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

Neil McKenzie, Larry McMurchie, Carl Ebeling
Department of Computer Science and Engineering
University of Washington
Seattle, WA 98195
mckenzie@cs.washington.edu

We describe a low-cost functional tester that provides a large number of programmable I/O signals for interactively testing and debugging chips, boards and subsystems. Testing a subsystem requires only minimal hardware setup which allows the tester to be shared by several users and reduces the overall cost of test equipment. Testing is performed under program control which can be driven either by a set of test vectors or by a user-supplied procedural test program. The tester system consists of the tester unit, a parallel interface card and a personal computer as the host. The tester architecture uses Xilinx FPGAs for the pin electronics and static RAM to enable testing of dynamic circuits. A simple but powerful software interface to the tester makes possible a wide variety of testing and debugging environments.

Introduction

The process of building and testing the prototype of a digital circuit is becoming increasingly difficult, given the increasing complexity of individual components, as well as the environment of the circuit. Testing of the prototype during its construction is usually performed at several levels.[1-3] At one level is functional testing,[4-8] which provides input stimuli in sequence to the circuit and verifies outputs; only the sequence and values of outputs are verified, not specific timing constraints. At another level is electrical and timing verification, which often occurs near the completion of the construction process. These two kinds of testing are fundamentally different and need to be addressed separately. Existing commercial testers focus on electrical and timing verification. Although functional testing and debugging dominate the verification of circuit prototypes, there has been comparatively little work in developing hardware and software environments for this kind of testing.

A functional test cycle consists of four parts: specifying the input vector, presenting the input vector to the pins of the device under test (DUT), latching the output vector, and verifying the output vector. Dynamic circuits, such as DRAMs, require the tester to sequence the input vectors at some minimum rate for a successful test. Functional tester hardware for dynamic circuits requires a dedicated sequencer and a memory to store an array of test vectors. We refer to this as *off-line* testing (Figure 1). Test vectors are determined

off-line by hand-calculation or by executing a program, such as a simulator. The host computer downloads these vectors into the tester's test vector memory. The tester sequences the input vectors into the DUT and records the behavior generated by the DUT. High performance testers compare the DUT behavior with the output vectors on-the-fly in hardware. Less expensive testers upload the raw DUT behavior to the host and verify the results in software. Tester memory usually consists of fast but expensive static RAM chips. Long tests and DUTs with a large number of pins require the tester to have a large number of these SRAMs which drives up the cost of the tester hardware.

On-line testing of static circuits (Figure 2) is an alternative to off-line testing. A test program that runs on the host computer controls the test vector cycle. The software that generates the test vectors is interleaved with the execution of the test. Another way to think of this is that the DUT is treated as a hardware subroutine. The sequencing rate can vary without affecting the logical operation of the DUT. There are many advantages to on-line testing. On-line tester hardware eliminates the off-line vector SRAM and is thereby less expensive. The number of test vectors is virtually unlimited so that a test may be left running over a long period of time. The test program employs the full power of a general-purpose CPU, including conditional branching, procedure calls, iteration and recursion. Asynchronous circuits and circuits whose timing is data-dependent are much simpler to test on-line, because the test programs adapt to different timing behavior. A variant of on-line testing is interactive testing, in which the user enters the loop and controls the generation of new input vectors on the basis of previous results. Another variant is a hybrid of off-line and on-line testing (Figure 3). The tester executes blocks of test vectors at off-line rates under a general on-line framework. This technique can be used to test dynamic circuits interactively, or to test *pseudo-static* circuits, which are dynamic circuits that are static under some conditions. Under hybrid testing, dynamic circuits can be tested with only a fraction of the SRAM needed for an equivalent off-line test.

Functional testing in both off-line and on-line forms provides a powerful framework for testing ASICs and other digital circuits. Functional test hardware can be substantially cheaper because precise verification of timing constraints is ignored. Unfortunately, existing commercial testers focus on off-line speed testing rather than functional testing. Test vectors are usually generated by a host system with a low-speed link to the tester which results in a long interaction loop. Often the proprietary nature of test equipment makes it hard to tailor a testing system to the environment in which the circuit has been designed. The result is that the user spends a lot of time context-switching between the design system and the test system. Worst of all, the cost of the commercial testers is often beyond the budgets of small companies and educational labs. In a recent review of ASIC test and verification systems, the lowest cost testers were priced at $1500 per pin[6], which puts the cost of a typical tester configuration of 64 pins near $100,000.

The high cost of test equipment usually means that functional testing must be performed in an *ad hoc* way. A special-purpose environment is constructed around the project to be tested, creating a self-contained system that can be operated, observed and debugged using a logic analyzer and/or an oscilloscope. Constructing this environment may be difficult and time-consuming. Shortcuts are taken that result in a hard-wired and hard-to-modify test system.

Commercial testers and the *ad hoc* approach both lack a convenient user environment for setting up functional tests. We believe that a procedural description for functional test provides both the expressiveness and flexibility required by a productive testing environment. A program specifying a test provides a high level of abstraction, where the behavior of different components and their interfaces can be encapsulated in procedures and functions. Small changes to the circuit are reflected in small and intuitive changes to the test program. Stand-alone test vectors by themselves are fragile since a trivial circuit change can alter every single test vector. Perhaps most important, errors found during test and debug can be related back to the part of the program modeling the failed component. More sophisticated test programs can even perform error analysis in addition to simple error logging. This form of error analysis is crucial when debugging a hardware prototype.

Once a programming language interface to a tester is constructed, one can envision all sorts of testing and debugging environments that can be built on top of it. In a windowing environment, the inputs and outputs of the device being tested can be displayed in separate windows and formatted in different ways, for example as a spreadsheet or a graph. In a simulation environment, the tester and the DUT can provide the real behavior for a device whose behavior is too complex to describe in a software model. This particular environment is useful to designers who wish to gain familiarity with complex devices before they are designed into a project.

In short, functional testing has enormous potential, very little of which has been realized by the existing commercial testers or the *ad hoc* environments that are often set up to test circuits. At the Laboratory for Integrated Systems (LIS) at the University of Washington, we have designed and constructed the UW MacTester to demonstrate that functional testing can be inexpensive, well-integrated and accessible under a wide variety of user environments.

Overview of the UW MacTester Design

The two primary goals for the MacTester were low cost and ease of use. Achieving a low-cost solution required the use of off-the-shelf parts even though it was tempting to design a custom ASIC to simplify the design.[7] However, we were able to approximate an ASIC architecture using FPGAs, which solved two of

the more difficult design problems, high pin count and board area, as well as reducing the parts cost. Using the Xilinx FPGAs yielded several other advantages. First, since Xilinx FPGAs are dynamically re-programmable, the tester hardware can be modified on-the-fly to meet the requirements of a particular test setup. Second, the Xilinx pins can be configured for either TTL and CMOS logic levels and thus can accommodate a wide variety of circuits.

Since the tester is controlled directly by software, an interface to the host with relatively high bandwidth and low latency is required. Using an interface like RS-232 or even SCSI would make testing unacceptably slow. The MacTester thus interfaces directly to the host bus via a simple parallel interface. We chose as host workstations the two most prevalent low-cost design stations, the Macintosh II and the IBM PC/AT. Although the IBM PC is in wider use in the design community, the Macintosh provides a simple, intuitive graphics interface with which an interactive testing environment can easily be constructed.

For the Macintosh, we designed a simple 32-bit parallel interface card for the NuBus. Test programs running on the Mac directly read and write test vector data at memory mapped addresses. We also designed an interface card for the PC/AT (ISA) bus. Due to “RAM cram” under MS-DOS, the precious memory map is not used; instead the tester is accessed through a small number of 16-bit I/O ports. The ISA card composes two 16-bit port accesses into a single 32-bit access to the tester. The overhead of the parallel interface for an on-line vector is on the order of ten microseconds, which is low enough to make on-line testing practical. The simple parallel interface was a key to the success of the MacTester project and the interface cards have been adopted by several other projects as well. Interface cards can be developed easily for other popular microprocessor busses such as SBus or VME.

The MacTester system allows a wide variety of testing and debugging environments. We developed a package of low-level tester interface routines, implemented as a C library module, that supports both on-line and off-line testing. Users can either write C programs or write in a higher-level language, such as C++ or Common Lisp, that can import the test package. Test programs are compiled, linked with the library, and then executed on the host computer. We chose C as the implementation language because commercial compilers are inexpensive and widely available for both platforms. The library is written entirely in C. The portability of C enabled us to create a single set of library source modules that compile for both the Mac and PC platforms, and test programs usually port without change across platforms.

With this low-level software interface in place, an interactive environment can be built for on-line testing, where each test vector is generated in response to a user's input. Graphical user interfaces are a natural extension. For instance, if the DUT is a real-time clock chip, the interface could look and act like a digital clock. As a demonstration of an interactive environment, we have integrated the tester into Capilano

Computing's *DesignWorks* graphical interface, which is described later in a later section. Other user environments include custom testing languages[4,8] and simulation-driven testing.

Another ease-of-use feature of the MacTester is the ability to test chips in a variety of packages. The ability to switch between different projects with a minimum of set-up effort is important, especially in educational labs where many students share a single tester. The tester provides this capability through a zero-insertion-force (ZIF) socket that is a 20 by 20 pin grid array with a pitch of 0.1 inch. The ZIF socket has pre-programmed mappings for nearly all common DIP and PGA chip patterns up to 128 pins (Figure 4). Each pin is bi-directional; the direction of each pin is independently controlled and it may be flipped for each test vector. For low-power chips, power and ground can be provided directly by the FPGA pin drivers. Hand-wiring is required to provide power and ground for higher-power chips. Larger projects that cannot plug directly into the ZIF socket require the construction of a special purpose interface, such as a daughter-board for a 64-pin DIP or a cable for a board-sized project.

In order to test dynamic circuits, the tester must store a sufficiently large number of vectors and sequence them through the DUT at a reasonable rate. Tester vector memory holds roughly 5400 vectors. Off-line tests run at 770K vectors per second when using a Mac II as the host computer. On-line tests run up to 50K vectors per second. In many cases the speed of on-line testing is sufficient to test dynamic circuits.

Although speed testing is not a primary goal of the MacTester project, it seemed important to allow at least a limited form to be done in conjunction with functional testing. Accordingly, the MacTester has the speed testing capabilities of a delayed latch on the output values. The test program controls the delay between the arrival of the input vector and the sampling of the output vector. In addition, the MacTester and a more powerful speed tester, such as the Tektronix DAS 9000, can be used in tandem. The MacTester provides the setup and environment for the low-speed part of the test. It then calls the DAS as a hardware subroutine to run the high-speed part of the test. The tester and the DAS handshake; the tester must tri-state the appropriate signals in order to hand off to the DAS. The capacitance of the signal wires is sufficient to hold the proper logic levels to the DUT during the hand-off.

Physical Design

Figure 5 is a photograph of the tester system. The tester unit is housed in a 2.5" by 14" by 10.5" metal case. There is a single interface cable with 80-conductor wire between the tester and the interface card. The cable is slightly over one meter long and uses a high-density (SCSI-2) style connector with a spacing between pins of 0.050". The tester is powered by an external 5 volt source which plugs into a five-pin round DIN connector. The tester draws about one amp. The case has a rectangular opening in the top that

allows the user access to the ZIF socket. Around the ZIF socket is a ring of 0.025" square pins that are electrically connected to pins in the ZIF socket. These pins allow the user to hand-wire power and ground and to connect oscilloscope probes to the DUT, or to connect the tester to an external board.

There are no physical switches in the tester unit. Power to the DUT is switched in software. The tester uses two low on-resistance FETs to provide power to the DUT. One FET controls direct power, and the other FET controls indirect power. The user can connect an ammeter between two test points and use indirect power to measure the increase in current flow when the user's circuit is plugged in. There are three LEDs visible inside the ZIF socket window. The red LED indicates that external power is supplied to the tester unit. The yellow LED indicates that the tester hardware is initialized and ready to test a circuit. The green LED indicates that power to the DUT is switched on.

The interface cards for both the Mac and the PC are half-size, i.e. 4" by 7". The interface cards bring out 32 bits of data, 16 bits of address and a variety of control signals. The interface card buffers the bus clock as a control signal, which eliminates the need for a clock crystal in the tester. The Mac NuBus provides a 10 MHz clock. Most PC compatibles provide an 8 MHz bus clock.

Hardware Architecture and Implementation

Figure 6 shows a high-level block diagram of the tester. Each of the 128 pins provided by the tester is driven by a tri-state driver whose data and enable values are given by registers that can be written by the host. These input registers are double-buffered so that a test vector set up via several writes by the host to the level 1 registers can then be presented all at once by shifting the contents of the level 1 registers to the level 2 registers. Note that each test vector can enable the tri-state independently and that this enable is specified separately for each pin. This allows maximum flexibility to test all types of bi-directional signals.

After a test vector has been strobed onto the output pins, the value of the pins are then latched into the level 3 registers after some user-specified delay. By setting this delay, the user can perform simple speed testing, for example, measuring the delay from a clock signal to an output value. Internally, the tester uses a 64-bit data bus. Since the host interface is 32 bits wide, data transfers between the tester and the host use only the upper or the lower half of the 64-bit internal bus. When the tester is running autonomously in off-line mode, data transfers between test vector memory and the tester registers use the entire 64-bit data bus.

During one step of an on-line test, the host first writes up to eight 32-bit values into the appropriate level 1 registers. These values are then transferred in unison to the level 2 registers, causing the entire test vector

to be driven onto the pins of the DUT. The level 3 registers are then latched after a user-specified delay. The host then reads back the results from the level 3 registers in four 32-bit data reads. The host then uses these results to validate the test or to modify the subsequent test vectors.

In off-line testing, the host copies an array of test vectors from program memory into the test vector memory. The host loads hardware counters with the starting address and the array length and signals the tester to initiate the off-line test. For each off-line test vector, the tester moves a test vector to the level 1 registers, moves the results from the level 3 registers back to test vector memory, increments the address counter, and decrements the length counter. When the length counter is zero, the tester sets a done bit to notify the host that the test is complete so that the response vectors can be uploaded. The off-line test can also be placed in a loop so that a particular test sequence can be monitored with an oscilloscope.

The data path is implemented in six Xilinx XC3020 FPGAs and the control logic in a seventh. The implementation would have been possible using fewer of the larger Xilinx chips, but since the cost per FPGA pin grows almost quadratically with the reduction in the number of chips the XC3020 turned out to be the most cost-effective size. Each of the six data path FPGAs drives 22 pins of the DUT and 11 bits of the internal bus (minus a few spares). The XC3020s run out of both pins and area under this configuration. Because of the limited number of I/O pins, we found it prudent to multiplex the memory. We use a total of eight 32K x 8 SRAMs with an access time of 85 ns. The SRAMs are multiplexed six ways and yield 32K/6 (about 5400) test vectors. Multiplexing the SRAMs causes a performance hit; the state machine takes 13 states to sequence a single vector which results in an off-line rate of just under 1 MHz (770 KHz). Using fewer states would increase the off-line testing rate, but would require more FPGAs and SRAMs and a larger printed circuit board. Because purpose of the tester is functional testing and not speed testing, we chose the lowest cost implementation.

Xilinx FPGAs are RAM-based rather than fuse-based and are programmed by downloading a stream of control bits generated by the Xilinx design tools. Users can retarget the behavior of the FPGAs by reprogramming them with various designs. For example, the delay between the level 2 and level 3 registers may be fine-tuned by downloading different circuits into the control logic FPGA. Reprogramming the FPGAs has yielded enormous benefit during implementation by allowing us to fix bugs and test different features without changing the printed circuit board or replacing chips.

The delayed latch timing is determined by two parts: a coarse delay and a fine delay. The coarse delay may be set in increments of a bus clock period (100 ns for the Macintosh version, 125 ns for the PC version). The fine delay is composed of a variable length chain of buffers inside the control logic FPGA. The fine delay may be either “hardwired” or set using a multiplexor inside the FPGA. The multiplexor version

provides incremental timing steps of about 10 to 15 ns. Both the coarse delay and the multiplexor delay can be set in the test program on a per-vector basis.

The tester implementation is extensible in several dimensions:

Number of pins. The internal data path can be easily increased in increments of 32 bits, along with the memory and datapath FPGAs required to extend the number of DUT pins by 64 per increment. More cycles would be required to write test vectors to the tester and read back responses, but off-line testing would proceed at the same rate.

Depth of the test vector memory. The depth of the test vector memory depends on the size of the RAM chips used. The current 32Kx8 RAMs can be replaced with larger chips to accommodate longer off-line test sequences. This change would also require the internal address bus to be extended to allow access to the larger memory.

Pipelining. In some cases, a short high-speed burst of test vectors suffices to test chips at speed. This can be implemented inexpensively by deepening the pipeline, i.e. adding more registers at levels 2 and 3. The host would prepare a high-speed test by downloading several test vectors. These would then be presented at high speed, at up to 40 MHz, and the responses captured by the level 3 register pipeline. We conjecture that almost all high-speed testing can be performed with only 4 test vectors in this high-speed burst.

Software Interface and Programming Examples

Initialization

When the tester is initially powered on, the FPGAs are uninitialized. The host computer initializes the tester by downloading a bit stream to the FPGAs to install the intended circuit behavior. When the user a launches a test program, the test package performs a self-test on the tester's memory. If the test fails, the test package downloads the FPGA bit files and reruns the self-test. A second failure means that there is a hardware problem and the program aborts. If it succeeds, the user is prompted to enter the latch delay value in nanoseconds. Next, a dialog box then prompts the user to select a power source to the DUT (Figure 7). The three power options are **Direct**, **Indirect** and **Xilinx**. **Direct** and **Indirect** select the appropriate FET to turn on to supply power to the DUT. Selecting **Xilinx** means that the DUT is powered directly by the Xilinx FPGAs. If **Direct** or **Indirect** are selected, the user is prompted to execute a power and ground self-test (Figure 8). This ensures that power and ground are correctly connected

to the intended pins of the DUT, to prevent the user from accidentally burning out a chip or board. One advantage to using Xilinx power is that with only 4 mA of output drive is available per pin, it is nearly impossible to blow up the DUT. The FPGAs themselves have been amazingly resilient, as we have burned out only one during the lifespan of the MacTester project.

After all the preliminary tests have been successful, the user is then prompted to plug in the circuit. At this point, the tester puts all its pins into a high-impedance state. For testing chips, the user raises the ZIF socket arm, places the chip in the socket, and the lowers the arm to make contact with the chip's pins. Care must be taken to place the chip in the correct orientation, as misplacement will lead to an unsuccessful test or possibly a blown chip. After placement, the user clicks the mouse button or types a carriage return, and testing begins. The tester first enables power and ground and then activates the input signals to the DUT. Testing proceeds until the user's code terminates, at which point the tester puts its pins into the high-impedance state, and the user can unplug the DUT safely.

One safety feature of the tester is that the power FETs are driven by gates in the control FPGA. If the control FPGA is uninitialized, the FETs cannot conduct, regardless of the state of the software or the interface control signals. This prevents the tester from being powered up in an incorrect state that would damage the pin FPGAs, the DUT or the FETs.

On-line testing

Figure 9 shows a simple C test program for an 8x8 combinational multiplier. The header file `<tester.h>` declares the interface to the test package. `<DIP40.h>` specifies the footprint of the DUT (i.e. a 40-pin DIP); it maps the standard (virtual) pin numbering of the chip to the physical pin numbering of the ZIF socket. Test programs have two entry points: `DefineSignals`, which the test package calls during tester initialization, and `MainTest`, which is the body of the user's test program.

To the user, the pins of the DUT appear as variables in the test program. In `DefineSignals`, the macro `Signal` declares each signal variable (`Multiplier`, `Multiplicand`, and `Result`) to represent a collection of pins in the numbering scheme of the chip. The list of pins is declared from most to least significant using the string argument. The colon operator denotes a sequence of pins and the comma denotes concatenation. For example, the string `"5:8,11,14:12"` declares the list of pins 5, 6, 7, 8, 11, 14, 13 and 12, with pin 5 as the most significant bit and pin 12 as the least. Signal variables are declared as INPUT or OUTPUT, from the point of view of the DUT. Input variables thus manipulate the values of the DUT pins. Vdd and GND pins are also declared here. The tester provides no signals other than those

specified by the user's program. In particular, the test program must supply the clocks by explicitly changing a signal variable or set of variables.

In `MainTest`, the function `Set` changes input values, and `Get` accesses output values from the DUT. The function `SetDirection` changes the direction of bi-directional signals such as data busses. `SetLatchDelay` changes the latch delay value. `Set`, `SetDirection` and `SetLatchDelay` calls are buffered and take effect in parallel after calling `Next`. `Get` returns the value of the signal latched by the most recent `Next` call. The programmer's model of a test program step consists of a number of `Set` statements, followed by a `Next` statement, followed by a number of `Get` statements.

`Next` is the only function that directly activates the tester hardware. The actions of all the other functions are buffered in program memory. In on-line testing, `Next` performs the following actions:

- Copies the test vector in program memory to the tester input registers.
- Sets the latch delay.
- Executes a single test vector, driving the test vector onto the pins and latching the response vector.
- Copies the response vector in the tester output registers to program memory.

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.

Typical on-line test programs concurrently exercise the DUT and a software model that describes the behavior of the DUT. The program validates the DUT by comparing its behavior with that of the software model. In the example from Figure 9, the DUT is sequenced by loading its inputs `Multiplier` and `Multiplicand` with the iteration variables `i` and `j`. `Result` is validated with the software model, the expression `i*j`. In this example the circuit is stateless, as its outputs are a function of its inputs only. In the second example, the circuit has an internal pipeline register added to increase the throughput. A two-phase non-overlapping clock sequences the pipeline. Figure 10 shows the clock generator and Figure 11 shows the other modifications to the test program. Changes are shown in boldface type. `Multiplier` and `Multiplicand` are loaded with `i` and `j`, a clock cycle is run and the result compared with the calculated value. Variables `lasti` and `lastj` are used to remember the inputs from the previous iteration. Auxiliary functions `fillpipe` and `emptypipe` (not shown) ensure that the multiplier is loaded correctly the first time and that the final test is completed.

Off-line testing

On-line testing may fail to test dynamic circuits in a multiprogramming environment where the tester task may be descheduled for an unbounded amount of time. The test program must change to invoke off-line testing in this case. The difficulty in off-line testing is the lack of concurrency between hardware sequencing and software modelling, which on-line testing provides on a vector-by-vector basis. To bridge this gap, we augment the semantics of `Next` to provide off-line test programs the look and feel of on-line testing. A two-pass strategy creates this illusion. During the first pass, known as the *generate* pass, through an off-line program segment each `Next` performs the following actions:

- Copies the test vector in program memory to the current vector location in tester memory.
- Increments the current vector location pointer.

Between passes, the program then:

- Sets the latch delay globally for the off-line array.
- Executes the array of vectors stored in tester memory, driving each onto the pins and copying the resulting response vector back into tester memory.
- Resets the location pointer to the first vector.

During the second pass, known as the *verify* pass, `Next` performs the following actions:

- Copies the response vector at the current location in tester memory to program memory.
- Increments the location pointer.

A single program can alternate between off-line and on-line mode, as the semantics of `Next` are determined dynamically. The third example (Figure 12) shows how the test program for the multiplier is converted into an off-line form. The `BEGIN_OFFLINE` and `END_OFFLINE` statements enclose an off-line array of vectors. The `BEGIN_OFFLINE / END_OFFLINE` pair are placed inside the outer loop because the test vector memory is limited to roughly 5400 vectors. This illustrates how a test program can use a hybrid approach to extend off-line testing beyond the size of the test vector memory. Another strategy could be used if our multiplier were pseudo-static rather than fully dynamic. For instance, if the multiplier is static when no clocks are active, then the off-line block could confined to the `clockchip` procedure.

In order to maintain the illusion of on-line concurrency, there are restrictions on the program statements in the off-line block. `Set` performs an action only during the generate phase, and `Get` returns a valid

response only during the verify phase. The macros `GENERATE` and `VERIFY` delimit blocks of statements such that they execute in their respective phases. Furthermore, the two passes of the off-line block must otherwise have the same initial conditions and compute exactly the same results. These restrictions can make off-line testing more complicated, but usually only small modifications of a program for on-line testing are required to produce one for off-line testing.

The *DesignWorks* Environment

The software interface of the MacTester makes possible a wide variety of testing and simulation environments. To demonstrate this capability, we chose to embed the MacTester into the simulation environment of *DesignWorks*, a schematic capture and simulation system written for the Macintosh by Capilano Computing. In *DesignWorks*, graphical I/O devices such as keypads, displays and timing diagrams are used to interactively test and debug a circuit. We used a *DesignWorks* library for writing simulation models to incorporate the MacTester into the simulator. In this way any device or board plugged into the MacTester can be incorporated into the schematic and the simulation just like any other device. The *DesignWorks* graphical I/O devices can then be used to test and debug the DUT. Moreover, the DUT can be tested as part of a larger system described by schematic drawings and simulation models.

Figure 13 shows an example of the test of a 4-bit TTL counter chip (74LS163) using the MacTester inside *DesignWorks*. At the top of the circuit is a generic 4-bit counter, whose behavior is described by a software model. A hex input display in the upper left provides the initial value for both counters. Toggle switches allow the user to load, clear and increment the counters. To the right of the counters is an adder and NOR gate which function like a comparator. Finally on the right is a register that simply stores the result of the comparison.

All signals supplied to the DUT, including power and ground, are specified in the schematic. When the simulation first starts up, the 74LS163 and the generic counter are likely to be out of sync. Toggling the *clr* line high for more than a clock cycle synchronizes both counters. By toggling other inputs, the functionality of the DUT can be determined quickly.

The specification of an entirely different device (with a different pin-out and/or package) requires little effort. The user simply uses the device editor to create a new symbol, links it to the tester interface code and declares signals to be input, output, bi-directional or tri-state. No software changes are required to the tester interface, which is generic across devices. The model employed in the MacTester interface is compatible with the simulator. For example, the tester will detect whether the DUT has placed a pin in a high impedance state and communicate that information to the simulator and the user.

An interesting use of the MacTester is to combine the programmed interface and the *DesignWorks* interface. For example, testing a Xilinx FPGA implementation requires the FPGA (as the DUT) to be first programmed via a bit stream generated by the Xilinx tools. This can be done as part of the program testing the FPGA implementation, or it can be done by a stand-alone program which is executed before the test program is executed. Similarly, if the FPGA is to be included in a *DesignWorks* schematic simulation, it can be initialized by this program before the simulation is started. In the simulation the programmed FPGA provides its own behavior in place of a software model. The impact is that the simulator runs faster and allows larger and more complex simulations to run,[9] as student projects have demonstrated.

On-line interactive testing using *DesignWorks* is useful for testing static chips; dynamic chips can also be tested in the same environment by a simple modification to the software interface. A test sequence begins with a reset of the DUT and a re-initialization of the tester memory. On every successive clock change, a set of new inputs is downloaded to the tester and stored in successive locations in memory. Then the entire sequence of input vectors is clocked into the chip, just as in off-line testing, and the results of the last input vector are uploaded to the host. Thus each change of the clock initiates a complete sequencing of the DUT through all the input vectors since the last reset. In this way the illusion of interactive on-line testing can be provided when testing a dynamic DUT.

The combination of the MacTester and *DesignWorks* allows designs that comprise only part of a system to be implemented in hardware, while the remainder of the system is described in a *DesignWorks* schematic. The entire system – hardware and schematic – can then be "simulated" as a single entity. The idea here is to isolate one component of the system, and test an implementation of that part in the context of the remainder of the system which is modeled in software.

Summary

Functional testing in both off-line and on-line forms is a valuable technique for testing and debugging circuit prototypes. Potential obstacles to functional testing are high cost and difficulty of use. The UW MacTester project overcomes these obstacles as it provides a low-cost integrated solution – one that is easily customized to a variety of interfaces. The parallel interface between the tester and the host provides a low-overhead link that makes on-line testing practical. FPGA technology increases the tester circuit density and makes possible a compact, low-power design. In conjunction with a simulator, the MacTester makes possible the construction and debugging of an electronics system partly described by software models and partly implemented in hardware.

Current Status

Five prototypes of the MacTester have been constructed, of which three are in use at Apple Computer Advanced Technology Group and two in the UW Computer Science and Engineering graduate student labs. Applied Precision Inc. of Mercer Island, Washington, is manufacturing and selling the production version of the UW MacTester. Contact Larry McMurchie (`larry@cs.washington.edu`) for further information.

Acknowledgments

We thank Apple Computer for donation of equipment as well as funds to support a research assistant. Ian Jones of the Advanced Technology Group of Apple Computer was instrumental in getting the project off the ground. DARPA provided funds for the prototype printed circuit board fabrication. NSF funded the development of the software under the Software Capitalization Program. The Washington Technology Center helped fund the commercialization of the UW MacTester.

References

1. Evanczuk, Stephen. "IC Prototype Verification: Test and Tribulation," *VLSI Systems Design*, April 1986, pp. 44-48.

2. Florcik, David, Low, David and Roche, Martin. "Prototype Debug Using ATE," *IEEE Design and Test of Computers*, May 1984, pp. 94-99.

3. Kinzelman, Paul M. "Behavioral Exercisers Expand the System-Test Design Kit," *High Performance Systems*, June 1990, pp. 54-62.

4. Ponik, Wayne. "Teradyne's J967 VLSI Test System: Getting VLSI to the Market on Time," *IEEE Design and Test of Computers*, December 1985.

5. Bassett, Robert W. et al. "Low-Cost Testing of High-Density Logic Components," *IEEE Design and Test of Computers*, April 1990, pp. 15-27.

6. DeSena, Art, "A Guide to ASIC Test and Verification Systems," *ASIC Technology and News*, July 1991, pp. 18-20.

7. Gasbarro, James A., and Horowitz, Mark A., "Integrated Pin Electronics for VLSI Functional Testers," *IEEE J. of Solid-State Circuits*, Vol. 24, No. 2, Apr. 1989, pp. 331-337.

8. Maurer, Peter M. "Dynamic Functional Testing for VLSI Circuits," *IEEE Design and Test of Computers*, December 1990, pp. 42-29.

9. Evanczuk, Stephen. "Mixed-Level Simulation Acceleration," *VLSI Systems Design*, February 1987, pp. 62-70.

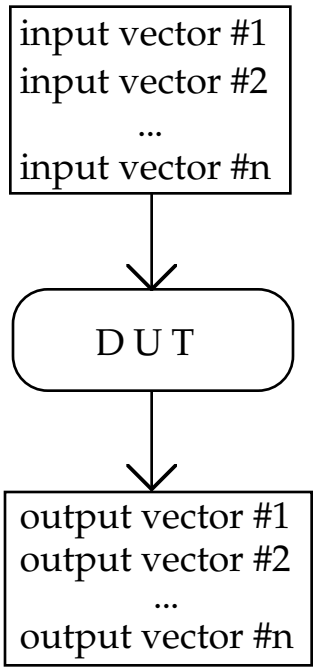

Figure 1. *Off-line testing*

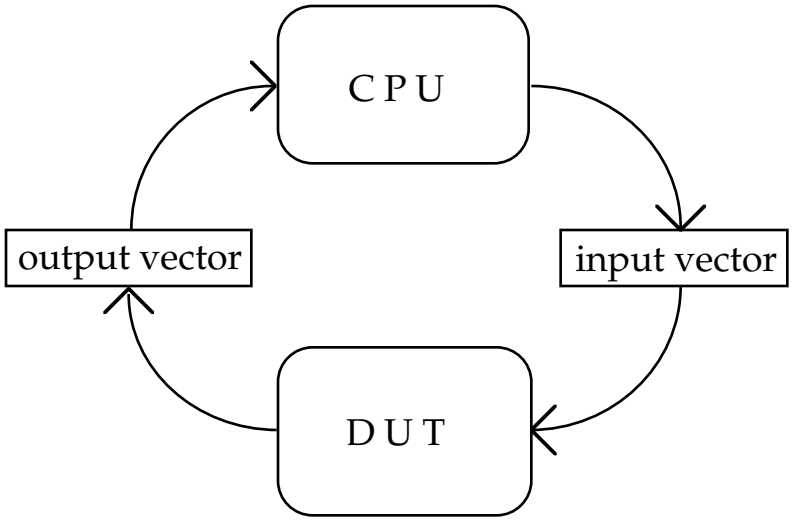

Figure 2. *On-line testing*

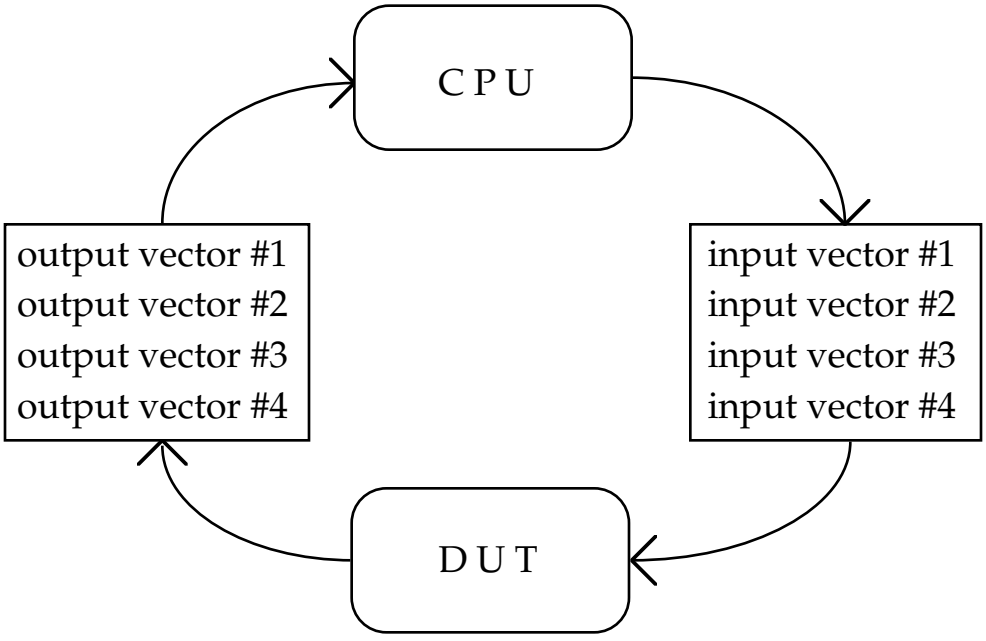

Figure 3. *Hybrid testing*

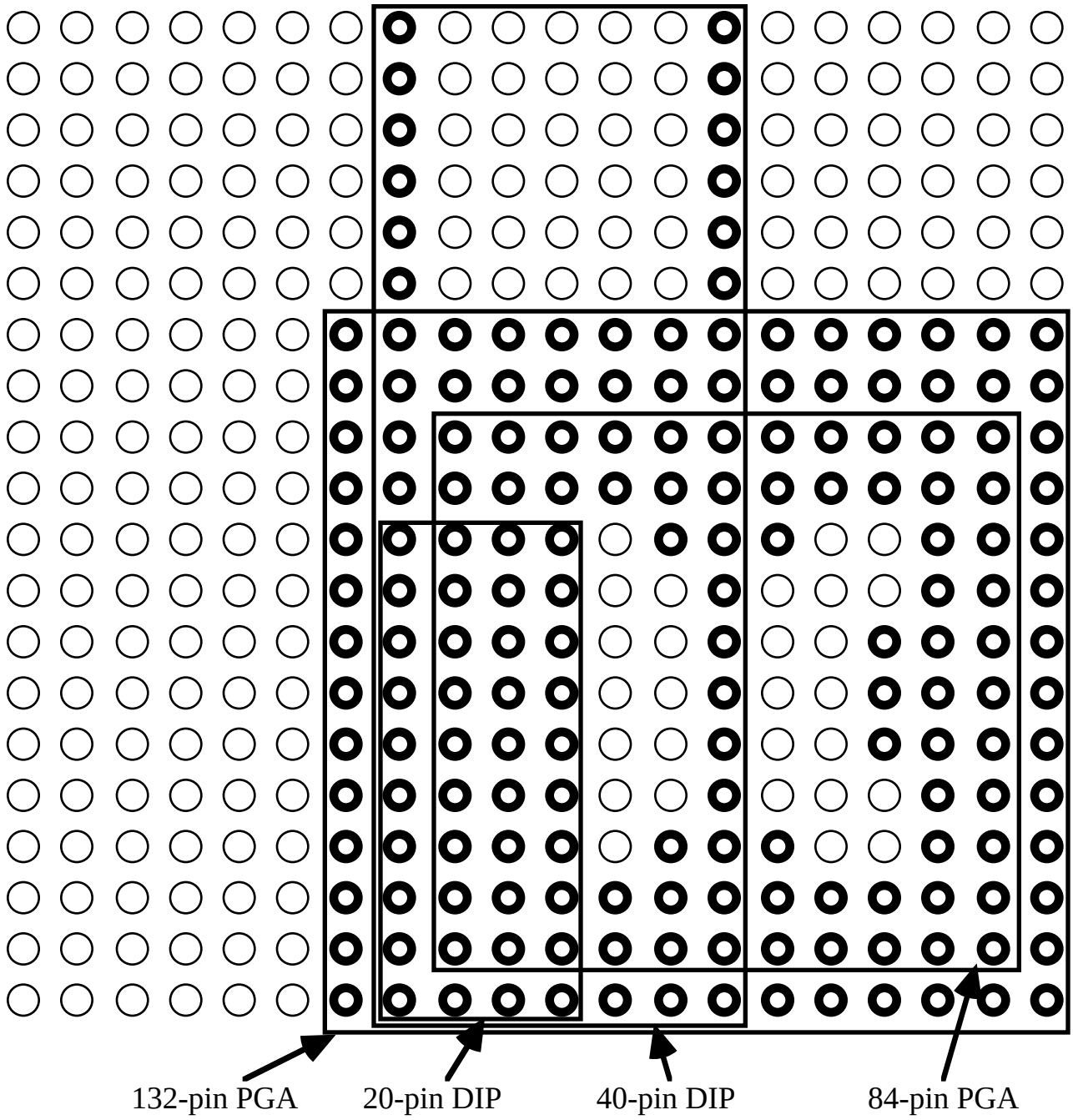

Figure 4. *ZIF socket mapping for DIPs and PGAs*

Figure 5. *Photograph of the MacTester*

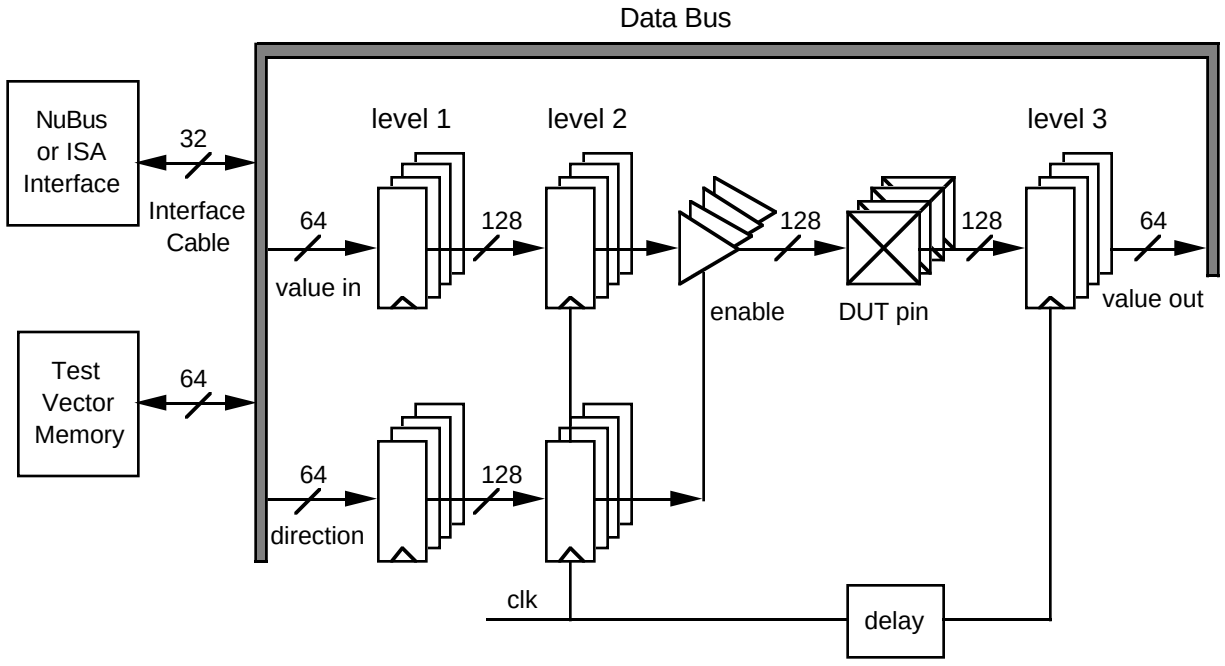

Figure 6. *The tester data path*

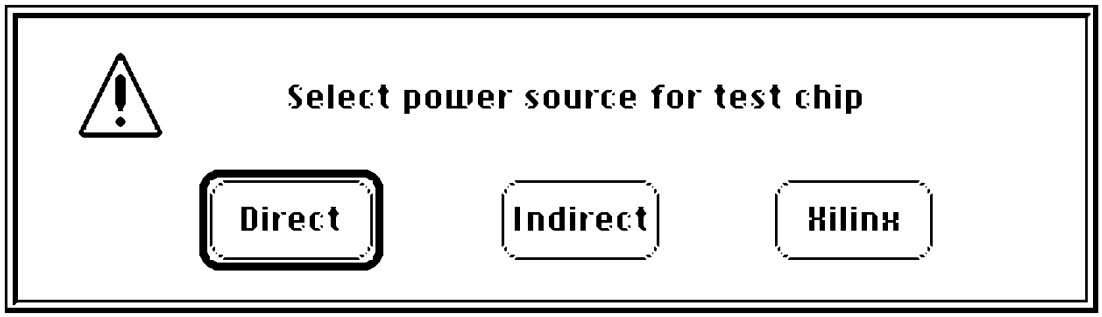

Figure 7. *DUT power selection dialog box*

Figure 8. *DUT power and ground test dialog box*

```
#include <tester.h>
#include <DIP40.h>
Sigptr  Multiplier, Multiplicand, Result;

DefineSignals()
{
       GND("20");
       Vdd("40");
       Signal(Multiplier, "8:1", INPUT);
       Signal(Multiplicand, "19:12", INPUT);
       Signal(Result, "39:31, 29:23", OUTPUT);
}

MainTest()
{
       int i,j;

       for (i=0; i<256; i++) {
              for (j=0; j<256; j++) {
                     Set(Multiplier, i);
                     Set(Multiplicand, j);
                     Next();
                     if (Get(Result) != i*j)
                            error();
              }
       }
}
```

Figure 9. *Test program for a combinational multiplier*

```
clockchip()
{
       Set(phi1, 1); Next();
       Set(phi1, 0); Next();
       Set(phi2, 1); Next();
       Set(phi2, 0); Next();
}
```

Figure 10. *A two-phase non-overlapping clock generator*

```
MainTest()
{
       ...
       fillpipe();
       for (i=0; i<256; i++) {
              for (j=0; j<256; j++) {
                     Set(Multiplier, i);
                     Set(Multiplicand, j);
                     clockchip();
                     if (Get(Result) != lasti*lastj)
                            error();
                     lasti = i;
                     lastj = j;
              }
       }
       emptypipe();
       ...
}
```

Figure 11. *Test program for a static single-stage pipelined multiplier*

```
MainTest()
{
       ...
       for (i=0; i<256; i++) {
              BEGIN_OFFLINE;
              fillpipe();
              for (j=0; j<256; j++) {
                     GENERATE {
                            Set(Multiplier, i);
                            Set(Multiplicand, j);
                     }
                     clockchip();
                     VERIFY {
                            if (Get(Result) != lasti*lastj)
                                   error();
                     }
                     lasti = i;
                     lastj = j;
              }
              emptypipe();
              END_OFFLINE;
       }
       ...
}
```

Figure 12. *Test program for a dynamic single-stage pipelined multiplier*

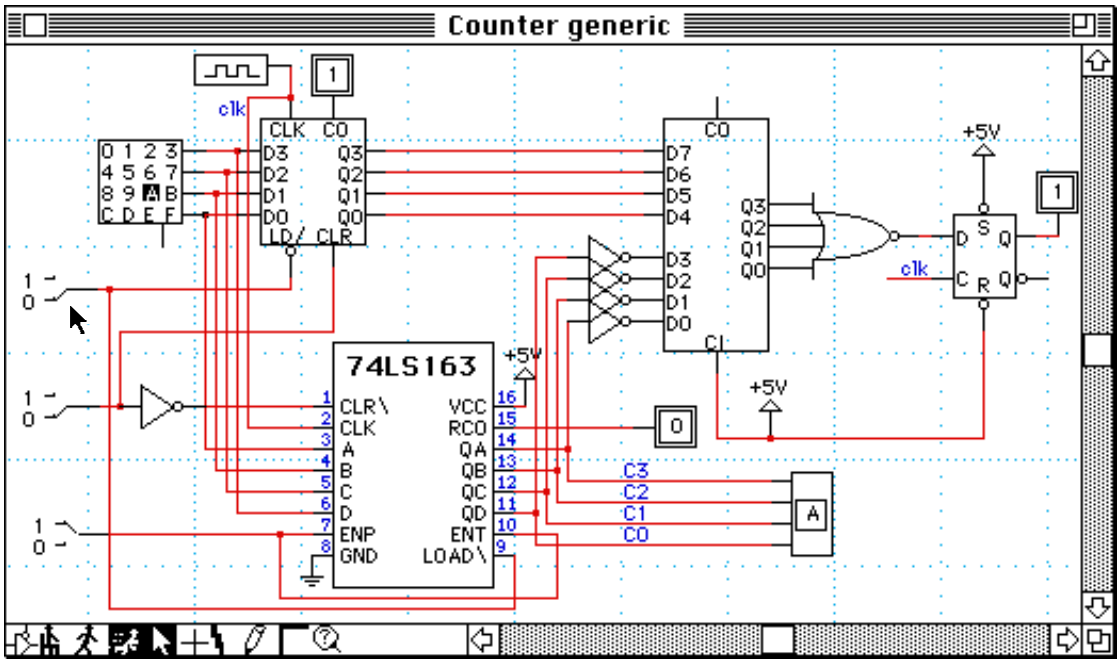

Figure 13. *DesignWorks session with the MacTester as 74LS163 block. The tester is loaded with a 74LS163 chip. The cursor is next to the toggle switch that controls the LOAD/ signal. Because LOAD/ is active low, both the simulated counter and the actual chip load the value hex A from the hex keypad upon the rising edge of the clock.*