The Use of the No-Instruction-Set Computer (NISC) for Acceleration in WISHBONE-Based Systems

Roko Grubišić, Vlado Sruk

Technical Report 11-14-2008

Department of Electronics, Microelectronics, Computer and Intelligent Systems
Faculty of Electrical Engineering and Computing, Zagreb, Croatia
{roko.grubisic, vlado.sruk}@fer.hr

Abstract: *General-purpose processors are often unable to exploit the parallelism inherent to the software code. This is why additional hardware accelerators are needed to enable meeting the performance goals. NISC (No-Instruction-Set Computer) is a new approach to hardware-software co-design based on automatic generation of special-purpose processors. It was designed to be self-sufficient and it eliminates the need for other processors in the system. This work describes a method for expanding the application domain of the NISC processor to general-purpose processor systems with large amounts of processor-specific legacy code. This coprocessor-based approach allows application acceleration by utilizing both instruction-level and task-level parallelism by migrating performance-critical parts of an application to hardware without the need for changing the rest of the program code. For demonstration of this concept, a NISC coprocessor WISHBONE interface was designed. It was implemented and tested in a WISHBONE system based on Altium's TSK3000A general-purpose RISC soft processor and an analytical model was proposed to provide the means to evaluate its efficiency in arbitrary systems.*

Keywords: NISC, WISHBONE, coprocessor, bus, interface

1 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 despite increasingly stringent design constraints and shrinking time-to-market. Traditional software-centered design approach offers high designer productivity, but low design quality in terms of performance and logic utilization. The key for increasing the design quality is the migration of computationally-intensive parts of the system's task to hardware. Since general-purpose processors are often unable to exploit the parallelism inherent in the software code, these hardware accelerators provide an effective way to meet the design's performance goals for certain classes of applications.

Hardware acceleration can be accomplished by adding application-specific coprocessors to the system or by customizing existing processors. 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. 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. Such design approaches include High-level synthesis (HLS), Application-specific instruction set processors (ASIPs) and the No Instruction Set Computer (NISC).

HLS and ASIP techniques are commonly used to design custom hardware, but their effectiveness and scope of use are limited. HLS usually supports only a subset of a high-level programming language and is only effective for small-size problems. Also, synthesis results tend to be unpredictable and so the designer can only try to meet the design constraints by trial-and-error methods. AISPs, on the other hand, rely on a limited number of custom instructions to increase the system's performance. The number of custom instructions is limited by the instruction decoder and some ASIPs even limit this number to a single custom instruction. Design of special datapath extensions is also required and it faces the same productivity vs. performance problems as designing any custom hardware.

NISC [1][2] addresses these limitations by eliminating the instruction abstraction and compiling programming language code directly to datapath control words using a special cycle-accurate compiler [3]. In this way, this approach keeps the best of both the world of general purpose processors and the world of custom hardware design. NISC processor's architecture can be manually modeled in an Architecture Description Language (ADL) called Generic Netlist Representation (GNR) [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 to be implemented in the desired technology, whether FPGA or ASIC.

NISC approach utilizes instruction-level parallelism (ILP) to speed up the execution of an algorithm and the generated processor is capable of executing several equivalent RISC instructions in a single clock

cycle. The additional prebinding mechanism enables simple I/O and interrupts [6] and in this way the scope of applications for which a NISC processor can be used is extended to control tasks of a classical microcontroller. The communication interface for the NISC processor [7] enables the use of multiple NISC processors in the system. It enables exploiting the system's available task-level parallelism by parallel execution of the system's tasks on different NISC processors which further accelerates the application.

NISC approach was designed to be self-sufficient and it eliminates the need for other processors in the system. However, that represents a problem when attempting to integrate a NISC processor into an existing system. In the case when a large base of non-portable legacy code optimized for a particular general purpose processor exists, we suggest a coprocessor-based approach to take advantage of the NISC technology and accelerate desired applications. This approach allows utilizing both instruction-level and task-level parallelism by migrating performance-critical parts of the application to hardware without the need for significant changes to the rest of the program code.

One of the major design challenges of the coprocessor approach is the integration of the NISC processor with other IP cores in the system. When designing SoCs, standard bus architectures are usually used, but the NISC processor itself isn't compatible with any of the specifications. 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. A standard bus interface for the NISC processor enables its simple integration in a wide variety of systems with different processors and peripherals.

In this paper we present the WISHBONE bus interface for the NISC processor [8] and propose the use of the NISC processor as a loosely-coupled application-specific coprocessor in WISHBONE-based systems. The NISC WISHBONE Interface was designed, implemented and tested in a WISHBONE system based on Altium's TSK3000A general-purpose RISC soft processor using an Altium LiveDesign Evaluation Board with a Spartan3 FPGA and a Xilinx ML506 board with a Virtex-5 FPGA.

2 WISHBONE bus architecture

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.

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 (e.g. Xilinx MicroBlaze and Altera Nios) have a WISHBONE-compatible variant available.

3 NISC WISHBONE Interface

In this section we introduce our approach to enabling the use of the NISC processor in a WISHBONE-based system. To make the connection of the NISC processor to the WISHBONE bus possible, we designed and implemented a NISC WISHBONE Interface IP. This IP enables the connection of the NISC processor to the WISHBONE bus, but for the NISC processor to be used as a coprocessor which performs useful tasks in the system, some changes in the NISC processor's architecture are required. We describe these architectural additions together with the software necessary to enable the communication from both the side of the NISC processor and the side of the main processor. We also propose an appropriate communication scheme, designed to ease the migration of application's functions between hardware and software and thus ease the process of design space exploration. The NISC WISHBONE Interface is compatible with the existing NISC design flow and requires no changes in the NISC Toolset.

The NISC WISHBONE Interface IP and the software communication scheme enable control and data transfer which are necessary for the use of the NISC processor as a coprocessor in WISBONE-based systems. The main processor can transfer the data to be processed (the function's arguments) to the coprocessor, start the execution of the coprocessor function, detect when the data processing is completed (either by pooling the coprocessor's status or by means of an interrupt) and retrieve the results.

The designed NISC WISHBONE Interface is targeted at fully synchronous WISHBONE systems and supports word size (32-bit), word granularity SINGLE READ/WRITE WISHBONE classic bus cycles. We describe two different methods for exchanging the data between the general-purpose processor and the NISC coprocessor. One method is suitable for simple general-purpose data transfer and the other for custom special-purpose data transfers tailored for a specific application. A design approach with two distinct parts of the WISHBONE interface was followed to enable this kind of flexibility. The first part is NISC Basic WISHBONE Interface which is responsible for control functions and special-purpose data transfers. The other part of the interface is the NISC Data Memory WISHBONE Interface and this one is responsible for general purpose data transfers, i.e. transferring large amounts of data directly to and from the data memory of the NISC processor.

3.1 NISC Basic WISHBONE Interface

The NISC Basic WISHBONE Interface enables coprocessor control control, i.e. starting or stopping data processing and detecting when the data processing is completed, and 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.

The advantage of using the NISC Basic WISHBONE Interface for data transfers is fast communication (one-cycle transfers) and the ability to bring the data directly to the input of the desired functional unit, in this way speeding up the execution of the algorithm. The disadvantage of this part of the interface is that it is only suitable for transferring small amounts of data, since this approach doesn't scale well beyond a few arguments and results.

3.1.1 NISC-side communication requirements

The accessible external signals of the NISC processor's RTL model are the clock signal, the reset signal, the halt signal and the configurable I/O ports. The NISC Basic WISHBONE Interface uses only these external signals to connect the NISC processor to the WISHBONE bus. Data transfer from the main processor to the NISC processor is achieved using input ports which can be read by the NISC's application using the prebinding mechanism. Starting the NISC's application execution is accomplished by resetting the NISC processor, i.e. by generating a falling edge on the reset pin. Completion detection is achieved by reading the halt signal which NISC automatically sets upon completing its task. The results are retrieved using output ports, also using the prebinding mechanism. After the NISC coprocessors completes, it writes the results to its output ports' registers to be read by the main processor using the WISHBONE interface. The output ports have an internal register inside the NISC processor's datapath, but because the input ports function only as proxies for external registers, it was necessary to provide additional registers as a part of the WISHBONE interface. These input and output registers represent the shared resources that enable the transfer of data between the main processor and the NISC coprocessor.

The designer's task is to add the external I/O ports to the desired datapath units as a part of the process of designing the architecture using the GNR ADL. A simple architecture with two input ports for the arguments and one output port for the result shown on Figure 1. This method of communication is convenient when the application requires intensive calculations with some of the arguments or when only simple argument exchange is required. In this case arguments can be brought directly to the input of the desired functional unit and selected when needed, using input multiplexers. This eliminates the need for the sequential transfer of the arguments to the register file for later use and thus eliminates unnecessary transfer of the data. One sequential transfer is already done when the main processor transfers the data to the interface's registers, so another transfer from the interface's registers to the NISC processor's register file or internal registers would double the communication expenses and thus limit the efficiency of the coprocessor. This approach also avoids register spilling for often-used values, which is especially important in area-limited systems with small register files. Output registers for the

results are used in a similar manner and can also be connected to the datapath in such a way to minimize communication expenses.

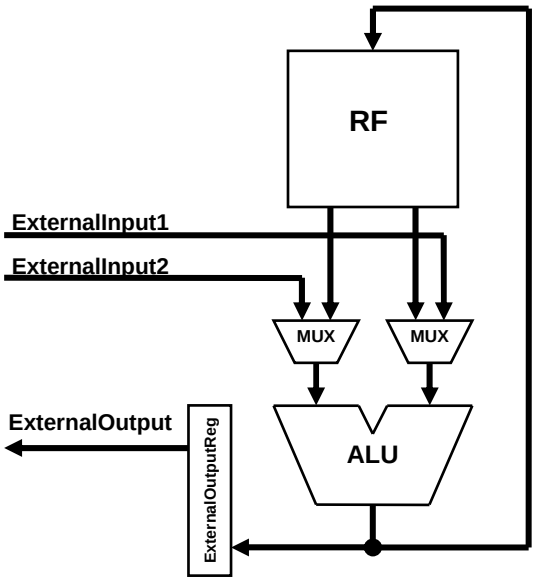

Figure 1: Simple architecture with argument inputs and a result output

Datapath connections for external inputs and external outputs enable NISC's hardware to access shared registers for communication, while the NISC Basic WISHBONE Interface enables the same access for the main processor. To enable the NISC's application to actually read these shared resources or write them, appropriate prebound functions for reading or writing shared registers are used. Listing 1 illustrates this concept for a simple application with two arguments and one result. First, the arguments are read from the external input ports into temporary variables and then they are used in a function to compute the result. The result is then written to the external output register to be read by the main processor when the application terminates and the halt signal is asserted.

```
void NiscMain(){
   //...

   a = ExternalInput1_read();  //get the first argument
   b = ExternalInput2_read();  //get the second argument

   //...

   c = do_stuff(a,b);          //calculate the result

   //...

   ExternalOutputReg_write(c); //return the result

}
```

Listing 1: Sample NISC application

3.1.2 NISC Basic WISHBONE Interface implementation

To enable the transfer of the data and control flow, the NISC Basic WISHBONE Interface's provides registers for input arguments and special control registers. Control registers are used to set the state of the NISC processor's reset signal and to read the state of the halt signal. The result output ports are simply routed to the interface's output multiplexer to enable the main processor to read the results. A simplified block schematic of the interface is shown in Figure 2. The figure shows the internal registers of the interface and the simplified circuitry to illustrate the path of data to and from the NISC processor's top module NiscSystem.

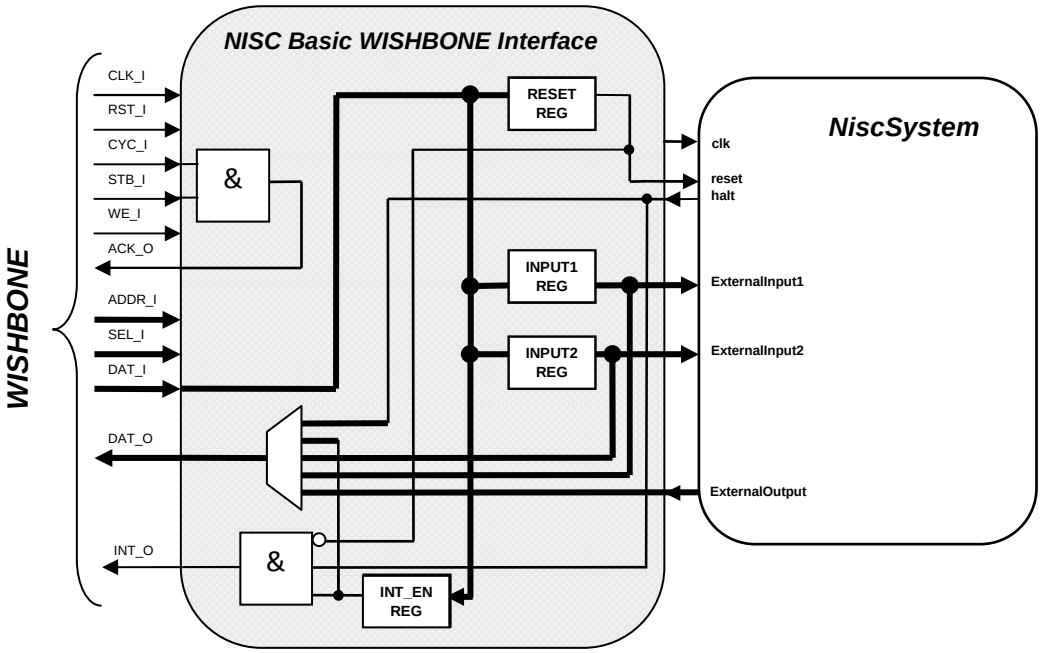

Figure 2: NISC Basic WISHBONE Interface

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 internal address decoding of the peripheral then determines which of its internal memory locations is addressed, and this is also the way the NISC Basic WISHBONE Interface operates. Reading the data from the interface is implemented using a 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 the interrupt enable (INT_EN) flag is set and the halt signal is 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.

3.1.3 NISC Basic WISHBONE Interface programming model

Using the NISC coprocessor attached to the WISHBONE bus in this way is very straightforward. 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 and the value '0' is written to the reset register to start the NISC's application execution. After that, the main processor can pool the value of the halt signal until it becomes active and then retrieve the results by reading appropriate registers. The reset register and the halt signal are mapped to the same memory location (location 0), and one is accessed when writing and the other when reading that memory address.

The interface's full memory map is shown on Table 1. The first memory location, at address 0 (plus the base address displacement) is reserved for the already mentioned reset/halt pair, the control register. Because 32-bit addressing is used, the address difference between two adjacent memory locations is 4 bytes. So, the next location's address is 4 and this is the location of the interrupt enable register. After that follow the argument and result registers. 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.

Table 1: NISC Basic WISHBONE Interface memory map

Address	Name	Width [bit]	Description
00	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
...	...	...	...
(N+2)*4	ARG N	32	*Read/Write* – Nth argument

Listing 2 shows an example *C* function offloaded to the NISC coprocessor to illustrate the designed communication scheme. The function call is inlined to eliminate the need for copying the arguments to local variables. This example function is blocking, i.e. it doesn't return until the coprocessor's task is completed and in that way the coprocessor's task behaves as a regular *C* function. Because of that, the programmer can use it from the calling function as though it is a software, rather than hardware, implementation of the algorithm. When using this communication scheme, only the body of the function needs to be replaced and the calling functions remain unchanged when migrating the desired function's implementation between hardware and software. In this way, it is very simple to experimentally evaluate design options and thus explore the design space.

For exploiting the task-level, data and function parallelism, more complex non-blocking versions of NISC coprocessor function calls should be used. In this way, the main processor can send the data to the NISC coprocessor, initialize the data processing and carry on with its own tasks. These tasks could include control functions, processing a part of the data (data parallelism), or performing a part of the processing on all the data (function parallelism).

```
// NISC declarations
volatile uint32_t * nisc_ctrl   = (uint32_t *)  Base_NISC;
volatile uint32_t * nisc_ie     = (uint32_t *) (Base_NISC + 4);
volatile uint32_t * nisc_result = (uint32_t *) (Base_NISC + 8);
volatile uint32_t * nisc_arg1   = (uint32_t *) (Base_NISC + 12);
volatile uint32_t * nisc_arg2   = (uint32_t *) (Base_NISC + 16);

//hide the pointers behind #define
#define NISC_CTRL      ( *nisc_ctrl )
#define NISC_HALT      ( *nisc_ctrl )    //alias for NISC_CTRL
#define NISC_IE        ( *nisc_ie )
#define NISC_RESULT    ( *nisc_result )
#define NISC_ARG1      ( *nisc_arg1 )
#define NISC_ARG2      ( *nisc_arg2 )

//constants
#define NISC_STOP      1
#define NISC_GO        0

//NISC function (blocking)
inline uint32_t nisc_function(uint32_t arg1, uint32_t arg2){

       NISC_CTRL = NISC_STOP; //stop the NISC
       NISC_ARG1 = arg1;      //write arg1
       NISC_ARG2 = arg2;      //write arg2
       NISC_CTRL = NISC_GO;   //start the NISC

       while(!NISC_HALT);     //wait until NISC completes

       return NISC_RESULT;    //return the result
}
```

Listing 2: Communication with the NISC processor

Interrupt-driven communication with the NISC coprocessor requires enabling the interrupts using the interrupt enable flag. When the arguments are sent to the coprocessor registers and the execution is started, the main processor can carry on with its regular tasks and retrieve the results in the interrupt service routine (ISR). As a part of the ISR, the main processor must also acknowledge the interrupt by clearing the NISC WISHBONE Interface's interrupt flag which is achieved by writing the value '0' to the NISC control register. Other standard interrupt acknowledge procedures must be followed depending on the processor's interrupt system architecture and whether the interrupts are level or edge triggered. The next batch of data to be processed can be sent in the ISR and the next execution can also be initiated.

3.2 NISC Data Memory WISHBONE Interface

The NISC Basic WISHBONE Interface was designed for applications which need to exchange a small amount of special purpose data with the main processor. The problem is that this approach doesn't scale well for applications that have a large amount of arguments and/or results. This approach would then require a large number of registers and large multiplexers, which would require a lot of chip area and would also limit the maximum frequency, especially when considering the negative implications of using large multiplexers in FPGAs.

A further drawback of the NISC Basic WISHBONE Interface is the inability to perform indexed addressing on the arguments and results without the need for data transfer between shared registers and the data memory. E.g. to perform indexed addressing on the arguments, NISC processor must copy the data from the interface's argument registers to internal arrays in the data memory and every piece of data thus has to be copied twice which imposes a large communication overhead when transferring large amounts of data. This especially represents an issue with algorithms that involve matrix calculations and the like.

3.2.1 NISC Data Memory Access Multiplexer/Arbiter

A simple solution for the problem of transferring large amounts of data to and from the NISC processor is the direct transfer of data to and from the NISC processor's data memory. To achieve this, a special module that enables external access to NISC's data memory was added to the design. The role of this data memory access multiplexer is to provide access to the NISC processor's data memory from the WISHBONE bus and to arbitrate access to the memory between the interface and the main processor. The data memory multiplexer is connected between the NISC processor's core (controller and datapath) and the data memory and it provides an additional set of pins that are connected outside of the NISC processor itself, as shown in Figure 3.

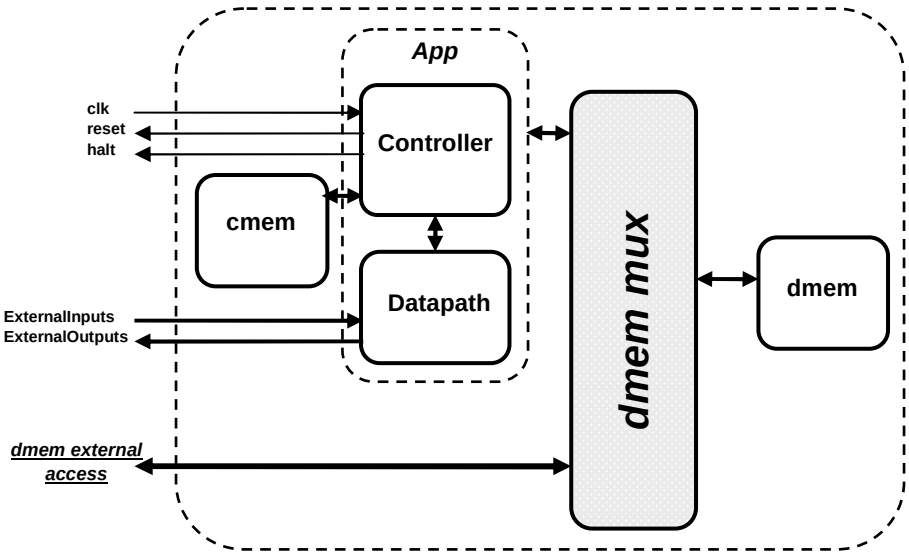

Figure 3: Multiplexing the access to the data memory

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 and 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 Figure 4.

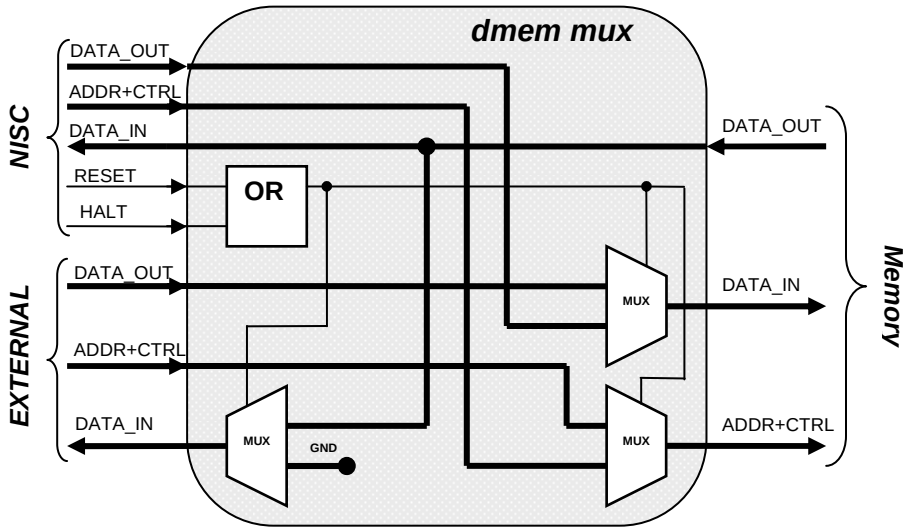

Figure 4: Data memory multiplexer implementation

3.2.2 NISC Data Memory WISHBONE Interface implementation

NISC Data Memory Access Multiplexer/Arbiter provides the means for external access to the data memory of the NISC processor and the NISC Data Memory WISHBONE Interface is connected to these external access pins. Its role is to handle the appropriate data conversion and handshaking to enable correct data memory access operations from the WISHBONE bus. NISC's data memory module encompasses FPGA block RAM's (BRAMs) and a memory controller module (namely, ByteAddressableDMemLogic Verilog module). The data memory module's ports are actually the memory controller's ports so the NISC Data Memory WISHBONE Interface is designed in such a way to operate with the memory controller and internal BRAMs.

Figure 5 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 1 cycle acknowledge delay to allow for the BRAM latency, since FPGA BRAMs are synchronous.

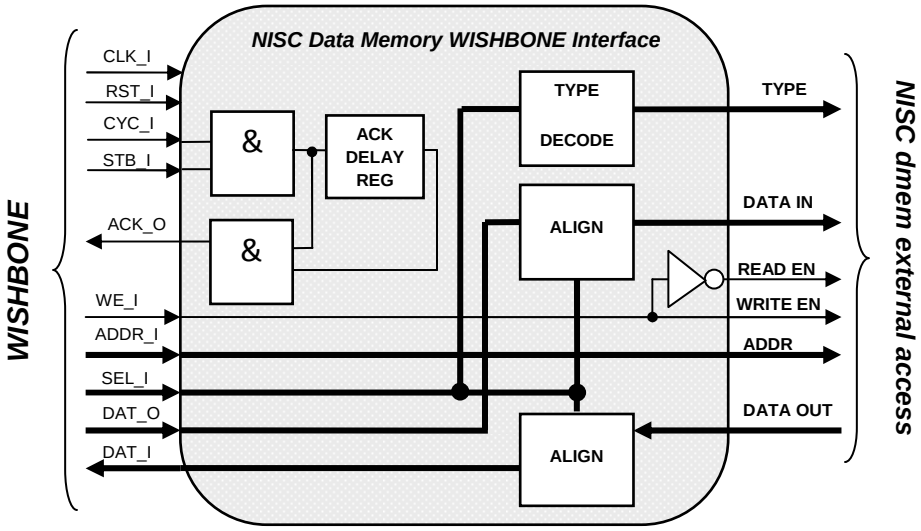

Figure 5: NISC Data Memory WISHBONE Interface

3.3 The complete NISC WISHBONE Interface and its programming model

The complete NISC WISHBONE Interface, as shown of Figure 6, consists of the NISC Basic WISHBONE Interface and the NISC Data Memory WISHBONE Interface. This approach with two distinct interfaces eliminates the need for address translations inside of the interface to differentiate between the shared registers space and the data memory space and thus simplifies the design. The simplest way to use the

complete interface is to first put the NISC processor in the reset state using the NISC Basic WISHBONE Interface and then send the data through the NISC Data Memory WISHBONE Interface. After that, NISC is taken out of reset state and the main processor waits for the application to complete and retrieves the results from the data memory. Special-purpose data could be also sent or retrieved using the NISC Basic WISHBONE Interface. It's also important to point out that the lack of hardware or software reset of NISC's data memory is vital for this sort of communication because the sent arguments would otherwise be lost when the NISC processor is reset.

To enable the main processor to read or write a particular bit of data in the NISC's data memory, its address must be known and it is used as an offset from the base address of the NISC Data Memory WISHBONE Interface in the main processor's memory map. NISC Toolset's Storage Bindings outputs are used to determine the address of a particular global variable or a global array in the data memory. Storage Bindings provide this information in the form of a table and the required information can be found in this table and used in the program code. The whole data memory of the NISC processor is mapped to a contiguous external memory block in the main processor's address space and is thus easily accessible through pointer manipulation. After the main processor writes the arguments to the global data structures in the NISC processor's data memory, the NISC processor operates on this data. The NISC processor then writes the results to designated global data structures which are afterwards read by the main processor. This approach requires no changes in the NISC's algorithm, provided that it uses global variables or global arrays. Global structures can also be passed to the functions as arguments, which eliminates the need for any changes in the code of any of the functions.

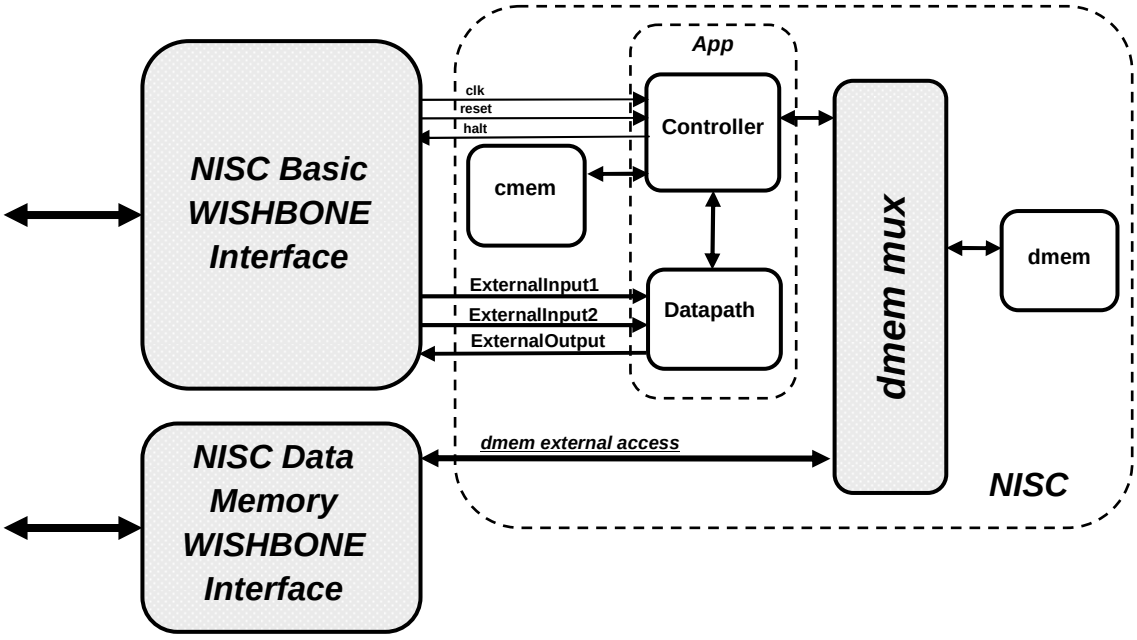

Figure 6: NISC WISHBONE Interface architecture

Listing 3 shows an example implementation of the communication using the NISC Data Memory WISHBONE Interface. After asserting NISC's reset signal to ensure access to the data memory, arguments are copied to the data memory using pointers and the data processing is started by deasserting the NISC reset signal. After completion detection, using pooling in this example, results can be retrieved for further processing. If there is no need to immediately start processing the next batch of data, the data can be used directly from NISC's data memory, thus eliminating the need for copying. If, however, we want to utilize task-level parallelism and process another batch of data in parallel with the task of the general-purpose processor, copying becomes necessary.

```
volatile uint32_t *nisc_dmem_arg = (uint32_t*) (nisc_dmem_base + arg_offset);
volatile uint32_t *nisc_dmem_res = (uint32_t*) (nisc_dmem_base + res_offset);

//NISC function (blocking)
inline void nisc_function(uint32_t *arguments, uint32_t *results){

       NISC_CTRL = NISC_STOP; //NISC stop

       //send the arguments
       for(int i=0; i < N; i++){
              nisc_dmem_arg[i] = arguments[i];
       }

       NISC_CTRL = NISC_GO;   //NISC start

       while(!NISC_HALT);     //wait until NISC completes

       //return the results
       for(int i=0; i < N; i++){
              results[i] = nisc_dmem_res[i];
       }
}
```

Listing 3: Communication using the NISC Data Memory WISHBONE Interface

3.4 TSK3000A WISHBONE System with a NISC coprocessor

The complete NISC WISHBONE Interface was implemented using VHDL and integrated with a generated NISC processor in a self-sufficient WISHBONE compatible module which can 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 (unlike FPGA vendors' proprietary processors, such as Xilinx MicroBlaze and Altera Nios). It is designed for seamless integration with Altium's configurable Wishbone Interconnect module which was also used in this proof-of-concept system. Main processor-side communication routines were developed as presented earlier in this chapter 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 Figure 7.

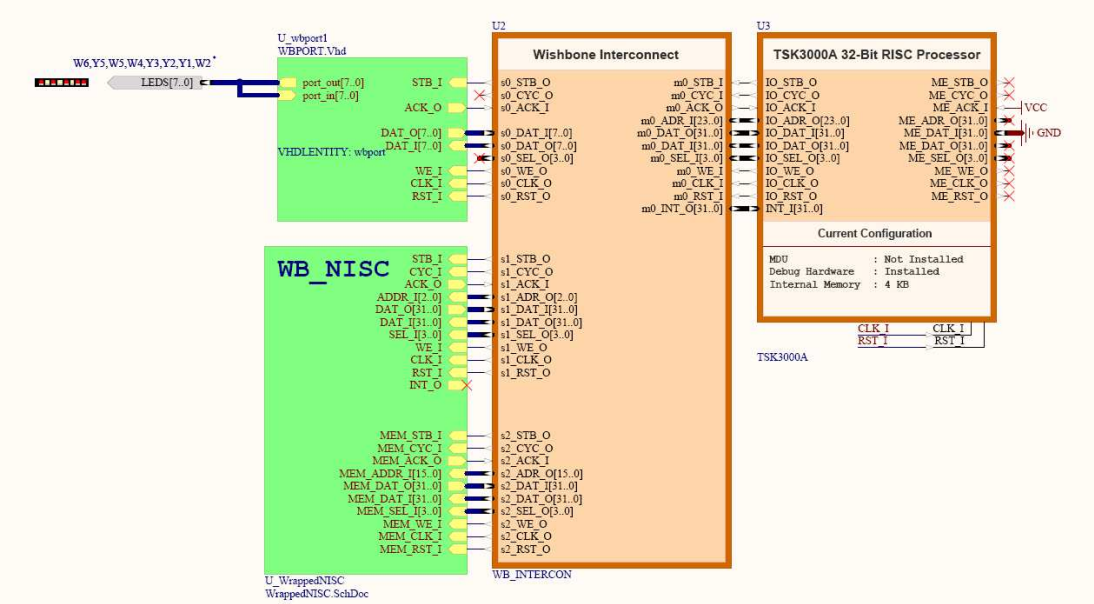

Figure 7: TSK3000A with a NISC coprocessor

4 Performance analysis

The designed NISC WISHBONE Interface provides the means for using the NISC processor as a loosely-coupled coprocessor in WISHBONE-based systems. To justify the use of the NISC coprocessor in the system, the speedup of the algorithm must justify the cost of the extra hardware. Furthermore, the speedup of the NISC implementation of the algorithm in question must be such to cancel the negative effect on the performance imposed by the main processor to coprocessor communication overhead and still provide overall system speedup. Of course, what is actually justified also depends on the achieved speedup rate and design constraints. E.g. for area-constrained systems a small achieved speedup would not justify the use of a NISC coprocessor, but it could be justified in systems where area is not an issue and every bit of performance improvement counts.

To ease the design space exploration and provide the means of estimating the system's performance and the margins of the speedup required by the NISC processor in the early stages of the project, an analytical model of the system's performance was devised. This model is intended to be used together with the profiling information and it enables the designer to estimate the benefits of using a NISC coprocessor for a particular part of the system's task and quickly explore the design space before the actual system is built. It also predicts the minimum performance limit for the NISC processor which can be used to quickly eliminate unsuitable architectures and thus drive the hardware-software co-optimization process in the right direction considering the system-level requirements.

The NISC WISHBONE Interface was designed in such a way as to extract maximum performance in systems with a general-purpose processor which uses WISHBONE classic bus cycles, like the Altium TSK3000A. This means that an access to the interface's register takes one clock cycle and an access to

the data memory takes two clock cycles (because of BRAM latency). WISHBONE classic bus cycles don't allow for distinct address and data transfer phases and any sort of bus transaction pipelining to improve performance. Because of that, there are no means for further acceleration of the data transfer according to the WISHBONE classic specification, not even with additional circuitry. The number of clock cycles required for the communication is defined by the number of cycles required for the transfer of arguments, the number of cycles required for the transfer of results and the number of cycles required for the control operations. The control overhead consists of only two cycles, one for stopping the NISC processor (i.e. putting it into reset state) and one for starting it (i.e. taking it out of reset state).

To estimate the performance impact of the communication overhead, we use the worst-case scenario for the transfer of the arguments and results, i.e. we use the slower mode of data transfer, the transfer of data to and from the data memory of the NISC processor. The communication time is defined by the cycle count for the communication and the frequency of the communication clock, i.e. the bus frequency, which is the same as the system frequency in fully synchronous systems.

Fully synchronous systems offer the advantage of easier design process and simpler verification. The drawback of using a single clock in the system is the negative impact on performance. The only way to extract the maximum performance is to use asynchronous clock domains for different parts of the systems, each running at its maximum frequency and use synchronization logic and FIFOs for crossing the clock domains, which often leads to design problems and difficult-to-track timing issues. The fully synchronous approach eliminates timing problems inside of the system and it was our intent to analyze and determine the boundaries of this design approach when using NISC coprocessors. The major speedup should be achieved by the means of parallelism and architecture optimizations with cores sharing the same clock and asynchronous design should only be considered when this approach fails to meet the design constrains.

Since the designed NISC WISHBONE Interface is 32 bits wide, the model presumes 32-bit arguments and results but this reasoning is, of course, valid for the arguments with widths of less than 32 bits. Transfer of data wider than 32 bits can be split up in 32 bit data transfers with some software support and can thus also be modeled. It is also important for the model to include additional software overheads for the communication, e.g. loop overhead. There are different approaches to accelerating this sort of communication, using both software and hardware. These approaches include loop unrolling, using custom loop instructions with zero control overhead (depending on the processor architecture and configurability) or using additional DMA engines to drive the data transfer. To model these additional parameters, a scaling factor for the number of transfers was added together with a constant that includes any additional control overheads (such as DMA setup) to better estimate the system's performance. The basic expression for estimating the communication time is given by the equation (1).

$$T_{comm} = \frac{CycleCount_{comm}}{f_{BUS}} = \\ = \frac{(ArgCount + \mathrm{Re}sCount) \times (2 + SWOverhead) + CtrlOverhead}{f_{BUS}} \quad (1)$$

To analyze the benefits of the NISC implementation of a particular function, we must compare the execution time of the algorithm for a general-purpose processor and the execution time when using the NISC processor. The execution time for the general-purpose processor is defined by the processor architecture and clock frequency. The processor architecture defines the number of instructions for the algorithm and the number of cycles required to execute an instruction (Cycles Per Instruction – CPI) and these define the total number of clock cycles for an algorithm. Equation (2) shows the expression for the execution time.

$$T_{CPU} = \frac{CycleCount_{CPU}}{f_{CPU}} \quad (2)$$

Similarly, the execution time for the NISC processor is defined by the number of cycles required for the particular application and the NISC processor's clock frequency, as equation (3) shows. The number of cycles is determined by the application itself and the NISC processor's architecture.

$$T_{NISC} = \frac{CycleCount_{NISC}}{f_{NISC}} \quad (3)$$

The total execution time when using the NISC processor as a coprocessor is defined by the sum of the execution time for the application itself and the time required for communication:

$$T_{COP} = T_{NISC} + T_{comm} \quad (4)$$

All these expressions depend on the frequency of a particular module. Since the NISC WISHBONE Interface is targeted at fully synchronous systems, the frequencies of all modules must be the same. Therefore, the maximum frequency of the system is determined by the minimum of all the core's frequencies, as shown in equation (5). This expression can be easily extended to systems with multiple NISC coprocessors.

$$f_{SYS} = \min\{f_{CPU,\max}, f_{NISC,\max}, f_{BUS,\max}\} \quad (5)$$

The execution time for the general-purpose processor can be expressed using (2) and (5), as follows:

$$T_{CPU} = \frac{CycleCount_{CPU}}{f_{SYS}} = \\ = \frac{CycleCount_{CPU}}{\min\{f_{CPU,\max}, f_{NISC,\max}, f_{BUS,\max}\}} \quad (6)$$

Similarly, the execution time for the NISC coprocessor can be expressed using (1), (3), (4) and (5), as follows:

$$T_{COP} = T_{NISC} + T_{comm} = \\ = \frac{CycleCount_{NISC} + (ArgCount + \mathrm{Re}sCount) \times (2 + SWOverhead) + CtrlOverhead}{f_{SYS}} = \\ = \frac{CycleCount_{NISC} + (ArgCount + \mathrm{Re}sCount) \times (2 + SWOverhead) + CtrlOverhead}{\min\{f_{CPU,\max}, f_{NISC,\max}, f_{BUS,\max}\}} \quad (7)$$

To determine the borderline of the coprocessor's effectiveness, we again turn to the worst-case scenario, which in this case is the blocking communication. When using blocking communication, there is no parallel execution of the algorithm on the main processor and the NISC coprocessor and because of that, this represents the worst case for the processor-coprocessor communication. For the hardware implementation of a function to be efficient, the obvious and necessary condition is for the function's execution time on the coprocessor to be shorter than the execution time on a general-purpose processor:

$$T_{COP} < T_{CPU} \quad (8)$$

If the coprocessor's implementation doesn't limit the system's maximum frequency (i.e. $f_{NISC,max} > f_{CPU,max}$ holds true) then the speedup of a part of an algorithm (taking into account communication and control overheads) certainly speeds up the execution of the whole program. From (6), (7) and (8), follows:

$$CycleCount_{NISC} < CycleCount_{CPU} - (ArgCount + \mathrm{Re}sCount) \times (2 + SWOverhead) - CtrlOverhead \quad (9)$$

The equation (9) defines the maximum number of clock cycles for the NISC coprocessor in order for that implementation not to reduce the effectiveness of the whole system. Using this expression, it is possible to eliminate unacceptable designs in the early stages of the project. For the coprocessor implementation to be truly effective, it must provide suitable speedup to justify the additional hardware expense. To quantify this, we must analyze the overall speedup factor. To do this, the program must be divided into two parts. One part will always be executed on the main processor. The other part can migrate between hardware and software implementations and it is for this part that we evaluate the effectiveness of the coprocessor implementation.

$$CycleCount_{PROGRAM} = CycleCount_{SW} + CycleCount_{HW/SW} \quad (10)$$

The speedup is defined as the ratio of execution times when using software or hardware implementation of the function in question. T_{CPU} is the execution time of a purely software implementation and $T_{CPU+COP}$ is the execution time when a part of the system's task is implemented in hardware, using the NISC coprocessor.

$$\begin{aligned} Speedup &= \frac{T_{CPU}}{T_{CPU+COP}} = \\ &= \frac{CycleCount_{SW} + CycleCount_{HW/SW,CPU}}{CycleCount_{SW} + CycleCount_{HW/SW,NISC} + CycleCount_{comm}} = \\ &= \frac{CycleCount_{SW} + CycleCount_{HW/SW,CPU}}{CycleCount_{SW} + CycleCount_{HW/SW,NISC} + (ArgCount + \mathrm{Re}sCount)\times(2 + SWOverhead) + CtrlOverhead} \end{aligned} \quad (11)$$

If the NISC coprocessor's implementation limits the system's maximum frequency then the speedup should be analyzed in this context. The NISC processor's implementation is in this case only effective if the overall execution time is shorter than that of a purely software implementation despite the lower overall system frequency.

$$T_{CPU+COP} < T_{CPU}$$

$$\frac{CycleCount_{SW} + CycleCount_{HW/SW,NISC} + CycleCount_{comm}}{f_{NISC,\max}} < \frac{CycleCount_{SW} + CycleCount_{HW/SW,CPU}}{f_{CPU,\max}}$$

$$\begin{aligned} CycleCount_{HW/SW,NISC} < \frac{f_{NISC,\max}}{f_{CPU,\max}}(CycleCount_{SW} + CycleCount_{HW/SW,CPU}) - \\ - CycleCount_{SW} - CycleCount_{comm} \end{aligned} \quad (12)$$

$$\begin{aligned} CycleCount_{HW/SW,NISC} < \frac{f_{NISC,\max}}{f_{CPU,\max}}(CycleCount_{SW} + CycleCount_{HW/SW,CPU}) - \\ - CycleCount_{SW} - (ArgCount + \mathrm{Re}sCount)\times(2 + SWOverhead) - CtrlOverhead \end{aligned}$$

The equation (12) defines the maximum number of clock cycles for the coprocessor for it not to degrade the system's performance and insure speedup, given the overall frequency of the system. If we're interested in the minimum frequency for the system that would insure speedup, (12) can be rewritten as follows:

$$\begin{aligned} &\frac{CycleCount_{SW} + CycleCount_{HW/SW,NISC} + CycleCount_{comm}}{f_{NISC,\max}} < \frac{CycleCount_{SW} + CycleCount_{HW/SW,CPU}}{f_{CPU,\max}} \\ &f_{NISC,\max} > \frac{f_{CPU,\max}\times(CycleCount_{SW} + CycleCount_{HW/SW,NISC} + CycleCount_{comm})}{CycleCount_{SW} + CycleCount_{HW/SW,CPU}} \end{aligned} \quad (17)$$

$$f_{NISC,\max} > \frac{f_{CPU,\max}\times(CycleCount_{SW} + CycleCount_{HW/SW,NISC} + (ArgCount + \mathrm{Re}sCount)\times(2 + SWOverhead) + CtrlOverhead)}{CycleCount_{SW} + CycleCount_{HW/SW,CPU}}$$

The system's speedup in the case when the NISC coprocessor limits the system's maximum frequency is as follows:

$$Speedup = \frac{T_{CPU}}{T_{CPU+COP}} = \frac{\frac{CycleCount_{SW} + CycleCount_{HW/SW,CPU}}{f_{CPU,\max}}}{\frac{CycleCount_{SW} + CycleCount_{HW/SW,NISC} + CycleCount_{comm}}{f_{NISC,\max}}} = \frac{\frac{CycleCount_{SW} + CycleCount_{HW/SW,CPU}}{f_{CPU,\max}}}{\frac{CycleCount_{SW} + CycleCount_{HW/SW,NISC} + (ArgCount + ResCount) \times (2 + SWOverhead) + CtrlOverhead}{f_{NISC,\max}}} \quad (18)$$

Similar reasoning applies to the case then the bus interconnect module limits the maximum frequency, e.g. due to its complexity increasing when introducing the NISC coprocessor in the system.

5 Conclusion

In this paper we presented our solution for using the No-Instruction-Set Computer (NISC) as an application-specific coprocessor in systems based on the WISHBONE open bus specification. This approach allows designers to easily use the NISC technology in existing systems with large amounts of unportable legacy code. We defined and explained the hardware and software extensions necessary for the integration of the NISC coprocessor in an arbitrary WISHBONE system. The flexibility of these extensions enables different ways of using the NISC coprocessor to accelerate the system's application. As a result, we enabled quick and easy integration of a hardware accelerator generated using the NISC Toolset into a given system to improve overall performance. With the previously designed interface and memory multiplexer hardware, the manual interface parameterization and integration process can't take more than an hour. This process is also a good candidate for automation using a simple software tool which can further simplify and speed-up the whole design process.

We also analyzed the performance implications of using such a coprocessor in a fully synchronous WISHBONE system. The effectiveness of using the NISC coprocessor and the borderline performance values required to achieve overall system speed-up were evaluated. We proposed an analytical performance model that could help quick exploration of the design space, elimination of unacceptable designs and setting the design targets even in early stages of the project. In this way, the hardware-software co-development and co-optimization process could converge more quickly towards the desired goal. Since the model requires only simple arithmetic calculations, it can easily be used to build a tool to help automate design space exploration.

6 Acknowledgements

The authors wish to thank the Unity through Knowledge Fund (UKF), Končar and SYSTEMCOM for supporting the project. This work builds on many years of NISC processor research at the Center for

Embedded Computer Systems (CECS), University of California at Irvine. We wish to thank CECS for allowing the download of NISC Toolset and Mehrdad Reshadi for his help and advice on how to use the toolset.

7 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, Irwine, 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/, (8.7.2008.)
6. 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.
7. B. Gorjiara, M. Reshadi, D. Gajski, "**NISC Communication Interface**", *Center for Embedded Computer Systems,* TR 06-05, March 2006
8. 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.
9. „**Specification for the: WISHBONE System-on-Chip (SoC) Interconnect Architecture for Portable IP Cores**“, Revision: B.3, Silicore/OpenCores.org, 2002.