TESLA REPORT 2005-16

# Software for development and communication with FPGA based hardware

Jaroslaw Szewinski, Pawel Kaleta, Przemyslaw Fafara, Piotr Pucyk, Waldemar Koprek,
Krzysztof Pozniak, Ryszard Romaniuk
Institute of Electronic Systems, Warsaw University of Technology, Poland

## ABSTRACT

This document describes aspects of creating software for communication with hardware, especially with FPGA based systems. Features unique for FPGA systems are compared with features of the traditional electronic systems. There are discussed topics like hardware interface definition, or address space description. Connecting client application is shown with Matlab as an example. Solutions with FPGA and embedded processors are discussed.

**Keywords:** fpga, Matlab, dll, mex, powerpc, linux

## 1. INTRODUCTION

The usage and popularity of FPGA systems is growing, because of high flexibility and high performance of this kind of electronic systems in such areas like control or digital signal processing. In comparison with the traditional digital logic systems, FPGA can be easily reconfigured after board manufacture, while changing functionality of the traditional electronic hardware system is equal to redesign and produce PCB board. The changes in traditional hardware may appear once per few months, while in programmable logic design changes may appear few times per day.
It is acceptable for digital logic systems, that software used for communication with the board from the PC computer is dedicated for each hardware design, and it is upgraded when new PCB project is made. In case of programmable logic designs, such a solution makes development very difficult, because it requires recompilation of the software for every change in the hardware. Because of that, software for FPGA systems should be enough flexible to handle as many as possible changes in the hardware, without recompilation. At this point, it is very important to describe hardware functionality in a such a form, that it acceptable by application at run time. In this solution, hardware functionality is described by Internal Interface.

## 2. HARDWARE INTERFACE DESCRIPTION

It is very important to define an interface of hardware system for communication with the computer systems. Interface definition should contains electrical description of physical connection and data flow description of protocol. Once produced hardware system, has fixed physical interface, but in case of FPGA system, protocol may freely changed, even ports direction may change. To avoid such a problems, systems developed by Warsaw ELHEP Group use *Internal Interface* (II) to organize the layout of the components of the address space. It is an advanced solution of designing FPGA systems. The idea is to describe the address space of the designed system in a file, using simple text format easy for edition by user. In this file the elements of the address space are represented by names (mnemonics), the addresses are generated later, during VHDL code generation. On the other hand, base text description is used by software to determine addresses of particular elements. On the user level, address space is available as a set of text based names (mnemonics). This way of implementing address space, saves a lot of time in comparison with writing whole design from the scratch. On the other hand, file with address space description is used by software system to determine the addresses of the requested items.
At the beginning, addresses where calculated by macros expanded by C preprocessor during the compilation process. As a result, the addresses were hardcoded in the application. A natural consequence was that after every change in the hardware interface, software recompilation was required because addresses had to be recalculated. This situation was uncomfortable for the system development, because hardware engineers were dependent on software engineers to

develop electronic equipment. Process of generation hardware implementation and software support for the address space hardcoded in application is shown on figure 1.

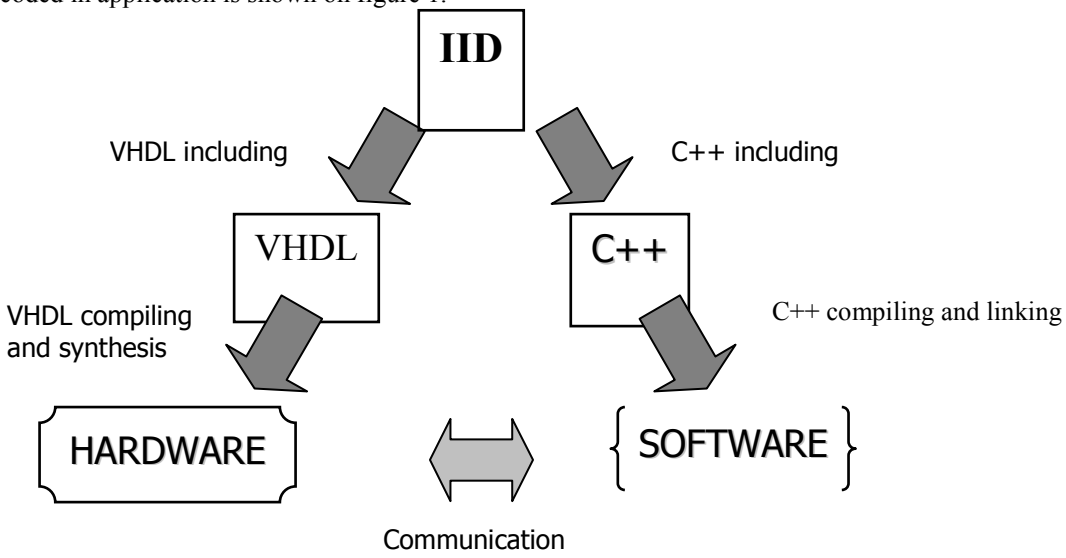


Figure 1. Internal Interface hard coded in software.

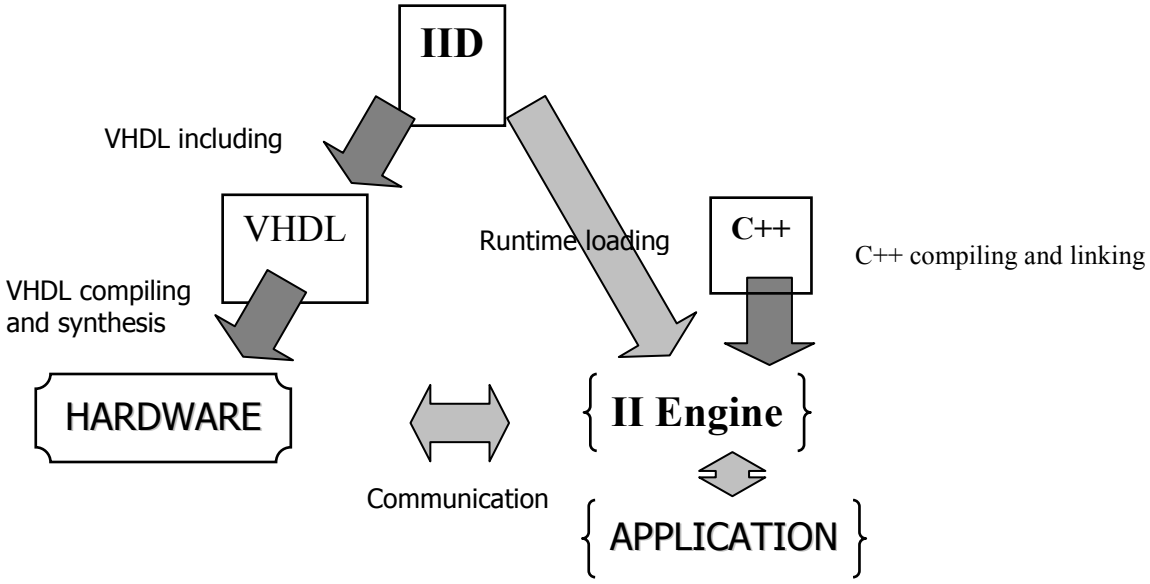


Figure 2. Internal Interface loaded at runtime.

The technique described above was not effective, so to improve the development process, the way of calculating addresses has been changed. The most important difference was that addresses were calculated at the run time (usually during system's startup). This solution does not require frequent software recompilations, after making change in the hardware, user simply replaces address space description file (IID file) and restarts application. This method is shown on the figure 2.

Run time address calculation is usually made at the startup because it can be time consuming (depending on complexity of the address space). During the run time, translated addresses are stored in the memory, so the access to this data is very fast.

## 3. SYSTEM ARCHITECTURE

To achieve better flexibility, system has been divided in to three logical layers: channel layer, II Engine layer and Application layer, between those layers interfaces have been defined.

- Application layer contains programs (i.e. Matlab, custom applications written in C++, etc.) which are clients of the system, they are initializing lower layer - II Engine. All requests to hardware are initiated by those programs.

- II Engine is responsible for translating mnemonic names of the II address space elements to addresses, and making requests to the hardware (thru the selected channel in the channel layer). This module loads the IID file right after it is initialized, and it calculates addresses. Calculated addresses are kept in the memory.

- Channel layer is responsible for accessing hardware using selected interface like EPP or VME. All channels have uniform API, so they can be used like plug-in. Changing channels is transparent to the higher layers of the system

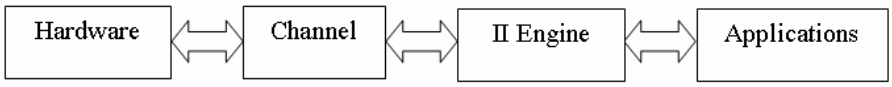


Figure 3. System Layers.

This solution gives end-user possibility to replace any element of the system with another one in the same layer, this operation is as easy as replacing files in the directory. It is possible, because channel layer and II Engine are implemented as dynamic libraries, witch can loaded at runtime – *Dynamic Link Libraries* on Ms Windows platform, and *shared objects* on POSIX systems. Application layer is usually implemented as executable programs which are able to load dynamic libraries, unless the system is used as a slave with other control system. In such a situation, a module (dynamic library) which acts a bridge, is placed in the application layer. This library is loaded by the master system, and then operates on libraries in lower level.
In future releases of the system, in POSIX family systems, channels will be implemented also as kernel mode driver (character device). This will enable more precise timing in communication, without effects of process scheduling in the operating system.

### 3.1 Matlab as the example of client using II Engine

In laboratory work, for processing measurement data, Matlab is used commonly. The obvious need is to transfer this data to the Matlab environment. To enable direct access from Matlab to the hardware (though the II Engine) there have been created a set of extensions for Matlab written in C language as a "Matlab Executables" - MEX-files, they appear in Matlab as a functions (like M-files).
Currently Matlab is used mostly on Microsoft Windows and Solaris platforms. On MS Windows it is useful that Matlab MEX'es are implemented as a Dynamic Link Libraries (DLL). If MEX source code is compiled using Win32-compatibile compiler, calls to WINAPI functions can be used. Additionally, if MEX itself is a DLL, it is guaranteed that it is linked at least with kernel32.lib library. II Engine implemented as a DLL, all is needed to use this library, is to include windows.h, and call LoadLibrary
The typical lifecycle of the MEX is presented below in figure 4.
Unfortunately Matlab uses the same thread to handle user input and MEX execution, because of that, Matlab is suspended (window blocked) for the time of MEX execution. The problem is when II Engine loads the IID file to translate names (mnemonics) of the FPGA items (registers and memory areas) into physical addresses. This operation is time consuming, it may take up to few seconds. This is extremely uncomfortable for user when reading or writing hundreds of registers in FPGA.

To walk around this problem the following technique has been used: there are two special MEX files, both of them does nothing, except that first (ii_lock) loads II Engine (unbalanced call to LoadLibrary(), without calling FreeLibrary() before exit), and second (ii_unlock) releases II Engine (unbalanced call to FreeLibrary()).

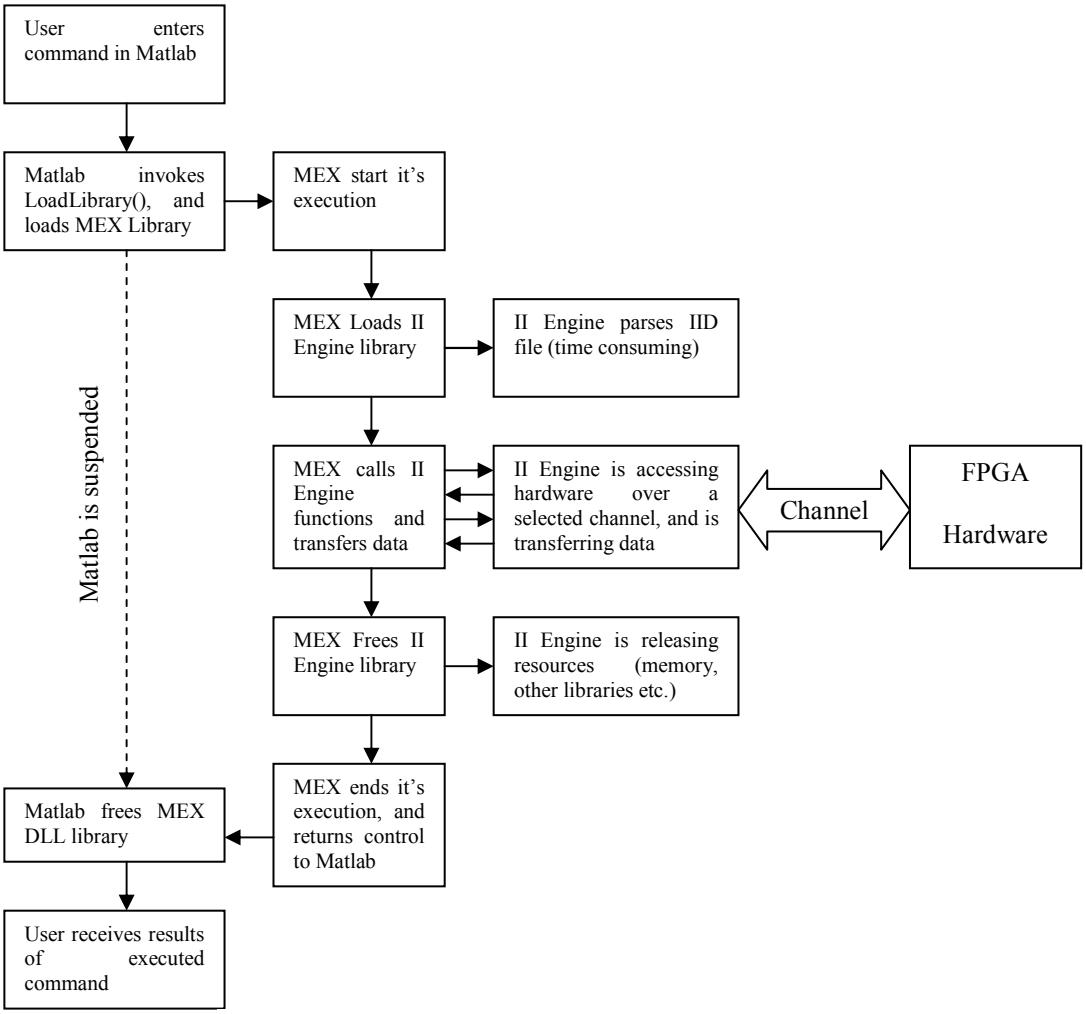


Figure 4. Lifecycle of MEX file.

When using this technique, MEX files are not loading the whole II Engine each time, but they only attach to preloaded library which has calculated addresses of all registers and memory areas.

Technique described above is possible, because calls to LoadLibrary() and FreeLibrary() is cumulative; FreeLibrary must be called as many times as LoadLibrary was, other way system will keep library in memory as long as number of calls to FreeLibrary() is less than number of call to LoadLibrary().
DLL libraries are mapped into address space of the calling process, the DLL_PROCESS_ATTACH
event is notified only during the first call of the LoadLibrary for calling process, and DLL_PROCESS_DETACH event is notified only during the last call of the FreeLibrary for the calling process.

In this case it is not a problem if II Engine is loaded by Matlab process using ii_lock, and then loaded again and released in each MEX files. Because all libraries are loaded in the address space of Matlab process, it works with full speed of available communication channel and cpu.

## 4. CLIENT – SERVER CONFIGURATION

For the simple operations, connection between an electronic system and computer (where user operates) can be made using single cable, this configuration is shown on figure 5 in point a). Such a solution is quite common, but there might be some situations where it cannot be used. The simplest example is situation, where equipment is distributed on large area, and user cannot easily work on multiple machines, or when user can not enter the restricted area where the equipment is placed.
For such a situation, remote access over the TCP/IP network is required. The easiest way to implement that access is to create TCP server, witch acts as a client of the system (loads II Engine) on machine which has hardware connected - this solution is shown on figure 5 in point b). On the operator's workstation must be an emulator of II Engine, which is loaded by client applications (i.e. Matlab). Lower layers (II Engine, channels) on the server machine and application layer on workstation does not notify this change, it is transparent for them. The II Engine emulator placed on operator's workstation does not perform any functional tasks other than sending requests to and receiving responses from the TCP server, which will pass requested actions to the real II Engine. This method is flexible and transparent for other parts of the system.

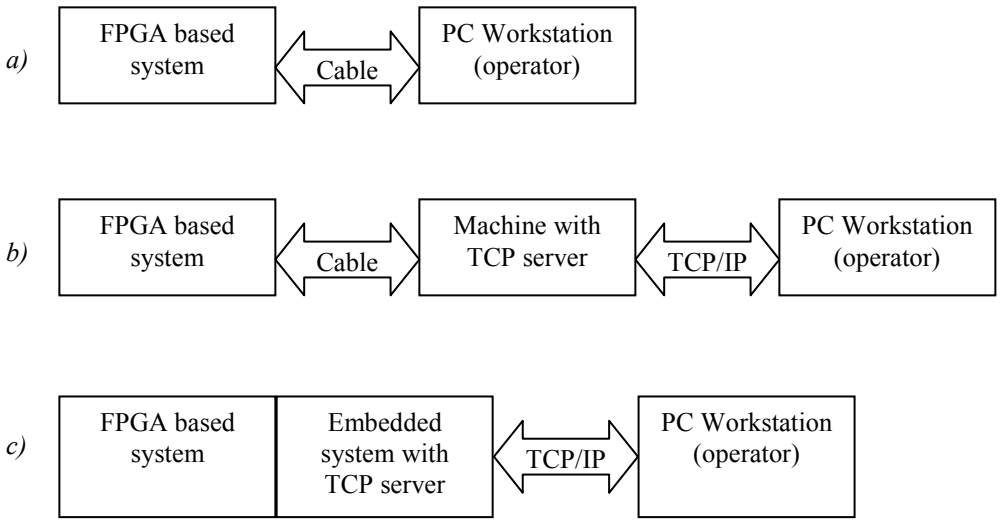


Figure 5. Access to hardware configurations.
a) local; b)remote, using PC server; c) remote, using embedded system

The method of remote access is not optimal, and can be improved. The PC-based server's performance and features (interfaces like soundcards, keyboard, USB, etc) are not used by TCP server application. The only required interfaces are: TCP connection (i.e. Ethernet controller), access to hardware devices and diagnostic channel (console on RS-232 serial line). These features are available commonly on embedded systems, so putting server on the same board with FPGA system reduces requirement for one PC machine, and enables higher performance in communication between FPGA system and embedded computer system – lower capacitance of PCB connections between chips results in faster communication.

## 5. EMBEDDED SYSTEMS

FPGA chip can cooperate with different embedded systems, but there are solutions that combine FPGA programmable logic and embedded processor in one integrated circuit. Altera Company offers the NIOS processor, which can be implemented in Altera's FPGA chips using programmable logic. Functionality of this processor is described by VHDL code, which is generated by SOPC Builder (System On Chip Builder) application. Xilinx Company offers similar solution called Microblaze, which is generated by EDK (Embedded Development Kit) application. Xilinx Company offers also another embedded-in-fpga solution, witch is PowerPC 405 processor available in Virtex II Pro and Virtex IV FX chips. Unlike Microblaze, this processor is not made in programmable logic, but is placed in silicon as a mask-

generated block of Virtex integrated circuit. PowerPC does not use FPGA resources, which all are available for custom logic. For all described examples, it is possible to run operating system, especially dedicated Linux distributions are available.
In both technologies (FPGA defined, and fixed in chip processor), it is necessary to configure FPGA logic, before booting operating system. In first case it obvious, because when FPGA is not configured, processor does not exist. In the second case, the processor exists, but many other peripheral devices are implemented in programmable logic. For example, in Virtex II Pro, core system peripheral devices like SDRAM memory controller, UART, Ethernet controller, interrupt controller, etc. are implemented in FPGA. Because of that, electronic systems must have on-board memory (usually an eeprom), which will configure the FPGA after turning the power on. In laboratory development it is possible and commonly used to configure the FPGA using the JTAG interface. Usually JTAG cable (ByteBlaster, BitBlaster, Parallel Cable III, Parallel Cable IV, etc.) is connected to personal computer's parallel port (LPT). This method of configuring the FPGA is slower than loading configuration from on-board memory, but it does require reprogramming the flash or eeprom memory, which in most cases has limited number of programming.

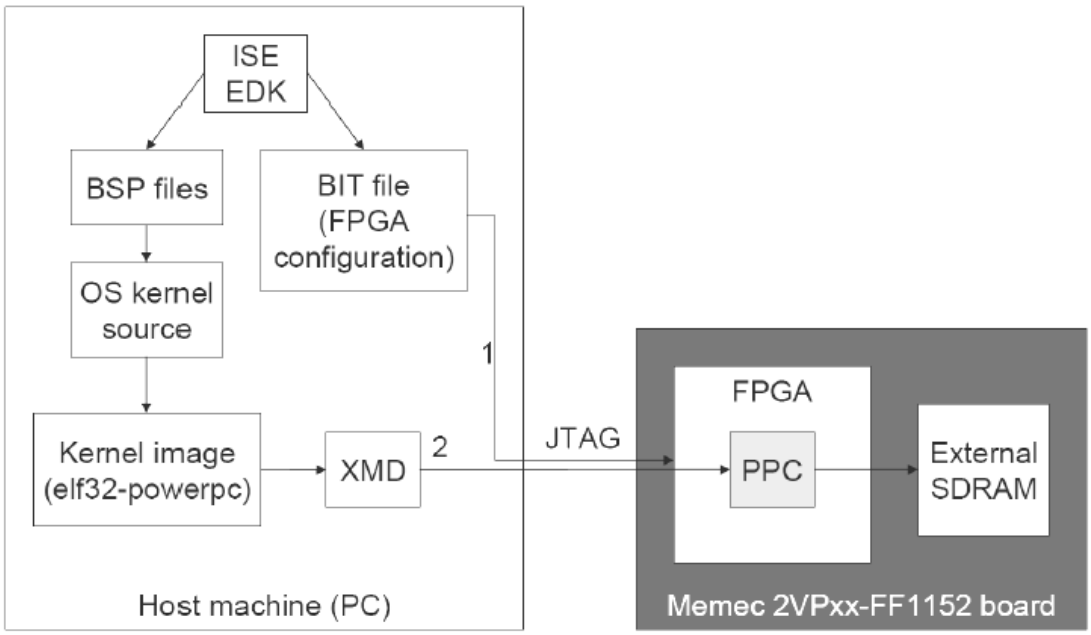


Figure 6. System generation.

After FPGA has been configured, the kernel of the operating system can be booted from the same on board flash or eeprom memory, or from the TCP/IP network using TFTP protocol (if the system has a network interface). To load the kernel from the network, it is required to load the bootloader from on-board memory, which will be able to download kernel image form the server. Bootloader must be able to control the network interface, and must have implemented TFTP protocol.

An example of generating system (peripherals implemented in programmable logic for the embedded processor) is shown on figure 6. This example describes system generation for the Xilinx Virtex II Pro (with embedded PowerPC). The first step is to define the system architecture in the Embedded Development Kit (EDK) – select peripheral devices to implement in FPGA, and set up connections between them. When the system is designed, it is possible to generate FPGA configuration data (*.bit file) at this point. On the other hand, user can generate BSP (board support package) files, which contain C source files with parameters definition, for example addresses of the FPGA implemented peripherals, algorithms for accessing those peripherals etc. These files do not contain whole drivers for Linux, but only core algorithms for Virtex specific hardware. Drivers for this hardware are written by companies which are developing Linux ports for embedded system. BSP files for Linux, generated by the Embedded Development Kit, are stored in directories,

which has the same structure as the part of Linux kernel directory tree. This feature helps transferring those files from Embedded Development Kit to Linux kernel source directory.
Once BSP files have been copied to the Linux kernel source directory, kernel should be configured properly. If kernel configuration has drivers of FPGA implemented device marked to compile in, and this device was not selected in Embedded Development Kit, then compilation will fail, because no information (i.e. base address) about this device is available in BSP (macro definitions required by kernel drivers are not defined).
Once both, Linux kernel and FPGA configuration are compiled, the system can be started up. First, FPGA must be configured with generated BIT file using the JTAG cable. When hardware is configured, kernel can be transferred using the same JTAG cable through the FPGA to the external RAM memory, when this operation is complete, system can be booted up.

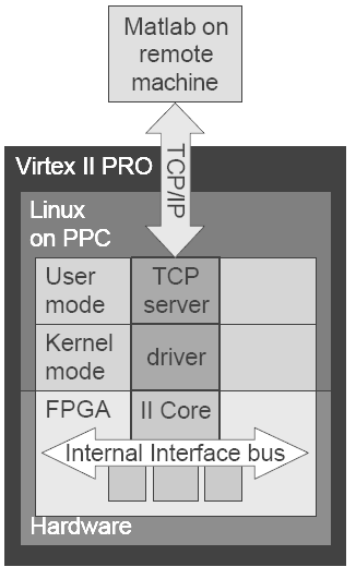


Figure 7. Virtex II Pro as an example of system combined of FPGA programmable logic and embedded PowerPC processor.

For all described above embedded in FPGA processors (NIOS, Microblaze, PowerPC), dedicated Linux distributions are available. This fact gives possibility of building very functional systems, because it is easy to balance functionality between hardware and software. Typical strategy of designing system is to put in to FPGA those parts of algorithms that can be described by VHDL types and constructions. These are usually fixed point (integer) arithmetic calculations, state machines, control algorithms, etc. Next step is to create a VHDL component with functionality for programmable logic, and define interface for connection with the processor. On the software side, a driver that is compatible with the VHDL component interface is required. Linux is an open source operating system, to create driver only a text editor and a compiler is required (no additional development kits or tools are required). All other parts of functionality that is difficult to fit into FPGA, floating point calculations, complex functions like logarithm, must be implement as a software applications in the operating system. These applications are using dedicated driver to access to hardware.
The sample implementation of this concept is shown on figure 7, on the top level Matlab is placed as a client application. It communicates with TCP server which is running on Linux on PowerPC. For accessing hardware, Matlab uses the same set of functions (MEX files) which was used for accessing hardware over the LPT or VME channels. Access to PowerPC was enabled by replacing the channel library, with one which communicates over the TCP/IP. The Matlab scripts did not required any modifications when connecting new hardware.
Linux running on PowerPC has TCP/IP protocols stack implemented, because of that, creating TCP server for embedded application was as simple as for PC machine. To access hardware, server uses kernel mode driver (a character device), though the entry in /dev directory. This driver was written especially for handling communication with FPGA Internal Interface Bus Controller Core.
Whole solution makes development of complex systems easier. To add new device to FPGA, designer needs only to connect this device to Internal Interface Bus, provide the description of address space (IID) for Matlab, and new device is ready to use.

## 6. SUMMARY

FPGA technology has many features, especially flexibility and performance. Usage of software dedicated for programmable logic designs makes FPGA systems more powerful. It makes faster and easier development and easier control. This kind of electronic may be attractive alternative for other kind of systems used for different purposes like Digital Signal Processing or Control. Usage of embedded systems together with FPGA, results in much better functionality. It gives possibility of easy balancing with implementation between hardware and software. The most powerful embedded solutions have embedded processor placed in the one integrated circuit (fixed in silicon, or implemented in the programmable logic). Most popular embedded solutions have Linux distributions available for it, this operating system is open source and does not require any additional development kits, which results in easier driver development.

## REFERENCES


1. Waldemar Koprek, Piotr Pucyk, Tomasz Czarski, Krzysztof T.Pozniak, Ryszard S.Romaniuk *DSP Integrated Parameterized FPGA Based Cavity Simulator & Controller for VUV FEL - SIMCON ver.2.1. Installation and Configuration Procedures; USER'S MANUAL* , TESLA note 2005-06.
2. Wojciech Giergusiewicz, Waldemar Koprek, Wojciech Jalmuzna, Krzysztof T. Pozniak, Ryszard S. Romaniuk, *FPGA Based, DSP Integrated, 8-Channel SIMCON, ver. 3.0. Initial Results for 8-Channel Algorithm* , TESLA note 2005-14.
3. Krzysztof T. Pozniak, Tomasz Czarski, Waldemar Koprek, Ryszard S. Romaniuk , *DSP Integrated, Parameterized, FPGA Based Cavity Simulator & Controller for VUV-FEL SC Cavity SIMCON version 2.1. re. 1, 02.2005 User's Manual*, TESLA note 2005-02.
4. K. T. Pozniak, M. Bartoszek, M. Pietrusinski, *Internal interface for RPC muon trigger electronics at CMS experiment*, Proceedings of SPIE, Bellingham, WA, USA, Vol. 5484, 2004, pp. 269-282
5. Waldemar Koprek, Pawel Kaleta, Jaroslaw Szewinski, Krzysztof T. Pozniak, Tomasz Czarski, Ryszard S. Ronmaniuk Software *Layer for FPGA-Based TESLA Cavity Control System (Part I)*, TESLA note 2004-10.
6. Tesla Linear Accelerator, http://tesla.desy.de/
7. Xilinx Company, http://www.xilinx.com/
8. Altera Company, http://www.altera.com/
9. Mathworks Company http://www.matworks.com/