BEYOND PCS: ACCELERATOR CONTROLS ON PROGRAMMABLE LOGIC

J. Dedič, M. Pleško, K. Žagar,
Cosylab, Ljubljana, Slovenia

Abstract

Based on experience gained on our products (a delay generators producing sub-nanosecond signals and function generators producing arbitrary functions of length in the order of minutes) and on our research projects (a prototype hardware platform for realtime Java, where Java runtime is the operating system and there is no need for Linux), we will speculate about possible future scenarios: A combination of an FPGA processor core and custom logic will provide all control tasks, slow and hard real-time, while keeping our convenient development environment for software such as Eclipse. We will illustrate those claims with designs for tasks such as low-latency PID controllers running at several dozen MHz, sub-nanosecond resolution timing, motion control and a versatile I/O controller - all implemented in real-time Java and on exactly the same hardware - just with different connectors.

INTRODUCTION

When performance is paramount, one of the techniques a skilled electronics engineer would use is *reconfigurable computing* [1]. In terms of performance, this approach is the second best to what is achievable with commercially available electronics, only surpassed by *application specific integrated circuits* (ASIC).

Pure programmable hardware implementations might be costly in terms of development effort, however. Some problems are inherently difficult to solve in hardware using programming languages such as VHDL and Verilog. To this end, FPGA vendors offer generic processor cores. In some cases, the cores are available in form of VHDL code (e.g., Altera NIOS or Xilinx PicoBlaze), whereas in higher-end FPGAs multiple powerful cores are fixed on the chip (e.g., Xilinx Virtex II includes PowerPC cores).

However, development in a C-like language is still cumbersome. Illegal use of memory, buffer overruns, memory leaks, relatively long compilation times, portability issues and structured programming approach contribute to decreased efficiency, which is in some cases even 2 to 10 times smaller than one achievable with higher-level programming languages, such as Java.

REQUIREMENTS

This section lists some of the requirements that a hardware Java platform would have to meet in order to retain high-level of development efficiency.

Standard Java constructs should be retained. I.e., no new keywords should be introduced into the language. This way, existing tools for Java development could be leveraged, such as high-productivity integrated development environments (IDEs, e.g., Eclipse or NetBeans), compilers, byte-code manipulation tools and code verifiers.

1. **Hardware** should be composed of **modules**. Each module would consist of the hardware part (templated VHDL files) and software part (configurable drivers). When a module would be instantiated, the software drivers would be automatically configured for the instantiated hardware (e.g., matching bus addresses, IRQ numbers, etc.).
2. **Static (compile-time) checking** should be possible. For example, it should be impossible to overlap register addresses of modules on a bus. Ideally, the register addresses would be assigned automatically.
3. **Support for debugging**. A debug console should be available through a serial port. In addition, Java virtual machine should support remote debugging using existing tools. JTAG diagnostics of hardware should also be possible.
4. **Field upgrades** of hardware and ROM software should be possible.

One-size-fits-all board: ideally, a general-purpose board design would exist, so that boards would not have to be developed for each application specifically. A modular board composition (IndustryPack, PC/104, VME) is a good approach to achieve this.

EXAMPLE APPLICATIONS

Nanosecond Resolution Timing

In particle accelerator controls, sub-nanosecond resolution timing is sometimes required due to high speed of particles whose orbit needs to be controlled. A particular application called for a controllable delay generator, capable of producing output signals that are delayed relative to a trigger signal for amount of time in the order of a nanosecond. Figure 1 shows an example of a trigger signal and the resulting output signals, which are delayed by t_A and t_B, respectively.

To this end, FPGA with a phase locked loop (PLL) can be employed. The PLL is capable of multiplying the clock frequency by a given factor. Thus, if a 500MHz external clock is used, the PLL can multiply it by 4, achieving a 2GHz clock (0.5 ns temporal resolution). The jitter of this clock is also very small (in the order of 50 ps), which makes FPGA technology a good candidate for this application.

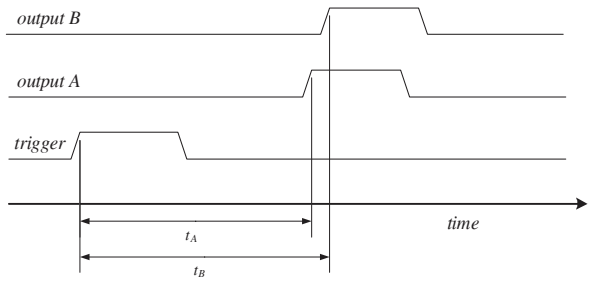

Figure 1: signals of a delay generator.

Since the delays t_A and t_B induced by the FPGA-based delay generator would have to be externally controlled (ideally through a SCADA-like system using a computer network), a pure VHDL solution is no longer a feasible option. Therefore, a co-design approach depicted in Figure 2 is a reasonable alternative. Here, a processor is monitoring communication over Ethernet, implementing TCP/IP or other network stacks required by the SCADA system, implementing their respective protocols. The processor then converts the requests from Ethernet to configuration for the delay generator through a bus internal to the FPGA. The processor, Ethernet MAC layer and delay generator are thus all contained in a single FPGA chip, whose input pin is a trigger, and whose output pins are correspondingly delayed. (Apart from these pins, also pins for reset, clock, ground, power supply, etc., are required).

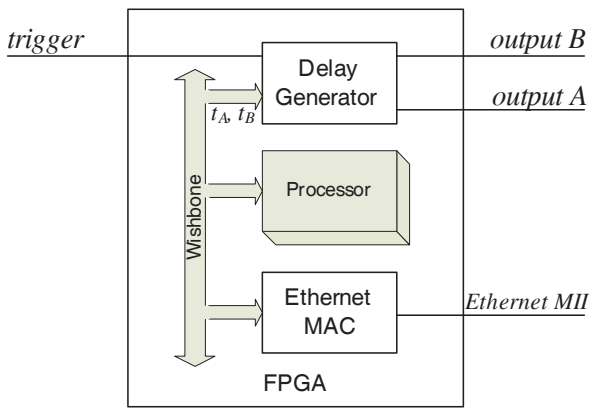

Figure 2: block diagram of a delay generator.

Versatile I/O Controller

In automation applications, integration with many kinds of devices is required. The devices are equipped with various control interfaces, ranging from analog current, via serial interfaces (e.g., RS-232, RS-485, etc) to more sophisticated busses, such as *General Purpose Input/Output Bus* (GPIB).

In some cases, for example control of particle accelerators, the number of devices under control is large, and there are tens of thousands of process variables that need to be controlled or monitored. Consequentially, the density of I/O channels is high, and entire racks are devoted to front-end control equipment (also called *Input/Output Controller*, IOC). This control equipment is responsible for performing simple tasks, such as communicating with devices with their respective protocol, initializing devices, converting the values returned by devices from raw to engineering units, etc. One of the control equipment's most important responsibilities is to make the connected devices available to a SCADA system via a computer network.

Implementing an IOC in programmable hardware might well be a very economic and efficient approach. The inputs and outputs of these controllers are then bound to pins of the FPGA, and from there to the IOC's board, where transceivers implementing the physical layer of communication are placed.

The pin count of FPGAs is fairly large – several hundred pins are available for application-specific purposes (e.g., Altera offers FPGAs from 484 to 1508 pins). One UART serial line requires 4 signals, which means that physically more than 100 serial connections could be handled by a single FPGA.

Since voltage levels of FPGA's pins are not arbitrary, the board would require transceivers to implement the physical layer of the communication stack. Also, some I/O controllers are not easily available or are difficult to implement. One such example is GPIB – in this particular case, integrated circuits are available, which can be integrated with FPGA through a standard bus (e.g., National Instruments' TNT5002, which uses PCI bus).

Well though-out modular design of the board would allow re-use of the same design for various I/O configurations. Such a board would either contain a very large pin bank to which connectors would be attached, or feature an extensible interconnect bus (VME, IndustryPack, etc.).

ACHIEVING HARD REALTIME

In automation**, hard real-time interlocks** are frequently a requirement. When an interlock is triggered, a reaction (e.g., a shutdown or switching-off of an output) must commence immediately. In FPGAs, such interlocks can be implemented directly in hardware. If they are implemented asynchronously, the reaction time is only limited by propagation delays, and doesn't even have to wait till the next period of the system clock.

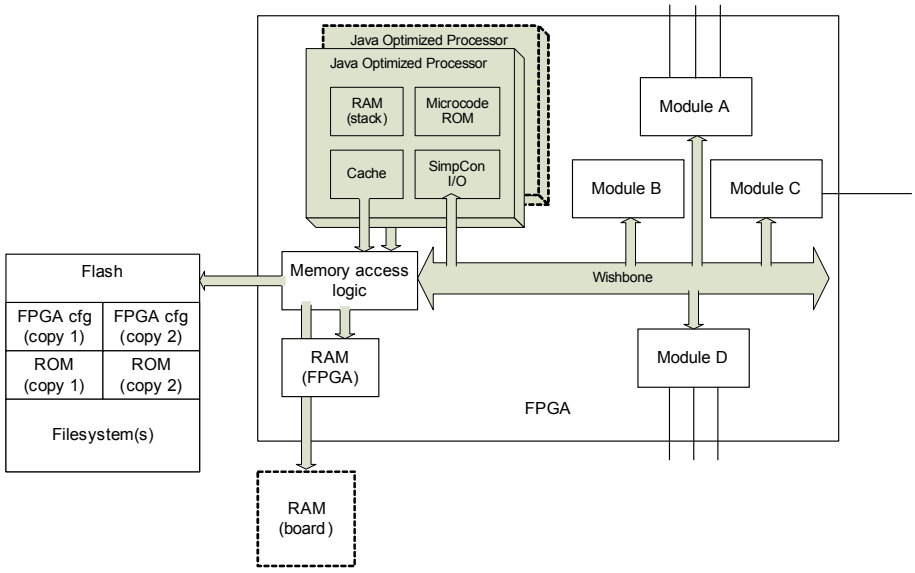

Figure 3: Overview of the hardware architecture..

VHDL code for this circuit (Listing 1) shows a VHDL design pattern where an interlock is implemented without affecting the rest of the logic – thus, the safety aspect of the system can be introduced in a design systematically, without affecting the design of the logic.

```
-- synchronous implementation of the logic
process(clock)
begin
      if rising_edge(clock) then
      begin
              logic <= …;
      end if;
end process;

-- asynchronous handling of an interlock
output <= '0' when interlock='0' else logic;
```

Listing 1: VHDL code of an interlock.

HARDWARE ARCHITECTURE

The architecture follows an established pattern: it features a CPU, memory/storage (on-chip and off-chip), various modules, and a bus that interconnects all of the components together (Figure 3).

For the CPU core, we propose using a standard, possibly open, implementation. Cores implementing the Java virtual machine specification in hardware already exist, for example *Java Optimized Processor* (JOP, [2]).

A good candidate for the bus is Wishbone [3]. Wishbone is a flexible, yet simple bus for interconnection of cores within a programmable chip. Since many hardware components (such as Ethernet MAC implementations) exist that offer a wishbone interface, supporting wishbone would allow leveraging these implementations.

CONCLUSION

In this paper, we have tried to illustrate the advantages and application potential of programmable logic. Since this approach offers a lot of freedom, a well thought-out architecture should be agreed upon to prevent unnecessary divergence of efforts and allow re-use of components and methodologies. Ideally, the approach would also leverage the standard integrated development environments, reducing the learning curve of engineers making use of the technology.

REFERENCES

[1] K. Compton, S. Hauck. “Reconfigurable Computing: A Survey of Systems and Software”, ACM Computing Surveys, Vol. 34, Np. 2, June 2002, pp. 171-210.

[2] Martin Schöberl. “JOP. A Java Optimized Processor for Embedded Real-Time Systems”, PhD thesis, Vienna University of Technology, January 2005, http://jopdesign.com.

[3] opencores.org. “WISHBONE System-on-Chip (SoC) Interconnection Architecture for Portable IP Cores”, revision B.3, September 2002.