

# **NISC Toolset User Guide**

## **Version. 2007.07**

Center for Embedded Computer Systems  
University of California at Irvine,  
Irvine, CA - 92697  
<http://www.cecs.uci.edu/~nisc>

## **Preface**

This document describes how to use the NISC toolset developed by Center for Embedded Computer Systems at University of California, Irvine. It helps the user understand how to setup the NISC toolset, understand different elements of the toolset and start using the toolset.

## **Audience Description**

This guide is intended for students, researchers, and IP designers. This guide assumes basic knowledge of software, and hardware simulation and synthesis process using hardware definition languages. (Preferably Verilog).

## **System Requirements**

To run the NISC toolset you need the following:

### **Hardware:**

Minimum 512 MB RAM

Pentium 1 GHz or faster

### **Software:**

Microsoft Windows 2000/XP/2003

Microsoft Visual Studio.NET 2003 or 2005

Xilinx (Or any other synthesis tool)

ModelSim (Or any other simulation tool)

NISC Toolset User Guide Version. 2007.07.....	1
Preface .....	2
Audience Description .....	2
System Requirements .....	2
1 Background .....	4
1.1 IPs and SoC .....	4
1.2 NISC .....	5
2 NISC Toolset.....	5
3 NISC Toolset Components.....	5
4 Installation.....	6
4.1 Directory structure.....	7
5 Application Source .....	7
6 Architecture: Generic NetList Representation (*.gnr).....	9
7 Input Configuration: Nisc System Xml arguments (*.nsx).....	10
7.1.1 Structure of NSX files .....	11
7.1.1.1 Argument “outputDir”.....	11
7.1.1.2 Group “libs” .....	11
7.1.1.3 Group “topModule”.....	12
7.1.1.4 Group “compiler” .....	12
7.1.1.5 Group “verilog” .....	13
7.1.1.6 Search paths .....	13
8 References.....	13
Appendix I: The DefaultNiscSystemConfig.xml .....	14
Appendix II: Benchmarks .....	17

# 1 Background

## 1.1 IPs and SoC

Performance of applications can be improved by exploiting their inherent horizontal and vertical parallelism. Horizontal parallelism occurs when multiple independent operations can be executed simultaneously. Vertical parallelism occurs when different stages of a sequence of operations can be overlapped. In processors, horizontal parallelism is utilized by having multiple functional units that run in parallel and vertical parallelism is utilized through pipelining. On the other hand, the power consumption of an embedded application can be reduced by moving as much computation as possible from runtime to compile time, and by customizing the microprocessor architecture to minimize number of cycles.

Over the past years, the trend of processor design has been to give compiler more control over the processor. This is more evident in transition from CISC (Complex Instruction Set Computer) to RISC (Reduced Instruction Set Computer) and from Superscalar to VLIW (Very Long Instruction Word) processors. While in CISC, instructions perform complex functionalities, in RISC, the compiler constructs these functionalities from a series of simple instructions. Similarly, while in superscalar, instruction scheduling is done in hardware at runtime, in VLIW, the compiler performs the instruction scheduling statically at compile time.

Increasing the role of compiler and its control over the processor has several benefits:

1. The compiler can look at the entire program and hence has a much larger observation window than what can be achieved in hardware. Therefore, much better analysis can be done in compiler than hardware.
2. More complex algorithms (such as instruction scheduling, register renaming) can be implemented in compiler than in hardware. This is because first, the compiler is not limited by the die size and other chip resources; and second, compiler's execution time does not impact the application execution time. In other words, compiler runs at design time, while hardware algorithms run during application execution.
3. The more functionality we move from hardware to compiler, the simpler the hardware becomes, and the less the runtime overhead is. This has a direct effect on power consumption of the circuit.

Currently, in VLIW processors, the compiler controls the schedule of parallel independent operations (*horizontal control*). However, in all processors, the compiler has no control over the flow of instructions in the pipeline (*vertical control*). Therefore, the vertical parallelism of the program may not be efficiently utilized. In Application Specific Instruction-set Processors (ASIPs), structure of pipeline can be customized for an application through custom instructions, but these approaches also have complex design flow and impose limitations on the extent of customizations. Furthermore, in all processors, no matter how many times an instruction is executed, it always goes through an instruction decoder. The instruction decoder consumes power and complicates the controller as well.

Another important limitation of processors is that they are not always suitable for designing embedded IP blocks. Processors cannot be customized beyond a certain point because they always assume a minimum behavior and an instruction decoder. This means that most often they impose an unnecessary overhead for implementing custom dedicated hardware blocks. Today, for such IP blocks, designers mostly use RTL designs along with different types of synthesis tools. However, these approaches also have their own limitations [5] and cannot efficiently address the development requirements of today's complex IPs.

The No-Instruction-Set-Computer (NISC) technology is proposed to address the above issues and to move as much functionality as possible from hardware to compiler. In NISC, the compiler determines both the schedule of parallel independent operations (horizontal parallelism), and the logical flow of

sequential operations in the pipeline (vertical parallelism). The compiler generates the control words (CWs) that must be applied to the datapath components at run time in every cycle. In other words, in NISC, all of the major tasks of a typical processor controller (i.e. instruction decoding, dependency analysis, and instruction scheduling) are done by the compiler statically. Since, in NISC, the compiler decides what the datapath should do at every clock cycle, we call it a *cycle-accurate compiler*. The NISC cycle-accurate compiler compiles the application directly to the datapath. It can achieve better parallelism and resource utilization than conventional instruction-set based compilers.

NISC technology can be used for low-power application-specific processor design, because: (a) the compiler-oriented control of the datapath, inherently minimizes the need for runtime hardware-based control, and therefore, reduces the overall power consumption of the design; (b) NISC technology allows datapath customizations to reduce total number of cycles and therefore total energy consumption. The extra slack time can also be used for voltage and frequency scaling, which result in more savings; and (c) NISC does not limit the number of custom functionalities that can be implemented on its datapath because instead of using custom instructions and then relying on the decoder in hardware to generate the control signals, in NISC the compiler generates the control signal values.

## 1.2 NISC

NISC technology is an enabler for the IP market. It provides a common parametrizable architecture and corresponding compiler and simulator for any IPs that are implemented using NISC technology. The NISC technology allows perfect tuning to any application since the architecture can reflect the structure of the application program while the compiler will optimize executable for such specific architecture. NISC architecture is defined by a netlist of RTL components such as registers, register files, memories, ALUs, shifters, buses and others. NISC compiler converts C language program into control words in the control memory for the NISC architecture. The content of the control memory together with the netlist can be used as an input to any FPGA, or ASIC standard tools. Since NISC architecture is defined before compilation it can be defined for manufacturability. NISC represents a new processor technology since it eliminates the instruction set, the last interpretation step between the programming language and the hardware that executes it.

## 2 NISC Toolset

This is the third major release of the NISC toolset with several new or improved features as well as many bug fixes. You can use the NISC toolset to quickly convert your C codes into simulatable/synthesizable Verilog code. There are several architectures and design examples provided in this version that you can use or modify for implementing your applications. You can also use this toolset as a backend for a synthesis tool that converts high level system or design descriptions into set of executable C codes and custom datapaths. The goal of this document is to help a new user understand different elements of the tools and start using the toolset. This document assumes that you are running the tools locally (e.g. from command shell). Many of the provided information does not apply to the Web based NISC Toolset interface [1].

## 3 NISC Toolset Components

Figure 1 shows the NISC flow of designing a custom architecture for a given application. The datapath can be generated (allocated) using different techniques. For example, it can be an IP, specified by the designer, reused from other designs, or generated automatically by algorithms similar to HLS. The datapath description is captured in a Generic Netlist Representation (GNR) [6]. A component in datapath can be a register, register-file, bus, multiplexer, functional unit, memory etc. The program, written in a high level language such as C, is first compiled and optimized by a front-end and then mapped on the given datapath. The compiler generates the stream of control values as well as the contents of data

memory. The generated results and datapath information are translated to a synthesizable RTL design that is used for simulation and synthesis. After synthesis and Placement and Routing, the accurate timing, power, and area information can be extracted and used for further datapath *refinement*. For example, the user may add functional units and pipeline registers, or change the bit-width of the components and observe the effect of modifications on precision of the computation, number of cycles, clock period, power, and area. In NISC, there is no need to design the instruction-set because the compiler automatically analyzes the datapath and extracts possible operations and branch delay. Therefore, the designer can refine the design very fast.

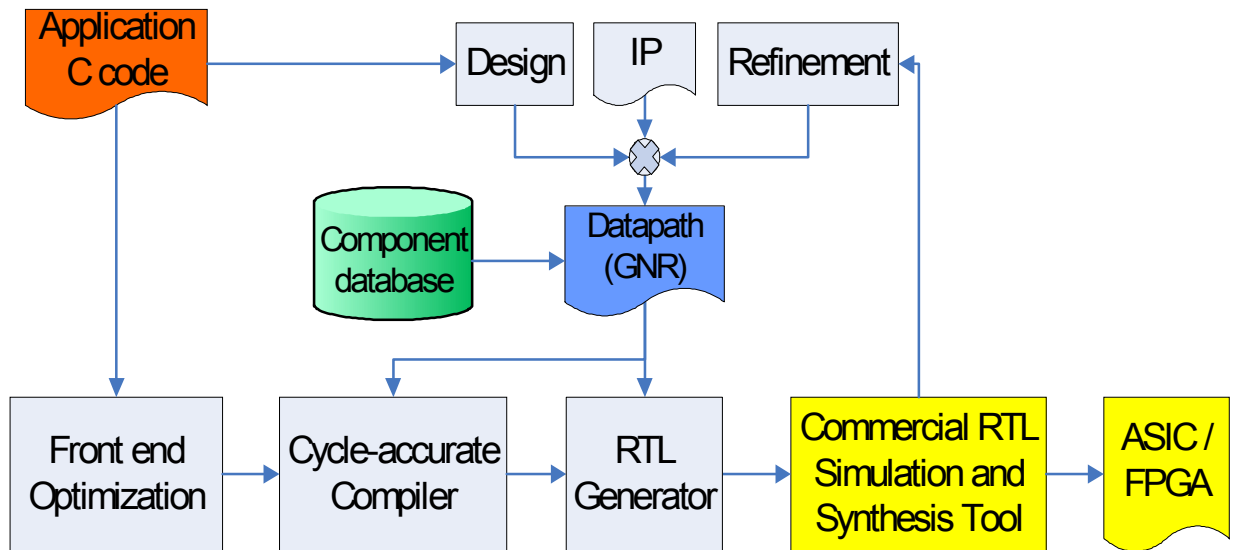


Figure 1- NISC toolset flow.

## 4 Installation

The NISC toolset and other components required for running NISC toolset can be downloaded from <http://www.cecs.uci.edu/~nisc/toolset/>. To install NISC toolset follow these steps:

1. Install at least one of the following front-ends:
  - a. Install Visual Studio.NET 2003. This will also install .NET framework 1.1.
  - b. Alternatively you can install Visual Studio.NET 2005.
  - c. Alternatively you can install the free Visual C++ 2005 Express Edition.
2. You may also need to install "Microsoft Core XML Services (MSXML) 6.0". However, in most cases, this program is already installed on your machine by default.
3. Download the latest version of NISC Toolset ZIP file and unzip it to any desired directory (e.g. C:\NiscToolset).
4. Run "`<%NiscToolset%>\NiscEnvironmnet\bin\Install.wsf`".

Here and in the rest of this document, we use %NiscToolset% to refer to the installation directory of the toolset. For example, if you are installing the toolset in the "C:\NiscToolset", you should replace "%NiscToolset%" with "C:\NiscToolset". The NISC toolset supports long filenames with spaces. However, you should consider the requirements of your Verilog simulation or synthesis tools as well when deciding about the installation directory. For example, the Xilinx Synthesis Tool does not support spaces in the directory names.

## 4.1 Directory structure

There are three main directories in the %NiscToolset% directory:

- *NiscArchitectures*: This directory includes the Generic Netlist Representation (GNR) of the components that are general and can be used in any design. It contains datapath components, several generic NISC architectures and two sample system modules that can be used as top modules of the design. The file %NiscToolset%/NiscArchitectures/Documents/index.htm contains documentation of these libraries and their components.
- *NiscEnvironment*: This directory contains the executable and other related files of the NISC toolset. The main file is %NiscToolset%/NiscEnvironment/bin/Release/Nisc.exe. The .nsx file extension must be associated to this executable to run properly. The installation script does that automatically. Alternatively, you can double click on an .nsx file and associate it with the above executable. In this new release, you can have multiple versions of the NISC toolset running side by side. If you need this feature, please contact us for more information.
- *NiscDesigns*: This directory contains benchmarks that have been tested and simulated with the NiscToolset. In each directory, you will find one or more .nsx file that calls the NiscToolset with proper arguments. You can execute them by double clicking on them. To see the output messages of the tools, it is recommended that you open a command prompt (cmd.exe), go to the corresponding directory, type the name of the .nsx file and then press enter. The output of the tools will be generated in the output directory that is specified in the .nsx file (See Section 7 for information about structure of .nsx files). These benchmarks also serve as examples for the new user. They may also be used as the starting point for developing new designs. Refer to Appendix II for details of these benchmarks.

## 5 Application Source

The source code of application developed in C is the input to the NISC compiler. There is a template application project file available in the %NiscToolset%/NiscDesigns/SingleCore/Apps/ApplicationProjectTemplate directory that has all the correct project settings and can be used as a starting point for creating new projects. Every such project has 4 configurations: *Debug* configuration is for compiling the application on NISC without any optimizations; *Release* configuration is for compiling the application on NISC after performing architecture independent optimizations (currently performed by MS VC++); *Release-Simulate* and *Debug-Simulate* configurations are used for compiling and running the application on PC with and without optimizations. You should specify the application information in the NSX file or the corresponding GNR module in the form of :

```
<App n="App-name" msil="path-of-exec-file">
  <Msvc
    project=".vcproj project file"
    config="Release or Debug configuration names"
    solution=".sln solution file"/>
</App>
```

Attribute `msil` is optional. The NiscToolset automatically detects what version of Visual Studio must be called and where the generated executable file is located. However, depending on the structure of the project file, in some case the toolset may not be able to find this output file and so, having the `msil` attribute can direct the tools to the correct file. The GNR and NSX file formats are explained in sections 6 and 7 0respectively.

Any NISC application must include *NiscStartup.c* and *NiscStandardDefinitions.h* that are available in %NiscToolset%/NiscDesigns/BaseLib directory. Also, the application must have `void NiscMain()`

and `void NiscInterrupt()` functions. The `NiscMain` function is the main top-level function of the application. The `NiscInterrupt` function is the main top-level interrupt handler routine which will be called whenever an interrupt is raised. More than one interrupt can be handled in this routine by checking the interrupt number. You can refer to the `%NiscToolset%/NiscDesigns/XilinxBoard_VideoStarterKit` and the communication examples in `%NiscToolset%/NiscDesigns/SimpleCommunication` to see examples of interrupt handling routines.

In this release of the tools, everything in C language is supported except a few features. These limitations are temporary and will be resolved in the future versions of the toolset. The following is a list of unsupported features:

- **Function Pointers:** Function pointers are not supported in the current version of the NISC compiler.
- **Global Pointer Initializations:** global pointer values that are initialized in the declaration are not supported. For example the following code is not supported:

```
int I;
int* pI=&I;
```

However, global pointers can be initialized in a function. For example the previous code can be written as:

```
int I;
int* pI;
void f() { pI = &I;}
```

Note that this also includes string array initializations. For example, the following declaration is not supported:

```
char* days[]={`mon`, `tue`, `wed`, `thu`, `fri`, `sat`, `sun`};
```

Instead, you can assign each string value in a function as shown below:

```
void f() {days[0]=`mon`; days[1]=`tue`; ...}
```

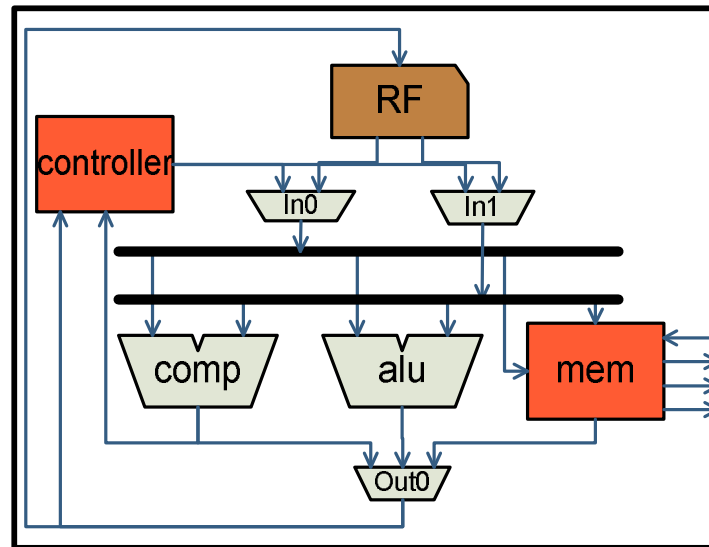
Please note that the following string declaration is valid and supported:

```
char* str=`This kind of declaration is supported!`;
```

- **Standard libraries:** you cannot `#include` any of the standard libraries such as `stdio.h`. Note that we are compiling the application on a single processor that does not have an OS. Therefore, function calls such as `printf` or OS calls such as `malloc` are meaningless in this setup. However, other standard library functions, e.g. string manipulators, can be used in the program if the source code of the body of these functions is also included in the input program. We have included a *Display* component in the *MainLib* library that simulates several display functions in Verilog. All of the processors in the `%NiscMain%/NiscArchitectures/Processors` use this component. When ever this component is used in a processor, you can call functions such as `__$stdout_display_int` to for example print an integer value. You can see example of using these functions in the `%NiscMain%/NiscDesigns/SingleCore/Apps/OnesCoutner` application.
- **Operation translations:** Supporting other features of the C language depends on the input NISC architecture. For example, to support floating point operations, your input architecture must have a floating point unit performing the corresponding operations. As another example, if your application contains division, you need to have a proper divider unit in your input architecture. NISC compiler does not automatically add software implementation of such operations.
- *Double* and *Long* types are not currently supported by the basic components in the `%NiscToolset%/NiscArchitectures/ComponentLib/`. You can add support for these types by adding your own components. The bit-width of almost all basic components in the *ComponentLib* is parametrizable and hence you can reuse them for *Long* types. But you need to design your own memory proxy component that supports 64-bit load/store operations.

## 6 Architecture: Generic NetList Representation (\*.gnr)

The Generic Netlist Representation (GNR) can capture a single IP or a system composed of several IPs as a parametrizable hierarchical netlist in XML format. GNR models a system as a hierarchical description of components (objects) and their connections (composition) i.e. the components used in a GNR module are described by another GNR module. GNR contains a set of predefined component-types and port-types that are used for enforcing the composition rules. For more information about the GNR format you can refer to [6]. The NISC toolset also comes with a XML Schema with which you can verify and validate



your GNR files using any standard XML processor.

**Figure 2: Sample IP module**

Any GNR module can be specified as the top-level module of the system. You can specify the top module and set its parameter values using an NSX file (See Section 70). In GNR, predefined types are represented by XML tags (in the form of `<tag-name>`). Figure 2 shows the block diagram of a sample IP. The GNR representation of this IP can be found in Appendix IV. A `<ProgrammedNisc>` component can be top module, or more than one of them can be included inside a system module. Each `<ProgrammedNisc>` specifies a `<NiscArchitecture>`, the application running on that architecture, and the data/control memory implementation of the corresponding NISC component. Each `<ProgrammedNisc>` is processed by the NISC compiler separately. The compilation results for each component are stored in a separate directory in the output directory specified in the NSX file.

When using a single NISC component, the simplest way is to use the system module available in `%NiscToolset%/NiscArchitectures/Systems/System1Nisc.gnr`. The actual application and NISC architecture must be passed as parameters to this module. To see an example of using this component, check the `%NiscToolset%/NiscDesigns/SingleCore/Arguments.nsx` file.

The GNR file corresponding to the Sample IP module shown in Figure 2 is show below:

```
1 <CustomIP type="simpleIP">
2   <Ports>
3     <Clock n="clk" bitWidth="1"/>
4     <InPort n="reset" bitWidth="1"/>
5     <InPort n="dm_r" bitWidth="32"/>
6     <OutPort n="dm_addr" bitWidth="32"/>
7     <OutPort n="dm_w" bitWidth="32"/>
8     <OutPort n="dm_readEn" bitWidth="1"/>
9     <OutPort n="dm_writeEn" bitWidth="1"/>
```

```

10 </Ports>
11 <Netlist>
12   <Components>
13     <Instance n="controller" type="Controller"/>
14     </Instance>
15     <Instance n="RF" type="RF2x1">
16       <SetParam n="BIT_WIDTH" val="32"/>
17       <SetParam n="REG_COUNT" val="32"/>
18     </Instance>
19     <Instance n="In0" type="Mux">
20     </Instance>
21     <Instance n="In1" type="Mux">
22     </Instance>
23     <Instance n="Out0" type="Mux">
24     </Instance>
25     <Instance n="comp" type="Comparator">
26     </Instance>
27     <Instance n="alu" type="ALU">
28     </Instance>
29     <Instance n="mem" type="DataMemProxy">
30     </Instance>
31   </Components>
32   <Connections>
33     <Conn src="controller" sPort="cw" dest="In0" dPort="i0" extend="signed" s="9" e="0"/>
34     <Conn src="comp" sPort="o" dest="controller" dPort="status" s="0" e="0"/>
35     <Conn src="RF" sPort="r0" dest="In0" dPort="i1"/>
36     <Conn src="RF" sPort="r1" dest="In1" dPort="i0"/>
37     <Conn src="Out0" sPort="o" dest="RF" dPort="w0"/>
38     <Conn src="In0" sPort="o" dest="comp" dPort="i0"/>
39     <Conn src="In1" sPort="o" dest="comp" dPort="i1"/>
40     <Conn src="comp" sPort="o" dest="Out0" dPort="i0"/>
41     <Conn src="In0" sPort="o" dest="alu" dPort="i0"/>
42     <Conn src="In1" sPort="o" dest="alu" dPort="i1"/>
43     <Conn src="alu" sPort="o" dest="Out0" dPort="i1"/>
44     <Conn src="In0" sPort="o" dest="mem" dPort="addr"/>
45     <Conn src="In1" sPort="o" dest="mem" dPort="w"/>
46     <Conn src="mem" sPort="r" dest="Out0" dPort="i2"/>
47     <Conn src="" sPort="dm_r" dest="mem" dPort="dm_r"/>
48     <Conn src="mem" sPort="dm_addr" dest="" dPort="dm_addr"/>
49     <Conn src="mem" sPort="dm_w" dest="" dPort="dm_w"/>
50     <Conn src="mem" sPort="dm_readEn" dest="" dPort="dm_readEn"/>
51     <Conn src="mem" sPort="dm_writeEn" dest="" dPort="dm_writeEn"/>
52     <!--## 2 clock connections##-->...
53     <!--## 13 control connections ##-->
54     <Conn src="controller" sPort="cw" dest="alu" dPort="ctrl" s="10" e="10"/>
55     ...
56   </Connections>
57 </Netlist>
58 <Compiler-aspect defaultIntegralRF="RF" defaultDMem="mem">
59   <CwFields n="cwFields">
60     <Field n="const0" bitWidth="10"/>
61     <!--## 13 control field ##-->
62     <CtrlField component="alu" ctrlPort="ctrl"/>
63     ...
64   </CwFields>...
65 </Compiler-aspect>
66 </CustomIP>

```

**Figure 3 – GNR representation of Simple IP shown in Error! Reference source not found.. Lines 3 to 9 specify the input and output ports of the top level module. Lines 13 to 30 specify the components of the IP. Lines 33 to 54 specify the connections between those components. Lines 59 to 64 specify the aspects. For further information about aspects refer to [4]**

## 7 Input Configuration: Nisc System Xml arguments (\*.nsx)

The NSX files are the top-level configuration files for running the NISC toolset. The files are cascadable, i.e. you can include other NSX files in one NSX file and then only specify a subset of required arguments. The arguments are copied from the included files in the same order they are included. If an argument is

specified twice (or in multiple files) the last value will be considered. This feature is very useful when you want to share a set of argument values among several designs. The main tool specific arguments are defined in the %NiscToolset%/NiscEnvironment/bin/DefaultNiscSystemConfig.xml, provided in Appendix I. We strongly recommend that you always include this file at the top of your own NSX file, and then specify design specific parameters.

### 7.1.1 Structure of NSX files

The NSX file has XML format and its arguments can be categorized in hierarchical groups. The root tag is <Args> and it can contain <Include>, <Group>, and <Arg> tags. The <Include> tag copies another NSX file in its place (the same way #include works in C). A <Group> tag can contain one or more <Arg> or nested <Group> tags. When merging two NSX files, the <Group> tags in different files that are in the same level and have the same name (i.e. same “n” attribute) are merged at the end. Therefore, the following two specifications are equivalent.

<pre>//File1.nsx &lt;Args&gt;   &lt;Group n="g1"&gt;     &lt;Arg n="a1" val="v1"/&gt;   &lt;/Group&gt; &lt;/Args&gt;  //File2.nsx &lt;Args&gt;   &lt;Group n="g1"&gt;     &lt;Arg n="a2" val="v2"/&gt;   &lt;/Group&gt; &lt;/Args&gt;</pre>	<pre>//File.nsx before merging &lt;Args&gt;   &lt;Include file="File1.nsx"/&gt;   &lt;Include file="File2.nsx"/&gt; &lt;/Args&gt;  //File.nsx after merging &lt;Args&gt;   &lt;Group n="g1"&gt;     &lt;Arg n="a1" val="v1"/&gt;     &lt;Arg n="a2" val="v2"/&gt;   &lt;/Group&gt; &lt;/Args&gt;</pre>
---	--

Figure 4 shows a sample NSX file that includes the default arguments values and then overwrites or specifies the design specific arguments values. In addition to the arguments specified in the *DefaultNiscSystemConfig.xml*, you typically need specify a minimum set of arguments. These arguments are explained in the following sections.

#### 7.1.1.1 Argument “outputDir”

The results and log files of the NISC toolset will be generated in an output directory that you must specify in your NSX file by <Arg n="outputDir" val="..."/>. Note that all file or directory paths in the NSX files can be relative to the location of the NSX file itself and so you do not need to specify full paths. So, for example, if your output directory does not start with a drive letter or a root symbol (‘\’ or ‘/’), then the directory path will be considered relative to where the NSX file is located.

#### 7.1.1.2 Group “libs”

You must specify the module name and file path of all components used in your design in a <Group n="libs"></Group>. If you have already included the *DefaultNiscSystemConfig.xml* then all of the standard components of the %NiscToolset%/NiscArchitectures/ComponentLib are already included and hence you only need to specify the top module and processor file name. Each module must be included in this group in the following format:

```
<Group n="libs">
  <Arg n="module-name" val="module-file.gnr"/>
```

```
...
</Group>
```

```
<Args>
  <Include file="../../NiscEnvironment/bin/DefaultNiscSystemConfig.xml"/>
  <Arg n="outputDir" val="Outputs"/>
  <Group n="libs">
    <Arg n="Systems" val="Systems/System1Nisc.gnr"/>
    <Arg n="Nisc" val="Processors/GN_5.gnr"/>
  </Group>
  <Group n="topModule">
    <Arg n="lib" val="Systems"/>
    <Arg n="type" val="System1Nisc"/>
    <Group n="params">
      <Arg n="NiscType" val="GN_5"/>
      <Arg n="APPLICATION_PRJ">
        <App n="OnesCounter">
          <Msvc
            project="Apps/OnesCounter/OnesCounter.vcproj"
            config="Release"
            solution="Apps/OnesCounter/OnesCounter.sln"/>
        </App>
      </Arg>
      <Arg n="RF_REG_COUNT" val="32"/>
      <Arg n="CMEM_DEPTH" val="300"/>
      <Arg n="DMEM_DEPTH" val="2000"/>
      <Arg n="CONST_WIDTH" val="20"/>
    </Group>
  </Group>
  <Group n="compiler">
    <Group n="default">
      <Arg n="genCfg" val="+"/>
    </Group>
  </Group>
</Args>
```

Figure 4-A sample NSX file

### 7.1.1.3 Group “topModule”

You need to specify the top module information in the `<Group n="topModule"></Group>`. The library and name of the top module are specified using `<Arg n="lib" val="lib-name"/>` and `<Arg n="type" val="module-name"/>` tags respectively. The values of the parameters of the top module must be specified in the `<Group n="params"></Group>` inside the *topModule* group. Note that the parameters of a module can also have a default value inside the GNR file of that module as well. So, if you do not specify a parameter in the NSX file, the NISC tools will use the default value from the GNR.

### 7.1.1.4 Group “compiler”

The `<Group n="compiler"></Group>` contains the arguments for configuring the NISC compiler. All main arguments are already set in the *DefaultNiscSystemConfig.xml* file. But you can control the outputs of the NISC compiler by overwriting the settings in the arguments that appear in the `<Group n="default"></Group>` inside the *compiler* group. Any argument that is specified in the *default* group is directly passed to the NISC compiler. For a list of these arguments and their descriptions please refer the *DefaultNiscSystemConfig.xml*. The output and log files of the NISC compiler will be stored in the `%outputDir%/NiscCompiler` directory where the `%outputDir%` is the one specified in the NSX file.

### 7.1.1.5 Group “verilog”

The `<Group n="verilog"></Group>` contains the arguments for configuring the NISC HDL generator. The `<Arg n="generateSimulatable" val="true"/>` enables the generation of simulatable Verilog in the `%outputDir%/Verilog/Simulate` directory where `%outputDir%` is the one specified in the NSX file. You can run the `%outputDir%/Verilog/Simulate/simulate.bat` file to simulate the results using your ModelSim simulator. If you want to change the settings of this batch file for all future executions of the NISC toolset, you can modify `%NiscToolset%/NiscEnvironment/Lib/simulate.bat` according to your own setup.

The `<Arg n="generateSynthesizable" val="true"/>` enables the generation of synthesizable Verilog in the `%outputDir%/Verilog/Synthesis` directory where `%outputDir%` is the one specified in the NSX file. Currently the tools generate synthesizable code for Xilinx FPGAs. In this case, the arguments inside `<Group n="xilinx"></Group>` are used to specify the target package information. You can get the correct values of these arguments from your Xilinx ISE toolset. The synthesizable Verilog files are generated for the Xilinx ISE tool. Therefore you need to have this tool installed on your machine. You may install the free ISE WebPACK available at <http://www.xilinx.com>. To synthesize the design on the selected Xilinx FPGA target follow these steps:

4. Run the `coregen.bat` file. This batch file calls the `CORGEN.EXE` installed by the ISE. It may take several minutes to finish. Please be patient until all the cores are generated.
5. Create a Xilinx ISE project file with the exact "Device Family", "Device", "Package", "Speed" values that you used in the `<Group n="xilinx"></Group>` in your NSX file.
6. Add the `Design.v`, `Testbench.v`, and the `*.xco` files to your project file.
7. Synthesize your design.

Note that these steps must be done in the specified order.

We have tested the included benchmarks (Refer Appendix III) on Virtex II, Virtex II Pro, and Virtex 4.

### 7.1.1.6 Search paths

Every NSX file can have one `<search-paths></search-paths>` tag that specify the directory paths that the NISC tools should look for the specified files. In each NSX file, you can specify the file-path relative to that NSX file, or alternatively, only specify the file name where needed and then add the relative or full path (starting by a drive letter followed by a `:\`) of the file's directory to the `<search-paths></search-paths>` section. It is not recommended to use the search paths, but if you need to do so, please check the `DefaultNiscSystemConfig.xml` file for examples of how to define yours.

## 8 References

- [1] Web-based NISC toolset: <http://www.cecs.uci.edu/~nisc/demo>.
- [2] B. Gorjiara, M. Reshadi, D. Gajski, "Designing a Custom Architecture for DCT Using NISC Technology", Asia and South Pacific Design Automation Conference (ASPDAC), Design Contest, January 2006.
- [3] MiBench benchmark: <http://www.eecs.umich.edu/mibench/>
- [4] B. Gorjiara, M. Reshadi, P. Chandraiah, D. Gajski, "Generic Netlist Representation for System and PE Level Design Exploration", International Symposium on Hardware/Software co-design and System Synthesis (CODES+ISSS), October 2006.
- [5] M. Reshadi, D. Gajski, "A Cycle-Accurate Compilation Algorithm for Custom Pipelined Datapaths", CODES+ISSS, 2005

[6] 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.

[7] NISC based designs: <http://www.cecs.uci.edu/~nisc/designs>.

## Appendix I: The DefaultNiscSystemConfig.xml

```
<?arg version="2.0" processor="Release/SystemGenerator.exe"?>
<Args>
  <Group n="tools">
    <!--
      The path where the Microsoft Visual Studio 2003 is installed. The MSVC++ compiler is used as
      front-end. -->
    <Arg n="msvc7CompilerExec" val="C:\Program Files\Microsoft Visual Studio .NET
2003\Common7\IDE\devenv.exe"/>
    <!--
      The path where the Microsoft Visual Studio 2005, or Microsoft Visual C++ 2005 Express Edition
      is installed. The MSVC++ compiler is used as front-end. -->
    <Arg n="msvc8CompilerExec" val="C:\Program Files\Microsoft Visual Studio
8\Common7\IDE\devenv.exe"/>
    <!--
      The path where the NISC compiler is installed. -->
    <Arg n="niscCompilerExec" val="NiscCompiler.exe"/>
  </Group>
  <!--
    Specifies where the root directory of the toolset is and is used by different tools to find
    their own files.-->
  <Arg n="NiscEnvironmentRootDir" val=".."/>
  <!--
    Specifies whether the final config file should be saved in the output directory after expanding
    all <Include> tags. -->
  <Arg n="saveUsedConfig" val="true"/>
  <!--
    Specifies where the path of log file. You can use val="" to write to the console. -->
  <Arg n="logFile" val=""/>
  <!-- this enables or disables the module checking! -->
  <Arg n="enableModuleValidator" val="true"/>
  <!--
    Describes where each library is located. -->
  <Group n="libs">
    <Arg n="ClockedMemLib" val="ComponentLib/ClockedMemLib/ClockedMemLib.gnr"/>
    <Arg n="ControllersLib" val="ComponentLib/ControllersLib/ControllersLib.gnr"/>
    <Arg n="FUsLib" val="ComponentLib/FUsLib/FUsLib.gnr"/>
    <Arg n="MainLib" val="ComponentLib/MainLib/MainLib.gnr"/>
    <Arg n="NiscWrapperLib" val="ComponentLib/NiscWrapperLib/NiscWrapperLib.gnr"/>
    <Arg n="CommunicationLib" val="ComponentLib/CommunicationLib/CommunicationLib.gnr"/>
    <Arg n="FloatingPointLib" val="ComponentLib/FloatingPointLib/FloatingPointLib.gnr"/>
    <Arg n="SimpleFUsLib" val="ComponentLib/SimpleFUsLib/SimpleFUsLib.gnr"/>
    <!--
    <Arg n="Systems" val="Systems/System1Nisc.gnr"/>
    <Arg n="Systems" val="Systems/System1Nisc1CI.gnr"/>
    -->
  </Group>
  <!--
    The top module information -->
  <Group n="topModule">
    <Arg n="saveExpandedSystem" val="false"/>
    <Group n="params">
    </Group>
  </Group>
  <!--
    The compiler arguments -->
  <Group n="compiler">
    <!--
      To reuse previous compilation results and only running other NISC tools, set val="false" -->
    <Arg n="runCompiler" val="true"/>
    <!--
```

```

To reuse previous front-end compilation results and only running other NISC tools, set
val="false" -->
  <Arg n="runFrontendCompiler" val="true"/>
  <!--
To reuse previous NISC Compiler compilation results and only running other NISC tools, set
val="false" -->
  <Arg n="runNiscCompiler" val="true"/>
  <!--
The NISC Compiler arguments -->
<Group n="default">
  <!--
##### OUTPUTS #####
-->
  <!--
Generate the cycle by cycle values of control words and control signals of each component--
>
  <Arg n="genCWTable" val="-"/>
  <!--
Generate the call graph for functions of the program-->
  <Arg n="genCallGraph" val="-"/>
  <!--
Generate HTML CDFG and data dependencies of each function in the application.-->
  <Arg n="genCdfg" val="-"/>
  <!--
Generate HTML CFG of each function in the application.-->
  <Arg n="genCfg" val="-"/>
  <!--
Generate the usage diagrams for components of the architecture.-->
  <Arg n="genCompUsage" val="-"/>
  <!--
Generate the utilization diagrams for components of the architecture.-->
  <Arg n="genCompUtils" val="-"/>
  <!--
Generate netlist graph of the architecture.-->
  <Arg n="genDotyArch" val="-"/>
  <!--
Generate schedule diagram information of the program.-->
  <Arg n="genOpSchedDiagram" val="-"/>
  <!--
Generate HTML table showing the operation slacks in each function in the application.-->
  <Arg n="genOpSlacks" val="-"/>
  <!--
Generate utilization diagrams for operations in the program.-->
  <Arg n="genOpUtils" val="-"/>
  <!--
Generate HTML table showing the flow of data values and their corresponding sequence of
machine actions in each clock cycle.-->
  <Arg n="genPipelineRegisterDetails" val="-"/>
  <!--
Generate HTML table showing the flow of data values in the registers in each clock cycle.--
>
  <Arg n="genPipelineRegisters" val="-"/>
  <!--
Generate a summary of scheduling results.-->
  <Arg n="genScheduleSummary" val="-"/>
  <!--
Generate a report showing where every constant, global variable, or local variable is
assigned to.-->
  <Arg n="genStorageBindings" val="-"/>
  <!--
##### CUSTOMIZE BEHAVIOR #####
-->
  <!--
It is possible to configure the tools to compile only one function in the design and
disable all function
call supports. In this mode, the tools become similar to traditional High Level Synthesis
tools. The following
arguments can be used for this purpose.
Note that in this case, that single function:
  -cannot have any argument
  -cannot return any value-

```

```

    -must call halt() at the end
-->
<!--
    Instead of compiling all of the functions in the design, you can select which functions to
be compiled.
    The selected functions can be specified as val=":f1,f2,...,fn,". Note that the string
starts with a ':'
    and after each function name, you need a ',':
    Example: <Arg n="includeFunc" val=":NiscMain,"/>
-->
<!--
    Instead of compiling all of the functions in the design, you can exclude some functions
from compilation.
    The excluded functions can be specified as val=":f1,f2,...,fn,". Note that the string
starts with a ':' and after
    each function name, you need a ',':
    Example: <Arg n="excludeFunc" val=":NiscMain,"/>
-->
<!--
    If you have only one software function in the design, you can use includeFunc argument,
and also tell the compiler
    no to generate the prolog/epilog of the function for storing/restoring the context.
    To do so, set val="-"-->
<Arg n="addPrologEpilog" val="+"/>
-->
<!--
##### MAIN BEHAVIOR #####
-->
<!--
    val="+" enables the scheduling and output generation. You can use val="-" to disable
scheduling
and only generate basic html outputs, e.g. for debugging.-->
<Arg n="schedule" val="+"/>
<!--
    Enables the binary control word generation. This is necessary for generating HDL. But if
you only want
    to use the HTML outputs, and have disabled the HDL generation, you can set this argument to
"-". But it
    is recommended that you do not change the value of this argument!-->
<Arg n="genCW" val="+"/>
<!--
    Enables the optimizations that are implemented in the NISC Compiler. These optimizations
are applied after NISC related code are added to the application. -->
<Arg n="optimize" val="+"/>
<!--
    To generated the pre-bound C functions and variables to be used in your application, set
val="+". Enabling
    this feature will disables all other compilations and output arguments. -->
<Arg n="genPreboundCFiles" val="-"/>
</Group>
</Group>
<!--
The Verilog generator arguments
-->
<Group n="verilog">
<!--
    Enable/disable generation of simulatable Verilog with true/false as the value.-->
<Arg n="generateSimulatable" val="true"/>
<!--
    Enable/disable generation of synthesizable Verilog with true/false as the value.-->
<Arg n="generateSynthesizable" val="true"/>
<!-- additional file that will be copied to the simulation and synthesis directories-->
<Group n="additionalFiles">
    <!--from the Lib directory (see the paths below) -->
    <Arg n="simulationBatFile" val="simulate.bat"/>
</Group>
<Group n="xilinx">
<!--
    If generating synthesizable Verilog is enabled, the following enables generation of xlinx
files.-->
    <Arg n="generateXilinxFiles" val="true"/>

```

```

<!--
The following arguments specify the Xilinx FPGA package information.-->
<!--<Arg n="deviceFamily" val="virtex2"/>
<Arg n="device" val="xc2v2000"/>
<Arg n="package" val="ff896"/>
<Arg n="speed" val="-6"/>
-->
<Arg n="deviceFamily" val="virtex4"/>
<Arg n="device" val="xc4vsx35"/>
<Arg n="package" val="ff668"/>
<Arg n="speed" val="-12"/>
<!--
<Arg n="deviceFamily" val="virtex4"></Arg>
<Arg n="device" val="xc4vlx15"></Arg>
<Arg n="package" val="sf363"></Arg>
<Arg n="speed" val="-12"></Arg>
-->
<!-- additional file that will be copied to the synthesis directory-->
<Group n="additionalFiles">
  <!--from the Lib directory (see the paths below) -->
  <Arg n="copyAll2Xilinx" val="copyAll2Xilinx.bat"/>
  <!--from the Lib directory (see the paths below) -->
  <Arg n="copyVerilogs2Xilinx" val="copyVerilogs2Xilinx.bat"/>
</Group>
</Group>
</Group>
<!--
The search paths in which the looks will look for the require or specified files. The file name
or
its relative path is combined with the search paths to find the actual file.-->
<search-paths>
  <!-- The NiscEnvironmentRootDir -->
  <path>..\</path>
  <path>Release\</path>
  <path>..\..\NiscArchitectures\</path>
  <path>..\..\NiscArchitectures\ComponentLib\</path>
  <path>..\..\NiscArchitectures\ComponentLib\ClockedMemLib</path>
  <path>..\..\NiscArchitectures\ComponentLib\CommunicationLib</path>
  <path>..\..\NiscArchitectures\ComponentLib\ControllersLib</path>
  <path>..\..\NiscArchitectures\ComponentLib\FloatingPointLib</path>
  <path>..\..\NiscArchitectures\ComponentLib\FUsLib</path>
  <path>..\..\NiscArchitectures\ComponentLib>MainLib</path>
  <path>..\..\NiscArchitectures\ComponentLib\NiscWrapperLib</path>
  <path>..\..\NiscArchitectures\ComponentLib\SimpleFUsLib</path>
  <path>..\..\NiscArchitectures\Processors\</path>
  <!-- the Lib directory that contains additional files -->
  <path>..\Lib\</path>
</search-paths>
</Args>

```

## Appendix II: Benchmarks

This directory %NiscToolset%\NiscDesgins contains benchmarks that have been tested, simulated, and synthesized with the NiscToolset. The benchmarks included in this version of the toolset are:

1. *CustomDCT*: The .nsx argument files in this directory show the sequence of steps for designing a custom DCT block. For more information about this design, please refer to [2].
2. *FpMp3*: This directory contains three experiments with fixed point MP3 decoder: (I) a single core implementation, (II) a coprocessor-style implementation, and (III) a pipelined implementation of the decoder. For the two latter designs, in addition to the .nsx files, the top-level system modules (.gnr file) are also included.
3. *MiBench*: This directory includes benchmarks from MiBench [3] suite. The source codes of these benchmarks are modified in order to include the input data as well.

4. *NoC*: This directory includes three examples for making a network-on-chip design using NISC components. The designs includes a 1×1, 1×2, and 3×3 set of Cells each including a NISC processing element and a NISC router. A test application is provided with the example, which you can extend to apply your own. You can get the full synthesizable Verilog of your NoC this way.
5. *SimpleCommunication*: This directory contains two simple examples of multi-NISC systems: (I) Two NISCs communicating over one bus, and (II) five NISCs communicating over two busses.
6. *SingleCore*: This directory contains several simple examples for running on a single NISC component. The applications in this directory can be executed on almost all of the NISC processors in %NiscToolset%/NiscArchitectures/Processors. To create a new application, copy and rename %NiscToolset%/NiscDesigns/SingleCore/Apps/ApplicationProjectTemplate directory. Do not create a new Visual Studio Project file unless you are familiar know how to transfer all of the settings from the template project file.
7. *XilinxBoard\_VideoStarterKit*: This directory contains three examples that use interrupts for handling the LEDs and the switches on a Xilinx Video Starter kit board. The batch files in this directory show the sequence of the steps needed for implementing this example. For more details refer to [7].