# Giano: The Two-Headed System Simulator

Alessandro Forin
*Microsoft Research*

Behnam Neekzad
*University of Maryland*

Nathaniel L. Lynch
*Texas A&M University*

September 2006

Technical Report
MSR-TR-2006-130

Microsoft Research
Microsoft Corporation
One Microsoft Way
Redmond, WA 98052

# Giano: The Two-Headed System Simulator

Alessandro Forin
*Microsoft Research*
*sandrof@microsoft.com*

Behnam Neekzad
*University of Maryland*
*behnam@umd.edu*

Nathaniel L. Lynch
*Texas A&M University*
*nathaniel-lee-lynch@neo.tamu.edu*

## Abstract

*Giano is a simulation framework for the full-system simulation of arbitrary computer systems, with special emphasis on the hardware-software co-development of system software and Real-Time embedded applications. It allows the simultaneous execution of binary code on a simulated microprocessor and of Verilog code on a simulated FPGA, within a single target system capable of interacting in real-time with the outside world. The graphical user interface creates the interconnection graph of the user-provided simulation modules in PlatformXML, an XML-based platform description language. Experience with several projects reveals that the tool is effective in reducing the development and maintenance time for system software and for embedded applications. The most visible benefits are a shorter modify-compile-test cycle, better support for performance tuning and improved flaw detection. Giano is freely available in source and binary form for non-commercial use.*

## 1 Introduction

A number of companies have introduced microcontrollers that integrate a processor and some amount of programmable logic in a single package. In these "System on a Chip" (SoC) devices the I/O pins are connected to the programmable logic side of the device and it is the final user that defines how the microcontroller interfaces to the external world, not the manufacturer. The manufacturer provides libraries for timers, counters, serial communication ports (UART, $I^2C$, IrDA, and SPI), CRC generators, amplifiers, ADCs and DACs, filters, DTMF tone generators, PWM modulators, LCD drivers. The user is then free to create more.

The SoC is but the natural evolution of a common practice in the embedded space, namely the offloading of the most demanding computations and I/O control functions to a field-programmable gate array (FPGA). In fact, FPGAs have become so powerful that they can eliminate the microcontrollers as separate chips and integrate them directly as "soft-cores". Applications are therefore built of two parts, one in C that executes on a microcontroller and one in Verilog that "executes" on the programmable logic.

The ultimate products of the two compilers for C and Verilog are the binary code to execute on the CPU, the binary file to configure the FPGA, and little confidence that the two will actually work together. Programming FPGAs is fairly complex; it requires familiarity with an additional set of tools for coding, verifying, synthesizing, placing and routing, downloading, and testing the Verilog or VHDL "programs". These processes are time consuming and testing is a challenge due to the growing imbalance between a reduced I/O pin count and the ever growing on-chip functionality

System integration and testing of the complete product reveal functional and performance flaws, forcing these processes to repeat. Notice that we could eliminate this costly repetition if the functional development of software was complete *before* the hardware design had stabilized. Unfortunately, development and testing of system software cannot even begin until a working prototype of the hardware is available. One way to break this impasse and to realize this ideal hardware-software co-development is to use a full-system simulator, one capable of realizing both the instruction level behaviors of instruction set processors, and the behaviors of models defined in a Hardware Description Language (HDL).

We have identified the following six necessary requirements for such a tool: it should be capable (1) of simulating hardware cores for an FPGA or other device, (2) of executing large bodies of code, and (3) of simulating a complete system including (4) a variety of I/O devices and (5) communicating in real-time with the outside world. We expect users to make changes and extensions to the simulator, (6) the availability of source code is essential. While a number of existing simulators can match one or more of these six requirements, none is able to match them all. For instance, *Icarus Verilog* [26] is the only hardware simulator that is currently freely available in source form, but it does not simulate a full-system. Some commercial products allow extensive system simulation, but none can interface in real-time with the outside world. There are many CPU and full-system simulators available in source form but none of them understands Verilog or any other HDL. We therefore resolved to write a new hardware-software simulator, which we named *Giano*[1] [14].

In the rest of the paper we introduce the usage scenarios that motivated the creation of the tool in Section 2, describe the design goals and challenges in Section 3, the current status of our implementation in Section 4, the practical use of the tool in a few case studies in Section 5, and we report on our experiences using Giano over the last few years in Section 6. Section 7 summarizes the related work and our conclusions are found in Section 8.

## 2 Usage Scenarios

An *extensible* simulation system that can easily *leverage* other existing tools, including HDL simulation can adequately support many practical usage scenarios. In this section we describe some of the scenarios where the tool supports our own research. Section 3 shows how these two goals of extensibility and leverage have guided the design of Giano.

[Prototypes] Research at the boundary of Architecture and Operating Systems is hard to evaluate when it is not supported by sufficient empirical evidence [9]. But until a hardware prototype is available it is not even possible to collect this evidence. Only a simulator for a realizable model of the proposed hardware can break this circularity. For instance, the Giano distribution includes a novel Memory Management Unit (MMU) that is fully synthesizable and allows full software evaluation.

[Full-System prototyping] A number of embedded systems are controlled by a microcontroller and an FPGA that offloads some of the most demanding I/O control functions [10]. Giano can prototype the entire control section, and with the addition of tools like LabVIEW [15,16] the whole system including stepper motors and other external devices. Analog signals can be simulated with MATLAB [36] or captured by an A/D converter. As for individual peripherals, a manufacturer can provide (the DLL of) a C model for software developers to work with, in addition to the often unclear specifications in a human language.

[Architectural simulations and benchmarking] Giano can simulate two different CPUs within the same system environment, possibly at the same time. This has rarely, if ever, been achieved with real system implementations. With a full-system simulator, the architect can compile and execute large bodies of code and make meaningful comparisons between different architectural solutions. For instance, running the Doom [6] videogame with ARM or MIPS processors within the same simulated microcomputer produces only slightly different instruction counts but remarkably different frame rates. In a test run, the MIPS system uses 2,339 million instructions to paint 1,514 frames, the ARM system 2,325 million instructions for 2,705 frames. Various factors affect this result, and they *can* be accurately investigated because the simulator improves the visibility of all phenomena under study.

[Hardware/Software Co-Design] Section 5.3 is a case study illustrating precisely how Giano supports the hardware-software co-design processes. In essence, when using a co-simulator coding and formal verification can proceed in parallel with functional simulation, system integration, and testing. If the simulator is flexible enough it is possible to explore alternate solutions before committing to the final configuration.

[Coprocessors] Some processor architectures (MIPS, ARM) define a standard way to interface the on-chip coprocessors. These might be new types of floating point coprocessors, vector processors, multimedia instructions extensions, DSP-like functionality, or cryptographic functions. Giano can simulate these coprocessors in Verilog, while the rest of the system is simulated in C. Test programs and application code and libraries are developed long before the actual device becomes available. The performance implications for the full system are accurately evaluated. A faster simulation of the coprocessor in software can be used for validation against very large test programs and/or to collect valuable traces.

[Rapid software development] Once the first Giano implementation was available we found it surprisingly effective in helping us develop software for our embedded boards. There are three simple practical reasons for Giano's success with our users. In the first place, the compilation times for FPGA devices are quite long, in the order of tens of minutes if not hours. Giano's HDL interpreter might be slow, but it requires zero compilation time. In the second place, downloading and rewriting the FLASH devices in our ARM development boards takes time because they are interfaced via a serial line, like most other boards. A five-to-ten minute reprogramming time goes to zero when using Giano: the FLASH image is simply read into simulated memory and executed. In the third place, a bad software error in the new FLASH image can make the boards unusable and impossible to re-FLASH in software. It becomes then necessary to use a FLASH programmer or some other time-consuming repair procedure. With Giano we can re-FLASH an image to an actual board after we have already tested it, eliminating the hazard.

[1] In the Roman mythology, Giano was the guardian god of doors and temples; the two opposing faces on his head could stand watch simultaneously in two opposite directions.

[Reconfigurable Computing] A desirable property of an FPGA device is to be able to perform partial reconfiguration while the chip is operational. Very few existing devices have this property and none of their simulators. With Giano we can modify the Verilog interpreter to perform partial reconfiguration, at least at the behavioral level.

## 3 Design Challenges

The simulator was designed with *extensibility* as its primary goal. From the beginning we expected that we would be using multiple CPU architectures, MMUs and memory subsystems, different byte-orders, multiple hardware cores, multiple I/O devices, busses and interrupt controllers, and eventually multiple Hardware Description Languages. This goal has affected both the design of the structure of the system and the choice of implementing it in a portable programming language.

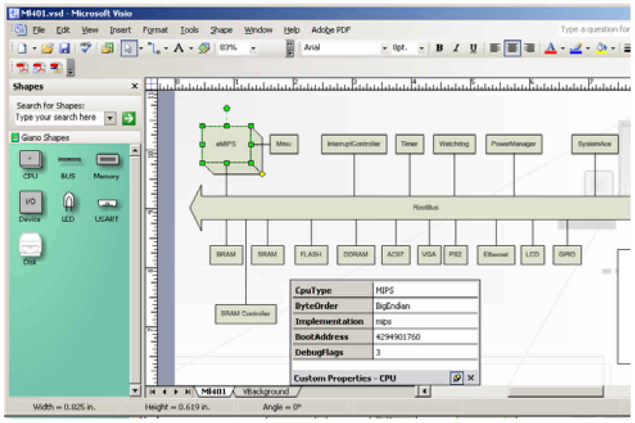
Because of this goal Giano is more of a *simulation framework* than a simulator properly. The core executable is mainly devoted to parsing of the configuration graph expressed in PlatformXML and creating and initializing the graph in memory. The semantic of any given simulation is actually provided by the nodes of the graph, which are dynamically loaded DLLs written by the user. Giano does expect all nodes to derive from a common *GianoModule* class, which provides a method to notify the node of its siblings in the graph and a method to start and terminate the simulation. The core executable is also a good repository for shared functionality and tools that a module implementer would have to write anyways.

A second design goal was to *leverage* the functionality provided by other (large) tools rather than recreating it afresh. The user interface, HDL interpreter, mathematical modeling, physical object simulation can all be realized by separate tools that already exist and we should just find a way to incorporate them in Giano. The same principle applies to individual modules as well. A lot of work has been done to simulate CPUs and caches and their various properties with various degrees of accuracy. Ideally we should be able to incorporate all this work with minimal programming effort.

Selecting such an open-ended basic design for the tool required us to answer a number of questions both at design and implementation time. We will illustrate the most relevant design problems and how these goals guided our solutions in the rest of this section.

### 3.1 User Interface

The first problem for a tool that relies heavily on configuration data is to create and manage the configurations in a simple and intuitive way. Given that the core structure is a graph it was natural to select a graphing tool as the basic user interface. Rather than creating a new one, we chose to leverage Visio, a commercially available tool that is well established, easy to use and is easy to customize. Figure 1 is a screenshot of the resulting User Interface. We wrote a simple function (a macro in Visio parlance) to parse the Visio diagram and generate the PlatformXML file and otherwise just relied on Visio to solve the management problems for us. A second macro solved the problem of starting the simulation, activating the core executable and passing the configuration file name as argument. With just these two macros we allow the user to create a graph depicting an arbitrary computing system and to simulate it.



| | |
|---|---|
| CpuType | MIPS |
| ByteOrder | BigEndian |
| Implementation | mips |
| BootAddress | 4294901760 |
| DebugFlags | 3 |

Custom Properties - CPU

**Figure 1: A Screenshot of Giano**

A second issue for the user interface was passing arguments to the individual simulation modules. Some options can be chosen at compile time but others are better left for the users to select at run-time. A CPU module might have a selectable byte-order for instance, or a memory bank might have a selectable size. Clearly we could not anticipate all of the options all of the modules would define, and passing them all in a command line is certainly possible but impractical for most configurations. Our solution was to use Visio's custom properties to define a list of name-value pairs and to (visually) associate one such list to each individual node in the diagram. Selecting a different module automatically updates the window of the custom properties, which makes data entry and editing simple and natural. The CPU module in Figure 1 is currently selected and the central sub-window shows the list of its properties. These custom properties are added into the PlatformXML file by the parsing macro or manually with a text editor. At runtime, each simulation module is passed a pointer to its own list of name-value pairs, and it is also possible to retrieve the argument list for sibling nodes.

We expected that our users would often want to override the values provided by the configuration file, for instance to change the values of some options or to add new options. We provided a simple mean to do this on the command line by specifying pairs of the form (NodeName::PropertyName Value). We were concerned that the user interface might prove unpopular or too inflexible. In actual use we found quite the opposite, very few users know this command line feature even exists.

### 3.2 Configurations

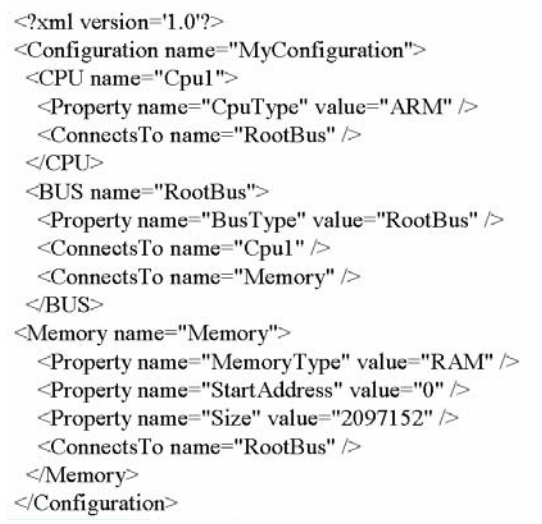
The choice of using XML to describe Giano configurations was based on two goals: we wanted to propose a standard way to describe hardware platforms for use by other simulators as well as ours, and we did not want to restrict the ability to express unforeseen data or any properties that might be required in the future. XML [34] is widely understood and supported, and it is indeed extensible and therefore a good match for our purposes.

A simple example of a PlatformXML configuration file is given in Figure 2. The node tags we defined match the basic classes found in Giano, namely CPU, BUS, MEMORY and DEVICE but they do not actually differ in their internal syntax. They all expect a Name and a list of Property and ConnectsTo clauses. ConnectsTo is used to link nodes by-name; Names must be unique.

### 3.3 Cycle-accurate?

Many users ask whether Giano simulations are "cycle-accurate" or not. The simple answer to this simple question is that Giano does not make any decision on this issue; it actually depends on the choice of modules. The basic framework provides the means for carrying a central clock throughout the entire configuration, but it is then up to the modules themselves to "run" this clock. It is also possible to create multiple clocks and make them visible system-wide, but again it is up to the modules to run and synchronize them. A more complete answer is that a user must identify the specific events she is interested in and to make sure that they are reproduced with the correct timing relationships, at the desired level of precision. Section 3.7 describes in some detail how external events can be handled with accuracy to realize a Real-Time simulation system. Section 3.11 deals with collection of traces and their analysis.

We have realized three different types of CPU simulation modules, at different levels of precision. The first type is intended as a functional simulator that trades speed for accuracy. This is typically written in C++ and/or using dynamic code generation [19] and might, for instance, assume that all instructions and all memory accesses take a single unit of time. A second type of module is realized leveraging SimpleScalar [5]. This is believed to be a more accurate model, even though it does not reflect any real implementation. Leveraging the ARMulator [2] as a Giano CPU module might realize a more accurate system, or at least one that is as accurate as the manufacturer is willing to provide. A third type of module is realized using an HDL simulation of the CPU module. This approach trades accuracy for much lower speed. Section 5.3 describes in some detail how one project used this approach. Note that the accuracy is still predicated on the HDL code being synthesizable for the specific technology/target; a simple behavioral simulation can be just as inaccurate as any C model.



```
<?xml version='1.0'?>
<Configuration name="MyConfiguration">
  <CPU name="Cpu1">
    <Property name="CpuType" value="ARM" />
    <ConnectsTo name="RootBus" />
  </CPU>
  <BUS name="RootBus">
    <Property name="BusType" value="RootBus" />
    <ConnectsTo name="Cpu1" />
    <ConnectsTo name="Memory" />
  </BUS>
<Memory name="Memory">
    <Property name="MemoryType" value="RAM" />
    <Property name="StartAddress" value="0" />
    <Property name="Size" value="2097152" />
    <ConnectsTo name="RootBus" />
  </Memory>
</Configuration>
```

**Figure 2: A Simple Configuration**

Similar considerations apply to the other simulation models. For instance, an accurate cache module designed with Cacti [33] could be interposed between the memory bus and a CPU module.

### 3.4 Multiprocessors

We selected to provide multiprocessor support from the beginning, even though most embedded systems are single-processor and some CPU architectures have never been realized in an actual multiprocessor system. The motivations behind this decision were to be in line with our extensibility goals and the hope that the tool might be useful to perform scalability analysis of existing and/or new software. The tool was indeed used to develop system software for a new multiprocessor system (the Xbox360 gaming platform), validating our choices. In practice, this decision had limited effect on the structure of the system but it did affect the implementation and the eventual performance of some CPU modules. For these, we can use a compile-time conditional to generate a separate uniprocessor version that is somewhat more efficient but this option is not normally used. Consider for instance the multiprocessor synchronization instructions of the MIPS and of the PowerPC processors. Both architectures provide a pair of instructions of the form load-with-reservation and conditionally-store-to-memory. Logically, the load instruction sets a flag to remember the destination address and the store instruction checks that the address is still valid, otherwise it does nothing. Any store to the reserved address by any processor will cancel the reservation and therefore must be monitored. This potentially means serializing all stores to memory from all processors, a serious scalability bottleneck and an overhead on all store instructions. In practice, reservations are rarely used and almost never contested, a fact that we leveraged to eliminate the lock in most instances and consequently improve performance. Processor reservations are a shared class exported by the core executable, which favors reuse. The class can be used either by a cache module or directly by an integrated CPU-cache module.

Notice that it is trivial to create a hybrid multiprocessor system with Giano, the user simply selects a different DLL for each CPU node in the configuration. A hybrid built of the example modules for the PowerPC and MIPS processors would be memory-coherent as well.

The inter-CPU clock synchronization is another issue that strongly affects performance. Ideally, a faithful simulation would dispatch instructions on the various CPUs with the same timings as on the real system. In practice, software is always written to be independent of the relative speeds of the various processors because bus contention, stalls and interrupts all contribute to make such relative speed unpredictable. Taking advantage of this fact, it is possible to ignore the synchronization issue altogether and use independently scheduled threads to simulate all CPUs in parallel. It is left to the simulated software to take care of synchronizing them. This approach provides best performance, especially on a multiprocessor system with a number of real processors matching or exceeding the number of simulated CPUs. The downside of this approach is that the simulation is imprecise and any performance data could be rather misleading, especially when executed on a uniprocessor where threads can easily get out of step.

One alternative is to use a single thread to simulate all the CPUs, for instance by leveraging the work of the PolyScalar project [32] and integrating it into Giano as a CPU module. With this approach, one instruction each for all cores is dispatched in the main loop and all cores are therefore naturally synchronized. All CPUs collectively look like a single CPU module to the configuration system, which treats PolyScalar as a single entity. The bus, memory subsystem and I/O peripherals can remain separate modules.

An intermediate solution is to realize a “close enough” simulation but with better performance. Rather than synchronizing on each instruction the various CPUs can execute a fixed number of instructions and then synchronize. Notice that the deviation from reality is still unbounded, even though the clock skew is now bounded.

## 3.5 Interfacing C and Verilog

When using Icarus Verilog we have the ability to realize any desirable interface between C and hardware models, because we can change the Verilog simulator itself. This is not the case when using simulators for which we do not have access to sources, e.g. all commercial grade simulators. In addition, these simulators require running as individual processes, not as DLLs within our Giano process.

The IEEE 1364 standard Verilog Programming Language Interface (PLI) [23] was defined to solve this and other interface issues for hardware simulators. We used it to interface Giano with a commercial Verilog simulator, ModelSim v5.8c [13]. According to the standard, we created a DLL to be loaded by ModelSim. The DLL exports a Verilog runtime function that is invoked by a Verilog module that desires to interface to the other C model(s). While many other interfaces are possible, we have provided as an example a Verilog module that is based on a dual-ported memory abstraction. We do not attach any semantics to the values or particular locations of the dual-ported memory. They can be used as the register file of a device, or a FIFO or as packets for transactions over an asynchronous bus. A VHDL model can interface to C models by leveraging the Verilog module.

The PLI runtime function passes a handle to the dual-ported memory array to the DLL code for later use. This function is only called once at simulation start time and does not otherwise affect the simulation timings. Also at this time the DLL will attempt to connect to Giano using a TCP connection, possibly on a separate machine. Using two separate machines for the C and the Verilog simulations is often desirable because ModelSim uses the full CPU during simulation and can require large amounts of system memory.

Once the connection is established, an asynchronous read request is kept posted at all times to accept requests from the CPU side. At the end of each simulated step ModelSim invokes a function that quickly checks for completion of the posted read. When data is available, it contains the necessary parameters for performing either a read or a write access to the dual ported memory. A second runtime function can be used to report cycle counts back to the CPU model.

Although we have only tested this extensible interface with ModelSim it should allow us to leverage all the other hardware simulators that support PLI.

## 3.6 Asynchronous Events

Many simulation modules are passive and become active only when the CPU issues a read or write reference to their register file. At that point, they perform all the operations necessary to complete the read or write access in a synchronous manner, namely in the context of the Fetch/Store method call on their object instance. Memories, busses, frame buffers and other output-only devices, and even permanent storage can possibly be realized in this simple way.

Some devices are more active and a synchronous simulation is not feasible. Consider the case of a serial line with programmable baud rate. Not only is data arrival an asynchronous event, but to be accurate the individual bytes of data must be "received" by the USART at a maximum frequency defined by the baud rate. Another example is a programmable timer, a device that generates interrupts at times defined only by the simulated software. Even when it is feasible, synchronous simulation might not be desirable if it is not accurate enough for the user's needs. For instance, a real disk completes operations in times that have a large deviation from any set average. If the behavior of the disk affects the accuracy of the results it must be simulated with greater accuracy.

To deal with these asynchronous modules Giano provides an eventing facility linked to the core clock. Modules can create events that are stamped with the desired time of activation. When the clock reaches that time a method is invoked to return the event to the module. A serial line uses this asynchronous method invocation to deliver the next character, a timer to generate an interrupt to the CPU and a disk to complete the I/O operations at the correct time.

The module that controls the clock is logically responsible for triggering events as well. This is often a CPU module, but it is not required. A simple way to realize the control loop for a CPU thread is as follows:

- Check if any event has triggered and execute the corresponding callback function;
- Check if the CPU has any pending interrupts; these can be generated either synchronously during a previous I/O access or during the callback, or asynchronously by a separate thread simulating an I/O device;
- If present and enabled, use the MMU to translate the program counter's virtual address to a physical address;
- Access the memory bus to fetch the next instruction;
- Execute the instruction. Load and store instructions again access the MMU and the memory (or I/O) subsystem.
- An exception could be triggered by a failed MMU translation, an access to non-existent memory, an unaligned access, or some other arithmetic or protection error.

The eventing facility supports our extensibility goal by adding more flexibility to the implementations. It also helps us leverage existing simulation codes that are rarely, if ever, multi-threaded. Asynchronous events are usually invoked by the CPU module and therefore never run in parallel to other methods like Fetch/Store.

## 3.7 Real-Time Simulation

All existing simulators work on a best-effort basis. Their designers assumed from the start that the simulator would be slower than the real system and no thought was given to matching the temporal behavior of the target system. Many embedded and Real-Time systems operate at relatively slow clock frequencies because cost and power consumption is often a primary concern. Giano already emulates an EB63 board at the correct speed on a relatively slow 800 MHz PC. On more modern PCs it can execute instructions much faster than the original board, which is not at all a desirable property.

To address this issue we can rate-limit the speed of the CPU module, but paying attention to realize a control loop that can be fine-tuned to relatively small speed differentials. A simple scheme is as follows.

- Every *M* thousands of instructions spin idle for *D* microseconds;
- Every *N* millions of instructions check the execution rate against the target rate and adjust the delay *D*.

In this way, the effective delay on each instruction is *D/M*, which can be as small as a few picoseconds and therefore would be impossible to realize on a per-instruction basis. The inevitable overhead is also spread over a larger number of instructions. Checking the execution rate involves reading the current time, which on many Operating Systems is an expensive operation compared to incrementing a counter and performing an integer division. The adjustment of the delay value *D* is based on a first-order filter, which seems sufficient to provide quick convergence without excessive fluctuations. Again, the cost of this computation is amortized over *N* million instructions. It is worth mentioning that this type of feedback loop takes care, at least in part, of the fluctuations in execution speed due to multiprogramming on a general-purpose and non-Real-Time commodity Operating System. The CPU module executes at the target number of MIPS while the user can continue with other activities.

There is a second aspect to Real-Time Simulation that Giano uniquely supports. Real-Time programs are defined correct not only with respect to their internal behavior but also in relation to the timing of their interactions with the external world. A simulator for Real-Time programs must therefore be able to interact with the external world and it must reproduce faithfully the external behaviors of the programs being simulated.

There are two parts to this support: a calibration phase and notification events. Using the rate-limiting algorithm above, a CPU is able to generate a clock with known frequency and to keep it stable (within the limits in accuracy of a general-purpose commercial Operating System). All things being equal, the CPU executes at the same speed on the same system every time the simulator is started. We can therefore record on disk what the clock speed is the first time the simulator is used and reuse this value as a first order estimate on each subsequent run. This provides faster convergence and therefore better accuracy even for short-running experiments. Modules with Real-Time requirements can lookup the actual clock speed and schedule events at the appropriate times in the future. On receipt of a notification event such modules re-post all outstanding events, with an adjusted expiration time.

Notification events are needed to adjust for variations in execution speed due to load changes and also for the inevitable differences in the costs of memory accesses among different simulator configurations. In practice, we observed that a relatively coarse threshold of 1 MHz in clock speed change is a good trade-off between accuracy and the overhead of rescheduling events.

Unfortunately, support for Real-Time Simulation is an innovation that works against our goal of leveraging existing tools. We cannot leverage Real-Time in existing tools because they simply do not support it. It also means that a non-Real-Time module can disrupt the timing properties of an otherwise correct simulation, because it must be aware of time and it is not. Real-Time Simulation does not work against extensibility because it is an optional feature. It does make it more complicated to reuse existing code in some cases, but only if we want to *add* this feature into the existing module.

## 3.8 Power management

The eventing facility can be used to implement power management, as in the case of the At91m63200 microcomputer. On this device, the clock to the CPU core can be halted programmatically and it will automatically resume on the first subsequent interrupt. A similar principle applies in a number of other systems though the interfaces can differ; some use special “halt” instructions, others use special sequences of instructions and yet others use explicit clock frequency control units. The Power Management Controller of the At91m63200 (PMC) is a device that exposes control both of the CPU clock and of other peripheral’s clocks. In Giano, the PMC sends a special internal “power management interrupt” to the CPU when the clock to the CPU is turned off. The CPU unconditionally recognizes this interrupt on the next instruction and then (1) asks the underlying host Operating System to context switch with a minimal sleep, and (2) bumps the clock forward to the next scheduled event.

This extensible scheme has two additional benefits. In the first place, we do not waste time simulating a well-known idle condition and we can speedup some challenging scenarios. For instance, with the MMLite software system (MIC) [8] the serial line plays the triple role of console I/O, file system I/O and networking I/O. In this configuration the serial line is the device that most affects simulation performance.

The second benefit is less intuitive but quite useful in practice. Because the CPU simulation thread is context-switched out during idle time its performance profile matches the one of the simulated CPU. Therefore a host CPU load visualization tool such as XPerf or TaskManager will graphically render the load profile of both the real and the simulated CPU. This also works on a multiprocessor system, provided each simulated CPU thread binds to a different underlying CPU.

Dynamic insertion and removal of modules is important for extensibility and can be considered a form of power management. For instance, the PCI bus supports “hot-plugging”, e.g. the insertion and removal of I/O cards without turning off the power to the whole system. In Giano this is supported in the *ChangeDependent()* method, a base method that every module must implement. This method informs of the addition or deletion of a sibling node, and a PCI bus will take the appropriate actions upon such notification. This method is also used when peripherals on the PCI bus are re-assigned their physical addresses via configuration space write operations or on the EBI bus when the CPU changes the mapping between physical addresses and chip-select numbers. The bus module notifies its parent bus of the change and the device mapping tables can be updated at all levels of the memory hierarchy; in this way all devices are just one function call away from the root bus.

## 3.9 Debugging

We expected to use the tool for debugging software programs too but we did not really expect that this would be quite so popular. Initially we just provided support for remote debugging via the simulated serial lines, or the simulated Ethernet and IEEE 1394 Firewire NICs. This was in line with our stated goals of leveraging existing components without reinventing them. While that worked, we frequently received complains that either the debuggers were not available or that they were not able to help in the most difficult cases. A more promising idea was to attach a debugger over a simulated JTAG connection, but we found that either the tools did not support JTAG at all, or that the interface information was not readily available.

We therefore built some debugging aids directly into our CPU modules to provide symbolic tracing and printing. Tracing can be turned on and off at selected memory locations and it is a per-processor condition, one processor can proceed at full speed while another is being traced. This simple facility has proven quite valuable for debugging interrupt routines, monitoring stores to selected memory locations, and for race conditions that are otherwise hard to capture and/or reproduce. While it is not meant to replace a debugger but only to augment it in corner cases, it nonetheless proves that leverage alone does not always work.

We also built the *Debug* module, a generic module that can be added to any configuration. It uses a command-line window to interact with the user. It can dynamically attach to any of the modules in the graph, stop it, resume it, and read and write to it. This tool is especially useful when there is no CPU module in the simulated system; it still allows inspection of any peripheral register, memory location, or bus control register.

## 3.10 Testing with Oracles

The idea of a "JTAG connection" to the simulator has actually been used in Giano, but for an entirely different purpose. We wanted to test the instruction simulators as extensively as possible, to gain trust that the simulator is faithful to the real hardware. This is a daunting task even for the relatively small number of opcodes that constitutes the ARM instruction set. Doing this manually is not only tedious but also error prone and vulnerable to mistakes and/or misunderstandings in the specifications.

The ARM CPU module creates a TCP socket where we can feed "test-cases" that are automatically executed and verified. In case of a discrepancy, the test case contains all the information needed to reproduce the error and it quickly leads us to the cause of the error. Test-cases are automatically generated by a separate program, which is executed on an "oracle" machine. Tests quite often involve just a single instruction. The test generator encodes the instruction with the desired flags and registers, and specifies the processor state before execution in terms of register values and the contents of the relevant memory locations. The generated test is then executed on the oracle machine and the oracle records the resulting register state as well as the memory locations that were modified by the instruction. This constitutes the *expected result* of the test. All this information is sent off to the target simulator and test generation continues. Billions of tests can be generated, executed and verified automatically in a 24-hour period when using two average performance personal computers. To test the ARM CPU simulator we used the ARM Ltd. ARMulator [2] as the oracle machine and its limited speed turned out to be the bottleneck of the resulting automated testing system. In retrospect, we should have used a Pocket PC that can run an actual ARM processor at a much higher speed.

## 3.11 Collecting Useful Traces

We had implemented two different tracing facilities for Giano, both of which collect their data on every instruction step. This had a noticeable negative effect on performance and we found that the resulting information was only of limited use to our software developers. Number of accesses to SRAM, FLASH, cache, I/O and so forth do explain the CPI but do not readily help locate the source code most responsible for it.

Checking for the start of a basic block is more efficient than invoking the tracing facility on every instruction because it is only done on instructions that do branch. We have therefore developed a simple set of tools for performing code analysis and profiling at the basic block level. These tools are available as part of the Giano distribution. One of these tools analyzes an executable image and generates a corresponding basic block database file. Giano CPU modules can use this database during execution and update the dynamic execution counts for each basic block reached during actual execution. Loading and unloading of executables inside the simulated system is recognized using the standard debugger interface that is provided by the executing Operating System, e.g. some special trap or the invocation of a certain function. Therefore the disassembly and profiling data that Giano generates is not limited to the initial Operating System/runtime image but it is extensible and covers dynamically loaded programs as well. This is a feature that is not found in any other simulator. In a similar fashion, the symbol tables for the executable can be used to generate symbolic traces even for dynamically loaded software programs.

After execution is complete, other tools sort and inspect the updated databases to reveal the most frequently executed basic blocks and their relationship to the original source code. It is conceivable to use Giano itself to identify the basic blocks using dynamic rather than static analysis of the executable images, but their relationships to the names of functions and methods is only known when looking at the symbol tables contained in the executable images.

## 3.12 Portability

The hardware simulation side should execute as many "hardware programs" (hardware designs written in HDL) as possible, just like the CPU simulation can execute arbitrary "software programs". In Giano the hardware model is selected at run time and is not compiled-in into Giano. Modifying the model and restarting the simulation is a matter of seconds even for large designs. According to our extensibility goals, the hardware simulator is not linked statically into the program; it is loaded dynamically at execution time. This feature was particularly useful when we added support for commercial simulators, for which we do not have source code but we still can leverage (see Section 3.5).

Giano is coded in C++ and so are all the example modules that are provided in the distribution. This does not preclude a user from choosing a different implementation language for a module, provided it interfaces correctly to the other modules. We have avoided making any design decisions that would hamper portability. For instance, most existing CPU simulators only simulate 32-bit target processors because this maps efficiently to a 32-bit host processor. Giano supports full 64-bit addresses but it can be compiled for either a 32-bit or a 64-bit IA86 processor. This benefits the 64-bit PowerPC simulation when running on a 64-bit PC and exacts only a limited performance penalty in other cases.

Even the inevitable dependencies on the underlying Operating System are relatively benign. Using a separate DLL for each simulation module gives maximum flexibility but requires that the Operating System supports shared libraries or a dynamic loader. Various modules use separate threads, for instance the CPU modules and some peripherals. This requires that the Operating System supports multi-threading and it can result in much better performance on a multiprocessor system than on a uniprocessor system.

We expected that some of these decisions would negatively affect performance, but we chose to solve that problem separately and only once its relevance was understood. Given the advances in clock frequencies and the fact that the simulator spends all of its time in a relative small section of code it was hard to predict just how bad the performance would be. We measured about 20 MIPS on a 2 GHz Intel processor for an interpretive C++ ARM CPU module, which is both terrible and acceptable depending on the intended use. It is quite acceptable and in fact faster than an Atmel EB63 evaluation board that runs at 25 MHz on a 16-bit memory bus, resulting approximately in a 5 MIPS instruction execution rate. It is terribly inefficient for a PowerPC simulation of an Apple Macintosh G5. One way to boost the CPU performance is to implement a CPU module that uses dynamic code generation rather than pure interpretation [19,4].

### 3.12.1 Testing Portability

To validate the portability of the resulting system we have ported a subset of Giano to the Atmel EB63 development board, running under MIC. We dubbed this version *MicroGiano*, to emphasize the challenge on a board that supports at most 1 MB of SRAM memory. Within this memory budget we had to fit the Operating System, MicroGiano itself, and especially the simulated SRAM and FLASH devices.

MicroGiano simulates a MIPS big-endian processor, RAM and FLASH, a programmable timer, a serial line and little more. It executes MIPS-2 instructions at about 1 MIPS on a 25 MHz ARM board with a 16-bit bus. MicroGiano uses the second serial line connector on the EB63 as its primary serial line. A client connecting to this second serial line will talk to the simulated processor; the primary serial line is still available for the ARM processor. Section 5.3 describes how we leveraged MicroGiano in a hardware project. Yes, we developed MicroGiano using Giano on a PC to simulate an ARM/EB63 board while it was running MicroGiano simulating a MIPS/EB63 board.

# 4 Implementation

The main limiting factors while implementing Giano modules is adequate virtualization support from the host Operating Systems and exclusive-use of system devices. Sometimes the mapping between simulated and real devices is intuitive and easy to realize, as in the case of a graphic window for a frame buffer, or an audio mixer for an ADC converter, or a file for a permanent storage device. At times it can be less obvious; a simulated serial line can map to a physical serial line but then it cannot be used by multiple instances of the simulator because it is an exclusive-use device. Giano can also map serial lines to sockets, which are a more convenient software abstraction and do not create exclusive-use problems. At times the mapping is made impractical by the host Operating System, for instance in the case of the Universal Serial Bus (USB). We wanted to simulate a Host Controller Interface (HCI) device, which is the interface to the USB bus for the CPU. Unfortunately, this device is neither virtualized by the Operating System nor can it be accessed directly in exclusive-use mode like the serial lines. One way for simulating an HCI is to use a dedicated USB HCI card and to write a special kernel mode driver for it, but this is neither easy nor portable. The Ethernet creates a similar problem, but in this case we can leverage the Virtual PC [12] kernel mode driver, which solves both the virtualization and exclusive-use problems. Using the Packet Filter [35] and a raw socket is an alternative approach for the UNIX Operating System.

| | |
|---|---|
| CPU | 13 |
| MMU | 4 |
| Bus | 5 |
| Memory | 4 |
| I/O Device | 48 |
| Interrupt Controller | 4 |

**Table 1: Component Counts**

Besides being feasible, simulation requires minimal levels of performance in certain cases. For instance, audio should play without glitches. This is easy to achieve if we simulate a device without any hardware mixers and/or decoders, or if we can offload this task to the host audio device. The Doom videogame does contain a software mixer and when in use it reduces the frame rate by about 8%. Graphic performance can also be an issue. A simple 2D frame buffer simulation suffices for the graphical UIs (if any!) found in most embedded Operating Systems, but more modern graphics and 3D games require the simulation of complex 3D GPUs. The DirectX or OpenGL APIs can be used to realize the best performance, mapping the device's operations to higher-level APIs and taking advantage of the host's 3D graphic support [24].

In Table 1 we break down the count of components realized by our group into non-overlapping categories[2]. I/O device is the largest category and the fastest growing one. The count of CPU modules is somewhat inflated by the many ways in which the four supported architectures (MIPS, ARM, PowerPC and BlackFin) are realized.

[2] Due to licensing constraints for third party software not all these instances are included in the distribution.

| | Workin g Set | VM Size | Simulated Memory | Module s |
|---|---|---|---|---|
| SPOT | 1.2 | 5.8 | 3 | 15 |
| EB63 | 5.6 | 5.8 | 2.25 | 33 |
| ML401 | 4.3 | 77 | 73 | 20 |
| Xbox360 | 7.6 | 528 | 524.5 | 22 |

**Table 2: Memory Requirements**

Giano is accurate enough to pass all the test programs created by the manufacturer for the Atmel EB63 board. There are some thirty-three modules in this configuration, including some unusual devices such as a serial FLASH interfaced via parallel GPIO pins and a power management controller. Table 2 shows Giano's memory requirements in megabytes of host memory during simulation of this and three other selected systems. All the modules in these configurations are C models and the measurements were done using TaskManager under Windows 2000 SP4. Counts are all-inclusive, for the entire process. The virtual memory requirements match closely those of the simulated system but the working set remains limited even for the larger systems. There is some correlation in working set size between Giano and the simulated systems but the presence of graphic elements such as buttons, LEDs and frame buffer memory causes some strong variations.

## 5 Case studies

In this section we present first a short exercise in hardware-software co-development using Giano. This example is intended for readers not too familiar with mixed-mode simulation, but it also shows in detail the dual-ported memory interface described in Section 3.5, and how Giano is used in practice during code development.

In hardware-software co-development the designer must decide which components are assigned to software and which ones to hardware. The goal is to obtain a target system performance while meeting all the implementation constraints and cost budgets. It is not always clear which components would benefit the most from being realized in hardware in any given system. The case study presented in Section 5.2 tests how well Giano supports the assignment process via mixed-mode simulation.

Section 5.3 describes how Giano influenced the development of the eMIPS computer [11] and its real-time software. This project's final target is a Xilinx ML401 development board, but in the intermediate stages we use an EB63 board coupled with a Xilinx Spartan-3 FPGA board. In this case study, Giano is used for three different purposes: to support the incremental development of a complex hardware project, for hardware-software co-development, and for testing and validation. This example also shows components realized both in hardware and in software, but for reasons other than those illustrated in Section 5.2.

```
1.   module test;
2.   reg  [31:0] mem[3 : 0];  // dual ported memory
3.   integer counter;
4.   reg clock = 0;
5.
6.   initial
7.     begin
8.       //Initialize Giano interface
9.       $PassMemHandle(mem);
10.      // Initialize our state
11.      mem[`Counter_Increment] = 5;
12.      mem[`Counter_State]     = `Disabled;
13.      mem[`Counter_Reset]     = `NotReset;
14.      counter = 0;
15.    end
16.
17.    always @(posedge clock)
18.    begin
19.      case (mem[`Counter_State])
20.      `Enabled:
21.        begin
22.          if (mem[`Counter_Reset] == `Reset)
23.          begin
24.            counter = 0;
25.          end
26.          else
27.            counter = counter + mem[`Counter_Increment];
28.          mem[`Counter_Value] = counter;
29.        end
30.      endcase
31.    end
32.
33.  endmodule
```

**Table 3: Verilog code for the FRC**

## 5.1 Co-simulation Basics

In this exercise we want to provide an elementary timing facility for a system composed of a microcontroller and an FPGA connected via a memory-mapped I/O bus. A 32-bit free-running counter (FRC) shall be the basis of this facility. Table 3 shows the Verilog source code for the counter, to be realized on the FPGA. The dual-ported memory array that is the interface to the microcontroller is defined at line 2. The interface defines controls for starting and stopping the counter and for setting the units of time. The counter itself is declared at line 3. Line 6 starts the initialization block, which invokes the *PassMemHandle()* function that is part of our PLI interface DLL, described in Section 3.5. Line 17 starts the code that is executed on every clock cycle. Iff the counter's State is Enabled the Value field is changed. At line 27, the counter increments by the value of the Increment field, except if it is being Reset.

The test code to be run on the microcontroller itself is shown in Table 4. Note how the COUNTER structure maps directly to the dual-ported memory array in the Verilog code. After starting the counter at line 15 the program engages in some computation in the procedure *Spin()* not shown, then at line 17 reads the value of the counter and prints the results.

```
1.   typedef struct {
2.      volatile UINT32 Increment;
3.      volatile UINT32 State;
4.      volatile UINT32 Reset;
5.      volatile UINT32 Value;
6.   } COUNTER, *PCOUNTER;
7.
8.   #define TheCounter  ((PCOUNTER)0xfff04000)
9.
10.  int main()
11.  {
12.     UINT32 Elapsed;
13.     int SpinCount = parse_args();
14.
15.     TheCounter->State = ENABLED;
16.     Spin(SpinCount);
17.     Elapsed= TheCounter->Value;
18.     printf("Spin(%d) took %d ns.\n", SpinTimes,Elapsed);
19.  }
```

**Table 4: C test code for the FRC**

To execute this example we first start Giano and instruct it to load and execute the embedded Operating System' FLASH image. The Operating System will provide a simple command line interface or similar facility to run programs such as the one shown in Table 4. While the command line interpreter is waiting for input we start the ModelSim Verilog simulator and instruct it to load and run the Verilog model of Table 3. Once the model has connected back to Giano and it is executing we can run the C test and observe the results.

Notice that all of the familiar debugging tools are still available to the user. A debugger can be attached to the simulated microcontroller to debug the C code, and all of the debugging facilities built into ModelSim/Icarus Verilog are still available as well. Giano provides additional debugging tools for the more difficult cases.

## 5.2 Component Mapping

The MIC software system [8] is based on the notion of structuring software in components with clearly defined and controlled interfaces. As long as it implements the same interface, one component implementation is plug-compatible with any other. The idea of this test is to take a software component, remap it to hardware and measure the results within the same system, with the same test data. One such component is the *ICipher* interface for cryptographic cipher functions. We have at our disposal two software implementations of the AES cipher written in C and one hardware implementation written in Verilog. As a first step we add a dual-ported memory interface to the Verilog AES module as per Section 5.1, and debug the resulting core in isolation using Giano. The test vectors are simple commands for the RTOS debugger to read/write the dual-ported memory. The second step is to write a "proxy" implementation of the *ICipher* interface that communicates with the FPGA over the dual-ported memory.

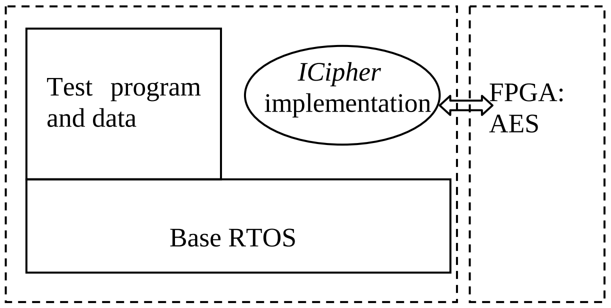


**Figure 3: Testing the AES Components**

The third step is to simulate and test the complete system with Giano, using the configuration illustrated in Figure 3. During testing, one of the three *ICipher* implementations is selected and loaded in memory at runtime. It is the only part that differs between the experiments; everything else is precisely the same. This gives us an exact measure of the reduction in memory utilization and an estimated speedup of 70x for the FPGA implementation over the software implementations. The final step is to synthesize the AES core for our FPGA, which requires a few changes to the purely behavioral portions of the original code. The new code is again tested with Giano, then executed and tested on the real system. The actual performance results are in line with the original projections. We have reached those results in a much shorter time and we have a more accurate understanding of what the system does.

## 5.3 The eMIPS Computer

The development process of the eMIPS computer [11] includes the five intermediate steps depicted in Figure 4. At every step we first use Giano to debug the code more quickly, then MicroGiano and an FPGA for hardware synthesis. In the initial configuration (Figure 4.a) all the components are written in C and are simulated by Giano; Verilog is not used. The modules are instruction decoding and execution (INST), virtual address translation (TLB), physical memory (MEM) and I/O peripherals (I/O). Software development begins as soon as the C models for the components are sufficiently developed. Software developers will continue using Giano through the whole project and abandon it only when the real, faster hardware becomes available. By that time, software is complete and its timing properties have already been verified. In addition to the application code, the software includes any software test programs developed to test individual hardware components, or to identify faults.

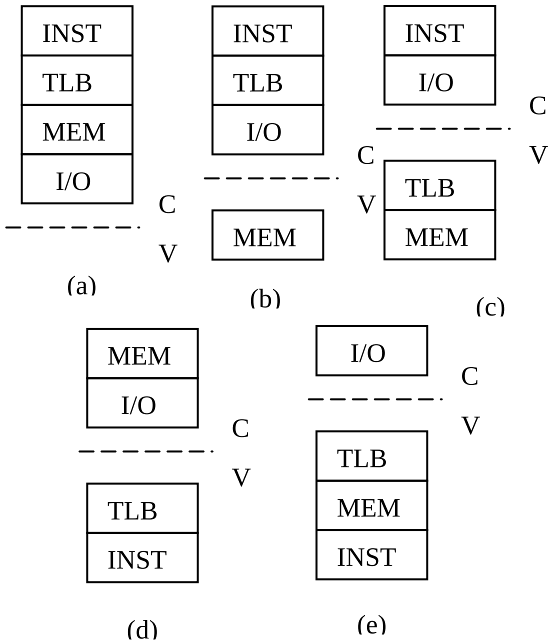


**Figure 4: Stages of the eMIPS Computer**

In the next step of Figure 4.b we used 4 MB of DRAM memory attached to the FPGA board as backing storage for MicroGiano's simulated memory. The DRAM memory interface is a Verilog module developed first with Giano/ModelSim, then realized with MicroGiano/FPGA. This module is augmented in Figure 4.c to include a TLB; at this point (Micro)Giano makes I/O accesses to the FPGA's dual ported memory to perform address translation. The next step is to move instruction execution onto hardware, Figure 4.d. Notice that to help debugging the MEM module has been moved back to a C model, which more easily allows us to collect and verify traces. After this configuration is debugged with Giano, MicroGiano can be replaced by a small program that simply awaits read/write requests from the FPGA side and performs them. In the next-to-final configuration of Figure 4.e the DRAM memory interface module is integrated back into the Verilog code. To move to the final configuration (shown in the screenshot of Figure 1) the I/O peripherals are changed to match the ones on the ML401 board.

The most complex module in this project is the INST module and testing it is a large task as well. We can make use of Giano to accomplish this task, in two different ways. Figure 5 shows a setting with two instances of Giano running separate sets of C models and sharing access to the Verilog model of the INST component. The C models are identical except for the CPU component.

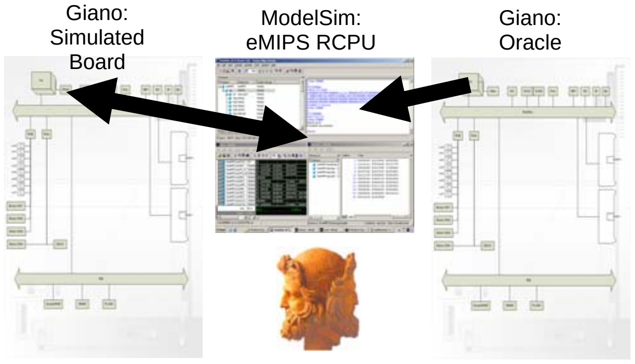


**Figure 5: Using Giano as a System Oracle**

The Giano instance on the right side of Figure 5 (Oracle) uses a true C model of the CPU; the one on the left uses an In-Circuit Emulator component acting as a slave to the Verilog component (center). The Verilog model performs the validation of its results at instruction retirement time, by checking the data produced by the Oracle model. This data includes the instruction's address and the values of all registers that are visible in the ISA.

This approach allows us to test the Verilog model by executing the final application software and any special functional tests. It if therefore a good common testing ground for the software and hardware engineers. Unfortunately, it does neither measure the test code coverage nor does it expressly cover any boundary cases. The diagram in Figure 6 shows an alternate setting, this time with a special TestGenerator Bus component that acts as a test generator. Both the CPU Oracle component and the Verilog component (FPGA) talk to this common Bus to access memory... which is not in the picture because there is none. The TestGenerator Bus will programmatically (or by using traces) create the test instruction sequences, provide the values for all memory reads and verify that memory stores are for the same addresses and with the same values.

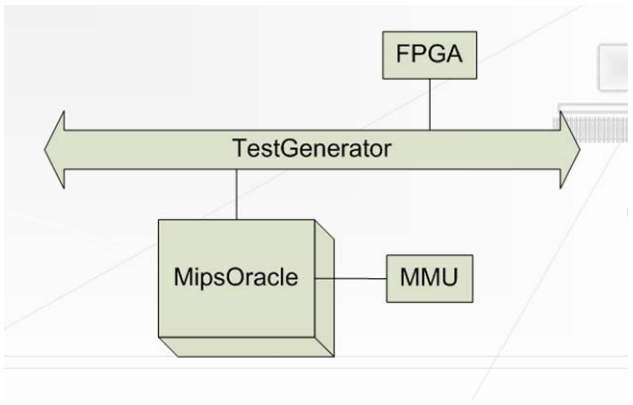


**Figure 6: Using Giano as a Test Generator**

## 6 Other Practical Experiences

Giano has been in use inside our research group for years and has been available to external parties now for two years. In addition to helping us carry out our research the tool has helped us and others in a variety of other ways:

- Giano has all but replaced the EB63 development boards for regular software development. Developers appreciate both the speed and the simplicity of use.
- We will be able to continue using the simulated boards long after the real ones are discontinued.
- In general, development boards for embedded processors are not meant to be real products; their life expectancy is limited. This creates a problem for software developers, who must invest their time on something of a moving target. Giano makes the platform stable for an indefinite amount of time.
- We have not yet found a case where something would work on Giano and not on the real hardware.
- With Giano we have found and fixed many subtle (timing) errors that had eluded us on the real hardware for some time.
- Regression testing with Giano is an automated part of the compilation process for the MIC system. When new changes are applied to the MIC software they are automatically tested by the individual developer with Giano before they are released to the rest of the developer's group.
- In at least two cases, the MIC real-time system was ported to a previously unsupported CPU architecture using Giano directly.
- Basic block profiling helps us make our software more efficient, often with surprising findings.

- All of our major technical demonstrations have used Giano in some way, including a secure consumer electronics system and a distributed real-time planning and execution system. We often use a mix of real and simulated boards; in one instance, we used a simulated board to replace a real one that had failed during the demonstration.
- The Xbox 360 software team used a simulator derived from Giano to develop the system software in advance of actual hardware availability. This has given them a net six months advantage on the software schedule and allowed them to meet an extremely demanding timeline for delivering the product on time for the Holiday season.
- The early versions of the Microsoft SPOT watches were similarly developed using the Giano simulator. System software was again ready well in advance of actual hardware availability.
- Researchers at Portland State University are using Giano for run-time verification of temporal logic properties using the Property Specification Language PSL [1], an IEEE standard. Giano can support PSL both on the software and the hardware side. They have also added support for mathematical modeling of input signals, using MATLAB [36].
- At Texas A&M University Giano is used in teaching a Microprocessor Design course at the senior undergraduate level.
- Students at Texas A&M are using Giano to simulate and test a complex add-on board cascading multiple-busses, including the PCI and ISA busses. This setup involves multiple cooperating instances of Giano, it is cycle-accurate and does not involve any CPU module.

## 7 Related work

Simulators are naturally CPU-intensive applications that have benefited from the performance improvements of the last decade. Full-system simulation is now possible [20,21,12,18] and the implementations use different approaches for best performance. In Giano performance does not dominate the design, functionality does. For instance, using the Virtual Machine approach to simulation is limiting to the one architecture being virtualized, namely the x86. With SimOS [20] it is possible to simulate an entire Operating System, but only if special device drivers are used; this prevents it from being able to execute existing commercial Operating System binaries and furthermore it cannot be used to test or develop new peripherals and/or their device drivers. None of these full-system simulators is easily extensible or supports Real-Time; none supports HDL modules, only SimOS is available in source form.

The SimpleScalar [5] simulator is used in computer architecture education and research. It simulates a MIPS-like processor, either in big-endian or in little-endian mode. A simulation run under SimpleScalar is equivalent to execution within a UNIX process and most UNIX system call traps are supported, except for *fork/exec*. Many commercial CPU simulators provide a similar level of abstraction. SimpleScalar is neither synthesizable nor does it allow simulation of the Operating System code and of the I/O devices; consequently it cannot support Real-Time. It has been extended in a number of ways, for instance for power usage modeling [27] and for multiprocessor simulation [32].

With ARMulator [2] and Simics [25] the user can define programmatically the memory model, which allows for I/O peripheral simulation. These simulators keep an account of the cycles spent on each read/write operation by the CPU, but do not support Verilog, do not support multiprocessors and are not Real-Time. Their sources are not available, but both are user-extensible.

ModelSim [13] is popular with hardware designers; it has both a graphical and a command line UI. It can simulate both behavioral and synthesized Verilog and VHDL code. Fast CPU simulation is available for some "soft-cores". ModelSim is not a full-system simulator, does not support Real-Time and it is not available in source form. Giano similarly relies on a separate module (Visio) to provide the graphical UI. Seamless [37] improves on ModelSim by allowing full-system co-simulation but it is not Real-Time and not available in source form.

Atmel's FP*SLIC* [3] is a development kit for an 8-bit SoC that includes a CPU and a CPLD. The kit includes a simulator, but only for the CPU side.

SIMH [22] is a project aimed at building simulators for historical computers; it started with the PDP-11/23 "fuzzball" routers of the old ARPANET and now covers many other retired computers. Some of the embedded computers that Giano simulates have a similar long life expectancy.

Games are interactive programs, they must provide an adequate response time or they are just not playable. Game consoles often use hardware solutions that are difficult to reproduce in software, for instance for audio and graphic effects. Many game consoles have a simulator that can play their games [7], sometimes for the love of the games and sometimes to cheat and pirate, but always as a challenging engineering enterprise.

## 8 Conclusions

Giano is a Real-Time, full-system, hardware-software co-simulator that we have developed to support our research in Embedded Systems and in Reconfigurable Computing. We identified six necessary requirements for such a tool: it should be capable (1) of simulating hardware cores for an FPGA or other device, (2) of executing large bodies of code, and (3) of simulating a complete system including (4) a variety of I/O devices and (5) communicating in real-time with the outside world. To let users make fundamental changes to the simulator in the future (6) the availability of source code is essential. No existing simulator possessed all these features.

Giano's design was guided by the two goals of maximizing the tool's extensibility and to leverage other existing tools to the maximum extent possible. These goals were rarely at odds with our requirements and led to some innovative solutions. Giano is the first simulator that supports Real-Time Simulation and the symbolic performance analysis of dynamically loaded software. Practical experience has demonstrated that the tool is effective in shortening the modify-compile-test cycle, in supporting performance analysis and tuning, and to detect flaws more quickly and more accurately.

## References

[1] Accellera *IEEE P1850 PSL*.
[2] ARM Ltd. *The ARMulator.* ARM Ltd, Cambridge, UK.
[3] Atmel *FPSLIC Programmable System-Level Integration* Starter Kit P/N ATSTK94.
[4] Bond, B. *Windows CE Device Emulator* available at http://msdn.microsoft.com/mobility/windowsmobile/downloads/emulatorpreview/default.aspx
[5] Burger, D., Austin, T. M. *The SimpleScalar Tool Set, Version 2.0.* Technical Report 1342, June 1997, University of Wisconsin-Madison.
[6] Carmak, J. *Doom* source at http://www.doomworld.com
[7] See for instance http://www.emulator-zone.com
[8] Helander, J., Forin, A. *MMLite: A Highly Componentized System Architecture.* Eight ACM SIGOPS European Workshop, Sintra, Portugal, September 1998.
Download at http://research.microsoft.com/invisible/
[9] Koldinger, E. J., Chase, J., Eggers, S. J. *Architectural Support for Single Address Space Operating Systems.* ASPLOS 1992, New York, NY, pp. 175-186.
[10] Liu, S. et al. *Marionettes*
at http://faculty.cs.tamu.edu/jcliu/web_462/marionette.mov
[11] Pittman, R. N., Lynch, N. L., Forin A. eMIPS, a *Dynamically Extensible Processor*. Report no MSR-TR-2006-143, Microsoft Research, WA.
[12] Mann, A. *The Rational Guide to Microsoft Virtual PC 2004*. Rational Press, Rollinsford, NH, 2004.
[13] Mentor Graphics *ModelSim* at
http://www.mentor.com/products/fpga_pld/simulation/index.cfm
[14] *Microsoft Giano* at http://research.microsoft.com/downloads/ and http://www.ece.umd.edu/~behnam/giano.html
[15] Morong, C. *LabVIEW for Dummies*
at http://www.iit.edu/~labview/Dummies.html
[16] National Instruments *LabVIEW 8*
at http://www.ni.com/labview/
[17] PCI SIG *PCI Local Bus Specification* Rev 2.2, December 1998.
[18] Pratt, I. *Xen* at http://www.xensource.com/
[19] Reshadi, M. et al. *Instruction Set Compiled Simulation: A Technique for Fast and Flexible Instruciton Set Simulation*. DAC 2003, Anaheim CA.
[20] Rosenblum, M. et al. *Complete Computer System Simulation: the SimOS Approach.* IEEE Parallel and Distributed Technology: Systems and Applications, Vol 3.4, Winter 1995, pp. 34-43.
[21] Rosenblum, M. et al. *VmWare Virtual Platform*
at http://www.vmware.com/support/pubs/
[22] Supnik, B. *The Computer History Simulation Project*
at http://simh.trailing-edge.com/
[23] Sutherland, S. *The Verilog PLI Handbook.* Kluwer Academic Publishers, Norwell, MA 2002.
[24] Tong, X. MSR Asia, personal communication.
[25] Virtutech Inc. *Simics* at http://www.virtutech.com
[26] Williams, S. *Icarus Verilog* at ftp://icarus.com/pub/eda/verilog//v0.7
[27] Ye, W. et al. *The Design and Use of SimplePower: A Cycle-Accurate Energy Estimation Tool.* Proceedings of the 2000 Design Automation Conference, pp. 340-345.
[28] Reshadi, M. et al. *An Efficient Retargetable Framework for Instruction-Set Simulation*. CODES+ISS'03, October 2003.
[29] Engel, F. *A Generic Tool Set for Application Specific Processor Architectures*. CODES'02, May 2000.
[30] Wieferink, A. et al. *A Generic Tool-Set for SoC Multiprocessor Debugging and Synchronization*. ASAP'03.
[31] Braun, G. et al. *A Universal Technique for Fast and Flexible Instruction-Set Architecture Simulation*. IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems, Vol 23-12, December 2004.
[32] Franklin, D. "PolyScalar" available at
http://www.csc.calpoly.edu/~franklin/PolyScalar/Home.htm
[33] Shivakumar, P and Jouppi, N. P. *CACTI 3.0: An Integrated Cache Timing, Power, and Area Model*. WRL Research Report 2001/2.
[34] Bray, T. et al. Extensible Markup Language (XML) 1.0. W3C Recommendation, February 1998. Available at http://www.w3.org/TR/REC-xml-20060816
[35] McCanne, S., Jacobson, V. *The BSD Packet Filter: A New Architecture for User-level Packet Capture*. Proceedings of the Winter'93 USENIX Conference.
[36] The Mathworks *MATLAB*. Natick, MA.
[37] Mentor Graphics *Seamless*. Wilsonville, OR.