# A Hardware-Software Co-simulator for Embedded System Design and Debugging

A. Ghosh, M. Bershteyn, R. Casley, C. Chien, A. Jain, M. Lipsie, D. Tarrodaychik, O. Yamamoto

Mitsubishi Electric Research Laboratories, Inc.

Sunnyvale, CA 94086

## Abstract

**One of the interesting problems in hardware-software co-design is that of debugging embedded software in conjunction with hardware. Currently, most software designers wait until a working hardware prototype is available before debugging software. Bugs discovered in hardware during the software debugging phase require re-design and re-fabrication, thereby not only delaying the project but also increasing cost. It also puts software debugging on hold until a new hardware prototype is available.**

**In this paper we describe a hardware-software co-simulator that can be used in the design, debugging and verification of embedded systems. This tool contains simulators for different parts of the system and a backplane which is used to integrate the simulators. This enables us to simulate hardware, software and their interaction efficiently. We also address the problem of simulation speed. Currently, the more accurate (in terms of timing) the models used, the longer it takes to simulate a system. Our main contribution is a set of techniques to speed up simulation of processors and peripherals without significant loss in timing accuracy.**

**Finally, we describe applications used to test the co-simulator and our experience in using it.**

### 1 INTRODUCTION

Design of embedded systems containing both hardware and software requires solving several unique and difficult problems [4] [5] [12]. One of the interesting problems is that of debugging embedded software in conjunction with hardware. The traditional co-design process, where the software is debugged after hardware is fabricated, produces large design delays due to late discovery of errors in the hardware and in the interface between hardware and software. Integration on a chip will make this problem worse because currently used tools like In-Circuit Emulators (ICE) cannot be used and signals on a chip cannot be easily observed. There is an obvious need for a change in the co-design methodology whereby software and hardware can be jointly debugged earlier in the design cycle. However, this change in methodology can only happen when appropriate design tools are available.

There are two approaches to debugging hardware and software without building the actual hardware. The first one is based on emulation of hardware using, for example, Field Programmable Gate Arrays (FPGA) and using a separate board for the processor and memory. A designer can generate a prototype relatively quickly and debug software and interfaces on the prototype. After bugs are detected, the entire system can be recompiled within a relatively short time. In most cases, hardware emulators run only an order of magnitude slower than the actual system, allowing the designer to test the system with a large number of test cases. However, due to its high cost, this technique is economically feasible only in certain cases. It also cannot be used to model timing constraints accurately and in many cases designs have to be modified to suit emulation. Moreover, re-compiling hardware takes more time than compiling software or a HDL (Hardware Description Language) model for simulation. Finally, it is not always possible to observe the internal state of the circuit, both in the FPGA and in the processor, making debugging complicated and slow.

A complementary approach is to build software models for all the components of the system and use simulation to analyze behavior. There are many advantages of this approach. First, software can be combined with behavioral-level hardware descriptions to detect bugs as early as possible in the design phase. Hardware, software and interface routines can be designed and debugged in parallel. Second, timing constraints can be accurately modeled. Third, re-compilation of either hardware or software is quick. Detailed debugging, where internal states of all components can be accessed and altered at all time points can be easily supported. Finally, this approach is not as expensive as emulation.

Simulators have been mostly used for the design of hardware and there are few tools for co-simulation. In this paper, we describe a hardware-software co-simulator that can be used in the design, debugging and verification of embedded systems. This tool contains a simulation backplane which can be used to integrate processor, hardware (HDL) and dedicated simulators for peripherals, forming a co-simulator capable of efficiently simulating hardware, software and their interaction. Each simulator implements debugging functions like setting breakpoints, examining and altering internal states, single stepping, *etc*. In order to feed stimulus to the system and to observe its response a set of virtual instruments have been created. The co-simulator and the virtual instruments can be used to create a virtual laboratory that will provide users with a platform for rapid virtual prototyping. Performance metrics (like clock cycles needed to execute software) can be easily evaluated, allowing the user to explore different algorithms, hardware and software implementations and hardware-software trade-offs.

The main drawback of simulation is its speed. In many cases, simulation runs orders of magnitude slower than the actual system. Simulation time depends on the (timing) accuracy of the models, with time increasing with increased accuracy.

Therefore, reducing simulation time without sacrificing timing accuracy becomes a very important problem. Our main contribution is a set of techniques to speed up simulation of processors and peripherals without significant loss in timing accuracy. Processor simulation speed is improved by accurately (in terms of timing) simulating only those cycles where there is interaction with peripherals and by caching results of instruction decoding. Suppression of periodic signals and other techniques to be described speed up simulation of peripherals. Simulation overhead is kept low by managing time more efficiently.

We expect this tool to be used at any point after the initial architecture is determined. Software designers may use behavioral hardware models for initial debugging, evaluation and exploration of algorithms and implementations. System architects may use the tool to determine hardware-software tradeoffs. Hardware designers can use prototype software to evaluate, test and debug their hardware. Finally, when hardware and software are ready, designers can work on testing and debugging the entire system.

The rest of this paper is organized as follows. Previous work is described in Section 2 followed by a description of the co-simulation framework in Section 3. Simulator coordination is the topic of Section 4. Simulation of processor is described in Section 5 followed by simulation of custom hardware in Section 6. Simulation of standard peripherals is described in Section 7. Interface to other simulators is described in Section 8. Example applications used to test the co-simulator are described in Section 9. Conclusions and directions for future work are presented in Section 10.

## 2 PREVIOUS WORK

In [7], a debugging tool for embedded system software is presented. The software is cross-compiled for the embedded processor and then executed on a model of the system. The system is modeled completely in hardware and simulated using a hardware simulator. During simulation, which may take several days, all interaction between the processor model and the surrounding hardware is logged. After simulation, the designer switches to a software debugging environment on a host workstation where the code is compiled for the host and re-linked to pseudo hardware drivers that interact with the logged information. The primary advantage of this approach is that during debugging, software can run at the host computer speed. However, when a bug is fixed, the entire system may have to be re-simulated, thereby increasing the debugging time. Further, during debugging, there is no way of interactively affecting system behavior by feeding the system a different set of inputs. In our opinion, such a debugger has limited usefulness.

An interesting approach presented in [1] is based on distributed communicating processes modeling hardware and software. Software is run on a host workstation and all interactions with hardware are replaced by remote procedure calls to a hardware simulator process. The main drawback of this approach is that there is no notion of timing accuracy as neither the software execution speed nor the interface between hardware and software are accurately modeled.

The Poseidon co-simulator is described in [4]. An event driven simulator is used to co-ordinate the execution of a hardware and a software simulator. The processor simulator is tied closely to the DLX microprocessor [4] model. There is no special handling of standard peripherals and little information regarding the debugging environment, simulation speed and accuracy is available.

In [6] the use of Ptolemy [2] in hardware-software co-design for a digital signal processing (DSP) application is described. The emphasis in [6] is on the use of the capabilities of Ptolemy for heterogeneous simulation and code synthesis for single and multiple processors. After code generation and hardware synthesis, co-simulation is performed using the hardware simulator *Thor* [13] and a simulator for the digital signal processor DSP56000. It is our belief that though what is described here in terms of the backplane and what is provided by Ptolemy may be similar in principal, Ptolemy does not address the efficiency issues related to hardware-software co-simulation, especially the simulation of processors and peripherals. From [6], few details are available regarding speed of simulation, accuracy, the way standard peripherals are handled and about the debugging environment.

The use of virtual instruments was introduced in [3] in the context of simulation of hardware systems. Currently, the tool described in [3] does not have any capabilities for hardware-software co-simulation. Use of a simulation backplane in mixed mode simulation is described in [10] and similar backplanes for the integration of hardware simulators are commercially available.

## 3 CO-SIMULATION FRAMEWORK

In designing the co-simulator the main goals were:

- to provide fast and timing-accurate simulation;
- to provide an extensible and flexible simulator-independent framework where new simulators can be easily integrated;
- to provide adequate debugging capability for both hardware and software;
- to provide virtual prototyping capability through the use of virtual instruments;
- to provide means for evaluation of performance metrics.

The architecture of the co-simulator is shown in Figure 1. We believe that different parts of an embedded system will be simulated using different simulators and therefore we need to allow for heterogeneity in the simulation environment. To

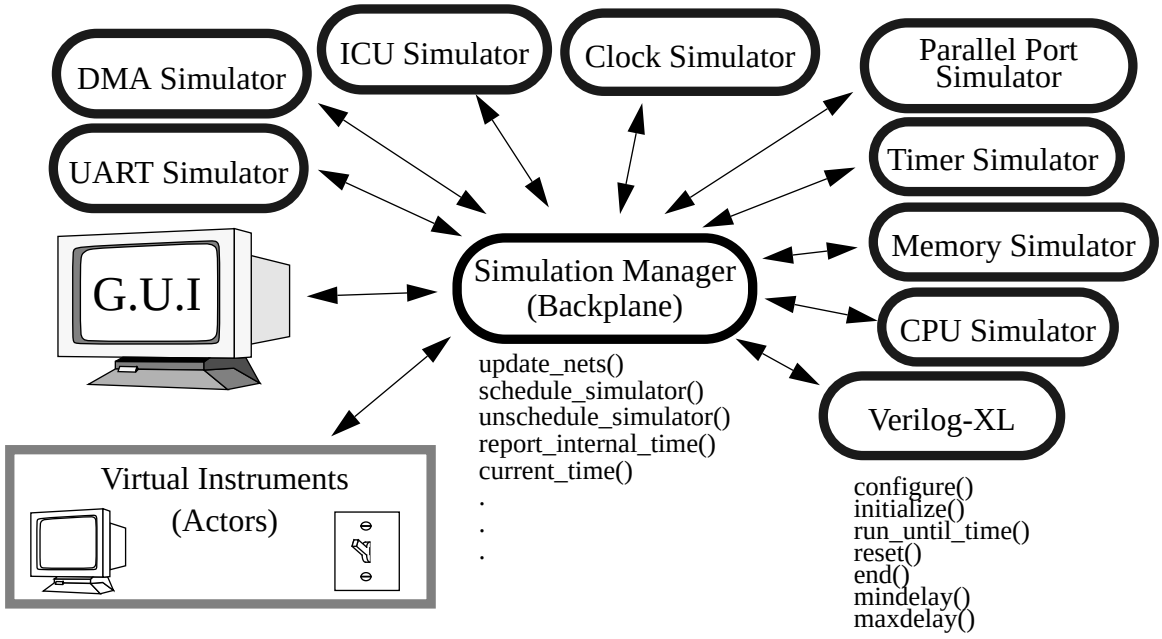


**FIGURE 1. Architecture of the co-simulator**

allow different simulators to interact with one another, a simulation backplane is used. This backplane, also called the simulation manager, is the main component of the tool. It manages simulation and debugging as well as communication with virtual instruments. A well-designed Graphical User Interface (GUI) makes the use of the co-simulator easy and natural for both software and hardware debugging. At the time of writing, only one processor simulator for an M16 microprocessor [8] and several dedicated simulators for standard peripherals have been integrated. A commercial simulator, Verilog-XL [14], is used for the simulation of hardware described in Verilog HDL.

The system to be simulated can be broadly divided into electrical and mechanical components (or even chemical components). The electrical components could be either hardware or software. The hardware could further be digital or analog. For example, to simulate a motor control system, we need to simulate the controller (electrical) as well as the motor (mechanical). Simulators for standard components are provided. It is our assumption that for special components like motors, engines, *etc*., the user will be supplying their own models and/or simulators.

The input to the co-simulator is a description of the system to be simulated. It consists of the following items: a list of blocks and their simulators; a list of nets connecting the blocks; a list of virtual instruments and their connections; and a list of source/object files used by the software debugger and source/library files used by the hardware simulator. The simulation manager reads the system description, allocates necessary data structures and initializes all the simulators that would be needed to simulate the system. Once the system is loaded, the user may interact with any simulator, setting breakpoints, examining registers, *etc*. During simulation, virtual instruments are used for human interaction. When a breakpoint in any simulator is reached, simulation is stopped and the user is prompted for commands. Whenever a prompt is displayed, the user can issue commands for any simulator. Batch mode simulation can also be selected when no interactive input is required.

The co-simulator is implemented as a multithreaded program to allow easy integration of stand alone simulators. The simulation manager and some dedicated simulators constitute the main thread. Verilog-XL and the simulator for M16 are separate threads.

#### 3.1 SIMULATION MANAGER

The simulation manager, hereafter SM, is the backbone of the co-simulator. It performs the following important functions.

- It manages the simulation and debugging session. All user commands are relayed by the GUI to the SM. It understands commands for loading the system to be simulated, for running simulation as well as for debugging (*e.g.* setting breakpoints at certain times). The SM also directs commands to simulators.
- It manages communication between the co-simulator and virtual instruments.
- Simulation of a system involves coordinating the activities of simulators, each of which is responsible for simulating a part of the system. The SM controls when a simulator is invoked, what events are passed to it, *etc*. This is the most important function of the SM and is discussed in Section 4.

#### 3.2 GRAPHICAL USER INTERFACE

The graphical user interface, built using *Tcl/Tk* [9], allows the

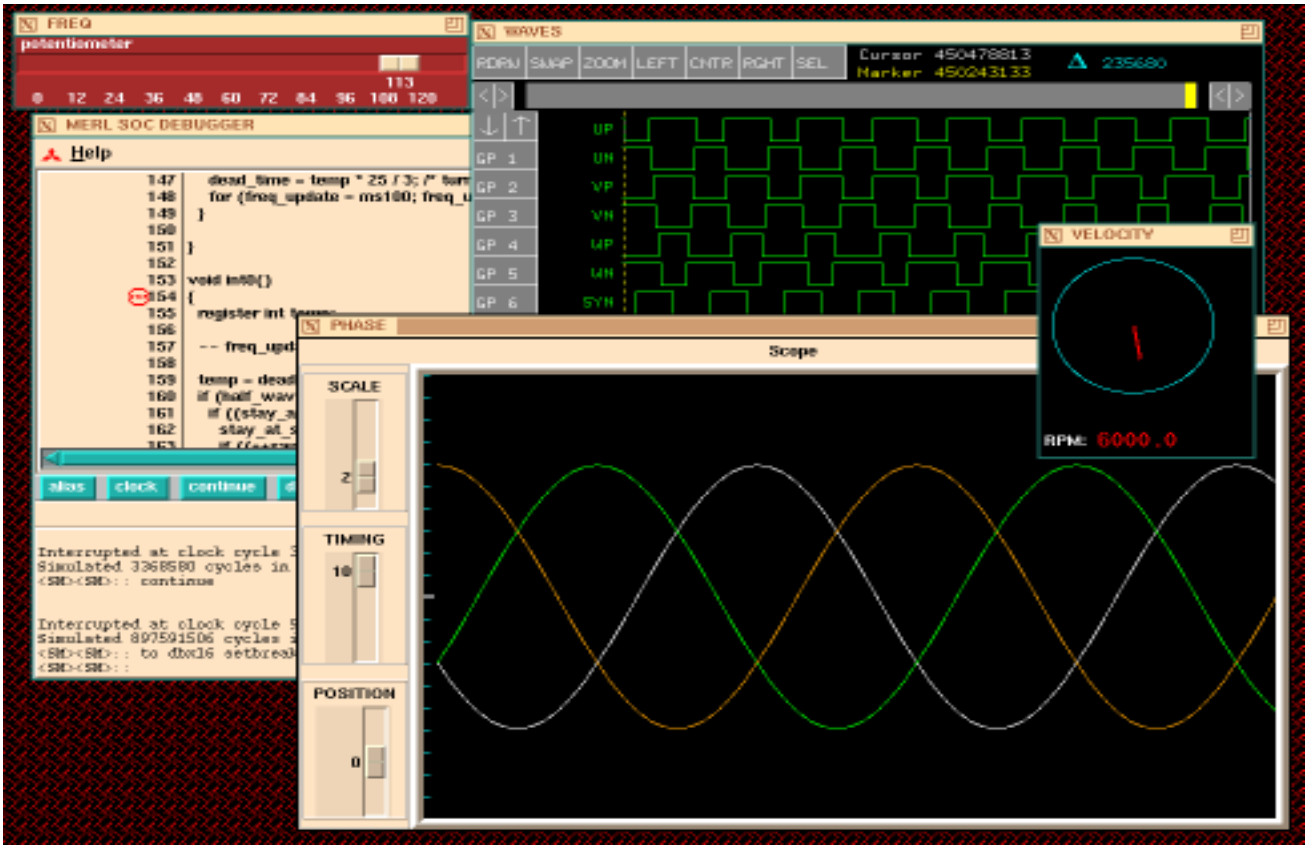


**FIGURE 2. The graphical user interface, virtual instruments, and waveform display window**

user to interact with the SM and the simulators easily and effectively. A snapshot of this interface is shown in Figure 2. The GUI consists of a source display window where source (for both software and hardware) and assembly-level code is displayed. There is also a command window for entering commands and a configurable button panel for frequently used commands. The source display window is used to display breakpoints, the current line where execution has stopped, and other relevant information found in most software debuggers. Additional windows are used to display variables, waveforms, *etc*.

### 3.3 VIRTUAL INSTRUMENTS

Virtual instruments, also called *actors*, are used primarily for human interaction with the system being simulated. They are used to provide stimulus as well as to observe response. As such they model parts of the environment with which the system interacts and enables the user to use the co-simulator as a virtual laboratory. They are implemented using *Tcl/Tk* [9]. Each virtual instrument is a separate process that communicates only with the SM using Unix sockets. The SM manages the socket traffic as well as the starting and termination of each actor. The virtual instruments that have been implemented include a variable voltage/current source, a switch, a simple LED probe, a meter, an oscilloscope, a video monitor, an electric motor and an automobile engine. The voltage source, electric motor and oscilloscope actors are used for a virtual prototype of a 3-phase motor control system shown in Figure 2. The voltage source actor (FREQ) has a slider that can be pulled to change the value of the voltage generated. Waveforms are viewed on the oscilloscope actor (PHASE). The motor actor (VELOCITY) shows the current motor r.p.m. Using virtual instruments, users can get both a quantitative measure as well as a qualitative feel for the system. In the example of Figure 2, the user can see the actual waveforms that would be generated by the system without building hardware and using an oscilloscope.

### 3.4 Simulators

It is possible to represent an entire system, including processor, memory, peripherals and custom circuitry in a HDL like Verilog and simulate it using a simulator like Verilog-XL. Using the right models, simulation can be accurate but will be very slow [7]. Our approach to speed up simulation is to create dedicated simulators for standard components like processors and peripherals and integrate them into a co-simulator using the simulation backplane.

The simulator for M16 is also a software debugger with sophisticated debugging capabilities. It allows both source-level as well as assembly-level debugging. It can also evaluate performance metrics like the number of clock cycles needed to execute a piece of code. Verilog-XL is a hardware debugger with capabilities that include display of waveforms (as shown

in the WAVES window in Figure 2), monitoring of signal values, determination of set-up and hold time violations at latches, *etc*. Simulators for peripherals allow very primitive debugging like examining and setting internal registers. The debugging capabilities of simulators combined with those of the SM provide a powerful debugging and verification environment for embedded systems. It should be emphasized that a natural debugging environment is provided for both software and hardware, so neither the software nor the hardware designer is at a disadvantage.

#### 4 SIMULATOR COORDINATION

The interface between the SM and a simulator consists of a set of functions, some implemented in the simulator and some in the SM (shown in Figure 1). A simulator simulates one or more blocks of the same type, with each block having a set of input and output pins. From a simulator's point of view, it is given a set of events at a particular time, which indicate a change in signal value on the input pins, and asked to simulate until some time in the future. During simulation, if the signal value at one of the output pins of a block changes, the simulator reports to the SM the new value and the time this event happened and stops simulating further. The SM sees the system as a set of blocks connected by nets. Whenever there is an event on a net, simulators for the blocks affected by the event are invoked. Simulator coordination includes determining which simulators to invoke, what events to pass to them and the simulation time when a simulator should stop simulating and return control to the SM. Simulator coordination overhead can be reduced by decreasing the number of events, allowing simulators to run uninterrupted for as long as possible, and managing time efficiently.

To manage time efficiently, the SM counts time in units of a fixed time called the *simulation period* and also bounds the size of the timing wheel. This has several important consequences. Since events can be produced only at certain times and a limited time into the future, the number of unique times to manage is smaller. It allows us to statically allocate the timing wheel before simulation begins. This reduces the run-time overhead in managing time and the timing wheel. Discretization of time allows us to take advantage of the cycle accuracy of processor and peripheral simulators which produce events only at discrete times. However, when timing accurate simulators (like circuit simulators) are used, events can be produced at any time. The interface routines round event times to the nearest discrete time value, thereby introducing errors in simulation. A small enough *simulation period* can reduce this error, but may offset the benefit obtained from using discrete time.

Simulator coordination and synchronization can be understood by following a co-simulation session. After the system description is read, the SM determines the simulators that need to be run and calls the **configure()** routine to let the simulators know that their services would be needed. Subsequently, for each block, the SM calls the **initialize()** routine with a pointer to the block, the pins of the block and the nets connected to the pins. This allows simulators to initialize their internal data structures and their interface routines. After this, the SM allocates and initializes the timing wheel. Each simulator is asked to report the minimum and maximum delay of each block it is going to simulate through the **mindelay()** and **maxdelay()** functions. The minimum and maximum delays are the minimum and maximum time required, respectively, for any event at an input to propagate to an output. Simulators that can ascertain the value of minimum and maximum delay may report it and the rest (like a circuit simulator) report a negative value, indicating unknown delays. The *simulation period* is decided on the basis of the timing accuracy required for simulation and is usually chosen to be the time between successive clock transitions of the processor/bus clock. The maximum of the maximum delays is used to guide the selection of the size of the timing wheel. This size is advertized to all simulators which can then use it during self-scheduling (to be described shortly).

During simulation, the SM first determines events at a particular time and the simulators that need to be run. If there is only one simulator to run, the SM determines the time for the next event on the timing wheel (if there is no event on the timing wheel, this time is considered to be infinity). It then calls the **run_until_time()** function in the simulator with an event list and a variable *stop time* set to the time of the next event on the wheel. If there are more than one simulator to run, the SM determines the minimum of the minimum delays of the simulators. This minimum delay is added to the current time to determine the *stop time*. This ensures that no simulator simulates beyond a time where an external event for it may be produced, thereby obviating the need to roll back simulation time. This is called running in lock step.

Each simulator, during **run_until_time()** transfers all external events to its internal event queue and simulates until the *stop time*. If an event on an external net is produced at or before the stop time is reached, the simulator suspends itself and reports the event to the SM by calling **update_nets()**. It reports the time at which it has stopped by calling **report_internal_time()** and then passes control back to the SM. When a simulator stops, if there are events to be processed in its internal queue, the simulator requests that it be called again at a specific time in the future (as determined by the time of the earliest internal event) by calling **schedule_simulator()**. This procedure, called self-scheduling allows simulators to stop before exhausting all internal events. A simulator can schedule itself at any (discrete) time in the future provided it does not exceed the current time by the advertized maximum size of the timing wheel. Simulators that schedule themselves in the future but are invoked before that time by events at their inputs can remove their self-scheduling events by calling **unschedule_simulator()**. Note that when a simulator returns control to the SM, it is required to save its internal state so that simulation can be continued from where

it was stopped. For simulators that run as separate threads, state is automatically saved on a thread switch. Other simulators have to implement this feature explicitly.

Apart from coordinating simulators, the SM controls the trade-off between simulation accuracy and speed. As will be explained in Section 5, the simulator for M16 has the capability to choose the appropriate level of speed and accuracy when the processor is trying to read from or write to a certain address. When the address is in the range of memory, no signals are produced on the bus, but when the address is outside the range, phase-accurate bus signals are produced. This is adequate for the simulation of most peripherals. However, there are certain peripherals, like a DMA controller, that 'listen' to the bus in order to detect vacant bus cycles and perform cycle-stealing DMA. For such situations, even when the processor is accessing memory, signals on the bus have to be produced. Therefore, each simulator like the DMA is marked as a bus listener. Whenever a bus listener has to be run in lock step with a processor simulator, the SM sets a special flag indicating to the processor simulator that bus signals should be produced. This ensures correct simulation of systems with DMA controllers and other bus listeners.

Another important function of the SM is the mapping of internal values of simulators to a uniform representation and back to allow mixed-level (*e.g.* gate and transistor) and mixed-mode (*e.g.* analog and digital) simulation. It should be noted that standard templates are provided for the interface functions that make the job of integrating simulators easier.

## 5 PROCESSOR SIMULATOR

Processor simulators can be divided into three categories depending on accuracy and speed of simulation.

- Instruction Set Simulator (ISS) simulates the instruction set and values in memory and registers accurately. Signals at the pins of the processor can be produced only at the boundaries of instructions. It does not model superscalar ordering effects, delayed branch, pipeline stalls, wait states, and cache access. Therefore accurate clock cycle count for code execution cannot be determined. However, it is the fastest processor simulator and can be used for pure software simulation and debugging.
- Cycle-Accurate Simulator (CAS) can simulate the instruction set, the pipeline and the local cache of a processor and can provide the signals at the pins of the CPU at each clock transition and also provide accurate clock cycle counts. Superscalar ordering effects, pipeline stalls and wait states can be simulated accurately. However, it can be more than an order of magnitude slower than an instruction set simulator. In addition to software simulation, it can be used to model interaction with hardware components, though there might be inaccuracies in timing. A variation of a cycle-accurate simulator is a phase-accurate simulator (PAS) where the behavior of the processor in each clock phase is accurately simulated.
- Timing Accurate Simulator (TAS) can simulate the complete functionality of a processor with full timing accuracy. Because each pin can change at potentially unique times and the detailed timing behavior of the CPU together with the instruction set and the pipeline has to be simulated, this is the slowest of all simulators.

For M16, which is a scalar processor without a local cache, assuming that all memory accesses take the same amount of time, an ISS can be used to simulate the processor with little loss in accuracy. This is also based on the assumption that the interaction between processor and memory does not have to be debugged. However, an ISS cannot be used to simulate interaction with peripherals.

The choice between CAS/PAS or TAS depends on the level of accuracy required. Since a CAS/PAS produces signals at pins only at discrete times, the internal model for a CAS/PAS can be simpler and can run faster. The extra accuracy gained by using a TAS is that the signals can be produced in between clock cycles at the exact time they would be produced by the processor. Since the price for this increase in accuracy is steep, it is worthwhile investigating when full timing accuracy is required and when a CAS/PAS is adequate.

To determine whether a CAS/PAS is adequate, the first question to be answered is whether it is possible that certain signal transitions may not be generated or caught by a CAS/PAS. The M16 processor uses a synchronous bus protocol for the transfer of data to and from memory and peripherals. Address and data are latched by the processor and peripherals only at certain clock edges. The few fully asynchronous pins (like Data-Complete, Interrupt, Hold) are internally synchronized and therefore have to be active for at least one clock cycle. In other words, two events on the same net or that affect one another never happen without a clock edge in between. Our initial study of other processors indicates that this is true for the Intel i960 processor family and the Motorola MC68030 processors. Therefore, for these processors, a CAS/PAS that produces and samples bus signals only at each clock transition is equivalent to a TAS except for timing accuracy.

When the user is interested in determining if set-up and hold times are being violated, or when he/she is debugging an ASIC with tight timing constraints, the exact time when inputs arrive and when outputs are produced are important and there is no alternative to using a TAS. Therefore, a CAS/PAS can be sufficient only when the system has been designed so that set-up and hold times are not violated and all custom circuitry and peripherals meet their timing constraints. The M16 processor ensures that set-up and hold times are not violated in its peripherals by producing signals on the bus well in advance of the clock edge where they would be latched. Users manual also require that peripherals produce data a certain time before the clock edge where it will be latched by the processor. If a system is carefully designed and conservative design rules are

followed, there may be few set-up and hold time violations. These violations can be detected using bus functional models and timing accurate simulation. Therefore, with an appropriate design methodology, the use of CAS/PAS may be sufficient for hardware-software co-simulation. We are conducting further study to validate this assertion.

The simulator for M16 is an integrated ISS and a PAS. Each processor clock cycle is divided into six periods and the PAS produces bus signals at the boundary of each period, while the ISS does not produce any bus signals. During execution of a program, depending on the instruction and operand address, the simulator automatically switches from ISS to PAS and *vice versa*. The ISS is used to simulate program execution when nothing but memory is accessed. Whenever the processor tries to access some region that is outside the address range allocated to memory or when the SM sets a flag that indicates that signals on the bus have to produced, the PAS is used. Note that switching between ISS and PAS requires that the ISS maintain some information about the state of the pipeline during execution. The PAS consists of a pipeline simulator and a bus interface module. The pipeline simulator simulates the pipeline of the CPU accurately while the bus interface generates the appropriate signals. Using the less accurate but fast ISS when only memory is accessed and switching to the more accurate but slower PAS only when required cuts down on the number of events too and speeds up simulation by more than an order of magnitude in most cases.

Most ISS can simulate anywhere between 2000 and 20,000 instructions per second [11]. In order to speed up the ISS and PAS for M16, we exploited the locality of reference in the program memory. Many embedded programs execute a group of instructions over and over (as in a loop). Each instruction, which includes opcode and operand(s), is decoded and the result is stored in a cache. Before decoding, a new instruction, it is looked up in the cache. For a cache hit, the decoded form is used directly, thereby avoiding the simulation of the complicated and time consuming decoding phase. This can increase the execution speed of the ISS and PAS by about a factor of 2. Currently, the M16 ISS can simulate about 50,000 instructions per second for typical programs on a Sun Sparcstation 10. The PAS can simulate about 4,000 instructions per second. The PAS does not simulate the instruction fetch cycle, assuming that no events for peripherals can be produced during this time and that instruction memory can only introduce a fixed number of wait states.

## 6 SIMULATION OF HARDWARE

Custom hardware represented using Verilog HDL is simulated using a commercial simulator, Verilog-XL [14]. Since a commercial simulator is designed to be a stand alone tool and does not implement the interface functions required by the SM, its integration poses certain problems. For Verilog-XL, the interface functions were implemented using the Programming Language Interface (PLI) for the simulator [14]. The PLI allows user defined functions (written in C) to be called from Verilog-XL during simulation. It also allows these functions to call certain functions for simulation control in Verilog-XL. The details of the implementation are skipped for the sake of brevity.

In our implementation, the user is required to call the function **$codebug** in an **initial** block of the top level module in the custom circuit description. There are some requirements on the way input, output and bi-directional lines are represented. There is no other restriction, and hardware can be represented at any level of abstraction allowed in Verilog. Verilog-XL is currently the only timing accurate simulator in our framework. Since other simulators are only phase-accurate, the interface functions for Verilog-XL may introduce errors during rounding of event times if proper care is not exerted in describing the hardware.

## 7 SIMULATION OF STANDARD PERIPHERALS

Embedded processors are often used in conjunction with a set of standard peripherals. Instead of describing them in some HDL and using a hardware simulator, we use dedicated simulators to simulate each type of peripheral. Each simulator consists of a behavioral model written in C and a bus interface. The behavioral model simulates the phase-accurate behavior of the peripheral and the bus interface generates the appropriate signals at every clock transition.

There are several advantages of using dedicated simulators. First, multiple instances of the same standard peripheral can be simulated more efficiently. Consider, for example, a system that has several parallel ports. When the processor writes to one of them, events are generated for each parallel port which then decode the address to determine the recipient. In most cases, only one parallel port will respond to the write while others will ignore it. Therefore, for all but one parallel port, decoding of the address is a useless operation that cannot be prevented if a hardware simulator is used. Using a dedicated simulator, all parallel ports can be simulated together so that when a processor writes an address on the bus, only one set of events is created for all the parallel ports and given to the simulator. The simulator decodes the address only once to determine which one of the parallel ports the CPU is talking to. Therefore, not only is the number of events reduced, but useless decode operations are avoided.

The second advantage of dedicated simulators is better handling of periodic signals. Such signals impair simulation efficiency by increasing simulation overhead. In [15], it was shown that suppression of periodic signals during concurrent fault simulation can produce significant savings in simulation time. We adopt a similar approach here. Each clock generator advertises its clock signal as a triple, describing the period, the rise time and the fall time. The use of this information is illustrated by the timer simulator. A timer is a counter that is initialized with a value corresponding to the number of clock pulses to be counted. On receipt of a start signal, the timer

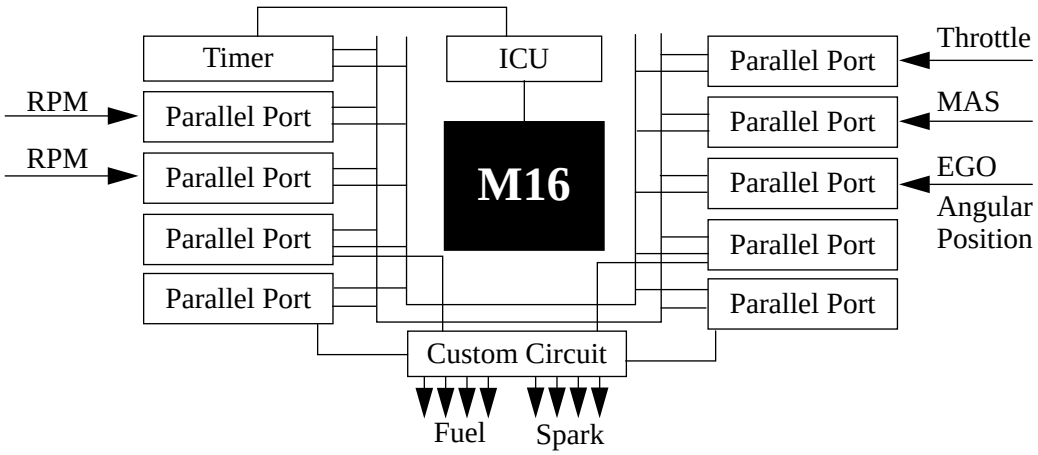


**FIGURE 3. Architecture of an Engine Control Unit**

starts to decrement the value of the counter at each positive/ negative edge of the clock. If simulated using a hardware simulator, clock events have to be fed to the counter periodically. However, a dedicated timer simulator can use the advertised clock signal and the value of the counter to determine at what time the counter is going to expire. It can then schedule itself at the right time in the future to produce the appropriate event. This decreases the number of events generated, the number of simulators invoked to handle each event, and the time spent in simulating the timer. For the motor control application to be described in Section 9.2, this can reduce the number of events per revolution of the motor from 129,640 to 840. The other advantage of this method is that other simulators, like the processor simulator can run uninterrupted during the time the timer is counting, thereby reducing synchronization overhead further. Note that it is not possible to avoid the generation of the clock signal at all times, *e.g.* when the clock is an input to custom circuitry. In such situations, we use a local clock generator which uses the advertised clock signal to generate a clock only for the module that needs it. Once again, this reduces simulation overhead because periodic signals are produced locally where they are needed.

The third advantage of dedicated simulators can be illustrated using an Interrupt Control Unit (ICU). The algorithm for interrupt priority resolution requires complicated and deeply pipelined hardware. Simulation of this hardware takes more time than executing the algorithm directly in the simulator. The advertized clock signal is used to determine the state of the pipeline and how long it takes to generate an interrupt signal. For an example application, replacing the dedicated ICU simulator with a RTL Verilog model slowed down simulation by two orders of magnitude. Though a part of this slowdown can be attributed to Verilog-XL and its interface to the backplane, this result is still significant. Also, this technique is fairly representative of the techniques that can be used to speed up simulation.

It is obvious from the discussion above that dedicated phase-accurate simulators for standard peripherals may be able to speed up simulation in ways that HDL simulators cannot. However, there are certain drawbacks. For every new peripheral a new simulator has to be written and integrated into the backplane. Also, it is not always possible to implement the kind of techniques mentioned above for all standard peripherals. We are working on a tool that will solve the first problem by providing the standard boiler-plate needed for a simulator. For the second problem, we rely on the ingenuity of the simulator developer.

## 8 OTHER SIMULATORS

We have developed an interface between the co-simulator described in this paper and the Tsutsuji hardware simulation system [3]. The Tsutsuji system is capable of efficiently modeling and simulating signal processing functions. Systems that have both control and signal processing functions, like motion detectors, can be easily simulated. We are also in the process of developing an interface to the Ptolemy simulator to allow us to use the heterogeneous simulation environment of Ptolemy.

In addition, a simulator for a three-phase electric motor and for a rudimentary automobile engine has been developed for the design and debugging of motor and engine control systems. It is our hope that as this system finds more and more use, a large library of simulators for diverse application areas will develop and will increase the usefulness of this tool.

## 9 EXAMPLE APPLICATIONS

Several applications were used to test the capabilities of the co-simulator. They include an engine control unit, a three-phase motor control unit, a real-time operating system for the M16 microprocessor, a motion detector and a computer modem. The first three applications and our experience in using the co-simulator are described briefly in this section.

### 9.1 ENGINE CONTROL UNIT

The operation of an engine is controlled by varying the air-flow, the duration for which fuel is injected into each cylinder and the spark time. The engine control unit receives inputs from the mass air flow sensor (MAS), the RPM sensor, the

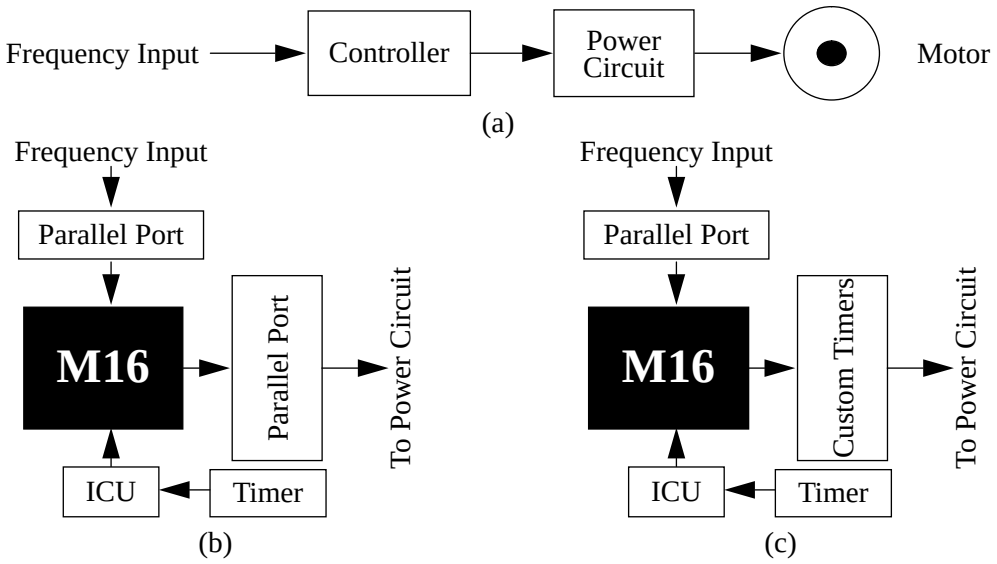


**FIGURE 4. 3-phase motor controller (a) block diagram (b) first and (c) second implementation**

exhaust gas oxygen sensor (EGO), the throttle position sensor and the crankshaft angular position sensor. The controller controls the idle valve (not shown in the figure), the throttle-body fuel injectors, and the spark plugs.

An architecture of a simplified engine control unit is shown in Figure 3. It consists of an M16 processor, a timer, an ICU, nine parallel ports and some custom circuitry. The custom circuitry can be implemented in approximately 2000 gates. The C source code for the engine controller is about 1000 lines long.

The software for the controller and the RTL description of the custom circuit were developed and debugged solely using the co-simulator. 1500 CPU cycles (approximately 300 machine instructions) could be simulated per second on a Sun Sparcstation 10. At this speed, it takes 40 minutes to simulate the behavior of the engine and the controller as it goes from 0 to 7000 r.p.m. This represents a slowdown of about a factor of 400 over real time operation, an adequate speed for debugging. Note that the emissions from the engine were not modeled and a simplified dynamic control algorithms was used for the controller.

#### 9.2 THREE-PHASE MOTOR CONTROL

A three-phase motor controller, shown in Figure 4(a), takes as an input the desired frequency of rotation and produces pulse width modulated signals which are demodulated by the power circuit, producing three sinusoidal signals at the required frequency but phase shifted 120 degrees with respect to one another.

An implementation of this controller using a microprocessor and standard peripherals is shown in Figure 4(b). All computation required to produce the pulse width modulated signals is performed in the microprocessor. At high frequencies, the demodulated waveforms show a mean square error of 8% from an ideal sine wave because the processor cannot keep up with the required rate of calculation. An alternative architecture is shown in Figure 4(c) where some custom circuitry is used in conjunction with the microprocessor. The calculation for pulse width modulation are still performed in the processor but the actual generation of the signals is moved to custom hardware. The demodulated waveforms now show a mean square error of less than 1% from an ideal sine wave at all frequencies. The amount of ROM required to store the program and the tables is also smaller. This is a good example of how the co-simulator may be used to determine hardware-software trade-offs at the implementation level.

The controller of Figure 4(b), can be implemented in 600 lines of C code and simulation runs about a factor of 3200 slower than the actual system. The controller of Figure 4(c) can be implemented with only 200 lines of C code while the custom circuit is represented using 100 lines of behavioral-level Verilog. Simulation runs about 7400 times slower than the actual system for the second implementation, showing the effect of Verilog-XL on simulation time. It has been our experience that use of custom hardware significantly slows down simulation. Note that the power circuit and the motor is simulated using a special simulator. A screen image of this simulation is shown in Figure 2.

#### 9.3 RTOS AND DEVICE DRIVER DEBUGGING

Traditionally, operating systems and device drivers have been debugged using working hardware. A part of the real-time operating system kernel and device drivers for a microcontroller based on the M16 processor has been debugged using the co-simulator. The hardware used for this purpose consists of an M16 CPU, an ICU, three timers and two parallel ports. Interrupts are fed to the system from two external buttons and

are also generated by the timers.

The software running on this system consists of six tasks and the real-time OS. Task1 is invoked when there is an interrupt from any timer and counts the number of timer interrupts. Task 2 is invoked when there is an interrupt from the first button and counts the number of button interrupts. Task 3 is invoked when there is an interrupt from the second button and resets the count kept by task 2. The rest of the tasks, numbered 4 to 6 are scheduled in round robin fashion. The task number being executed is displayed through one parallel port and the number of button interrupts is displayed through the other one.

Simulation of the RTOS can be performed at a speed of 23,000 instructions per second. This represents a slowdown of 1500 compared to the RTOS running on an M16. This speed is adequate for the debugging of the RTOS. The debugging environment is natural for a software developer and the greater observability of the internal state of the processor during simulation also helps debugging.

Our experience so far suggests that a PAS is adequate for debugging the interface between hardware and software. However, we recommend the use of more accurate timing simulation using bus functional models in conjunction with co-simulation.

#### 10 CONCLUSIONS AND FUTURE WORK

We have presented a hardware-software co-simulator for embedded system design and debugging. This tool provides a natural environment for joint debugging of software and hardware and is also useful for evaluating system performance, selection of algorithms and implementations and also for exploring hardware-software trade-offs. We have addressed the problem of simulation speed and have outlined various methods to speed up simulation. The improved speed of the co-simulator comes from various sources. First, our co-simulator is targeted towards phase-accurate simulation. Switching between ISS and PAS during simulation, caching of decoded instructions and not simulating instruction fetch cycles all contribute to the increased speed of simulation of processors. Use of dedicated simulators, suppression of periodic signals and associated events, and specific short cuts reduce the time required for simulation of peripherals. Making time discrete and using a statically allocated timing wheel helps keep coordination overhead low. We have demonstrated the use of the tool in three design examples and have shown that the simulation speed is adequate.

The usefulness of this tool will depend on several factors. First amongst these is the availability of simulators for standard components. Second, is the adequacy of cycle-accurate simulation in system verification. We are continuing our research in this area. We feel a co-design methodology with conservative design rules, use of bus functional models to ensure compliance and an overall design style to aid simulation may be required.

Apart from the items mentioned before, in the future we are looking at incorporating other processor and hardware simulators into our framework. We believe that the next major increase in simulation speed will come from compiled simulation and we are investigating promising techniques in this area, especially in the simulation of processors. We are also investigating the use of a network of workstations to speed up simulation. There is ongoing work on a better user interface that includes system schematic capture, dynamic attachment of virtual instruments, *etc*. so that a virtual laboratory can be created on the desktop. Improving the efficiency of the simulation backplane is another area of ongoing work. The actor library is being enhanced to include commonly used components in embedded system design. We are also developing links to compilers and hardware design tools so that the co-simulator can be easily integrated into a design methodology.

### References


[1] D. Becker, R. K. Singh and S. G. Tell, "An Engineering Environment for Hardware/Software Co-simulation", *Proceedings of the 29th Design Automation Conference*, Anaheim, CA, 1992.

[2] J. Buck, S. Ha, E. A. Lee and D. G. Messerschmitt, "Ptolemy: a Framework for Simulating and Prototyping Heterogeneous Systems", *International Journal of Computer Simulation*, special issue on "Simulation Software Development," January, 1994.

[3] W. B. Culbertson, T. Osame, Y. Ohtsuru, J. B. Shackleford and M. Tanaka, "The HP Tsutsuji Logic Synthesis System", Hewlett-Packard Journal, August 1993.

[4] R. K. Gupta, C. N. Coelho Jr. and G. De Michel, "Synthesis and Simulation of Digital Systems Containing Interacting Hardware and Software Components", *Proceedings of the 29th Design Automation Conference*, Anaheim, CA, 1992.

[5] IEEE Design and Test Magazine Roundtable, "Hardware/Software Codesign", IEEE Design and Test Magazine, March 1993.

[6] A. Kalavade and E. A. Lee, "A Hardware/Software Codesign Methodology for DSP Applications", *IEEE Design and Test*, September, 1993.

[7] Y. Kra, "A Cross-Debugging Method for Hardware/Software Co-design Environments", *Proceedings of the 30th Design Automation Conference*, Dallas, TX, 1993.

[8] The M31000S2FP Users Manual, Mitsubishi Electric Corporation, Japan.

[9] J. K. Ousterhout, *An Introduction to Tcl and Tk*, Addison-Wesley Publishing Company, 1994.

[10] H. El Tahawy, D. Rodriguez, S. Garcia-Sabiro and J-J. Mayol, "VHDeLDO: A New Mixed Mode Simulation", *Proceedings of the European Design Automation Conference*, CCH Hamburg, 1993.

[11] J. A. Rawson, "Hardware/Software Co-simulation", *Proceedings of the 31st Design Automation Conference*, San Diego, CA, 1994.

[12] D. E. Thomas, J. K. Adams and H. Schmit, "A Model and Methodology for Hardware-Software Codesign", IEEE Design and Test of Computers, September, 1993.

[13] *Thor Tutorial*, VLSI/CAD Group, Stanford University, 1986.

[14] *Verilog-XL Reference and Programming Language Interface Manuals*, Cadence Design Systems, 1992.

[15] T. Weber and F. Somenzi, "Periodic Signal Suppression in a Concurrent Fault Simulator", *Proceedings of the European Conference on Design Automation*, Amsterdam, 1991.