NISC Technology Online Toolset

Mehrdad Reshadi, Bita Gorjiara, Daniel Gajski

Technical Report CECS-05-19

December 2005

Center for Embedded Computer Systems
University of California Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{reshadi, bgorjiar, gajski}@cecs.uci.edu

NISC Technology Online Toolset

Mehrdad Reshadi, Bita Gorjiara, Daniel Gajski

Technical Report CECS-05-19

December 2005

Center for Embedded Computer Systems
University of California Irvine
Irvine, CA 92697-3425, USA
(949) 824-8059

{reshadi, bgorjiar, gajski}@cecs.uci.edu

Abstract

We are currently developing the NISC technology at CECS at UC Irvine. The main goal of this technology is to simplify the design of custom processors. It simplifies the process by removing the instruction abstraction from processor. Consequently, the compiler for a NISC processor is more complex because it must combine compilation / synthesis techniques. The current online toolset (at http://www.cecs.uci.edu/~nisc) demonstrates this technology. At this link, the user can provide the program C code and processor netlist (user may upload these files or select from predefined examples) and the tool compiles the program on the datapath and generates several outputs including views of pipeline and simulatable Verilog codes.

Table of contents

NISC Technology Online Toolset

Mehrdad Reshadi, Bita Gorjiara, Daniel Gajski

(reshadi, bgorjiar, gajski)@cecs.uci.edu

1 About the online toolset

We are currently developing the NISC technology at CECS at UC Irvine. The main goal of this technology is to simplify the design of custom processors. It simplifies the process by removing the instruction abstraction from processor. Consequently, the compiler for a NISC processor is more complex because it must combine compilation / synthesis techniques.

In the NISC design flow, an application written in C as well as the netlist of datapath is input to the NISC toolset. Using these tools, first the application is compiled on the datapath and then the proper control word for each cycle of execution is produced. Finally the tools generate (a) the simulatable and synthesizable RTL (Verilog) of the complete NISC processor and (b) the contents of NISC control / data memory.

The current online NISC toolset (at http://www.cecs.uci.edu/~nisc) demonstrates this technology. At this link, the user can provide the program C code and processor netlist (user may upload these files or select from predefined example) and the tool compiles the program on the datapath and generates several outputs including views of pipeline and simulatable Verilog codes.

The NISC compiler, used in the online tool, is a stable version that does not include any code generation optimizations. Exclusion of optimizations also reduces the work load on our server. Only standard front end optimizations on the input C code is included in this online demo. In this document, we first explain how to run the tool in Section 2 and then describe the inputs and outputs of the tool.

2 How to run the online tool

The interface of the tool is simple and straight forward. To run the tool, follow these steps:

1. Press the ***Start*** to go to the ***Application*** page.
2. Input your code or select one of the available examples and then press ***Next***.
3. Upload your processor description file or select one of the available examples and then press the ***Next***.
4. Select the proper compiler options and then press the ***Compile*** button.
5. Wait until the results are ready. If there was any error, check the log files to resolve the error.

Notes:

***After the first run of the demo, you can repeat the above steps in any order using the menu on the top of each page.

*** At the top of each page, you can find a link to the help file that explains the contents and operations of that page.

3 The input application

The C code is first compiled to a 3-address code and then mapped to the given datapath. The supported operations and data types in the program depend on the components in the architecture. For example, to compile a C code that has a multiplication or division, the given architecture must have at least one multiplier or divider, respectively.

Currently, every thing in C-language is supported, except:

- 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 or I/O (yet). Therefore, I/O 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.
- Function pointers and indirect call/jumps: at this point, we do not support indirect jumps, therefore (a) function pointers, (b) indirect call/jumps, and (c) switch statements that are converted to a jump table (this may happen if the case statements form a sequence of integral values) are not supported.
- Memory block copy / init: a `struct` or `string` variable that is treated as a normal variable (and not a pointer) may cause a compiler error indicating that "a `cpblk` or `initblk` operation is not supported". Some of the cases that may cause this problem includes:
  - Initializing an array of strings in the declaration. For example `char x[][]={"S1", "S2"};`
  - Copying one `struct` variable to another.
  - Passing a `struct` variable as the parameter of a function, or returning a `struct` value in a function.
- Floating point operations and variables: At this point we do not have Verilog components with floating point operands in our Verilog library. Therefore, floating point is not supported.

4 The input processor

You can either upload your own processor description file, or select one of the example processors.

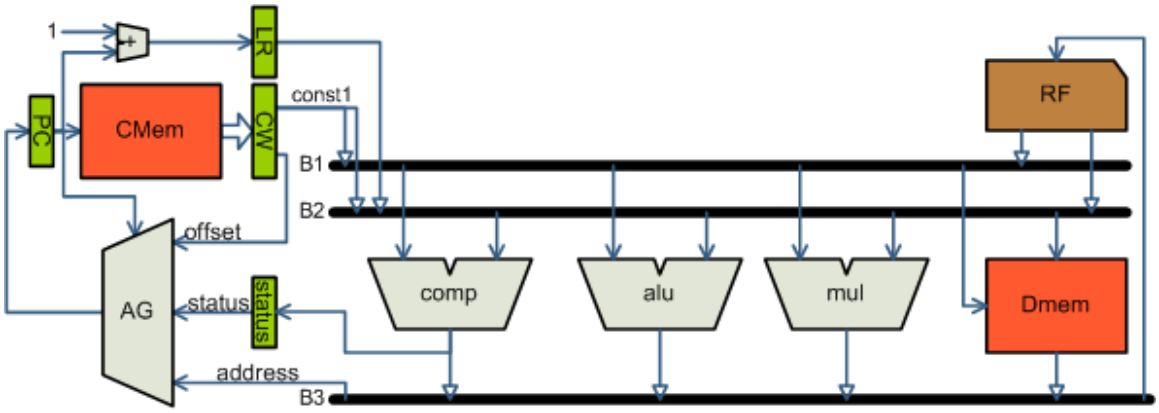
4.1 Uploading processor description file

You can specify your own processor using our Generic Netlist Representation (GNR) [?]. GNR is a set of (a) component instantiations, (b) component connections, and (c) annotation specifications. One good approach is to select an example processor, download and save the corresponding file on your hard disk from the results page, modify the file, and upload it back in the ***Processor*** page.

4.2 Example processors

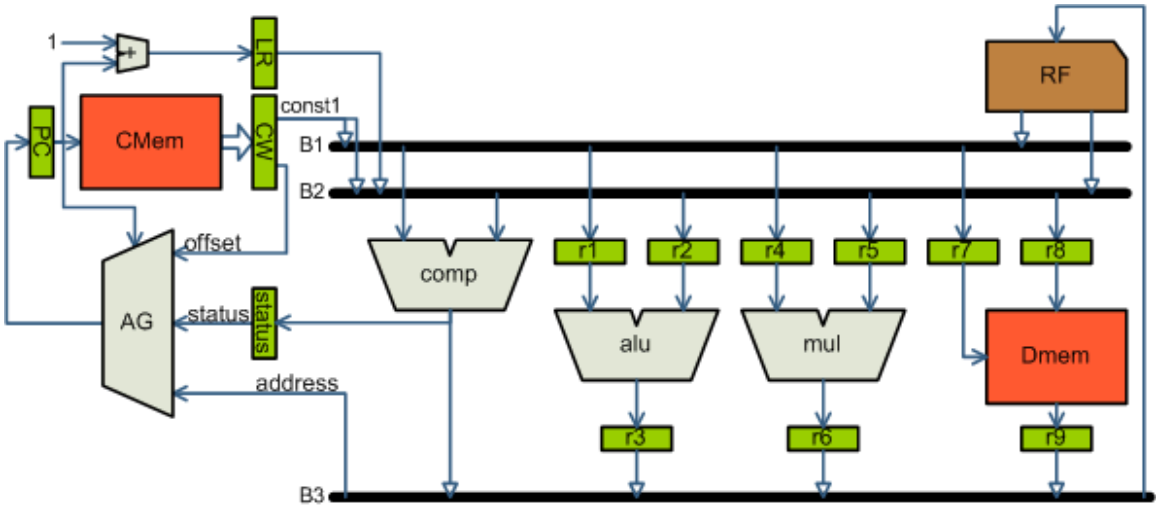
4.2.1 Controller Pipelining

The datapath of this processor is not pipelined. The control path (activated for jumps) has two registers (control word register and status register).



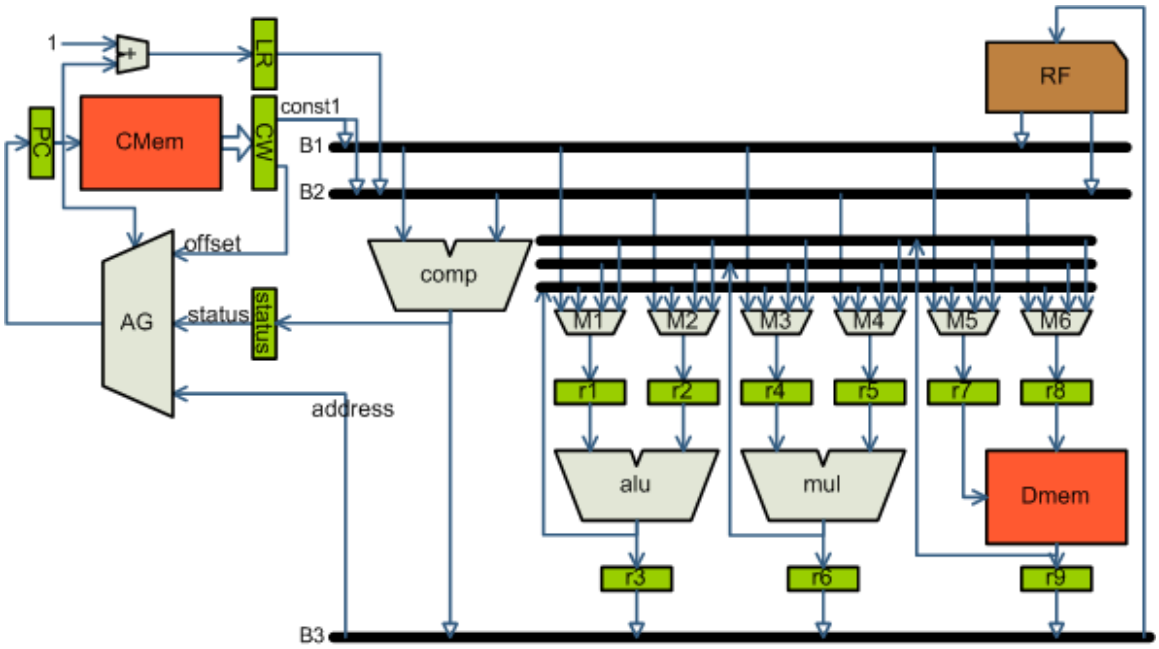
4.2.2 Controller and Datapath Pipelining

Both datapath and control path of this processor are pipelined.



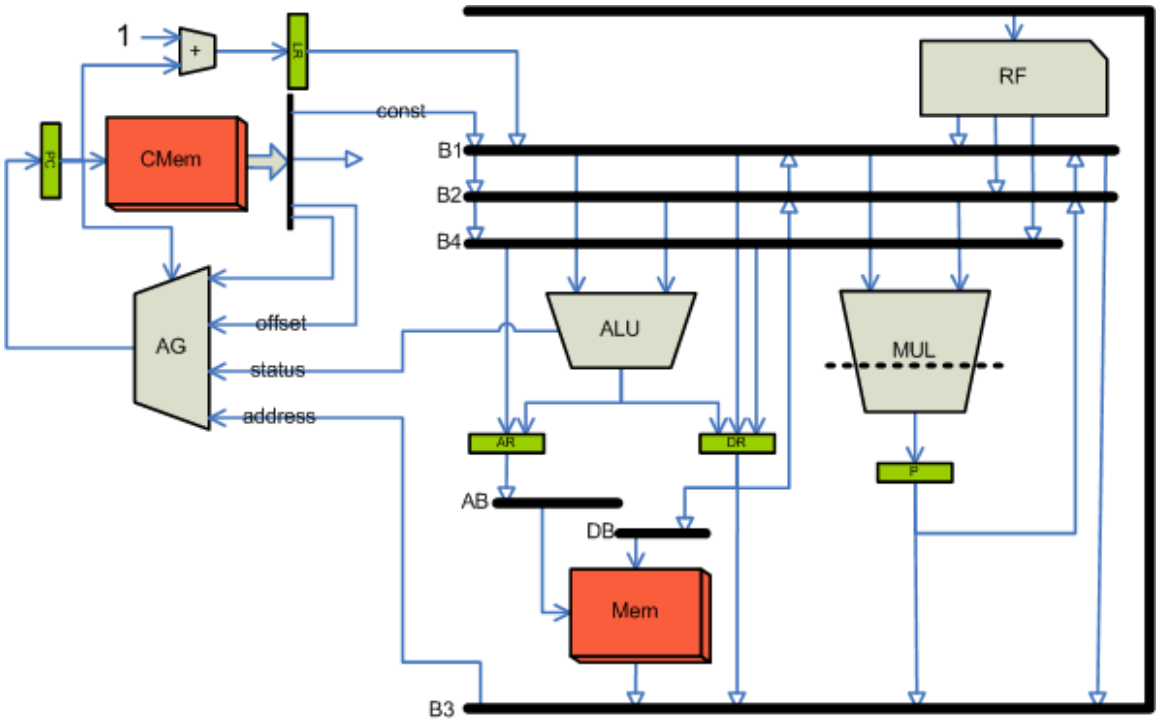
4.2.3 Controller and Datapath Pipelining + Forwarding

Both datapath and control path of this processor are pipelined. Additionally, the outputs of components are forwarded to the input of other components.



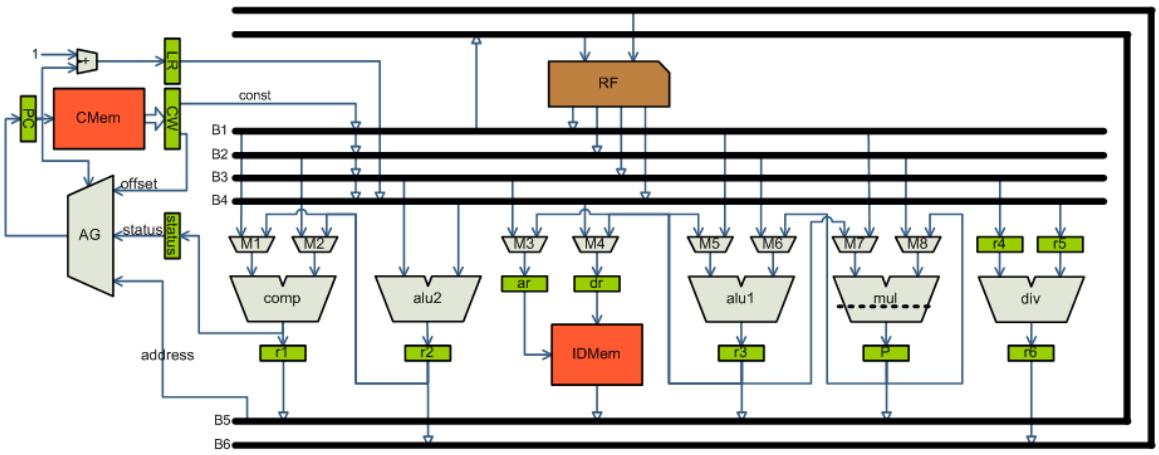
4.2.4 NISC style MIPS

The datapath of this processor is the same as the datapath of a MIPS processor. However, the controller is a NISC style controller and does not have any instruction decoder.



4.2.5 NISC style extended MIPS

This is an extended version of the MIPS processor datapath. An extra ALU and a DIVIDER component are added to the datapath. Note that in this processor, the data forwarding paths are not uniform. The output of *alu1* is forwarded only to its left input (and not both inputs). Similarly, the output of *mul* is forwarded to its right input and the right input of *alu1*.



5 The options of the tool

In the ***Compile*** page you can select different options that control the behavior of the tool. In this page, you can also see the program code and the processor GNR file that are used as inputs of the NISC toolset.

You can select the following options:

- **Optimize application code:** by selecting this option the C code of the application is optimized. These are standard architecture independent optimizations that are implemented by the front end. We use Microsoft® Visual C++ 2003 as front end.
- **Generate CFG/DFG graphs:** by selecting this option, the Control Flow Graph (CFG) and Data Flow Graph (DFG) of each function in the C code are generated.
- **Generate Verilog:** by selecting this option, the Verilog files of the processor and its control/data memory contents are generated.

6 The outputs of the tool

In the ***Results*** page, there are four groups of files. Each group is explained in the following sections.

6.1 Input files

This group contains the following files:

- Application source code: the C code of the input program
- Processor code: the GNR file of input processor.

6.2 Log files

This group contains the log files that report the status of different tools. These files include:

- GNR processor log shows the warnings and errors that are encountered during processing the processor GNR file. It also includes the errors that might have been encountered by the (a) C compiler front end, (b) NISC compiler, or (c) the Verilog generator.
- C compiler build log shows the status of the C compiler and the list of possible errors and warning that are encountered during processing of program C code.
- NISC compiler log shows the timing and warnings corresponding to various phases of the NISC compiler that maps the program (after converting to 3-address operations) on the given processor.

6.3 Compiler outputs

6.3.1 Binary output

The “Binary output” file shows the contents of control/data memory.

6.3.2 Compilation summary

The “Compilation summary” file shows some statistics for each function in the program.

6.3.3 Views of pipeline

The NISC compiler generates several HTML output files that represent different views of the pipeline components during execution of the program. These pipeline view (PV) files have a similar structure but present different set of information.

Functions of the program are listed at the top of a PV file. Each function in the program has a section in the PV file that includes: name, storage binding, CFG/DFG of that function, and the status of pipeline during execution of basic block of that function.

Figure 1 shows the first three columns of the PV table of a sample `main` function. The first column of the table shows the clock cycle starting from the beginning of a basic block and the third columns shows the value of program counter (PC) in that clock. The second column shows the schedule of 3-address operations of the program. A 3-address operation is an expression that performs an operation on variables, constants and registers. For example, in Figure 1, in clock cycle 0, the value of stack pointer (`__$SP`) is added with constant 4 and stored in the temporary variable `t146`. The registers of the architecture are shown with `__$reg-name`. For example, `__$SP` is stack pointer (that points the top of stack), `__$FP` is frame pointer (that points to the beginning of the stack of current function), `__$LR` is link register (that stores the return address of a function call), and `__$RF(3)` is register index 3 in register file `RF`.

main | storage bindings | CFG | DFG | *Methods list* |

clock	operation	PC
0(10): $bb0		
0	t146=(__$SP+4);	10
1	t143: *(__$SP)= __$LR;	11
2	t147: *(t146)= __$FP;	12
3	*no-op*	13
4	t125: __$FP:= __$SP;	14
5	t150=(__$FP+8);	15
6	t128=(__$SP+main.stackSize);	16
7	t129: __$SP:= t128;t151: *(t150)= __$RF(3);	17
8	*no-op*	18
9	*no-op*	19
10		
1(20): $bb1		
0	t115=(__$FP+-4);	20
1	*no-op*	21
2	t116: *(t115)= 0;	22
3	*no-op*	23

Figure 1- The common part of pipeline view (PV) files.

In the rest of this section, we describe the specific PV tables that are generated by the tool.

6.3.3.1 Control words

The "Control words" PV file shows the control word and components' control bits in each clock cycle. The main structure of this PV and the first three columns of the tables are as described in Section 6.3.3. The fourth column of the tables shows the hexadecimal value of the control word for the corresponding PC value. The rest of the columns of the tables, show the binary values of the control ports of the components of the datapath. The header of each column shows the name of the corresponding control port as well as the bit-slice in the control word that corresponds to that control port.

6.3.3.2 Contents of pipeline registers

The "Contents of pipeline registers" PV file shows the value of registers in the datapath in different clock cycles. The main structure of this PV and the first three columns of the tables are as described in Section 6.3.3. The rest of the columns show the value of the registers. Figure 2 shows contents of pipeline registers after compiling a sample `main` function on the example architecture shown in Section 4.2.2. In this example, execution of 3-address operation `t146=__$SP+4` starts at clock cycle 0 when PC points to address

10. In cycle 1, the CW register contains a value 4 in its constant field. In cycle 2, registers `r1` and `r2` contain the value of `__$SP` register and constant 4, respectively. The result of addition (`t146`) is first loaded into register `r3` in cycle 3 and then loaded into register file `RF` in cycle 4. Note that the colors of data values correspond to the operations (in the second column) that is using or generating them.

main | storage bindings | CFG | DFG | *Methods list* |

clock	operation	PC	$0	LR	cwFields	r1	r2	r3	r4	r5	r6	r7	r8	r9	status	RF
0(10): $bb0																
0	t146=(__$SP+4);	10														
1	t143: *(__$SP)= __$LR;	11			4,											
2	t147: *(t146)= __$FP;	12				__$SP,	4,									
3	*no-op*	13						t146,				__$SP,	__$LR,			
4	t125: __$FP:= __$SP;	14										t146,	__$FP,			t146,
5	t150=(__$FP+8);	15														
6	t128=(__$SP+main.stackSize);	16			8,											__$SP,
7	t129: __$SP:= t128;t151: *(t150)= __$RF(3);	17			main.stackSize,	__$FP,	8,									
8	*no-op*	18				__$SP,	main.stackSize,	t150,								
9	*no-op*	19						t128,				t150,	__$RF(3),			t150,
10																t128,
1(20): $bb1																
0	t115=(__$FP+-4);	20														
1	*no-op*	21			-4,											
2	t116: *(t115)= 0;	22				__$FP,	-4,									
3	*no-op*	23			0			t115								

Figure 2- Contents of pipeline registers.

6.3.3.3 Data flow in pipeline stages

The "Data flow in pipeline stages" PV file shows the flow of values in pipeline. The main structure of this PV and the first three columns of the tables are as described in Section 6.3.3. The rest of the columns show what pipeline stages the data values are at in each clock cycle. Figure 3 shows the PV file for the example explained in Section 6.3.3.2. However, instead of showing the values for each register in the pipeline, this PV shows only the flow of data in the pipeline stages. For example, for the 3-address operation `t146=__$SP+4`, values of `__$SP` register and constant 4 are first loaded in pipeline stage 1 (registers `r1` and `r2`) and then the result goes through stages 2 and 3 (register `r3` and register file `RF`) in the next clock cycles. As always, the colors of data values correspond to the operations (in the second column) that is using or generating them.

main | storage bindings | CFG | DFG | *Methods list* |

clock	operation	0	1	2	3
0(10): $bb0					
0	t146=(__$SP+4);	10			
1	t143: *(__$SP)= __$LR;	11	4,		
2	t147: *(t146)= __$FP;	12	__$SP,4,		
3	*no-op*	13	t146,__$SP,__$LR,		
4	t125: __$FP:= __$SP;	14	t146,__$FP,		t146,
5	t150=(__$FP+8);	15			
6	t128=(__$SP+main.stackSize);	16	8,		__$SP,
7	t129: __$SP:= t128;t151: *(t150)= __$RF(3);	17	__$FP,8,main.stackSize,		
8	*no-op*	18	t150,__$SP,main.stackSize,		
9	*no-op*	19	t128,t150,__$RF(3),		t150,
10					t128,
1(20): $bb1					
0	t115=(__$FP+-4);	20			
1	*no-op*	21	-4,		
2	t116: *(t115)= 0;	22	__$FP,-4,		
3	*no-op*	23	t115,0,		

Figure 3- Data flow in pipeline stages.

6.4 Verilog outputs

The NISC tool-set also generates Verilog description of NISC processors for simulation and synthesis. In the online tool, the simulatable version is available. Synthesizable version may be provided upon request.

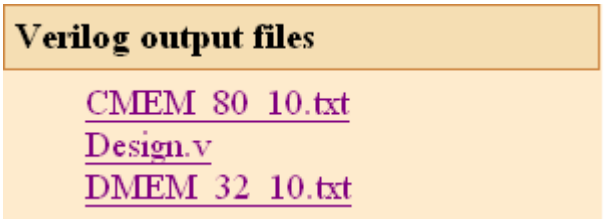


Figure 4. List of Verilog files generated for datapath of Section 4.2.1

The generated files include: contents of control/data memory, and Verilog code for controller, datapath, memories, and RTL components. Figure 4 shows the list of Verilog files generated for datapath of Section 4.2.1. The CMEM_80_10.txt and DMEM_32_10.txt contain the content of the Control Memory and Data Memory respectively, and they are generated according to a given application C code. The numbers in their filenames show the word size and address width of the required memories. The content of DMem_32_10.txt may be empty if no global variable is defined in the program.

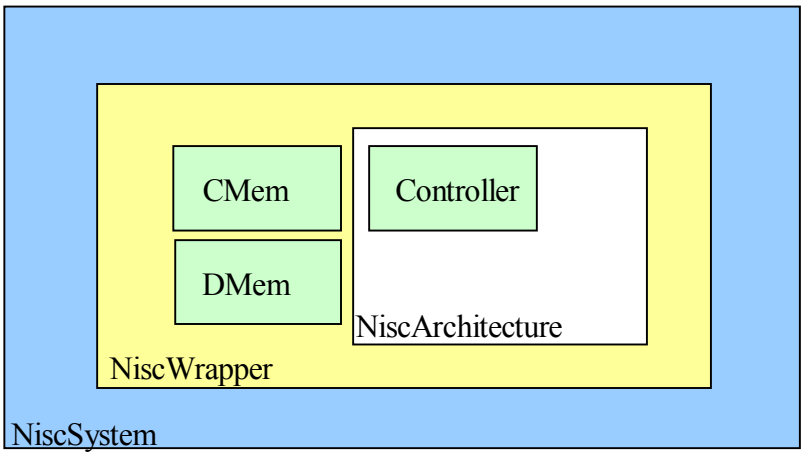


Figure 5. Hierarchy of main modules in Design.v

The *Design.v* file contains all the Verilog modules needed for simulation of a NISC processor. The top module in the design is *NiscSystem* which contains *NiscWrapper* module. The *NiscWrapper* contains the NISC architecture, data memory, and control memory. To simulate the design, a simple testbench file must also be created. Figure 6 shows the Verilog code of a sample testbench for *NiscSystem*. The *NiscSystem* has two input ports (i.e. *clk* and *reset*) and one output port (i.e. *halt*). The testbench simply drives the *clk* signal and resets the processor for one cycle. Then, the execution of program

starts. Finally, when the program finishes, the *halt* signal is set by the processor and simulation is terminated.

```
`timescale 1ns/1ps

module testbench();
  parameter halfClk = 10;
  parameter clock = 20;
  reg clk;
  reg reset;
  wire halt;

  integer clkCount;

  NiscSystem u1 (
        .clk(clk),
        .reset(reset),
        .halt(halt)
        );

  always begin
    #halfClk clk=~clk;
    if(clk == 1)
      clkCount = clkCount + 1;
  end

  initial begin
    clkCount = 0;
    clk = 0;
    reset = 0;
    #clock;
    reset = 1;
    #clock;
    reset=0;
  end

  always @(halt) begin
    if(halt ==1) begin
      $display("Halt!!!!!!!!!!");
      $display("No. of cycles:%d", clkCount);
      $finish;
    end
  end
endmodule
```

Figure 6- A sample test bench for simulation of NISC processors.

During simulation the trace of the Program Counter (PC) and memory read/write operations are displayed on the screen of the simulator tool. When simulation finishes, the final content of data memory is dumped into a text file called DMEM_32_10_dump.txt. Currently, the dumped file is used to check the correctness of produced results. The global variables defined in the C program are usually used to store

the outputs of the program. To check the produced outputs, first find their addresses in Storage Binding file produced by the NISC compiler. Then, read the content of the dumped memory file in that address. Storage Binding file can be accessed by clicking on “Control words” in “Compiler output files” section of the output page. Then click on the “Storage bindings” page.