Development of a Test Bench for VHDL Projects

Janos L. Grantner, Paolo A. Tamayo, Ramakrishna Gottipati, and Dave Florida

Department of Electrical and Computer Engineering
Western Michigan University
Kalamazoo MI 49008-5329, USA
janos.grantner@wmich.edu, p3tamayo@wmich.edu,
r0gottip@wmich.edu, david.florida@wmich.edu

Abstract

The objective of the course Digital Design (ECE355) is to develop the skills students need to design and verify digital systems using contemporary tools and devices. ECE 355 is a required course for students majoring in Computer Engineering. Along with combinational logic design, analysis and synthesis of both synchronous and asynchronous sequential circuits are covered in the course. The two major design projects are verified by simulation and implemented using Programmable Logic Devices (PLDs) and Field Programmable Gate Arrays (FPGAs), respectively, using breadboards.

Circuit designs are done in VHDL. In order to prepare the students to work with a professional development environment, the tools HDS Designer, ModelSim, and LeonardoSpectrum by Mentor Graphics along with the Xilinx ISE/WebPack are used to carry out the typical development tasks from describing the functions of the desired circuits in VHDL to physical implementation and verification.

In the graduate course Advanced Microprocessor Interfacing (ECE 605) students also work with the same set of development tools while taking on more challenging projects that are comparable to similar ones carried out in industry.

Prior to committing their designs to a CPLD or FPGA, students need to verify the performance of the circuits by developing a battery of simulation stimuli. A verification environment, referred to as a test bench, has been created to facilitate testing. The test bench is written in Verilog that encapsulates VHDL designs. Standard tests are included in the test bench as well as models of external devices, or controllers that the students' designs will interface to. The architecture and methodology of the test bench are discussed in the paper in detail.

In the first part of the paper, we briefly outline the key concepts to develop a verification methodology for teaching and research in the digital systems design area. The second and third sections of the paper focus on the test bench and provide for a few examples on how to use it. The fourth part of the paper concludes with a future assessment plan.

1. Introduction

The main focus of any digital design class is the theory and concepts behind the analysis and synthesis procedures and methods. However, there is a need to spend increasingly more time and effort on how to deal with contemporary design tools. Today's digital systems and the methodologies to develop them are rather complex. These are due mainly to the use of hardware design languages such as VHDL and Verilog and the very large programmable logic devices to develop and implement the systems. With the time constraints of a 3-credit hour, single semester course, it is often quite difficult to fit the necessary theory into the schedule along with a suitable coverage of the development tools such that students will be able to work with them with confidence. Aside from learning the theoretical aspects of designing digital systems students should also learn the syntax of the hardware design language, develop skills to create suitable stimuli to test their designs, and learn to use the synthesis and simulation tools.

The development of Tutorials on how to use the various tools helps in addressing these problems. However, the use of a standardized design and design verification methodology can help even more. The objective of an integrated design and design verification methodology is to establish a common test bench in which circuit designs and device models can be treated as components. In the test bench developed, Mentor Graphics electronic design automation (EDA) tools are invoked to carry out the simulation and debug tasks for the designs. The verified designs for the projects in the ECE 355 and 605 classes, respectively, are downloaded into Xilinx's FPGA and CPLD devices for demonstration.

2. Design Verification Methodology for Teaching and Research

The development of a design verification methodology creates a logical process that can be easily adapted to any digital design. The verification methodology creates an environment that facilitates testing. Using coding guidelines and proper partitioning of modules the architecture for the design is laid out. This makes the design verification environment generic. Along with device models and utilities, tests can be created easily to check particular functions of the designs.

Our verification methodology has been developed to test any digital design. Applications ranging from hierarchical, large-scale designs to single-level modules can be verified. Tests can be designed to run on module-level as well as system-level. This is a very important characteristic of verification when testing bus-based systems. In a classroom setting, simple projects are usually assigned. By using the methodology and verification architecture, a test bench can easily be created along with its required models and utilities. The methodology developed is generic and tool independent since it is implemented using industry standard languages (Verilog and VHDL). Coding has been done using Mentor Graphics HDL Designer for design entry and ModelSim for simulation. Synthesis has been carried out using Xilinx's Project Navigator for implementations on FPGA and CPLD devices.

The following discussion describes the methodology of creating the test bench. The methodology emphasizes the reuse of modules and tests. First the architecture is given, then it is followed by the detailed description of each module. Sample codes are provided that illustrate the coding guidelines supporting the reusability of the modules.

2.1 Verification Architecture

To facilitate easy assembly of the verification environment, the structure shown in Figure 1 is utilized. The Verification Architecture is composed of the following modules: the Test Module (Test Stimuli along with the Test Utilities) drives the Model and the DUT (Design Under Test) modules. A separate (optional) Monitor Module helps in observing the behavior of the DUT. The verification test bench is developed in Verilog. The DUT can be a VHDL module or a Verilog module. The use of Verilog for creating the test bench permits the driving of signals between Verilog modules without creating physical signals. This gives the verification tasks a great degree of flexibility to devise test sequences.

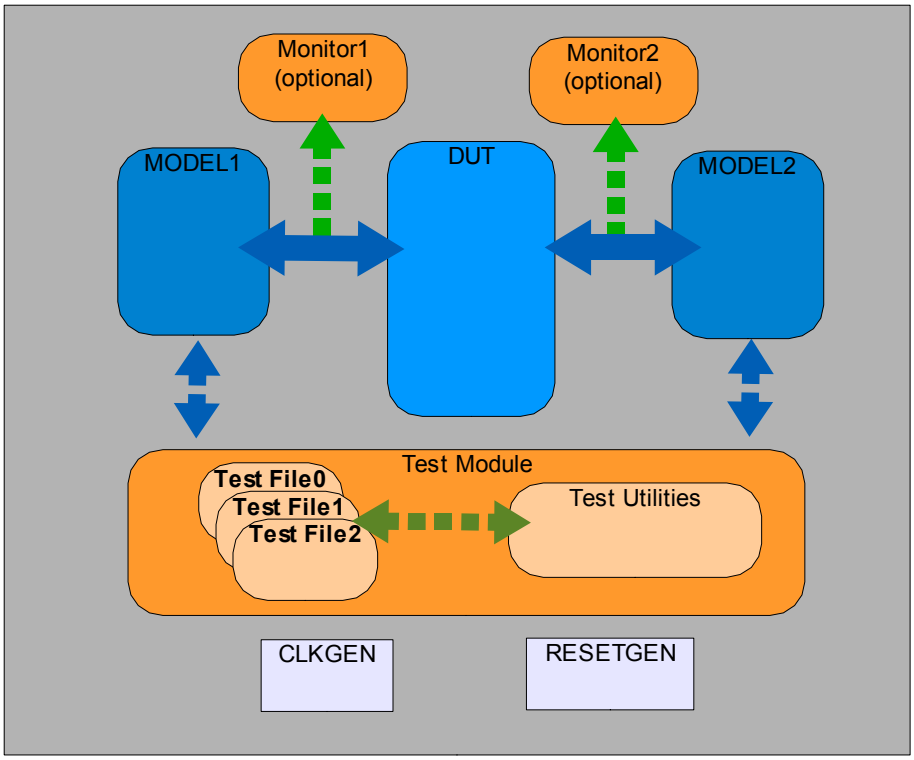

Figure 1. Verification Architecture

From Figure 1, one can see that the DUT's inputs and outputs can be driven and sampled, respectively, only by the models. The Test Module (through the predefined model utilities and functions) can access the models. By doing this, tests are devised without any dependency on the DUT but rather on the defined utilities and functions of the models. Hence, the models are generic. To connect a model to the DUT, an instance file is created. The file contains an instance of the model with the appropriate connections to the DUT. By adding the instances of the monitors and test modules the top-level module of the test bench is created. This file is unique to each new test bench since it contains the actual name of the ports of the DUT. The ports of the DUT make the physical connections to the generic and reusable models and tests.

The Test Module is made up of the Test Files and the Test Utilities. The Test Utilities are generic tests that can be used by the Test Files. The Test Files are developed to verify the

different functionalities of the DUT. Each test can focus on one particular feature of the DUT. This creates an opportunity for team verification to facilitate partitioning of verification tasks.

The Monitors are optional modules that come handy to catch protocol violations such as illegal bus timing and illegal signal combinations. The Reset Generator (RESETGEN) and Clock Generator (CLKGEN) modules are also generic modules that can be tuned to the requirements of the DUT. For example, by changing a parameter on the instance of the CLKGEN module the frequency of the clock can be configured.

Module Name	*Description*
DUT	This is the design that will be tested. It is possible that two or more DUTs are verified.
Models	Models are the modules created specifically for the DUT. They can generate the stimuli for the DUT or respond to the DUT to emulate the modules or devices interfaced to the DUT. These modules can also help in observing the DUT.
Test Module	It is the module that instantiates the Test Files and the Test Utilities.
Test Files	The actual tests created to start the stimuli from the models or from the DUT. The tests are written to emulate a real world scenario. They can vary from a simple test of a read or write sequence to an algorithmic test data that are fed into the inputs of the DUT by using the models or a complex sequence of events and scenarios.
Test Utilities	Utilities are created to aid testing and increase confidence in the design. Utilities can generate error messages, count the errors and point to the location of the error in time. Utilities can also be created to generate recurring sequences that are essential for the testing of the operation of the DUT. For example, since the reset of the DUT may require some special timing, a task can be written to accomplish that. It will then be placed in the Test Utilities. The reset task can then be conveniently called in the tests.
Monitors	Like the Test Utilities, Monitors can be added to the architecture to aid the verification of the DUT. Creating monitors can simplify the development of the tests since monitors have the ability to generate error messages whenever they encounter a problem in the signals being monitored. For example, if the DUT did not respond to the model, a monitor checking for bus inactivity would flag the error and inform the test that there is a bus timeout. The test can then terminate the sequence or carry out a system reset to recover and continue the tests on other items. This prevents a hang-state in testing.
Clock Generator and Reset Generator	These are essential modules in the system. They are, actually, special cases of models. They are modeling the clock module (for synchronous designs) and the reset module. These modules are dependent on their DUTs.

Table 1. Summary of Verification Architecture Modules

2.2 Models

Models are modules that emulate actual devices and modules that are interfaced to the DUT. A model can be a memory device, a memory controller or another bus master in a bus-based design. Models can be created along with utilities that can assist the verification. They are included in the architecture to make the test easier rather than to create another bottleneck in verification. For example, it is very helpful if a task can be called from the test to return the current contents of a memory device by giving the start address and the number of bytes. Through this task, actual data that are sent to the memory can be automatically checked by

comparing them with the expected ones during runtime. There will be no need to check the results after the simulation is completed.

In the case of an example for a bus master (a simplified MC68040V processor model), providing read and write tasks in a bus-based design helps in creating the tests. The Test Files can use the read and write tasks by calling them in the tests while providing the address of the memory location and data in the case of a write command. In the case of the read, the task will return the data read from the address provided. The following code shows an implementation of a 16-bt word read task.

```
task read_word;                          // word (16-bit word) read access task, this task
     input  [31:0] address;              // emulates the word access by a bus master
     output [15:0] read_data;
     begin
        @(posedge clk)                    // start of read access
          begin
             reg_rnw = 1'b1;              // access is a read access
             reg_addr = address;          // the address of the data in memory
             reg_ts = 1'b0;               // transaction start strobe
             reg_siz = 2'b10;             // size of transaction (one word)
             reg_tm = 3'b001;
             reg_tt = 2'b00;
             reg_mi = 1'b1;
          end

        @(posedge clk)                    // deassert transaction start strobe
          begin
             reg_ts = 1'b1;
          end

        @(posedge ta)                     // slave device acknowledges the transaction
          begin                           // check if data is from upper or lower word
                                          // (bus is 32-bits long)
             if ((address & 32'h0000_0002) == 32'h0000_0000)
               read_data = data[31:16];
             else
               read_data = data[15:00];
             reg_addr = 32'h0000_0000;  // removes the driven address on the bus
             $display(" Time %0d ns: Bus Word Read Access at address:%h
                      with data:%h", $time, address, read_data);

          end
     end
  endtask // read_word
```

In the test, the code below calls this task. The cpu_model is the instance name of the processor model. Note that the data is provided in the write_word task and the test_word variable is provided for the read_word task. The variable test_word is declared as a 16-bit long register.

```
cpu_model.write_word(32'h7001_FF00, 16'h0023);  // address is 0x7001_FF00 and data is 0x0023
cpu_model.read_word(32'h7003_1110, test_word);  // address is 0x7001_1110 and data read is
                                                // placed in test_word
```

Model instances are created to connect the models to the DUT. As mentioned earlier, these are usually done in the topmost module of the verification architecture.

2.3 Monitors

Models help in the verification by generating stimuli that drive the input signals of the DUT, or respond to the DUT's output signals. Monitors, on the other hand, help in observing the DUT (and the Models). They can serve as another check for the Model and DUT interface. Monitors are passive modules that are attached to the interface signals of the DUT, the Model, or between two DUTs communicating with each other. Monitors are usually implemented to observe bus protocols like monitoring bus master access to a slave device, or measuring the performance of bus designs. Monitors are also effective for checking the timing requirements of signals. A monitoring module may watch the signals and flag the test whenever a violation of the timing constraints is detected.

2.4 Test Module

The Test Module (an example is shown below) consists of two main sub-modules. These are the Test Files and the Test Utilities. The Test Module contains the instance file of the two sub-modules and the initial statement that runs the tests. This file grows in size as more tests are added to run simulations.

```
`include "test_util.v"            // file that contains the test utilities
`include "memory_test.v"          // file containing mem_test()
`include "access_test.v"          // file containing acc_test()

module Test_Module();

   test_util util();              // instance of the utilities
   memory_test mem_test();        // instance of the memory test
   access_test acc_test();        // instance of the access test


   initial
     begin

        mem_test.test();           // run the memory test
        acc_test.test();           // run the access test
        util.displayError();       // display in the log the errors and warnings

       $finish;                   // terminate the simulation

     End

endmodule
```

The test files can be implemented in a single file, or in separate files. It is easier to maintain separate files than managing a big one that contains all the tests of a module. Debugging is also easier with multiple files. In larger designs, test files can be partitioned to specific modules and each test file contains a series of tests for a specific module. The tests run sequentially with mem_test being first and it is followed by the acc_test.

2.5 Test Utilities

Test Utilities include generic tests and useful modules for verification. The displayError task shown in the sample code above is one example for a test utility. The number of errors that the tests have encountered is tracked. This provides a feedback for the verification if identified errors have occurred when the test was simulated. Tracking of the error count can be done by another utility such as the dutError task. This task is used by the test whenever a value or a condition different from the expected one is encountered. The code below illustrates the usage of such mechanism.

```
cpu_model.read_word(32'h5000_0010, test_word);
   if(test_word != 16'h1234)  // where 16'h1234 is the expected value
      util.dutError(1);       // Error utility is called if the data are not the same
```

The value passed to the dutError task can be defined such that either a warning count or an error count will be incremented. This provides versatility for tests through classifying errors into different types. Errors caused by minor problems can be classified as warnings, some others as errors, and more critical ones as fatal errors. For fatal errors, simulation can be halted and the problem should be fixed immediately. For warnings, simulations can continue and fixes can be integrated into the design later in time for another simulation run. One implementation of a test utility is given below where the dutError and displayError tasks are defined.

```
module test_util();
   integer dut_error;                // Error counter
   integer dut_warning;              // Warning counter

   initial
     begin
        dut_error = 0;                // initialized to zero
        dut_warning = 0;
     end
   // function to update the error and warning count
   task dutError;
      input level;
      begin
         if (level == 1)              // level defined to Error
           begin
              dut_error = dut_error + 1;
              $display("Time %0d ns: DUT ERROR = %0d", $time, dut_error);
           end
         else if (level == 0)         // level defined for warning
           begin
              dut_warning = dut_warning + 1;
              $display("Time %0d ns: DUT WARNING = %0d", $time, dut_warning);
           end
         else
           dut_warning = dut_warning; // if not defined maintain the current count
           dut_error = dut_error;
      end
   endtask // dutError

   // function to display the error and warning count
   task displayError;
      begin
         $display("DUT ERROR = %0d, DUT WARNING = %0d\n", dut_error, dut_warning);
```

```
      end
   endtask // dutError

endmodule;
```

Note that the simulation time is displayed in the simulation log whenever the dutError task is called. This saves a lot of time in checking the output file of the simulation and identifying where and when the error has occurred. Utilities such as these can also be used in monitors already mentioned.

Common tests can be identified beforehand and defined in the test utilities. This enables the author of the test to concentrate on the test scenario rather than the detailed implementation of the test. For example, a command to read a memory location is always followed by a comparison with the expected value of the data read. One can create a task in the test utility that carries out the read of the memory location along with a comparison with the expected data provided. The call to the dutError can also be incorporated in the task. However, caution must be observed when doing this type of code development since the code tends to be unusable for other designs because the utilities are now dependent on the tasks of the models. A way to separate the utilities for a specific DUT from the generic utilities is to create another module that contains the DUT specific utilities and instantiate them in the test module. This is illustrated in Figure 2.

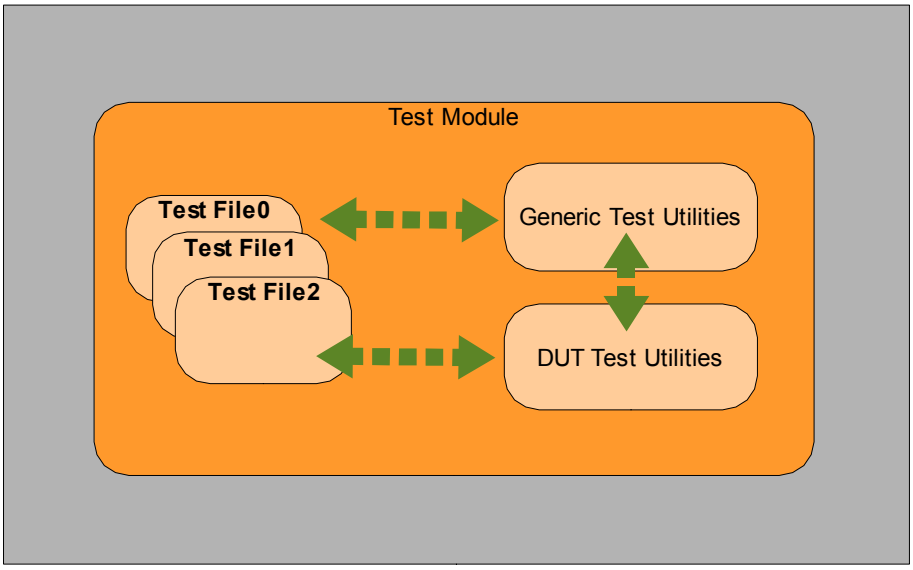

Figure 2. Test Module Implementation

2.6 Test Files

The Test Files are the test scenarios developed to check the functionality of the DUT. The tasks of the Models, Monitors and Test Utilities are combined to create the required sequences of events that will emulate real world scenarios. Tests are grouped into files and each file can focus on a particular feature of the DUT. For example, an SDRAM controller can be tested on its address encoding and access types. The operation of the data bus will be inherently tested when these two function blocks are checked.

It is important to observe the way the tests are coded. One can see from the discussion of the Test Module above that the test files are instantiated in that module. The task named test( ) is then called when the test is run. The task name must be the same in the test file to facilitate easy maintenance of the Test Module. It is illustrated in the sample code below. The code is taken from a bus-based design where the address decoding is checked. The memory location is written with arbitrary data and the same data is read back and compared with the expected value.

```
module test_file1();

   reg [15:0] test_word;

   initial
     begin
        test_word = 16'h1234;
        read_word = 16'h0000;
     end

   task test;
      begin
          $display("********* Time %0d ps: Test0 Started      *********\n", $time);
         resetGen.reset(150);
         cpu_model.write_word(32'h7001_FF00, test_word);  // write the word to memory
         cpu_model.read_word(32'h7001_FF00, read_word);   // read the written data
         if(test_word != read_word)                       // check if data was written
           util.dutError(1);                              // call error util if not
         #20 $display("********* Time %0d ps: Test0 Finished    *********\n", $time);
      end
   endtask // test
endmodule
```

3. Verification Methodology

A coherent and systematic creation of tests can be achieved with proper verification planning. When coded properly, the test files can be reused from block level verification to higher system level verification for larger systems. Figure 3 shows a procedure that must be followed to achieve this. The steps are given below.

Step 1
Develop test plans and identify the modules needed for verification from the specifications of the design. These modules are the monitor, models, test utilities and tests.

Step 2
Assemble the test bench from the coded and reused models, monitors and test utilities.

Step 3
Create the coverage list. It is just an enumerated list of features of the DUT. Each test file to be devised will usually cover for at least one item in the list.

Step 4
Develop the tests on the basis of the test plan and the coverage list.

Step 5
Run simulations and verify the results. Check to see whether all the items in the coverage list created in step 3 have been verified. Add tests if necessary and simulate again.

Sometimes, it is also necessary to modify the verification plan to achieve a higher degree of verification confidence in the DUT. This can occur when undefined DUT features are discovered during the verification coding, simulation and debugging. It is illustrated by the dashed line in Figure 3.

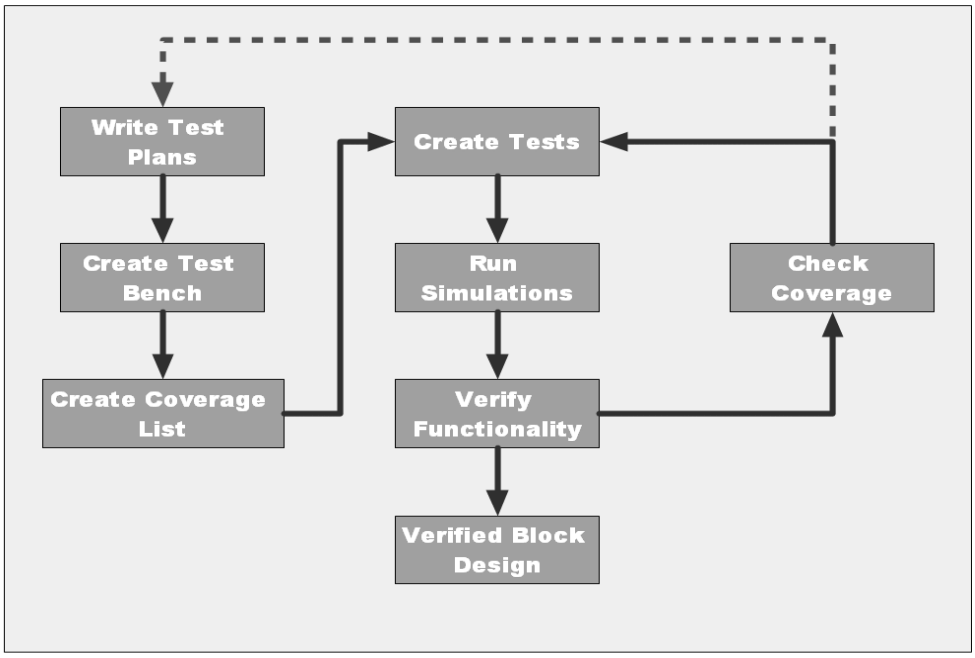

Figure 3. Verification Procedure

4. Assessment Plans

This test and verification methodology was not integrated in the courses ECE 355 and 605 in the Fall of 2004 because of time constraints. The authors are planning to deploy the test benches in the Fall 2005 Semester. The same questions that were taken from the ICES Catalog [7] in the Fall 2004 Semester will be used again to assess the impact of the test benches on the quality of the classes and its usefulness for the students to achieve their professional goals in the area of computer engineering. The authors are also planning to create a follow up paper to report on the lessons learned from the students' feedback and any potential improvements made in the methodology.

5. Conclusions

Today's complex design projects require an integrated approach for both development and testing. The creation of a verification methodology saves time in the development of test benches to check the design. By focusing on design reuse, future projects can be developed faster because the time needed to devise a test bench and the tests will be reduced. The use of proper

partitioning and coding techniques promotes the creation of generic modules that can be reused. Tests, when properly coded, can be applied for both module-level and system-level verification. Thanks to the industry standard Verilog and VHDL languages, the methodology presented is tool independent and portable to any development platform supporting them.

References

[1] P. Tamayo, HDL Testbench, Project Report in Problems in Electrical and Computer Engineering (ECE 697), Department of Electrical and Computer Engineering, Western Michigan University, Fall 2003 Semester.

[2] http://homepages.wmich.edu/~grantner/ece355 – ECE 355 Digital Design class Web page

[3] http://homepages.wmich.edu/~grantner/ece605 – ECE 605 Advanced Microprocessor Applications class Web page

[4] http://www.mentor.com/products/fv/abv/modelsim_se/index.cfm – Mentor Graphics compiler and simulator

[5] http://www.mentor.com/products/fpga_pld/hdl_design/hdl_designer_series/index.cfm – Mentor Graphics design entry tool

[6] http://www.xilinx.com – Xilinx synthesis and implementation tool

[7] The ICES Catalog, Divison of Measurement and Evaluation, University of Illinois at Urbana-Champaigne

Biographical Information

JANOS L. GRANTNER

Janos L. Grantner is a Professor of Electrical and Computer Engineering at Western Michigan University. Dr. Grantner received the Ph.D. degree from the Technical University of Budapest, Hungary, in Computer Engineering, and the advanced doctoral degree Candidate of Technical Science from the Hungarian Academy of Sciences, in Computer Engineering, respectively.

RAMAKRISHNA GOTTIPATI

Ramakrishna Gottipati is a Doctoral Student in the Department of Electrical and Computer Engineering at Western Michigan University. Mr. Gottipati received the MS degree from Western Michigan University, in Computer Engineering.

PAOLO A TAMAYO

Paolo A. Tamayo is a Masters Student in the Department of Electrical and Computer Engineering at Western Michigan University. Mr. Tamayo received the BS degree from the University of the Philippines, in Electrical Engineering.

DAVID FLORIDA

David Florida is the Lab Supervisor of Electrical and Computer Engineering at Western Michigan University. Mr. Florida received the MS degree from Western Michigan University, in Electrical Engineering.