WISHBONE Bus Interface for the No-Instruction-Set Computer (NISC)

Roko Grubišić and Vlado Sruk
Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing, University of Zagreb
Unska 3, 10000 Zagreb, Croatia
{roko.grubisic, vlado.sruk}@fer.hr

Abstract - General-purpose processors are often unable to effectively exploit the parallelism inherent to the software code. In such cases, additional hardware accelerators are needed to enable meeting the performance goals. To shorten time to market and enable meeting design constraints, designers today use special tools and technologies like the No-Instruction-Set Computer (NISC) to automatically generate custom accelerators. However, it is often difficult to integrate these accelerators into general-purpose processor systems and use them from the main processor's software. In this paper we present a simple and efficient method for using the NISC processor as a loosely-coupled coprocessor. To enable communication with the NISC processor, a simple set of coprocessor services exposed to the application programmer is defined. A hardware solution based on standard SoC bus architectures for implementing these services is described. Using a standard SoC bus interface enables simple integration of the NISC design flow into an existing design flow. A practical implementation of the NISC coprocessor WISHBONE interface was realized and tested in a system based on a WISHBONE-compatible general-purpose soft processor.

I. INTRODUCTION

Embedded computer systems are no longer used only as simple control devices. Instead, today's embedded systems have to efficiently perform complex tasks and algorithms within increasingly stringent design constraints and shrinking time-to-market. The widespread design approach based on implementing applications in software for general-purpose processors offers high designer productivity. However, this approach offers low design quality in terms of performance and power-efficiency because general-purpose processors are often unable to exploit the parallelism inherent in the software code. One of the solutions for increasing the design quality is migration of computationally-intensive parts of the system's task to hardware accelerators and creating custom Systems-On-Chip (SoCs). Custom hardware accelerators enable utilizing parallelism and provide an effective way to meet design's performance goals for certain classes of applications.

Custom hardware is conventionally designed at the register transfer level (RTL), which entails manually coding the RTL model in a hardware description language (HDL), such as VHDL or Verilog and exhaustive testing. This process is often tedious and error-prone and limits the designer productivity. To overcome this problem, it is necessary to raise the level of abstraction for the hardware design and use special tools to synthesize the actual circuitry. The most interesting design methodologies focus on synthesizing the hardware directly form software code written in a high-level programming language, such as *C*. Such design approaches include many High-Level Synthesis (HLS) tools and, more recently, the No Instruction Set Computer (NISC).

NISC approach [1,2] is based on the idea of elimination of the instruction abstraction and compiling programming language code directly to control words for a custom datapath using a special cycle-accurate compiler [3]. NISC processor's architecture can be manually modeled in the Generic Netlist Representation (GNR) architecture description language [4], automatically generated or selected from a library of standard or previously designed architectures. NISC Toolset [5] generates the RTL model of the processor and control words for the desired application written in *C* to be implemented in the desired technology. Advantage of the NISC approach is utilization of instruction-level parallelism (ILP) which enables speeding up the execution of an algorithm. The generated NISC processor is usually capable of executing several equivalent RISC instructions in a single clock cycle.

However, the NISC Toolset at present doesn't provide any standard facilities for communication with general-purpose processors. This represents a problem when attempting to use the NISC processor as a coprocessor and integrate it into an existing system to help meet desired design constraints. We propose using an interface to a standard bus architecture to enable NISC's simple integration in a wide variety of systems with different processors and peripherals. This approach allows using the NISC processor from the main processor's software through a simple set of coprocessor services. This enables simple migration of performance-critical parts of the application to hardware without the need for significant changes to the rest of the program code. In this way, it is easy to integrate NISC into an existing design flow.

In this paper we present the design of NISC WISHBONE Interface [6,7]. This interface provides communication facilities necessary for using the NISC processor as a loosely-coupled coprocessor in systems based on the WISHBONE standard bus architecture. Using this interface, the NISC processor was successfully connected to a WISHBONE system based on a general-purpose RISC soft processor using a commercial EDA tool. The example system was implemented and tested on two different FPGA boards. In this way, we have shown that the NISC processor can be connected to a standard bus architecture using a commercial EDA tool and used as a coprocessor from general-purpose processor's software.

II. STANDARD SOC BUS ARCHITECTURES

One of the major design challenges for any SoC design is system integration. Increased number of components used in such systems requires simple and efficient connection mechanism and generic interfaces to decrease design effort and promote reuse. Standard bus architectures simplify the process of system integration, shorten time to market and support portability and reuse of IP cores.

Renowned SoC bus architectures of the day include ARM Advanced Microcontroller Bus Architecture (AMBA), IBM CoreConnect, WISHBONE and Altera Avalon. The use of standard bus architectures also provides the opportunities for using special tools that automatically generate bus interfaces and interconnections, in this way further shortening time to market and reducing the possibility of human error.

These standard SoC bus architectures all share some common traits. Unlike traditional backplane busses, such as VME or PCI, SoC busses are usually defined as synchronous logical busses. Logical bus means that the functionality of a bus is defined only at the logical level and not at the electrical level or mechanical level. The logical specification defines logical signals, their meanings, active levels, the relationship between different signals and their relationship to the clock signal. This makes the bus interface independent on the way the hardware is described (e.g. RTL, schematic or a gate-level netlist) and on the implementation tools and technologies (e.g. FPGA or ASIC).

Standard SoC bus architectures define synchronous busses, i.e. they must adhere to the synchronous design discipline which states that the system changes state only on the active edge of the clock signal. Synchronous design principle enables simple and efficient design of digital circuits and increases the reliability of the circuit by eliminating race hazards. This is why synchronous design is the most widely used digital design methodology today and the methodology of choice for SoC busses.

Another common trait in standard SoC bus architectures is the master-slave architecture. IP cores connected to the bus are distinctly divided into master and slave modules. Master modules play the active role on the bus – they can initiate the transfer of data to and from slave modules, while slave modules only respond to requests. The distinctions between master and slave modules and master and slave bus interfaces enable a flexible base for the design of different types of systems.

Standard SoC bus architectures differ in the details of bus interfaces, supported data transfer protocols (single transfers, burst transfers, read-modify-write), supported bus interconnection topologies (e.g. point-to-point, multiplexed shared bus, crossbar switch) and whether they define a single bus or several different busses as a part of the specification. For example, AMBA and CoreConnect specifications provide a complex high-performance bus for processors and other fast modules and a simpler peripheral bus for devices like the UART or GPIO. On the other hand, WISHBONE defines only a single universal bus. High-speed busses like WISHBONE, AMBA Advanced High-Performance Bus (AHB) and CoreConnect Processor Local Bus (PLB) also support bus arbitration and multiple master modules.

The WISHBONE System-on-chip (SoC) architecture for portable IP cores [8] is a flexible design methodology developed by Silicore Corp. targeted at SoC integration and design reuse. This is accomplished by defining a standard interconnection scheme and data exchange protocols. WISHBONE specification defines a single, simple, logical, fully synchronous MASTER/SLAVE bus and IP core interfaces that require very few logic gates. It supports different technology-independent interconnection topologies ranging from simple point-to-point and shared bus interconnections to data flow interconnections and complex switch fabrics. It also supports a full range of standard data transfer protocols including SINGLE READ/WRITE cycles, BLOCK READ/WRITE cycles and read-modify-write (RMW) cycles with various data sizes and byte ordering. A handshake mechanism enables flow control and communication between different-speed cores.

Presented standard bus architectures provide a simple and efficient way of connecting coprocessors to general-purpose processor systems. This is also the approach we propose for the NISC processor to overcome integration challenges and enable coprocessor communication. WISHBONE bus architecture was chosen for the implementation of the NISC coprocessor interface mainly because of its flexibility and the fact it imposes no licensing and application restrictions. WISHBONE specification is currently maintained by OpenCores.org and today it represents a *de facto* standard for open-source hardware. It is also offers CAD tool support and a large library of free processors and other IP cores. Besides free processor IP, many popular commercial soft processors such as Xilinx MicroBlaze and Altera Nios have a WISHBONE-compatible variant available.

III. NISC WISHBONE INTERFACE

To be able to use the automatically generated NISC processor as a coprocessor we must provide a way to connect it to other IP cores in the system and a way for the main processor's software to access its coprocessor functionality. Since the NISC processor itself isn't compatible with any of the standard bus architecture specifications, we had to define necessary communication services and design special interface IP to enable connecting NISC to the chosen bus architecture - WISHBONE. Our approach enables mapping the NISC processor into the main processor's memory space and allows it to be controlled by the main processor's software.

The designed NISC WISHBONE Interface IP is targeted at fully synchronous WISHBONE systems, with the NISC coprocessor sharing a common clock with the main processor and the WISHBONE bus. Both the NISC Basic WISHBONE Interface and the NISC Data Memory WISHBONE Interface are SLAVE interfaces and support SINGLE READ/WRITE WISHBONE classic bus cycles with 32-bit data size and 32-bit data granularity. This enables connecting the NISC processor to a wide variety of WISHBONE-compatible 32-bit general-purpose soft processor cores. The designed interface is compatible with the existing NISC design flow and requires no changes in the NISC Toolset.

A. Interfacing overview

The interface between a main processor and a coprocessor must enable basic data and control flow transfer services. This interface is exposed to the application programmer via a memory-mapped programming model or a coprocessor API which must efficiently hide all the implementation details of the underlying hardware. And it must also enable simple migration of software functions to hardware. We have identified and implemented the following essential set of services:

1) Send the data to be processed to the coprocessor
2) Initiate the data processing
3) Detect processing completion
4) Get the results from the coprocessor

To implement these services, we have designed a communication scheme which takes advantage of the existing features of the NISC processor – the available external reset and halt signals and the possibility to use an arbitrary number of external input and output ports from NISC's application software using the prebinding mechanism [9]. Our goal was to eliminate the need for any modifications to the NISC Toolset and still make system integration as simple as possible.

Our complete interfacing solution integrated into NISC architecture with two input ports and one output port is shown in Fig 1. The parts of the NISC processor – datapath, controller and control and data memories are shown in the figure, together with the extensions we designed and implemented (highlighted in gray). The NISC Basic WISHBONE Interface enables control functions and enables transfer of small amounts of special-purpose data directly to and from the NISC processor's datapath. The NISC Data Memory WISHBONE Interface and the Data Memory Multiplexer enable transfer of large amounts of general-purpose data directly to and from the NISC processor's data memory.

These extensions, together with the software support from NISC's application enable the communication facilities necessary to provide the desired coprocessor services. We use NISC's reset signal to start data processing, halt signal for completion detection, the external I/O ports for transferring smaller amounts of special-purpose data and our Data Memory Multiplexer for transferring large amounts of data. Our communication scheme is designed to work in the following fashion:

1) Stop the NISC processor: The main processor asserts the NISC processor's reset signal to insure it is has exclusive access to NISC's I/O registers and data memory.

2) Send data to the NISC processor: The main processor sends the data to be processed to the NISC processor. Special-purpose data (special parameter variables, often-used constants and other function arguments normally passed by value) is written to shared registers using the NISC Basic WISHBONE Interface. These registers can be accessed through NISC's external input ports and used directly in the NISC processor's datapath. General-purpose data (arrays, structures and other function arguments normally passed by reference) are transferred directly to the NISC processor's data memory using the NISC Data Memory WISHBONE Interface and the Data Memory Multiplexer.

3) Start the NISC processor: Initiate data processing by de-asserting the NISC processor's reset signal.

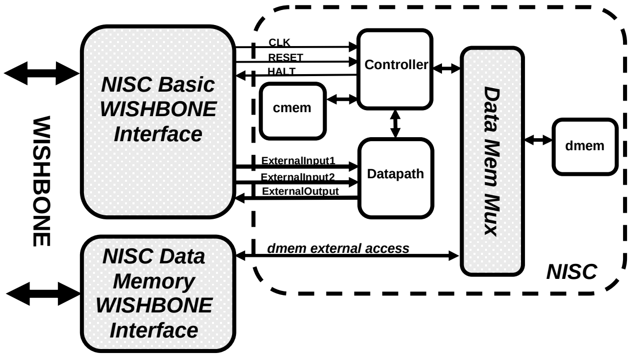

Fig. 1. The NISC WISHBONE Interface architecture

4) Detect processing completion: When the NISC processor finishes processing the data, it asserts the halt signal. The main processor can either detect this by polling the status of the halt signal using the NISC Basic WISHBONE Interface or by setting the interface to generate an interrupt when NISC sets its halt signal.

5) Get the results from the NISC processor: After the main processor has detected that the NISC processor has finished execution, it can transfer the results to its local memory for further processing. A smaller number of results (such as the return value in *C* programming language functions) can be stored to output registers by the NISC processor and then transferred to the main processor using the NISC Basic WISHBONE Interface. Larger amounts of data (results normally returned by reference) can be transferred directly from NISC's data memory using NISC Data Memory WISHBONE Interface and the Data Memory Multiplexer.

These five simple steps of our communication scheme translate almost perfectly to the four services we identified as necessary for using the NISC processor as a coprocessor. The only difference is the additional first step which is necessary to put the NISC processor into the reset state and enable other steps.

B. NISC Basic WISHBONE Interface

The NISC Basic WISHBONE Interface enables coprocessor control, i.e. starting or stopping data processing and detecting when the data processing is completed. It also enables transferring smaller amounts of special purpose data directly into the NISC processor's datapath. This is the main part of the NISC WISHBONE Interface and it is necessary for the communication since it enables coprocessor control and interrupt capabilities. It is designed for seamless integration with the RTL model generated by the NISC Toolset.

NISC Basic WISHBONE Interface implements fast single-cycle data transfers to and from shared registers. To enable the transfer of the data and control flow, the NISC Basic WISHBONE Interface's provides shared registers for input arguments and special control registers. Control registers are used to set the state of the NISC processor's reset signal, read the state of the halt signal and enable interrupts. Result registers are implemented using NISC external output ports which provide an internal register. This means that these registers are a part of the NISC processor and not the interface. So, the result output ports can be simply routed to the interface's output multiplexer to enable the main processor to read the results. On the other hand, external input ports work as proxies for external registers and that's why the actual registers are provided as a part of the interface.

In a typical WISHBONE system, the most significant bytes of the address are decoded in the interconnect module (the so called WISHBONE Intercon), which enables the desired peripheral using the active cycle (CYC_I) and strobe (STB_I) signals, and only the required number of address bits is forwarded to the peripheral's ADDR_I inputs. The peripheral's internal address decoding then determines which of its internal memory locations is addressed, and this is also the way the NISC Basic WISHBONE Interface addresses shared registers.

Fig. 2. shows the simplified internal organization of the NISC Basic WISHBONE Interface. The figure shows all the internal registers, output multiplexer and the logic for generating interrupts and the WISHBONE acknowledge signal. Addressing logic is not shown on the figure.

Reading the data from the interface is implemented using the output multiplexer which is controlled by the WISHBONE address lines. The write operation is implemented using register enable signals controlled by the address decode logic and the write enable qualifier signal WE_I. The WISHBONE clock signal CLK_I is routed to the NISC processor and the interface to create a fully synchronous system. The WISHBONE handshake mechanism is implemented using only combinatorial logic, as there are no slow modules in the design. I.e. none of the modules requires more than one cycle for read or write operations so no memory elements for delay are required. WISHBONE reset signal RST_I resets the NISC processor and the interface's internal registers.

Interrupt mechanism is implemented in such way that setting the interrupt signal INT_O only occurs when both the interrupt enable flag (the INT_EN register) and the halt signal are set. INT_O is not a standard WISHBONE signal, but it can easily be connected directly to the main processor and several commercial tools (e.g. Altium Designer) even provide interrupt connections as a part of the WISHBONE interconnect module.

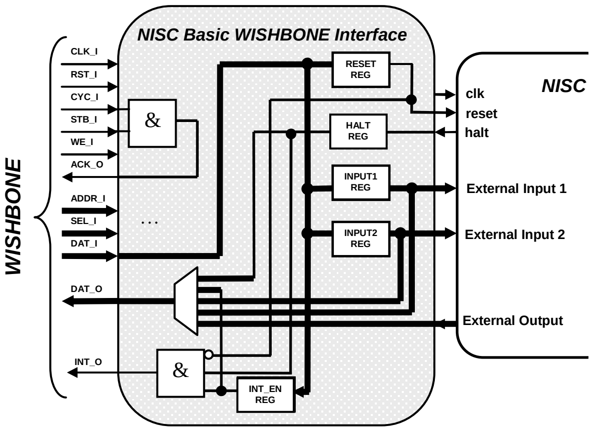

Fig 2. NISC Basic WISHBONE Interface organization

C. NISC Data Memory WISHBONE Interface and the Data Memory Multiplexer

NISC Basic WISHBONE Interface provides capabilities for transferring small amounts of arguments and results using shared registers. Transferring large amounts of data using this approach would require using a large number of registers, large decoders and multiplexers. This could result in lower performance, especially when implementing the design in the FPGA technology. The NISC Data Memory WISHBONE Interface was designed to overcome this by enabling transfer of data directly to and from the data memory of the NISC processor. In this way, arrays and structures can be transferred to the NISC processor and accessed as local variables from the NISC's application.

The role of the NISC Data Memory WISHBONE Interface is essentially converting WISHBONE read and write cycles to the protocol used by NISC's data memory controller. An additional IP, the Data Memory Multiplexer is used to multiplex and arbitrate the access to the data memory controller so it can be accessed by the NISC processor itself and also over the WISHBONE bus using external access ports (see Fig. 1.).

Fig. 3. shows the internal organization of the NISC Data Memory WISHBONE Interface. The address bits are forwarded directly from the WISHBONE address lines to data memory's address lines. Since NISC's memory controller doesn't require word access addresses aligned on 4 byte borders and requires the data with width less than 32-bits to be aligned to the lower data bits (unlike WISHBONE mechanism with byte-enables), special alignment logic was designed to handle the translation. Also, depending on the access type, WISHBONE byte enable lines are translated to appropriate type codes for the NISC memory controller. Write and read enable signals are derived from the WISHBONE write enable qualifier WE_I. The handshaking mechanism is implemented with one cycle acknowledge delay to allow for the Block RAM (BRAM) latency, since NISC uses synchronous BRAMs to implement data memory.

The Data Memory Multiplexer's arbitration scheme is quite simple: if the NISC processor's reset or halt signals are active, then the main processor has control over the data memory. Otherwise, the control is left to the NISC processor, the main processor has no means of writing to the data memory and reads the value of all data bits as zeroes. The implementation of the data memory multiplexer is shown in Fig. 4.

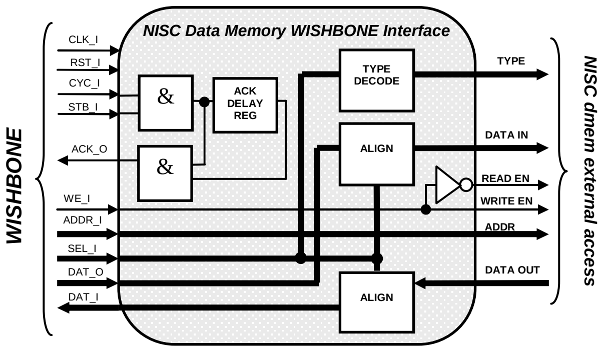

Fig 3. NISC Data Memory WISHBONE Interface organization

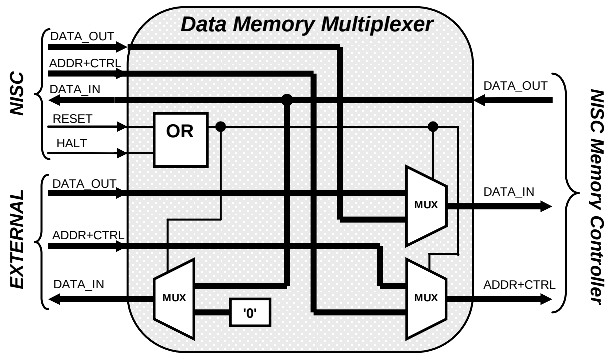

Fig 4. Data Memory Multiplexer organization

D. Programming model

Using the NISC coprocessor attached to the WISHBONE bus from the main processor’s software using the NISC WISHBONE Interface is very straightforward. It is used according to the five simple steps and the four essential coprocessor services presented earlier in this chapter. First, the value ‘1’ is written to the reset register to put the NISC processor in reset state and to insure that only the main processor has access to the shared registers. Then, the arguments are written to appropriate registers or transferred to the data memory. This is followed by writing the value ‘0’ to the reset register to start the NISC’s application execution. After that, the main processor can poll the value of the halt signal until it becomes active or wait on the coprocessor interrupt and then retrieve the results by reading appropriate registers or data memory locations.

The full memory map of the NISC Basic WISHBONE Interface is shown on Table 1. The first memory location, at the base address, is reserved for the control register, i.e. reset and halt registers. One register is accessed when writing and the other when reading that memory address. Because 32-bit addressing is used and registers are aligned on 32-bit boundaries, the next register’s address is displaced by 4 from the base address. This is the location of the interrupt enable register, INT_EN. After that follow the result registers and registers for arguments. The NISC Data Memory WISHBONE Interface maps the whole data memory of the NISC processor to the main processor’s memory space, starting from a separate base address.

The example register address layout shown in Table 1 enables simple implementations of traditional *C* functions with arbitrary number of arguments and one return value. Of course, for different applications with additional result values and matching result registers, different address layouts are possible with differently laid-out interface datapath and NISC I/O ports.

E. Design flow

The main purpose of the NISC WISHBONE Interface is to enable improving the system’s performance using the NISC processor if the purely software implementation on a general-purpose processor doesn’t meet the constraints.

TABLE 1
NISC BASIC WISHBONE INTERFACE MEMORY MAP

Addr	Name	Width [bits]	Description
00	CTRL RESET HALT !(INT_FLAG)	1	Bit0: *read*: status of the halt signal is read *write*: reset register is written to – '1' sets the NISC to reset state, '0' initializes the execution ('0' also serves as the interrupt acknowledge)
04	INT_EN	1	Bit0: *reads/writes* the interrupt enable flag – ‘1' enables the Interrupts, '0' disables them
08	RESULT	32	*Read/Write* – result
12	ARG 1	32	*Read/Write* – 1st argument
16	ARG 2	32	*Read/Write* – 2nd argument
...	ARG *N*	32	*Read/Write* – other arguments

Since designers tend to use familiar platforms and reuse software, NISC WISHBONE Interface was designed in such a way to enable easy integration in an existing design flow. In cases when the general-purpose processor’s software fails to meet design constraints using the existing design flow, designers first need to profile the code to identify the performance-critical functions. These functions are candidates for implementation in hardware as NISC processors using the NISC design flow.

During NISC design, GNR ADL is used to add I/O ports for the function’s arguments and results that are to be communicated through shared registers and connect them to appropriate datapath locations. Desired function’s code is modified to read the arguments and write the results using the prebinding mechanism. Static variables that will be accessed by the main processor through the NISC Data Memory WISHBONE Interface are provided for other results and arguments, e.g. arrays and structures. Then the NISC toolset is used to implement the actual hardware. The NISC WISHBONE Interface is parameterized to fit the application, i.e. appropriate number of argument and result registers is added. The resulting accelerators are connected to the WISHBONE bus using the NISC WISHBONE Interfaces.

Bodies of performance-critical function can then be replaced with coprocessor calls. Shared registers and shared variables in NISC’s data memory are used to transfer the arguments. NISC Toolset’s outputs identify the addresses of the shared variables (offsets from the NISC Data Memory Interface’s base address) so they can be accessed via pointers. Starting coprocessor execution and detecting completion is implemented using the NISC Basic WISHBONE Interface, as already described. This approach uses polling to detect completion and so the calls to these functions are blocking. After NISC finishes execution, results are retrieved from the appropriate shared registers and shared memory locations.

When the body of a function is replaced with blocking coprocessor call, there is no need to change any functions calling this function. This enables simple migration of software functions to hardware and exploiting available instruction-level parallelism. More complex communication schemes using non-blocking calls and interrupts could also be used to exploit task-level parallelism and achieve further speed-up.

IV. Experimental Results

The complete NISC WISHBONE Interface was implemented using VHDL, integrated with a generated NISC processor and wrapped in a WISHBONE compatible module to be used as a coprocessor in systems based on the WISHBONE bus architecture. The generated NISC processor was based on a generic NISC architecture with necessary I/O datapath extensions and simple applications designed to test the communication capabilities.

As a proof-of-concept for the presented coprocessor-based approach, we implemented and tested a WISHBONE system based on Altium’s TSK3000A general-purpose 32-bit RISC soft processor. TSK3000A is based on the MIPS/DLX instruction set and is FPGA vendor independent. Altium’s configurable Wishbone Interconnect module designed for integration with TSK3000A was the bus module used in this proof-of-concept system.

Communication routines for the main processor were developed and then tested in conjunction with the hardware using the Altium LiveDesign Evaluation Board with a Spartan3 FPGA and Xilinx ML506 with a Virtex-5 FPGA. The schematic for the complete system is shown in Fig. 5.

V. Conclusion

In this paper we presented a method for using the No-Instruction-Set Computer (NISC) as a loosely-coupled coprocessor in SoCs based on the WISHBONE open bus specification. This approach allows designers to easily use the NISC technology in general-purpose processor systems with very few modifications to the existing software code. Coprocessor communication services necessary to enable using NISC as a coprocessor were defined. Hardware and software extensions needed to implement these services were explained and the implementation of the NISC WISHBONE Interface was described. The proposed approach is compatible with existing design flows and enables quick and easy migration of software functions to NISC hardware for application acceleration. An example WISHBONE system designed using this approach was presented as a proof-of-concept.

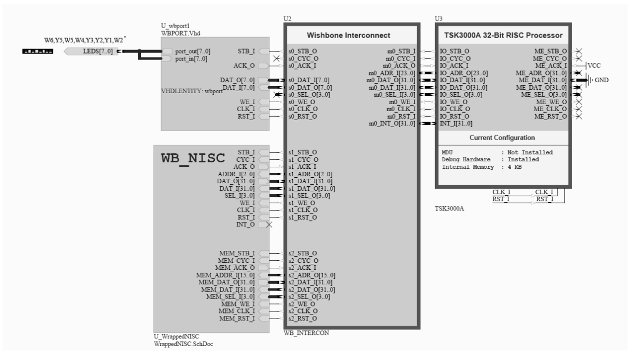

Fig. 5. TSK3000A with a NISC coprocessor

Acknowledgments

The work presented in this paper was done as a part of the project *Application-oriented Embedded System Technology* supported by the Unity through Knowledge Fund (UKF). This work builds on many years of NISC processor research at the Center for Embedded Computer Systems (CECS), University of California at Irvine. The authors wish to thank CECS for providing the NISC Toolset and Mehrdad Reshadi for his help and advice on how to use the toolset.

References

[1] D. Gajski, "NISC: The Ultimate Reconfigurable Component", Center for Embedded Computer Systems, TR 03-28, October 2003.

[2] M. Reshadi: „NISC Modeling and Compilation“, dissertation, University of California, Irvine, 2007.

[3] M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths", *International Symposium on Hardware/Software Codesign and System Synthesis (CODES+ISSS)*, pp 21-26, September 2005.

[4] B. Gorjiara, M. Reshadi, D. Gajski, "Generic Architecture Description for Retargetable Compilation and Synthesis of Application-Specific Pipelined IPs", *International Conference on Computer Design (ICCD)*, October 2006.

[5] NISC Technology & Toolset
URL: http://www.ics.uci.edu/~nisc/, availability verified in January 2009.

[6] R. Grubišić, „Design of the NISC processor WISHBONE Interface“, diploma thesis (in Croatian), University of Zagreb, Faculty of Electrical Engineering and Computing (FER), May 2008.

[7] R. Grubišić, V. Sruk, „The Use of the No-Instruction-Set Computer (NISC) for Acceleration in WISHBONE-Based Systems”, Technical Report, Department of Electronics, Microelectronics, Computer and Intelligent Systems; Faculty of Electrical Engineering and Computing, November 2008.

[8] „Specification for the: WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores“, Revision: B.3, Silicore/OpenCores.org, 2002.

[9] M. Reshadi, D. Gajski, "Interrupt and Low-level Programming Support for Expanding the Application Domain of Statically-scheduled Horizontally-microcoded Architectures in Embedded Systems", *Design Automation and Test in Europe (DATE)*, April 2007.