1. INTRODUCTION

Some embedded systems incorporate disparate commercial off-the-shelf (COTS) microprocessors; for example, a DSP to perform high-throughput audio processing, plus an inexpensive 8-bit microcontroller to manage the user interface and supervise the DSP. That such systems are difficult to test may seem surprising in view of the array of simulation and emulation tools available for COTS microprocessors. However, these tools are mostly designed for stand-alone use, and in consequence, developers cannot use simulation as a system testing technique at all. Instead, reliance has traditionally been placed on building a target platform with hardwired processor interconnections, and substituting emulator pods for the processors. Running two emulation packages concurrently does allow for some degree of controllability and observability, but if the packages are not integrated, system level debugging is extremely awkward. One could not, for example, arrange a breakpoint on one processor to halt both processors. Integrated debugging environments are more powerful, but they require target hardware; pure software simulation is still not an option. This state of affairs hampers the rapid development of products based on heterogeneous microprocessors.

This paper describes a new simulation system created to address this need in industry. Developed initially for two common processors—Intel 8051 microcontroller and Analog Devices ADSP-2181—the system is designed in terms of an object-oriented framework that facilitates its adaptation for other processor models.

2. BACKGROUND

We can further elaborate the motivation for this research by turning to the annual Buyer’s Guide of Embedded Systems Programming [1]. Though instruction-level simulators appear under the “Software Debuggers” section, it is important to distinguish simulation from debugging. Granted, one would ideally like both cases to have the same user interface, regardless whether the software is being simulated or executed on target hardware under debugger control. But pure software simulation brings two troublesome issues: (1) how to simulate interprocessor communication, and (2) how to synchronize multiple simulations to a common time base. These issues are sidestepped in ordinary debugging: the processor interconnections are hardwired, and the processors run according to their clocks.

Numerous vendors of single-processor simulators are listed in the Buyer’s Guide. “Multiple processor support” here generally means multiple instances of a given processor, i.e., the homogeneous case. The lack of support for heterogeneous processors is understandable from processor vendors, who no doubt prefer to focus on their own products. But what about third parties? Possibly the market for any particular pair of processors is too limited to warrant the development cost, and, conversely, the number of processor models needed for commercial viability of a heterogeneous simulation suite may be too large.

Only one vendor, Multiprocessor Toolsmiths Inc., is tagged as having “heterogeneous multiprocessor support.” Indeed, their literature maintains that “DSP systems invariably involve heterogeneous multiprocessing” [2]. However, their support involves layering the application software on top of a common operating system running in each processor, their own POSIX-compliant Unison OS. This approach homogenizes the interfaces as far as a common debugging environment is concerned, but no provision exists for the same heterogeneous support in pure simulation. Furthermore, one is required to commit to a particular OS, which also rules out running legacy assembly code.

Recently, Microtec released Version 4.0 of their XRAY Debug Suite [3]. XRAY also manages multiple het-
erogeneous targets for concurrent debugging. The targets need not be running the Microtec VRTX RTOS, but the Xtrace Software Monitor must be present (typically in PROM) to control processor execution and interface with the host-based XRAY. For pure simulation purposes, XRAY supports a “Virtual Target,” which is essentially a host-compiled simulation. This is only useful for high-level language (C, C++) code. XRAY does support a variety of 32-bit Instruction-Set Simulators (ISS), e.g., Motorola 68000 series, PowerPC, ColdFire, and ARM. While there is nothing to prevent a user from running several ISSs concurrently under control of the common XRAY debugger interface, there seems to be no provision for synchronizing or interconnecting them.

Another joint product release from Mentor Graphics and Microtec is relevant here. The Seamless Co-Verification Environment (CVE) [4] allows a software program running in an ISS to export a pin-level hardware model for bus and I/O transactions. These pin transitions are in turn integrated with a simulation (e.g., in VHDL) of the hardware portion of the embedded system. This achieves “cosimulation” in the hardware/software code-sign sense [5], and is quite useful for a system-on-silicon (SOS), but does not directly solve the problem of multiple processor simulations. One could conceivably use Seamless CVE to link two ISSs via a hardware interconnection model, but this would make the CVE Hardware Simulation Kernel a communication bottleneck between the two software simulations.

In academic research, instruction-level simulation is normally focussed on simulating a computer architecture, which may be a homogeneous multiprocessor, for experimental purposes. Embra [6] is representative. Heterogeneous system simulation, in contrast, is often connected with distributed system research. On the scale of embedded systems, Ptolemy [7] can be utilized to couple processor models, but the models would need to be written in C++ within the Ptolemy simulation paradigm, so there would be no speed advantage.

### 3. DESIGN CRITERIA

The following criteria were established through extensive consultation with an industrial collaborator who builds embedded systems specializing in real-time processing of singing (for example, karaoke applications). Thus, we believe our simulation system will be of particular interest to developers of data-dominated embedded systems.

**Functional criteria**

*Multiple “bare” processors.* The system runs two or more instruction-level processor simulations concurrently. Input is in the form of bootable binaries created by a commercial assembler. No particular operating system or executive can be assumed to be present in the target system, though the simulator should be able to run an OS if one is loaded.

*Simulated devices.* Simulations are provided for each processor’s associated I/O devices, typically via disk files. For example, a serial port could take input from a MIDI file, an A-to-D channel from a WAV file, interrupts from an event file, etc. Interprocessor communication facilities (e.g., DMA) are supported.

*Synchronization.* All processor and device simulations are synchronized to a common time base.

**Performance.** Desirable performance for DSP simulation is 100X real time or better.

**Common debugger interface.** The debugger interfaces for each processor simulator have a common “look and feel.” In addition to the usual range of debugger commands (breakpoints, conditional breakpoints, tracing, memory display/set, register display/set, watchpoints, etc.), breakpoints armed by conditions in another processor are supported.

**Architectural criteria**

*Extensibility.* It should be relatively easy to add simulation models for processors and their devices (including interprocessor communication models).

*Portability.* The user interface should run without modification on a variety of common desktop platforms—Windows PC, Macintosh, Sun. The simulation engine should require no more than minimal modifications related to host computer word length.

*Power.* When a multiple-CPU computer is available for running the simulation (e.g., a remote simulation host), the architecture should take advantage of multiprocessing in order to obtain maximum speed.

### 4. SYSTEM DESCRIPTION

A simplified block diagram of our simulation system is shown in Figure 1 below. In order to meet the power criterion, we have created a distributed system that consists of a user interface (UI) portion, intended to run on the engineer’s desktop computer (“UI host”), and a simulation engine running on a remote simulation host. If the UI host is powerful enough, it can also host the simulation engine. On the UI side, where speed is not an issue, we use Java in order to make it completely portable. The simulation engine also has a Java executive, and interprocess communication utilizes Java remote method invocation (RMI).

The heart of the simulation engine, where the concern for execution speed is paramount, is written in ANSI C++ and invoked via Java’s “native methods.” We have access to a SparcServer 690MP with 4 processors and shared memory (a symmetric multiprocessor). On such a system, separate threads are created for each processor model so that their execution can be overlapped. Each thread keeps its own nanosecond simulation clock, and synchronization is achieved through “self pacing”—the other clocks are examined prior to each simulated instruction cycle and threads which have gotten ahead will...
hold back. Devices requiring service at a future instant, as well as external interrupts initiated from a prepared event file, place their requests in a per-thread event queue, and these are serviced when their scheduled time arrives (similar to Emhra’s method).

Other simulation speedup techniques are employed based on the theme of expending simulation host memory to save run time. For example, each instruction is cached in an Instruction Record (IR) the first time it is decoded. Of course, executing a store into a simulated memory location invalidates any corresponding IR. As well as the operation code, an IR contains C++ references to all the register objects involved in the instruction, allowing predecoded instructions to be immediately dispatched on subsequent executions, such as in loops. This approach provides the same degree of acceleration as a complete predecoding pass through memory, but avoids the problem of distinguishing instructions from data in simulated memory, and works even if self-modifying code must be simulated.

Object-oriented frameworks

Both the UI and simulation portions are designed for maximum code reuse when new processor models are to be incorporated. This is accomplished through the use of two OO frameworks: one in Java for the UI, and one in C++ for processor models. This approach was considered preferable to the traditional provision of generality and flexibility by means of table-driven software.

The frameworks comprise a set of classes for building, respectively, a Java debugger UI for a particular processor, and its corresponding C++ simulation model. Both frameworks feature executive portions in Java and/or C++ that are non-model specific and thus completely reusable.

The Java UI classes consist of graphical UI widgets that are “bolted” together to construct a window through which to manage a single simulated processor. The various elements—register, memory, and source code displays, control buttons, and so forth—are linked to their underlying data structures in the C++ framework in a sort of client-server relationship. This enables the various window elements to display and modify simulated registers, memory, and so on, via automatic RMIs. Each element knows how to obtain the data it needs from the simulation model, and can display it in whatever format (hex, decimal, etc.) the user selects. In a similar way, the elements notify the simulation model of modifications (for example, when the user changes a register’s contents). Whenever the simulation halts, such as at a breakpoint, the elements may update their displays, but a caching mechanism is used to avoid redundant network update traffic.

In the C++ framework, templates and inheritance are used to construct a standard interface to elements such as registers and memory models which can be quite idiosyncratic in some processor architectures (notably DSPs). Since templates are resolved at compile time, an optimizing compiler (we use GNU g++) can produce very efficient assembly language and inline code even for methods that are parameterized, say by register bit length.

As an example of diverse bit lengths, the ADSP-2181 has 16-bit wide Data Memory and 24-bit Program Memory (which, nonetheless, can be used for data). It has 5-bit, 8-bit, 14-bit, and 16-bit registers, some signed and others considered unsigned. Some are ganged, such as the MR multiply result register, which consists of the 8-bit MR2 and 16-bit MR1 and MR0 registers together. Yet MR can also be treated as a 40-bit quantity. The complications go on: Three groups of registers have primary and secondary banks, access being determined by a flag in the MSTAT register. Another status flag can cause certain index registers to be read out bit reversed. Then there are

FIGURE 1. System block diagram (2-processor case)
the registers which are actually the tops of stacks: the PC stack, the loop stack, the status stack, and so on. These are all hardware registers, not memory locations.

To deal with the above complexity we created the C++ class hierarchy shown in Figure 2. The `Reg` abstract base class defines a simple interface for a register object in terms of two methods:

1. `(Sdata)` This conversion operator obtains the register contents in a universal signed data type, `Sdata`, which on our Sparc is just `int`.
2. `= Sdata` The assignment operator from type `Sdata`.

A simple kind of register is created via the template class `REG` which takes as its parameter the type of storage that the register data will occupy. That type is another template, `BITFIELD`, which defines a signed or unsigned integer of the specified bit width. `REG` implements the `Reg` interface with very elementary code:

```cpp
operator Sdata() const {return reg.bits;}
void operator= (Sdata data) {reg.bits = data;}
```

where `bits` is the subfield of `BITFIELD`. Now if we instantiate two registers as follows:

```cpp
REG<BITFIELD<unsigned,5> > A;
REG<BITFIELD<signed,8> > B;
```

then when we use A and B in expressions or assignments (e.g., A=B:), all necessary masking, shifting, and sign extension will be generated by the compiler using efficient code.

`REGFLAGS` extends `REG` by adding methods to test, set, and clear individual bits. `REGBANK` contains two `REG` instances. Its version of the `Reg` methods automatically reference whichever bank is selected in the system status register. `STACK` implements a stack of items to a specified depth, and automatically handles the specified overflow and empty flags. This is combined with `REG` to get a `REGSTACK`. With these tools, the definition of the DSP’s registers can easily proceed:

```cpp
REGBANK<BITFIELD<signed,16> > SI, SR1, SR0;
REGSTACK<BITFIELD<unsigned,14>,4,CSO,CSE> CS;
```

and so forth. The objective is to render `Reg` references completely interchangeable in the code that simulates instructions. The compiler handles the shifting and alignment with virtual methods, and the need for special case code is greatly reduced.

### 3. CONCLUSION & FURTHER WORK

In the context of CAE for embedded systems, we fill an important gap by providing system simulation facilities for heterogeneous microprocessor-based designs. Our solution applies OO framework software technology, along with traditional event-based simulation, improved to exploit multiprocessor computing power.

At this stage in our development, we still need to build useful device simulations in cooperation with our industrial partner. It also remains to be seen whether we can meet our performance target (100X real time for DSP) without further acceleration techniques, and what degree of speedup can be obtained with multithreaded simulation. Furthermore, the true value of the OO framework architecture will only be established as we attempt to expand the system to encompass additional models.

### References


---

**FIGURE 2.** Inheritance hierarchy of C++ register template classes

- `bf = BITFIELD<sign,width>`
- `Reg abstract`
- `STACK<item,depth,oflow,empty>`
- `REGBANK<bf>`
- `REG<bf>`
- `REGSTACK<bf,depth,oflow,empty>`
- `REGFLAGS<bf>`