INTRODUCTION
Emulators, like Mentor Graphics Veloce®, are able to run designs in RTL orders of magnitude faster than logic simulators. As a result, emulation is used to execute verification runs which would be otherwise impractical in logic simulation. Often these verification runs will include some software executing on the design – as software is taking an increasing role in the functionality of a System-on-Chip (SoC). Software simply executes too slowly to practically run anything but the smallest testcase in the context of logic simulation. For example, booting embedded Linux on a typical ARM® design might take 20 or 30 minutes in emulation. The same activity in a logic simulator would be 7 or 8 months.

With significant software being executed in the context of the verification run, there needs to be some way to debug it. Historically, this is accomplished using a JTAG (or similar interface) probe with a general purpose debugger. JTAG, which stands for “Joint Test Action Group”, is a standards organization which developed a standard interface to extract data from a board to an external system for testing purposes. This capability is used to extract the register state from a processor, which can be used by a debugger to present the state of the processor. The JTAG standard was extended to enable the debugger not to just view the state of the processor, but to be able to set the state processor and to control its execution. As a result, JTAG probes have become the “standard” method of debugging “bare metal” embedded software on a board or FPGA prototype. Through just a few signals, the debugger and the JTAG probe can download a software image to a board, single step through the program, and read arbitrary locations in memory.

JTAG AND EMULATION
Given that a JTAG probe is the standard method for debugging an embedded system prototype, it was natural that it would be used to debug systems in emulation as well. A system realized in emulation should have all the same characteristics as a physical prototype. Thus all the existing debugger support for boards could immediately be applied to debugging software in the context of emulation.

There are two ways to connect the JTAG probe to the emulator. One is to physically connect the probe to the JTAG signals in the design. This involves identifying the JTAG signals in the design and bringing them out to an “I/O card”, which allows the signal to be physically connected to a device outside of the emulator. For Veloce, Mentor Graphics provides an I/O card which has the JTAG 20 pin connector used by most JTAG probes. Often, when physically connecting external electronics to a design in the emulator some accommodation needs to be made for the fact that the emulator is running significantly slower than the final target system. Most JTAG probes, however, will accommodate a slower clock frequency without any issue.

The second method to connect the JTAG probe to the design in the emulator is to programmatically access the JTAG signals. Most emulation systems provide a method for sampling and driving arbitrary signals in the design. Using this facility, a program can be written which interacts with the debugger, which replaces the functionality of the physical probe and connection. This program will result in the same activity on the JTAG signals as the physical probe – it just involves less hardware. There are a couple of examples of this type of connection. ARM® has a product called “VSTREAM” which implements this “virtual” connection to an ARM® processor in an emulation system to their software debugger. Mentor Graphics has a “virtual probe” product which performs essentially the same function, but for a broader set of processor cores and debuggers.

PROBLEMS WITH PROBE DEBUG
While it was obvious to connect probe based debugger to a system being emulated, there are some short comings with this approach. The characteristics of an emulated target are not the same as a physical target.

A physical target will always be a complete system — it can’t be a work in progress. But in the context of emulation it is quite common, especially early in the design cycle, to deploy partial systems. However, a probe requires that the entire debug infrastructure be in place and debugged before it can become functional. For simple cores, this was simply a matter of connecting the
JTAG signals on the core to the JTAG signals in the larger
design. However, for modern complex cores like the ARM®
Cortex™ family, there is a large set of complex IP which
operates to enable the debug functionality needed for the
probe. ARM® refers to this IP collectively as Coresight™.
Coresight™ is a set of hardware that needs to be connected
to the rest of the design, and configured and programmed
correctly, before it will function. Coresight™ is so config-
urable, that it frequently is not implemented correctly. The
Coresight™ reference manual from ARM® is almost 400
pages long, and even at that is not a complete guide to its
implementation. Anecdotally, many commercial devices fail
to deploy Coresight™ correctly – leaving them with limited
or sometimes unusable debug capabilities. Because of this,
probe based debug is limited to late in the design cycle.

Another limitation of probe based debug is the method
by which it gets information about the core. In a physical
device it is not possible access the internal states. It is,
literally, a black box. So the JTAG committee devised a
way to use just a few signals to get access to the internal
states of the device. The trade off is that it takes a lot of
time to move all that data through the small interface – a
single wire for data transfer. With high (and increasing)
clock rates, and few alternatives it was deemed a good
trade-off for physical devices. However, in the context of
 emulation, and to some extent FPGA prototypes, there
are alternatives for getting at the internal states of the
device, while the clock rates – especially for the debug
circuitry - are not as high. Debug probe clocks often run
at even lower frequencies than the main processor clock.
And due to the length of the scan chains it can take a lot
of clocks to perform a single operation. Measurements
show that, for example, executing a single instruction on a
processor through a scan chain can take anywhere from
several hundred thousand processor clocks to over a million
processor clocks. Complex or aggregated operation may
take many times that amount. Not a problem when running
at hundreds of megahertz – but more limiting when running
at single digit megahertz rates, typical of emulation.

While the processor’s operation is suspended – it is put into
a “debug” state – the remainder of the design continues to
be clocked and its state advances. The millions of clocks
that go by to process the debug activity continues to be
driven into the rest of the system. For certain classes of
bugs, this is an irrelevant detail and can be ignored by the
developer. For synchronization and timing related bugs this
can make debugging extremely difficult.

Another issue is the lack of determinism. The activity of the
debugger is determined by the developer driving it. Since
the design continues to be clocked while the processor is
suspended, in debug mode, exactly when the developer
sends a step command, or how large a memory region
is retrieved can impact the state of the target (whether
emulated or physical). For intermittent or timing related
bugs this can make debugging an exercise in frustration.

Yet another problem with the probe style of debugging is
the fact that the debugger stops the system to enable the
programmer to examine the system. Emulators are valued
for their ability to run the design fast. Emulation vendors
all lead with performance numbers showing how many
megahertz they can achieve when running a given design.
But when a probe debugger attaches to the processor, it
halts all processor related activity – effectively dropping
the throughput to zero hertz.

The process of software debug consists of examination of
the source code and examination of the state of the system
at various points in time. There is a lot of thinking involved,
but not a lot of execution. Advancing the design happens
from time to time – to take the design to a new state where
it will be examined and considered. With an emulator
and probe-based debugger, the software debug work
happens not while the emulator is running, but while the
emulator is stopped. In most organizations, the emulator is
a shared resource, used by multiple developers and even
multiple projects. The goal will generally be to optimize
the throughput of the resource. Interactive software debug
conflicts with this goal. But until now, there have been no
practical alternatives.

**ALTERNATIVE SOLUTION**

Hardware debug on an emulation system generally consists
of running the design with some level of instrumentation
or signal tracing, and then looking at the signals after the
fact. To optimize the throughput – most users employ a
queuing mechanism. Jobs are loaded into a queue. LSF or
SunGrid are examples of this. As the emulator finishes one
job, it immediately starts the next to remain continuously
occupied. Hardware engineers will view the log files and
waveform traces after the fact.
Software debug with a probe does not support this model very well. Many times, the jobs are still entered into a queue. But in this case the software engineer waits on the machine (and it should be noted that the software engineer is often a more costly resource than the emulator). Common alternatives are rigid scheduling (each developer has a pre-defined time slot), but this will not map well to real-world needs. Or developers simply waiting for the emulator to be free – using an ad hoc scheduling scheme, which wastes both developer and emulator time.

The ideal solution is one where the job can be run on the emulator, in batch like hardware debug jobs, but software debug takes place after the job has completed – off line. This enables the emulator use to be optimized, as it can process a queue of jobs for both hardware and software debug. It also allows the software developer's time to be optimized, as they can debug the code at their leisure and not fight for time on the emulator.

TRACE BASED DEBUG
An alternative to using a probe and the hardware debug facilities of the processor being debugged, a trace based approach can be used which mitigates many of the problems described. This takes advantage of the emulator's ability to easily access the state of any signal or register in the design at any time. During an emulation run certain signals in and around the RTL for the processor are logged and conveyed to the host for processing. The data from these signals is used to reconstruct the state of the processor, and its memory space for any point in time during the emulation.

This data is interpreted by a “virtual target”. The virtual target mimics the state of the processor and it's memory subsystem in the emulation, and presents this to an embedded debugger. The same debugger that a developer uses to connect to the emulator through a JTAG cable can be connected to this “virtual target”. The debugger retains all the capability of stepping, running, setting break points, viewing memory, and viewing source code. From the software developer’s perspective just about everything works the same as connecting to the actual target. Even though a log of the processor's activity is recorded, the developer’s view is not a processor trace; they get a full featured software debugger view of what took place during the emulation run. The important difference being that while they debug the software, the emulator is free and available for other emulation jobs. The slow and thinking intensive process of software debug is no longer a limiting factor on emulation throughput.

At Mentor Graphics we have implemented this trace based emulation debug for the Veloce emulation system. The application is called “Codelink”.

THE WAITING GAME
When debugging software on an emulator with a probe something is always waiting. When the emulator is running, the developer is waiting for the design to get to the next breakpoint. When the software developer is looking at memory, variables, and source code the emulator is occupied (not available for other uses) but waiting for the next “run” command. With Codelink, most of this waiting is eliminated. The emulation runs straight through – never stopping. The developer will interact with the virtual target, not the emulator – so they will still need to wait for the virtual target to come to that same state. But the virtual target can be run on the same host as the debugger – eliminating the network and communications delays seen when using the emulator with a probe debugger. And since the virtual target is processing much less data than the original emulation it can run faster – more than 10 times faster. This means 10 times less waiting for the developer.

Codelink enables both expensive resources, the developer and the emulator, to be used much more efficiently.

LIMITATIONS
There are a couple of limitations with the trace based debug approach. Most notable is that the execution of the software cannot be changed – as what is being viewed in the debugger is a record of what happened. So the developer cannot change a memory location to a new value, or change the value of a variable and continue running.

BENEFITS
Debugging using a trace has some significant benefits as well.

Since the state of the target can be computed for any point in time, the “virtual target” can be run forward – as one would expect – but it can just as easily be run backwards. While the debugger can run in the normal forward mode,
it can also step and run the target backwards. This can make bug hunting a lot easier. One can start by setting breakpoints on exception handlers and error routines and then simply step backwards to find the source of the problem.

This also makes finding non-deterministic problems a lot easier. Once a problem has been captured in a trace it can be fully analyzed, without worrying about recreating the problem.

The trace is non-intrusive. Probe based debug activity introduces millions of clocks into the execution of the system, skewing performance and displacing events in time. This can introduce new bugs, or cover up existing ones. The trace based debug approach shows the system exactly as it will run. This is required for the analysis of synchronization between hardware and software – and between multiple cores.

A trace based debug approach provides a more responsive debug environment. Since the virtual target is processing less data to compute the state of the target system, it can provide a more responsive environment. It will run at least 10 times faster as compared to running live on the emulator. From the developers perspective, it delivers performance close to that of an FPGA prototype or development board.

Since the trace is performed on the processor itself, using the facilities of the emulation system, this approach does not rely on the debug circuitry which is part of the design itself. This means it can be used earlier in the design cycle – before facilities like Coresight™ are added to the design and before these debug facilities are fully debugged.

One of the most significant benefits is that the emulation run can be performed separately from the debug. This means that the emulation run can be put into a job queue and run when it is convenient for the emulator, and later the debugging can be performed when it is convenient for the developer. It also means that the debug operations will not consume emulator time.

Using a probe based debugger a 2 hour debugging session will consume 2 hours on the emulator. The entire time that the developer is using the debugger, the emulation resources are held exclusively. With the trace based approach, only the time needed to run the design is taken on the emulator – which is considerably less. The exact amount will depend on what is being debugged, and the nature of the hardware and software. Casual observation of a software developer in the debug process shows that most of the time is spent in examination of the system, and not running it. Measurements show that only 5 to 10 percent of this time is spent executing the system, while the remainder is spent examining and thinking. Thus, a 2 hour debug session may only require 5 to 10 minutes of emulation time. Using a trace based debug approach a single emulation seat can support many more software developers than a probe based approach.

CONCLUSION

Codelink – a trace based debug system – which gives software developers a traditional software debug view from a unique processor trace enables software developers using emulation to increase emulator utilization and enjoy a more productive debug experience. It is viable earlier in the design cycle as it can be used without having debug circuitry as part of the design. It is non-intrusive allowing visibility into synchronization and performance in a way not possible with probe based debug solutions.
20 Video Courses Available Covering

- Intelligent Testbench Automation
- Metrics in SoC Verification
- Verification Planning
- Basic and Advanced UVM
- Assertion-Based Verification
- FPGA Verification
- Testbench Acceleration
- Power Aware Verification
- Analog Mixed-Signal Verification

UVM and Coverage Online Methodology Cookbooks
Discussion Forum with more than 4400 topics
UVM Connect and UVM Express Kits

www.verificationacademy.com