MacTester: A Low-Cost Functional Tester for Interactive Testing and Debugging

Carl Ebeling and Neil McKenzie
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195

Abstract

We describe a low-cost, functional tester for the MacII that allows students to test and debug digital systems interactively. This tester provides a large number of programmable I/O signals and can be used to test chips, boards and subsystems. Test programs are easily written using a simple, intuitive interface and we expect a variety of interactive graphical testing and debugging environments to be built for the tester. We plan to make this tester generally available in kit form.

Introduction

Testing design projects in the University is typically done in an ad hoc way. Some sort of hardware environment is designed around the project to create a self-contained system which can be operated, observed and debugged using a logic analyzer or oscilloscope. Testing and debugging a partially completed system or a subsystem that operates in a complex environment can be very difficult. As the complexity of projects grows with the complexity of off-the-shelf parts and the capability of design tools, the problem of testing is becoming even more difficult. This is especially true for custom VLSI chip designs. Although commercial testers provide the necessary functionality, their cost is prohibitive and the usual testing environment is primitive.

The goal of the MacTester project is to remedy this situation by providing a powerful, interactive environment for functionally testing digital systems. Functional testing means that the *values* and *sequencing* of the I/O signals can be verified but that precise *timing* cannot. In particular, a functional tester cannot be used to determine how fast a system can run. While speed testing is clearly important, testing and debugging concerns tend to dominate in the classroom laboratory situation. We believe that the issues of functional testing and debugging can be separated from those of timing validation.

There are three styles of testing: offline, online and interactive. The usual method is offline testing where a list of test vectors is prepared and presented offline to the device under test (DUT). The resulting responses are collected and analyzed after the test is completed. Online testing is performed in real time where each test vector is generated and the response analyzed before succeeding test vectors are generated. Interactive testing allows the user to interact with the DUT on a vector by vector basis. The user can inspect the results of the current test vector to determine the next vector. Efficient debugging is facilitated by a good interactive testing environment. The only drawback to interactive testing is that it cannot be used for circuits with dynamic state.

The MacTester has been designed to allow all three types of testing, although we view interactive testing and debugging as the most important. The Macintosh computer is an ideal host for interactive testing and debugging because of its uniform, intuitive user interface and interactive graphics capability. The tester and the software interface provide the mechanism

with which a wide variety of testing environments can be built. We expect many different graphical and procedural interfaces to the tester to be built as users define their own testing needs.

Cost and functionality were often competing goals in the design of the MacTester. We decided to provide the following features that we felt were necessary for a tester used in design laboratories:

• The tester is easily shared by more than one project. Changing the test setup from one project to another takes only the few minutes needed to change the power and ground jumpers. (A pre-test can be performed to ensure that the power and ground setup matches the specification.) This ability to share reduces the number of testers that are needed and the overall cost.

• The tester provides a large number (128) of test signals so that even large projects can be accommodated.

• All signals are bi-directional and can be individually set to the input or output state. Although this makes the tester implementation more expensive, it facilitates sharing between projects since all pins can be redefined in software. Moreover, since the direction can be set dynamically, tri-state busses can be tested.

• Provision has been made for dynamic circuits by including an on-board memory capable of storing several thousand test vectors. The tester can independently present these vectors at a rate of approximately 1 MHz. Although dynamic circuits cannot be tested in a truly interactive fashion, the tester software includes a construct that allows the user to maintain the illusion.

We will begin by describing the software interface to the tester to show how the user views the device under test. We then outline the implementation of the hardware and the software.

The Tester Interface

In this section, we describe how the device under test (DUT) appears from the point of view of a test program and how the program manipulates the I/O signals. The tester software provides a simple, intuitive interface that makes it easy for programs to use the tester. These programs can range from simple test programs devised by the user to sophisticated interactive programs that supply a visual interface via schematics and timing diagrams. A simple test program for a combinational multiplier is shown in Figure 1.

To the user, the pins of the DUT appear as variables in the test program. The user first defines a set of signal variables in the `DefineSignals` section, each as a collection of pins, and then manipulates the values of the pins via these variables. Changing the value of a signal variable changes the values of the associated pins and reading the value of a signal variable reads the values of the pins. Although most signal variables are declared as input or output variables, they can also be declared to be bi-directional and the direction changed at any time.

A test proceeds as a series of test steps, each activated by the `Next` control statement. In each test step, a new set of values is first driven in unison onto the DUT input pins and then the resulting values of the DUT output pins are latched. These events are separated by enough time to allow the output values to become stable before they are sensed. In fact, the tester provides a very limited version of speed testing by allowing the user to set this time interval. The delay from the clock driving the input values to the clock latching the output values can be preset to any value from 20 ns. to 1 μsec. in increments of about 5 ns.

```
/* Define the chip signals */
DefineSignals
   Signal(Multiplier, "2,4,6,8,10,12,14,16", INPUT, 0);
   Signal(Multiplicand, "3,5,7,9,11,13,15,17", INPUT, 0);
   Signal(Result, "25:18", OUTPUT);
EndSignals

main()
{
   BeginTest;
   for (i=0; i<256; i++) {
      for (j=0; j<256; j++) {
         SetSignal(Multiplier, i);
         SetSignal(Multiplicand, j);
         Next;
         if (GetSignal(Result) != i*j) {
            printf("error: %d * %d ==> %d\n", i, j, GetSignal(Result));
         }
      }
   }
   EndTest;
}
```

Figure 1: *Test program for a combinational multiplier.*

The values of signals are changed via the `SetSignal` statement, and the values accessed via the `GetSignal` statement. `SetSignal` statements do not take effect immediately, but the values specified are collected and applied to the pins of the DUT by the `Next` sequencing statement, which performs all accumulated changes in parallel. The `GetSignal` statement returns the value of the signal latched by the most recent `Next` statement. Thus a step in the test program typically consists of a set of `SetSignal` statements followed by a `Next` statement followed by a set of `GetSignal` statements.

The direction of a signal declared in the `DefineSignal` statement can be `INPUT`, `OUTPUT`, or `BIDIRECTIONAL`. The direction of bi-directional signals can be changed at any time via the `SetDirection` statement. These changes are accumulated like signal value changes and applied by the next `Next` statement.

Each test step can be thought of as a test vector, except that each test vector is created on the fly by the program and the resulting response is available immediately and can be used to generate the next test vector. This makes testing truly interactive from the point of view of the test program and this view can be passed along to the user.

The tester provides no signals other than those provided by the user program. In particular, the test program must supply the clocks by changing a signal variable or set of variables explicitly. Thus a single clock cycle for a circuit using a two-phase non-overlapping clock requires at least four test vectors. Figure 2 gives a test program for a single-stage pipelined multiplier.

```
/* Define the chip signals */
DefineSignals
   Signal(Multiplier, "2,4,6,8,10,12,14,16", INPUT, 0);
   Signal(Multiplicand, "3,5,7,9,11,13,15,17", INPUT, 0);
   Signal(Result, "25:18", OUTPUT);
   Signal(Phi1, "26", INPUT, 1);
   Signal(Phi2, "27", INPUT, 0);
EndSignals
ClockChip()    /* Perform a single Phi1/Phi2 clock cycle */
{
   Next;       /* Assert data before clocking */
   SetSignal(Phi1, 0); Next;
   SetSignal(Phi2, 1); Next;
   SetSignal(Phi2, 0); Next;
   SetSignal(Phi1, 1); Next;
}
static int lastX, lastY, lastResult;      /* Previous test */
initmult()  /* Fill the pipeline */
{
   SetSignal(Multiplier, 0);
   SetSignal(Multiplicand, 0);
   ClockChip();
   lastResult = lastX = lastY = 0;
}
testmult(x, y) /* Perform one test of the multiplier */
int x, y;
{
   SetSignal(Multiplier, x);
   SetSignal(Multiplicand, y);
   ClockChip();
   if (GetSignal(Result) != lastResult) {
      printf("Error: %x * %x => %x[%x]\n", lastX, lastY,
         GetSignal(Result), lastResult);
   }
   lastResult = x * y;
   lastX = x;
   lastY = y;
}
main()
{
   BeginTest;
   initmult();          /* Fill pipeline */
   for (i=0; i<256; i++) {
      for (j=0; j<256; j++) {
         testmult(i, j);
      }
   }
   testmult(0, 0);      /* Flush pipeline */
   EndTest;
}
```

Figure 2: *Test program for a single-stage pipelined multiplier.*

Dynamic and Pseudo-static Circuits

Dynamic circuits cannot be tested interactively because the dynamic state is lost unless the circuit is operated continuously. Since the time for which dynamic state can be held is typically much less than a second, user interaction is not possible. Online testing may also be ruled out if the time between test vectors cannot be bounded. In particular, programs running under A/UX can be interrupted by the operating system for relatively long periods of time.

The MacTester accommodates dynamic circuits via onboard test vector memory which permits offline testing. In online mode, the `Next` statement causes accumulated values to be written directly to the DUT pins and the response to be read back. In offline mode, the test vectors generated by the `Next` statement are stored in test vector memory and presented all at once, independently from the test program. The responses are accumulated in the test vector memory and can be read back after the offline test has completed.

```
testmult(x, y)    /* Perform one test of the multiplier */
int x, y;
{
   SetSignal(Multiplier, x);
   SetSignal(Multiplicand, y);
   ClockChip();
   Verify
      if (GetSignal(Result) != lastResult) {
         printf("Error: %x * %x => %x[%x]\n",
                lastX, lastY, GetSignal(Result), lastResult);
      }
   EndVerify
   lastResult = x * y;
   lastX = x;
   lastY = y;
}
main()
{
   BeginTest;
   for (i=0; i<256; i++) {
      BeginDynamic;
      initmult();       /* Fill pipeline */
      for (j=0; j<256; j++) {
         testmult(i, j);
      }
      testmult(0, 0);   /* Flush pipeline */
      EndDynamic;
   }
   EndTest;
}
```

Figure 3: *The modified test program for a dynamic one-stage, pipelined multiplier.*

The tester software package contains a *dynamic block* construct that is used to maintain the illusion that the test program is online when in fact it is not. A dynamic block is created by wrapping an arbitrary set of statements with `BeginDynamic/EndDynamic` statements. A dynamic block is executed twice. In the first, *generate*, phase, all the test vectors generated within the block are accumulated and presented offline when the end of the dynamic block is encountered. In the second, *verify*, phase, the response vectors are available in the right sequence and statements that access these are executed.

Clearly this places restrictions on the program statements in the dynamic block since some should be executed only in the first phase, and others only in the second phase. The `SetSignal` statement is defined to have effect only in the generate phase -- other statements that are to be restricted to the generate phase can be protected using the `Generate` macro. The `Verify` macro is used to protect statements that are executed only in the verify phase.

The test program in Figure 2 can be changed to test a completely dynamic pipelined multiplier as shown in Figure 3. The dynamic block has been placed as shown because the test vector memory is limited to about 5300 vectors. Note that if the `Verify` protection macro is left out, the program will produce errors during the generate phase because the result of `GetSignal` is undefined during this phase.

Dynamic circuits can certainly make test programs more complicated. Fortunately, most dynamic circuits can survive online testing since if the computation required to generate each test vector is limited, online testing can proceed at about 50 KHz. This is true for programs under MacOS, but we were pleasantly surprised to find that test programs running under A/UX generally worked as well.

Pseudo-static circuits, that is, circuits that are static at some point in the clock cycle, can be tested interactively. In this case, the dynamic block is limited to a few statements and the test program can be interactive at the granularity of a clock cycle rather than each test vector. For example, if the multiplier in Figure 3 were pseudo-static, the dynamic block would be confined to the `ClockChip` procedure.

Implementation

A very high-level block diagram of the tester is shown in Figure 4. During one step of an online test, the appropriate level 1 registers are first written by the host. These values are then transferred in unison to the level 2 registers which are connected directly to the DUT pins. The level 3 registers are then latched after some user-specified delay. Finally, the results in the level 3 registers are read back by the host.

Offline testing is performed by first downloading the test vectors into the test vector memory and indicating the start and end address of the sequence. The tester then performs the offline test by transferring test vectors to the level 1 registers and results from the level 3 registers back into test vector memory. When the end address is encountered, a done bit is set in the interface control register and the host can upload the responses. The offline test can also be placed in a loop so that a particular test sequence can be monitored with an oscilloscope.

The hardware design was simplified by the use of Xilinx programmable gate array chips. The data path is implemented in six XC3020 chips and the control logic in a seventh. Although the implementation would have been possible using fewer of the larger Xilinx gate arrays, the cost per pin grows almost quadratically, and the XC3020 was chosen as the most cost-effective size. The choice of Xilinx gate arrays allows the data path and control to be changed by the user via

the Xilinx design software. For example, the programmable delay was implemented after the fact by using a multiplexor to specify the number of logic levels between the drive and latch clocks.

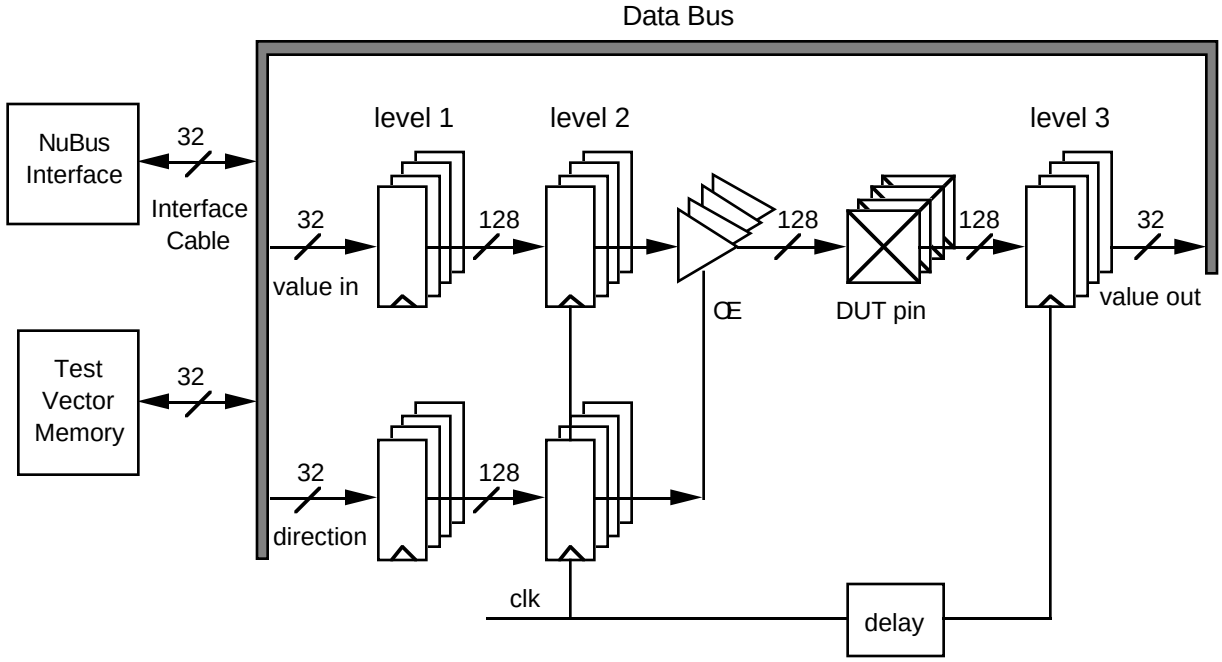

Figure 4: *The overall block diagram of the tester. The data path is 128 bits deep.*

The test vector memory is potentially a large and expensive part of the tester. We chose to trade cost for speed by multiplexing the RAM six ways. This reduces offline testing rate from a possible 5MHz to just under 1MHz, but reduces the number of static RAM's from 48 to 8 and reduces the number of I/O pins required for the datapath.

The tester is implemented as an 8x13 inch PC board as shown in Figure 5. This board is connected via a simple parallel interface to an ADEX NuBus interface board that plugs directly into the MacII. This interface is quite simple and interfacing the tester to other busses like the SBUS or VME bus is straightforward.

Software Interface Implementation

At the lowest level, the tester control and data registers are mapped directly into the address space of the user program. This is possible for both MacOS and A/UX programs and greatly simplifies the interface implementation. At the lowest level, a set of macros is provided for accessing these registers and for performing all the tester control operations. In addition, memory is allocated for collecting signal values and directions for each signal variable defined.

At the next level, user-visible operations are implemented using the low-level macros and interface memory. The `SetSignal`, `SetDirection` and `GetSignal` operations are implemented in two different ways depending on when the signal variables are defined. If they are defined dynamically, then these operations must be interpreted. Otherwise they can be pre-compiled into a much more efficient implementation.

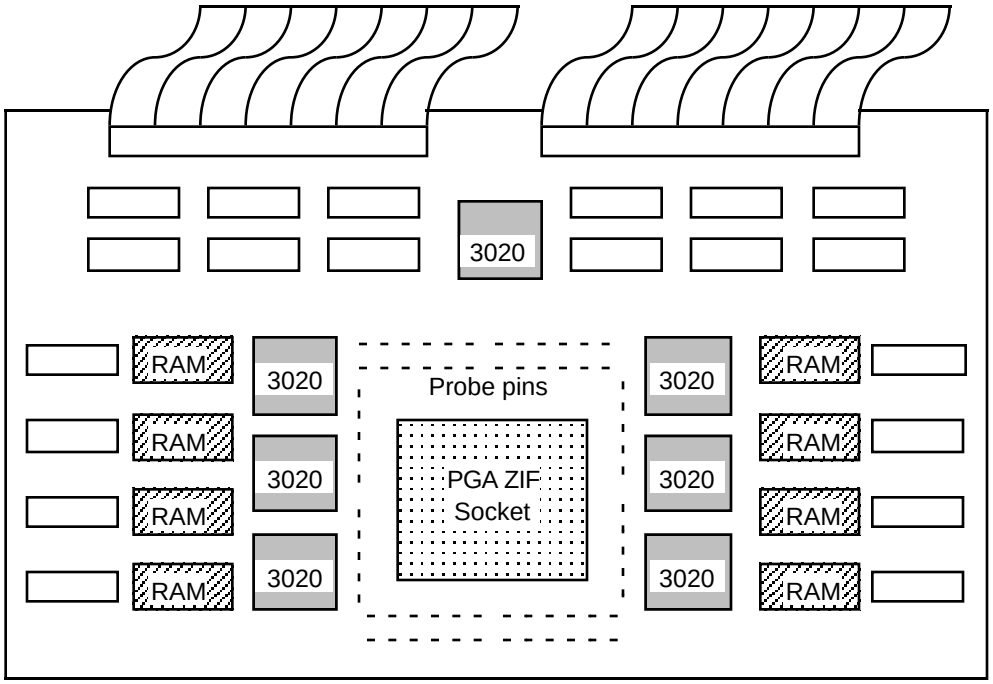

Figure 5: *Outline of the tester PC board.*

There is a wide range of possible testing and debugging environments that can be built on top of the MacTester. First, as shown in the sample programs, writing test programs is straightforward. In fact, a program is a very powerful way to specify the behavior of the environment of the system being tested. One such technique is to write a procedure that takes an abstract operation and maps it into the sequence of signal values that implements that operation. For example, if testing a dynamic RAM, one would write procedures for the read and write operations that generates the multiplexed address and the appropriate RAS/CAS. An effective way to familiarize students with the operation of the more complex off-the-shelf parts is by having them write such test programs.

We expect a variety of graphical testing environments to be developed as well. In such an environment, the inputs and outputs of the device being tested can be displayed in separate windows and formatted in different ways, for example as timing diagrams or a time line of values. For example, Capilano has graphical simulation tools which can interface to other software using what is called the Meda interface. Using these tools, the tester can be incorporated into the graphical simulation as a separate device. All the graphical I/O devices such as keypads, displays, and timing diagrams that are provided by the Capilano simulation tools can then be used to test and debug one's project in the tester. Moreover, the project can be simulated as part of a larger system as described by schematic drawings.

We also plan to use the tester in the RNL simulation environment. RNL provides an interactive Lisp environment for testing MOS circuits. First, the simulation can be replaced by the tester and the interactive Lisp environment used to test the project. Second, if there is a circuit description for the project, then the same program used to test the design can be used to test the finished product. Moreover, the simulation and the tester can be run in parallel and the results compared. This same strategy can be used for other simulators, notably the COSMOS simulator.

Status

The full 128 test pin PC board version has been completed and is currently being used to test two VLSI chips using test programs and the simple software interface. The interface to the Capilano software will be completed this summer, as well as RNL and COSMOS frontends. Anyone interested in obtaining this tester should contact the authors. We are currently planning to make the tester available in kit form along with a set of test software for the MacII.

Acknowledgements

This work was funded in part by Apple Computer, Inc., grant ER0030-65-5742, and NSF grant CCR8657589A02. The Northwest Laboratory for Integrated Systems is sponsored in part by DARPA under contract N00014-88-K-0453.