicoBlaze processor core and assembled PicoBlaze code.

Top-level interface module defines access points to I²C user interface pins on the target. The application code is written in assembler and translated into assembled HDL source which represents the instruction memory for the PicoBlaze processor core. Controller module serves as a wrapper for the assembled HDL code and PicoBlaze processor core and ensures their mutual mapping. During execution the processor core reads the instructions coded inside the assembled code and runs the application.

C. Altium Designer I²C Controller

The Altium Designer tool is digital design development environment with graphical interface and fully integrated libraries of Altium proprietary IP cores. These IP cores are logic and data blocks designed to support the design reuse concept in electronic design automation industry. In more complex designs' development process they are used as reliable and portable design elements. The Altium Designer environment provides the user with numerous soft IP cores what means that they can be instantiated in design schematic as boxes with well-known functionalities. There are core components libraries with generic components, instruments, memories, peripherals, processors, and board

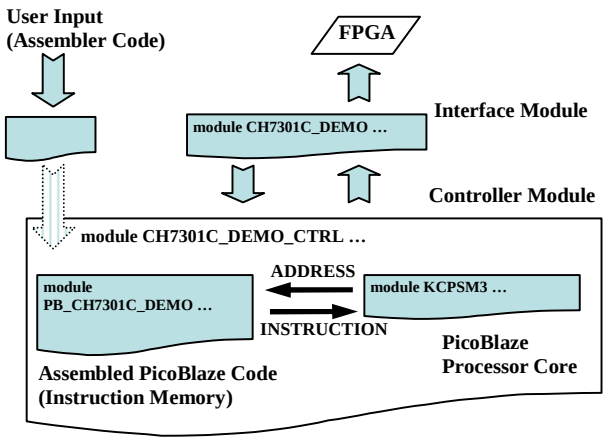


Fig. 5. The configuration of PicoBlaze based implementation.

port-plugins. A wide spectrum of present IP cores opens the horizon for rapid development of complex designs.

Altium I²C master core is implemented in two versions: non-Wishbone compliant (I2CM) and Wishbone compliant (I2CM_W) version. Schematic representation of the components in Altium Designer environment is presented in Fig. 6.

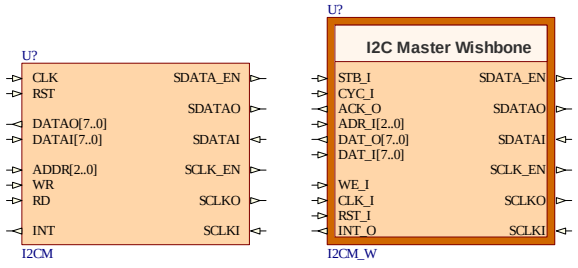
The non-Wishbone compliant core has the basic SCL and SDA I²C lines and interface lines for accepting user's demands; address and data buses, read/write select, reset and input clock. It is suitable for use in designs with simpler communication overhead and no demands for higher level bus standards.

The Wishbone compliant version introduces the additional Wishbone interconnect lines. Advantage of using standard bus is robust and flexible solution for enforcing compatibility between IP cores and allowing design reuse using standard bus components [6].

To implement the I²C master with Altium's IP cores a soft processor core is needed. Its task is to control the I²C master core. The final design includes RISC processor core, Wishbone interconnect, and I²C Wishbone-compliant controller. For the soft processor cores the environment provides adequate compiler support for assembler or C source code. Fig. 7. presents I²C controller architecture using the Wishbone interconnect features. Through the Wishbone interconnect the user application controls I²C master component.

The control of I²C sequences is done by C code application, as in portion of code below:

```
void SendI2CByte(unsigned char data, int last) {
    cmd = 0;

    i2c_int = 0;
    if (last) {
        cmd = I2C_CTRL_EN | I2C_CTRL_IEN | I2C_CTRL_WR |
        I2C_CTRL_STOP;
    } else {
        cmd = I2C_CTRL_EN | I2C_CTRL_IEN | I2C_CTRL_WR;
    }
    *I2C_WRIT_REG = data;
    *I2C_CTRL_REG = cmd;

    while(!i2c_int);
}

unsigned char GetI2CByte(int last) {
    cmd = 0;

    i2c_int = 0;
    if (last) {
        cmd = I2C_CTRL_EN | I2C_CTRL_IEN | I2C_CTRL_RD |
        I2C_CTRL_ACK | I2C_CTRL_STOP;
    } else {
        cmd = I2C_CTRL_EN | I2C_CTRL_IEN | I2C_CTRL_RD;
    }
    *I2C_CTRL_REG = cmd;

    while (!i2c_int);

    return *I2C_READ_REG;
}
```



a) b)

Fig. 6. I²C controller cores from Altium Designer library: non-Wishbone version (a) and Wishbone-compliant version (b).

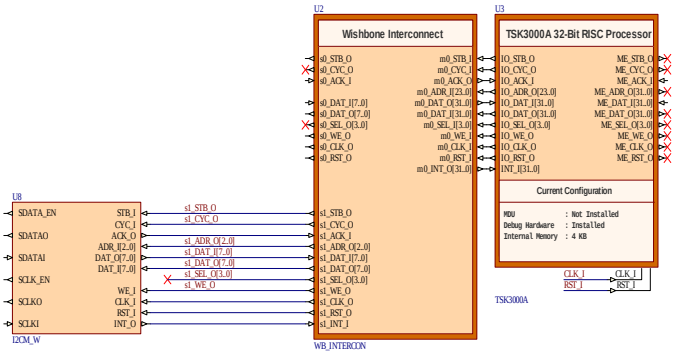


Fig. 7. Schematic of I²C Altium Designer based implementation.

The routines for sending (*SendI2CByte*) and receiving (*GetI2CByte*) data handle the communication sequences through a set of pointers. The user fills the pointer location *I2C_WRIT_REG* with byte of data to send, and receives the byte of data at *I2C_READ_REG* pointer location. The control of the communication is realized by combining special constants (*I2C_CTRL_EN*, *I2C_CTRL_IEN*, *I2C_CTRL_WR*, *I2C_CTRL_RD*, *I2C_CTRL_ACK*, *I2C_CTRL_STOP*) to the pointer location *I2C_CTRL_REG* which represents the control register. After issuing the commands, the communication is held inside those routines until variable *i2c_int* is set by interrupt from slave device. In such way the user is mastering the bus protocol in a higher level languages style of handling constants, variables and pointers. Since IP core takes care of low level protocol communication, there is no need of making the data sequences from bit-level pieces and coordinating them with the clock signal as in hardware description languages. Also, the C code higher level constructs allow that handling jumps, logic operations and moving the data between registers, as in a case of assembler code, are avoided.

IV. I²C PROTOCOL IMPLEMENTATION

Previously described implementations are investigated on two development platforms with different FPGA devices: Spartan-IIE and Virtex-5.

A. Spartan-IIE

The Spartan-IIE device is present on Altium's NanoBoard-NB1 platform developed for test and education purposes. The board utilizes I²C bus for accessing A/D and D/A converters.

Two successful implementations on Altium NanoBoard-NB1 were the OpenCores and Altium Designer based. The PicoBlaze processor core is not supported for Spartan-IIE device so the related implementation has been omitted. The information on device slices used and resulting cycles' periods after place and route phase is presented in Table I.

The assessment of presented results will be done in comparison with results on other platform in following sections.

TABLE I
IMPLEMENTATION RESULTS FOR SPARTAN-IIE

Solution	Occupied Slices (out of 3072)		Min. Period / ns	Max. Frequency / MHz
	#	%		
OpenCores	58	1	6.6	151.3
PicoBlaze	Not applicable			
Altium Designer	1885	61	31.0	32.3

B. Virtex-5

The Virtex-5 device populates the Xilinx's ML506 DSP development platform. The I²C bus is used for accessing EEPROM, Fan Controller, VGA Input, DVI Output and Serial Presence Detect (SPD) to the DDR DIMM.

The information on device resources consumed and resulting best possible timings after place and route phase is presented in Table II. The Virtex-5 platform has more available resources what reflects in slice occupation records and leaves space for development of more complex designs. Besides that, its slice organization is more advanced with 6-input LUT based logic what allows achieving the higher frequencies.

C. Implementation Approaches Comparison

Comparing proposed approaches the usage of Altium Designer environment with fully integrated proprietary IP cores eases the design process. It is especially useful when a soft processor core is already included in the design. However, the usage of such cores brings initial overhead to the device resources occupation. The solution based on Altium Designer environment on both platforms takes much more of available resources. Further, first two solutions are designed and optimized for use on lower levels of abstraction and consequently offer better performance in terms of resulting clock cycle periods. The size of solution reflects also on duration of synthesis and implementation process. The duration time for this process is 15 seconds for first two solutions, and over 1 minute for the third solution.

TABLE II
IMPLEMENTATION RESULTS FOR VIRTEX-5

Solution	Occupied Slices (out of 8160)		Min. Period / ns	Max. Frequency / MHz
	#	%		
OpenCores	37	1	3.1	322.9
PicoBlaze	66	1	8.5	117.6
Altium Designer	1073	13	13.7	73.0

TABLE III
DESIGN PHASE COMPLEXITIES

Solution	Written In	# Lines of Code
OpenCores	VHDL	500
PicoBlaze	Assembler	200
Altium Designer	C	200

Table III. reveals the complexity from application programming view in a sense of approximate code sizes and programming language levels. The inclusion of soft processor cores enables usage of conventional software programming languages and makes the job more familiar to application designer. In comparison to hardware description language (VHDL) such approach reduces the number of code lines to be written. The portion of C code given in Section III.C. particularly emphasizes the higher level approach in application programming.

V. CONCLUSION

The reality of embedded systems and hardware/software co-design environments assumes running of hardware base and development of software Electronic Design Automation (EDA) support. The hardware base could be the custom hardware component or development board based on flexible FPGA technology. The affordable development boards populated with FPGA device, memory elements and diverse peripheral devices lately became very popular choice for creating relatively quick solution prototypes.

To drive input and output peripheral ports the usage of soft processor cores often appears as a best choice [7, 8, 9]. Conventional approach of writing a HDL code that directly controls the components is moving towards higher levels of abstraction and is successfully replaced with a higher level code that runs on the processor core.

We explored approaches to I^2C bus protocol implementation on three different levels of abstraction. The prerequisites are different and each implementation has its own advantages and disadvantages. Implementation details for two different boards are presented.

The results are showing that writing a driver at lower levels of abstraction is still the most precise way which minimizes resources demands and maximizes the final frequencies. Unfortunately, it requires specific knowledge of special purpose languages, and consequently its design time is the most demanding. In the same time, introduction of soft processor cores represents a significant shift in designer's perspective. Considering more complex designs, with higher resource volumes and advances in FPGA device organization, the usage of soft processor cores essentially decreases the design time and improves the further integration in larger systems.

REFERENCES

[1] NXP, *UM10204 I2C-bus specification and user manual*, http://www.nxp.com/acrobat_download/usermanuals/UM10204_3.pdf

[2] OPENCORES.ORG, *I2C controller core: Overview*, http://www.opencores.org/projects.cgi/web/i2c/overview

[3] Xilinx, *PicoBlaze for Spartan-3 Generation, Virtex-4, Virtex-II, and Virtex-II Pro FPGAs*, http://www.xilinx.com/products/ipcenter/picoblaze-S3-V2-Pro.htm

[4] B. Timpe, "Hidden in Plain View", *XCell Journal*, Issue 67, p. 50 - 54, 2009.

[5] Altium, *I2CM Controller*, http://www.altium.com/files/learningguides/CR0105 I2CM Controller.pdf

[6] OPENCORES.ORG, *Wishbone Version B3*, http://www.opencores.org/projects.cgi/web/wishbone/wbspec_b3.pdf

[7] D & R, *FPGA Soft Processor Design Considerations*, http://www.design-reuse.com/news/?id=11622

[8] R. Chamberlain, J. Lockwood, S. Gayen, R. Hough, P. Jones, "Use of a soft-core processor in a hardware/software codesign laboratory", *Proceedings of the 2005 IEEE International Conference on Microelectronic Systems Education*, p. 97 - 98, 2005.

[9] H. Calderón and S. Vassiliadis, "Soft Core Processors and Embedded Processing: a survey and analysis", *Proceedings of ProRISC*, p. 483 - 488, 2005.