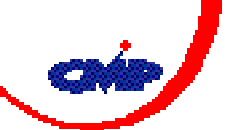
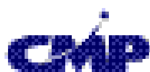




EmbeddedSystems
Conferences
San Francisco
Boston
Chicago
Europe
China

Executing an RTOS on Simulated Hardware using Co-verification

By David Harris, In-System Design

By DeVerl Stokes, In-System Design

By Russell Klein, Mentor Graphics

Presented on September 28, 2000 at ESC in San Jose, CA.

Abstract

Hardware software co-verification tools allow a designer to run software against a hardware design before a prototype has been built. They accomplish this by executing the software on a logic simulation of the hardware design. However, logic simulations are notoriously slow, typically running at about 1 to 10 clock cycles per second on complex designs. Co-verification tools mitigate this slowness by running only a small fraction of the bus cycles generated by the software against the simulated hardware. Most bus cycles, such as data references and instruction fetches, are executed against a simple memory array. This runs much faster and reduces the workload on the logic simulator. One consequence of this is that the "hidden" bus cycles are run without advancing time as far as the logic simulator is concerned. A problem occurs when the hardware has elements that depend on the clocks for proper operation, such as timers and counters.

This paper will detail our experiences with co-verification and some techniques that we used to maintain time synchronization between the hardware and the software execution. This synchronization is required to correctly model the hardware operating system tick and other timed hardware operations while running an embedded real time operating system on a virtual prototype of our system. We were able to observe the task switching and interrupt processing of the system while the software was interacting with the hardware as it will in the final target system.

Introduction

Co-verification is a relatively new technique that is used in the development of embedded systems. In the past, software developers often waited until a physical prototype of the systems hardware was available before doing much development of the code that would run on that system. But shrinking project development times and the increasing complexity of the software that runs on these systems has driven many developers to look for ways to begin the software development process sooner. Co-verification allows the software developer earlier access to the hardware design and provides increased visibility into the hardware design when problems are encountered.

Hardware Software Co-verification

Co-verification used to verify the interfaces between hardware and software for a new design, before a physical prototype is available. Co-verification combines a model of an embedded processor with an executable model of the embedded design. Within the model of the embedded system will be memory elements, RAM, ROM, FLASH, and perhaps other types of memories that will be loaded with the executable image of the software that will run on the embedded system. The model of the system is then "virtually" run and in this way the software and hardware can be observed and verified.

The model of the hardware is created using a hardware description language. Hardware description languages, or HDLs, are similar to the procedural languages used to create software. They have constructs that describe hardware elements, such as wires

and registers. They also have facilities for handling the concurrency inherent in hardware. While it may sound like a daunting task to create a representation of a hardware design, the hardware development team is probably already doing it if your design includes ASICs or medium to large capacity FPGAs. This hardware description language description of the embedded system can be executed or run in a logic simulator. This is typically what the hardware team will use to validate the operation of the hardware that they are designing.

The model of the embedded processor in a co-verification system provides a number of functions. First, it allows the software designer to view and affect the state of the processor and the software that is running in the co-verification system. To do this a graphical debugger is generally provided. The processor model also determines the behavior of the processor as it executes the software. This can be accomplished in a number of ways, the most common is to use an instruction set simulator. Finally, the processor model needs to produce the hardware activity of the processor as it interacts with the hardware design. This function is provided by what is called a bus interface model. The bus interface model allows the bus cycle activity of the processor to be translated into a sequence of pin change events that are compatible with the logic simulator being used. The bus interface model also needs to handle exceptions that might get generated by the hardware design, such as resets and interrupts. (See Figure #1) These components combined provide the full functionality of the processor being modeled. In some ways it can be thought of as a virtual in-circuit emulator that can be connected to a simulated circuit.

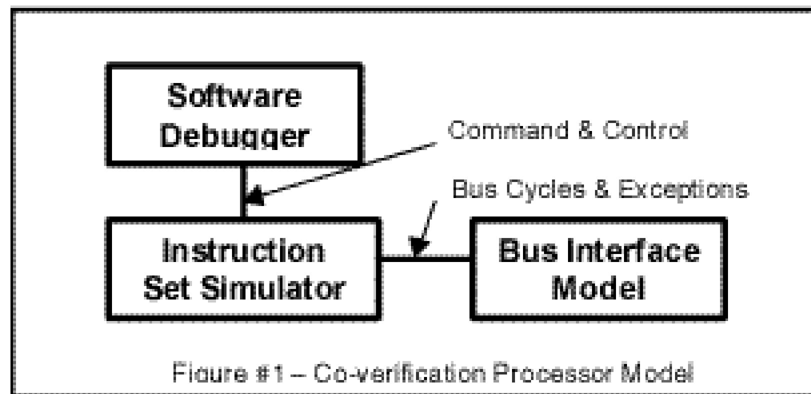


Figure #1 – Co-verification Processor Model

The co-verification tools links these two models together so that they can work to cooperatively run the hardware and software together. The co-verification environment starts the logic simulation of the circuit and the embedded debugger. Next it establishes the communication paths between the hardware and software of the system. The designer then runs the design by starting the logic simulator, then loading a software executable image into the design, generally through the debugger, and finally advancing the state of the system with the software debugger. The software debugger can be used to observe the internal registers of the processor, the state of memory and view variables symbolically. The debugger retains all of the debug functions that are available in a standalone version of the debugger. Likewise, the logic simulator in a co-verification environment supports all of the hardware debug features that the logic simulator would have in a standalone environment.

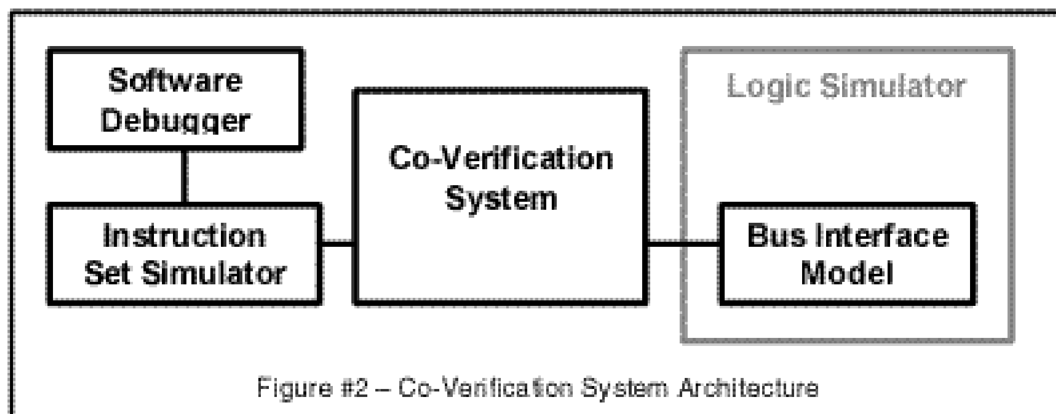


Figure #2 – Co-Verification System Architecture

While a complete description of the hardware design - with models of the processor, memory, control logic and the rest of the design - could be run in a logic simulator and would execute code loaded into the memories of the design, it would be too slow to run anything but a trivial amount of software. A real world design typically run at speeds of about 5 instructions per second. Doing some simple math we find that we could run about 144,000 instructions per 8 hours of compute time. As we consider the software that is in most systems today, this rate of execution is too slow to verify anything but the most trivial snippets of code. Another disadvantage of using just a logic simulator is that there is no debug visibility into the software.

Simulation Performance Optimization

Since logic simulation runs very slowly, co-verification tools need to provide some level of performance improvement to allow meaningful amounts of software to be executed while running a simulated design. All of the commercial co-verification approaches today rely upon "cycle hiding" to improve the performance of the system. This cycle hiding takes certain memory transactions and processes them against a memory array, rather then running them through the logic simulator. To illustrate this lets consider the example of an instruction fetch. This would generate a read cycle at the current program counter. If it were run in the logic simulator the bus cycle would propagate through the memory control logic and be driven against one or more memory elements and the memory contents are returned to the processor an interpreted as the next instruction. It is possible that the processor may update some internal cache information, but the state of the hardware design itself would not change - except in some very unusual circumstances. In processing this fetch cycle the logic simulator would process hundreds of simulation events and consume significant compute resources on the computer running the simulation. Since the bus cycle does not leave any state changes in the hardware, if the bus cycle were omitted from the logic simulation it would not affect the results of the simulation. A simple memory array could be created to hold the memory image and the instruction fetch in the instruction set simulator could get the op-code from this array quite efficiently.

As software executes it will generate a large number of instruction fetches, as well as data references and I/O cycles. One instruction in software may generate several bus cycles, a line of C-code may generate a dozen. Since hiding these bus cycles from the logic simulator results in much more efficient processing, we can see that this technique can significantly improve the performance of the simulation.

The instruction set simulator processes the op-codes and determines the resulting data movements or bus cycles that will get generated. The bus cycles are passed to the co-verification tool. The co-verification tool can then decide if the memory that the request is headed for is modeled in a simple data array, or what we will refer to as a "software" region. Alternatively, the bus cycle may be directed to the logic simulator, or what we will refer to as the "hardware" region. If it is in the hardware region, it is passed to the model of the pin interface and a series of signal transitions are driven into the logic simulation to emulate the bus cycle. If the bus cycle was a read, then the data is returned to the ISS, along with any interrupt or exceptions that may have occurred during the processing of the bus cycle. See Figure #3. Some commercial co-verification tools require a static configuration of these memory regions, other allow these regions to be changed during the simulation.

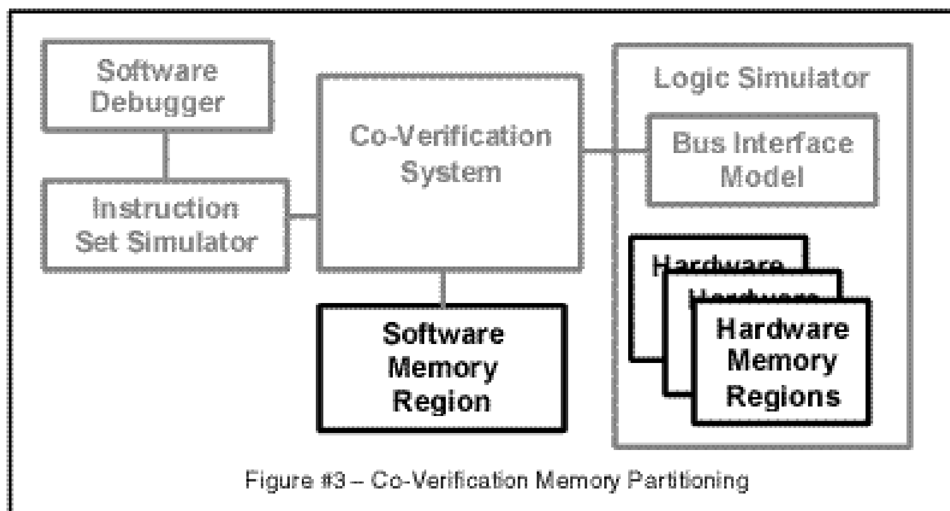


Figure #3 – Co-Verification Memory Partitioning

We have described co-verification from a theoretical standpoint. Now we will go over a brief example to show how things work. Consider the following code section. See Figure #4.

```
00F8    LD  R5, #0x8010
00FC    LD  R8, #0x550000
0100    LD  R9, @PC+0x20
0104    LD  R7, R8
0108    B   @R5
....
8010    LD  R1, @R8
8014    ADD R1, #4
8018    STR R1, @R8
```

Figure #4- Example Code Fragment

```
0000 - 011F   Code Space
0120 - 012F   Data Space
0130 - 8000   Unused
8000 - 8200   Code Space

550000  -     I/O Port
```

Figure #5- Example Memory Map

If we execute the code in Figure #4, starting at address 0x100 and executing 6 instructions, the hardware design would see a total of 11 bus cycle. See Figure #6.

```
Fetch  @ 0100          Instruction Fetch
Read   @ 0120          Data Reference
Fetch  @ 0104          Instruction Fetch
Fetch  @ 0108          Instruction Fetch
Fetch  @ 010C          Pipeline Prefetch
Fetch  @ 0110          Pip eline Prefetch
Fetch  @ 8010          Instruction Fetch
Read   @ 550000        I/O cycle
Fetch  @ 8014          Instruction Fetch
Fetch  @ 8018          Instruction Fetch
Write  @ 550000        I/O cycle
```

Figure #6- Bus Cycles

As we noted earlier, the co-verification tool would rarely run all of the bus cycles against the hardware design. If we take the code space and model this address region as "software" region - a fast memory array that is accessible to the instruction set simulator. The hardware design would not process any of the fetch cycles. In this case the hardware would process only 3 bus cycles – as shown in Figure #7.

```
Fetch  @ 0100          Instruction Fetch
Read   @ 0120          Data Reference
Fetch  @ 0104          Instruction Fetch
Fetch  @ 0108          Instruction Fetch
Fetch  @ 010C          Pipeline Prefetch
Fetch  @ 0110          Pipeline Prefetch
Fetch  @ 8010          Instruction Fetch
Read   @ 550000        I/O cycle
Fetch  @ 8014          Instruction Fetch
Fetch  @ 8018          Instruction Fetch
Write  @ 550000        I/O cycle
```

Figure #7- Bus Cycles

Just as we can hide the code space, we could also hide the accesses to the data space. If this were done the hardware would process only 2 bus cycles, as shown in Figure #8.

```
Fetch  @ 0100          Instruction Fetch
Read   @ 0120          Data Reference
Fetch  @ 0104          Instruction Fetch
Fetch  @ 0108          Instruction Fetch
Fetch  @ 010C          Pipeline Prefetch
Fetch  @ 0110          Pipeline Prefetch
Fetch  @ 8010          Instruction Fetch
Read   @ 550000        I/O cycle
Fetch  @ 8014          Instruction Fetch
Fetch  @ 8018          Instruction Fetch
Write  @ 550000        I/O cycle
```

Figure #8 –Bus Cycles

As we can see, using addresses or bus cycle types (or a combination) to partition how the memory is handled reduces the number of bus cycles that need to be processed by the logic simulator.

Now lets look at the implications of removing these bus cycles. One of the first things that we need to consider is what percentage of the bus cycles that are generated by the processor can be hidden from the logic simulation (or the hardware). It turns out that most bus cycles can be handled in this manner. We will go over several examples to illustrate this point.

The most hardware intensive code that can be written is a series of in-line store instructions. Consider the example section of code in Figure #9.

```
STR R5, @R4
STR R5, @R4
STR R5, @R4
```

Figure #9 – In-line Store Instructions

This code (Figure #9), would generate 6 bus cycles – 3 instruction fetches and 3 I/O cycles. Half of these bus cycles, the instruction fetches, could most probably be hidden from the hardware.

Putting that code snippet into a loop, which is a bit more realistic gives us the example in Figure #10.

```
LOOP:
  STR  R5, @R4
  B    LOOP:
```

Figure #10 – Store Instructions in a Loop

This would reduce the hardware cycles to 1/3 of the program. If you consider the effect of the pipeline stall which would occur as a result of the execution of the branch instruction and assuming a 3 stage pipeline the hardware cycles would be only 20%.

Let's consider a more realistic example – but still very hardware intensive - this section of code actually does something useful, copies a block of memory to a hardware port (see Figure #11).

```
  LD   R1, SRC + SIZE
  LD   R3, SRC
  LD   R4, OUT_PORT
LOOP:
  LD   R2, @R3
  STR  R2, @R4
  ADD  R3, #4
  CMP  R3, R1
  BNE  LOOP:
```

Figure #11 – Store Instructions in a Loop

Now we have 5 instructions in the body of the loop with only one being relevant to hardware – 20% (one in 8 if you add the pipeline effects, or 12.5%) A still more realistic example would actually check the status of the hardware port before and/or after accessing it and have some facility for handling error and exception conditions that inevitably occur. The point here is that most of what software does is reference instructions and data, not drive the hardware, even in sections of the code that are very dense with I/O cycles. The experience of co-verification vendors [1][2] is that 99.9% or more of the bus cycles generated by most embedded software can be masked from the hardware.

What this means is that the simulation of the system can proceed at a somewhat reasonable pace, especially when compared with logic simulation. The bus cycles that are run directly against the memory store can be run about 10,000 times faster than in than the same accesses run in the logic simulation. Lets consider a program that generates 1000 bus cycles. Lets further assume that the ratio of I/O cycles to code and data cycles is what has been typically observed – about 1 to 1000. Now for some simple math, 999 code and data references processed at a rate of 100,000 transactions per second and 1 I/O cycle processed at a rate of 5 transactions per second. This would take 0.2 seconds for the I/O cycle and 0.00999 seconds for the code and data references. Adding these values gives us a total run time of 209.99 milliseconds.

There are a number of conclusions that can be drawn from this example. First the logic simulator will be the bottleneck as we consider the performance of the overall co-verification session. In this example we are spending over 95% of our time in the logic simulator.

Many software developers perceive that the instruction set simulator running in the co-verification session will be too slow for their needs. An instruction set simulator is slow, however the co-verification runs will end up being more than 10 times slower than an instruction set simulator. While the performance will be slow – you will be running this weeks or months before you could have run the code waiting for a physical prototype. Contrary to what is assumed by many, replacing the ISS with some faster mode of execution of the software does not materially change the performance of the overall simulation. If we were to increase the performance of the ISS in the example by a factor of 1000, we would see a speed up the overall throughput from 209.99 milliseconds of run time to 200.00999 milliseconds. The improvement is less than 5% of the overall runtime. (Of course, the observant reader will note that we have simply provided an example of Ahmdal's Law.) While a 5 or 10 percent improvement in performance is nice, it would not warrant any loss of accuracy or debug capability. Replacing the instruction set simulator with a higher performance evaluation method would incur some loss of accuracy. Later in the paper we will discuss the need for this level of accuracy.

A final conclusion that we will draw from this discussion is that for co-verification to be and effective and viable technique, most bus cycles must be masked from the hardware. Fortunately, the nature of embedded code is such that this is possible. Most co-verification tools rely on the fact that the hardware and software are synchronized by events, and not the passage of time. However in most systems you cannot completely ignore the passage of time. Making a minor change in the performance of the system, say 10 or 20% would probably would have no impact on the operation of the system. Change it by a factor of 2 or 3 and it is quite probable that there will be some noticeable effects. Change it by a factor of 100 (which is what we are effectively doing) and something is bound to break. This does not mean that co-verification is not a valuable tool. Many designers are using it quite effectively. But as we run larger sets of software and operating the system in a more realistic fashion, we need to handle this "warping" of time in the simulated design.

Accounting for Hidden Cycles

As we discussed earlier, the hidden cycles shorten the logic simulations, as compared with a non co-verification simulations. This shortening of the simulations run time theoretically should not affect the logic simulation, since the memory transactions that are masked do not affect the state of the hardware system being simulated. However the passage of time can and does affect the state of the hardware. Consider the following set of bus cycles in Figure #13:

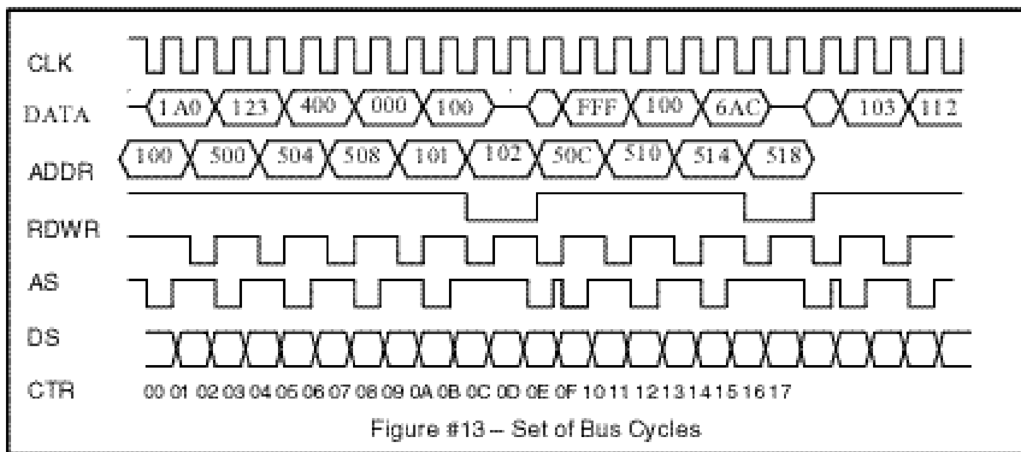


Figure #13 – Set of Bus Cycles

If we suppress the cycles in the range from 0x500 to 0xFFF we would see the hardware events as shown in Figure #14:

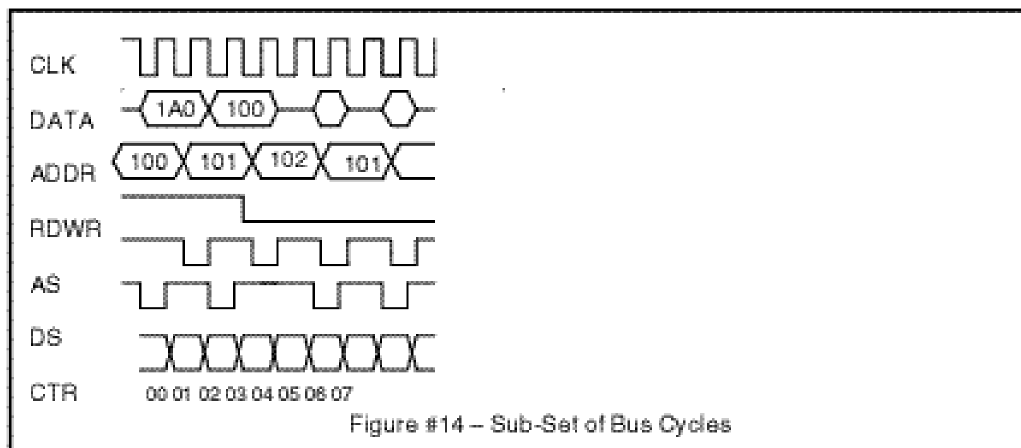


Figure #14 – Sub-Set of Bus Cycles

A timer that was counting the number of clocks in the first simulation would have received 24 clocks. The same timer in the second simulation would receive only 7 clocks. Recall that in actual co-verification sessions we will be suppressing over 95% of the bus cycles. This will put the timers badly out of sync with the software (typically we will be reducing 1000 clocks down to 1!)

There are 2 things that need to be done to compensate for this "time warping" effect of co-verification. First, the time associated with the missing bus cycles needs to be computed and updated into the registers in the hardware design that contain the timers. Second, we need to ensure that the logic simulation is active around the time of the critical event.

To accomplish this there are several facilities that we will need from the co-verification tool. First is the ability of the system to accurately compute the time that a given bus cycle would take on the system. This requires a model that takes into account all of the timing elements of the operation of the processor – including caches, pipelines,memory wait states and bus arbitration. Caches may change the time that a memory reference may take. If the data is in cache it can be read by the processor in fewer clock cycles than if a memory reference needs to take place. The pipeline can also affect the performance of the system, if the pipeline stalls waiting for data, or needs to be flushed due to a branch taken, it will change the timing of the bus cycles. The memory wait states will not be dependent on the processor, but will be dependent on the design. Another component of the time taken by a bus cycle will be the bus acquisition time, if there are other users of the bus, they can delay the completion of the bus operation.

The level of detail of the timing model that we need depends on what we are trying to verify. For keeping an OS timer updated, we could probably assume a given number of clocks per bus cycle and count bus cycles. This would give us a rough estimate that may be correct to within 10 or 20%. If we are trying to characterize the performance of the system, to determine interrupt latencies for example, a more accurate timing model would probably be required. The timing information that is available from the co-verification tool and processor model will depend upon the producer of the model and the tool. The model we used did include timing models for the internal pipeline of the processor, and could consider the effects of wait-states and bus acquisition times. Caches were not a consideration for us as our processor did not have a data cache.

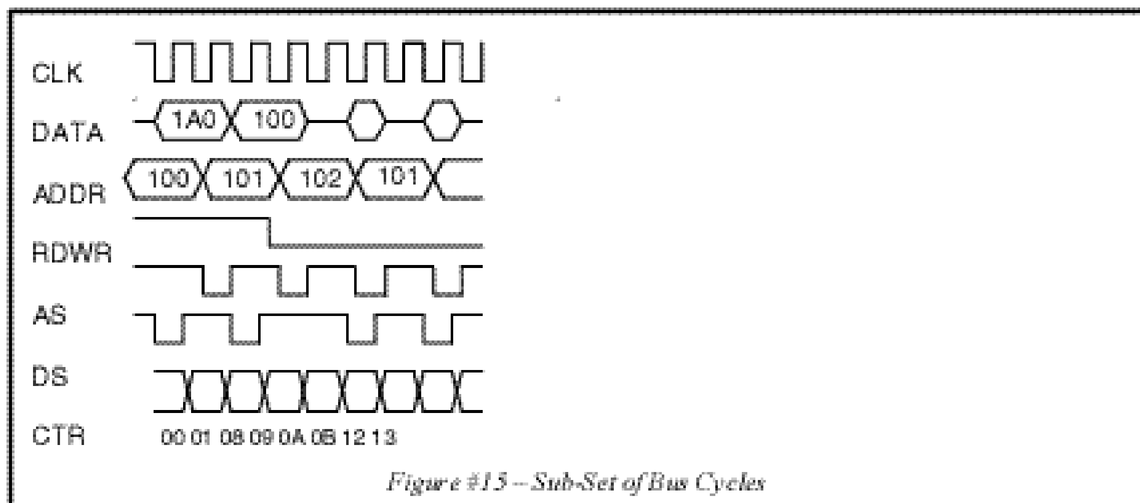


Figure #15 – Sub-Set of Bus Cycles

In addition to accounting for the time of the hidden cycles, we needed to be able to update the HDL timers with the adjustments needed for the hidden cycles. In Figure #15 we see what the simulation from Figure #14 looks like when we have the counter register updated with data from the co-verification tool. Notice that the timer no longer increments with the clock, but advances to match the waveforms from Figure #13, where we were driving all of the bus cycles into the logic simulator. Finally, we needed to be able to ensure that the hardware was not in a "hidden" bus cycle when the timer was to trigger an interrupt – as the interrupt would then be missed. Not all co-verification tools have hooks to get at such information. The tool we were using supplied HDL function calls to gain access to the time that advanced as cycles were hidden and allowed us to specify a time window around the timer trigger event.

The Design

The design that we used in our co-verification sessions was the "POSH" design. It is an SOC (System on a Chip) controller for an ink jet printer. It takes a PDL input (printer control language (PCL) or Postscript®) and builds the image to be printed. The image is composed of planar data, or as a separate image for each of the color planes. This is then used to create the raster image, which is in turn, printed by the print head(s). The hardware consists of an ARM based

embedded processor along with ASIC logic that implements the video frame processing logic (for which several patents were filed), memory control logic and external interfaces. The external interfaces include serial, parallel, USB, and Print Mechanism Controller (PMC). Also implemented in the ASIC is an interface to a front panel interface. See Figure 16.

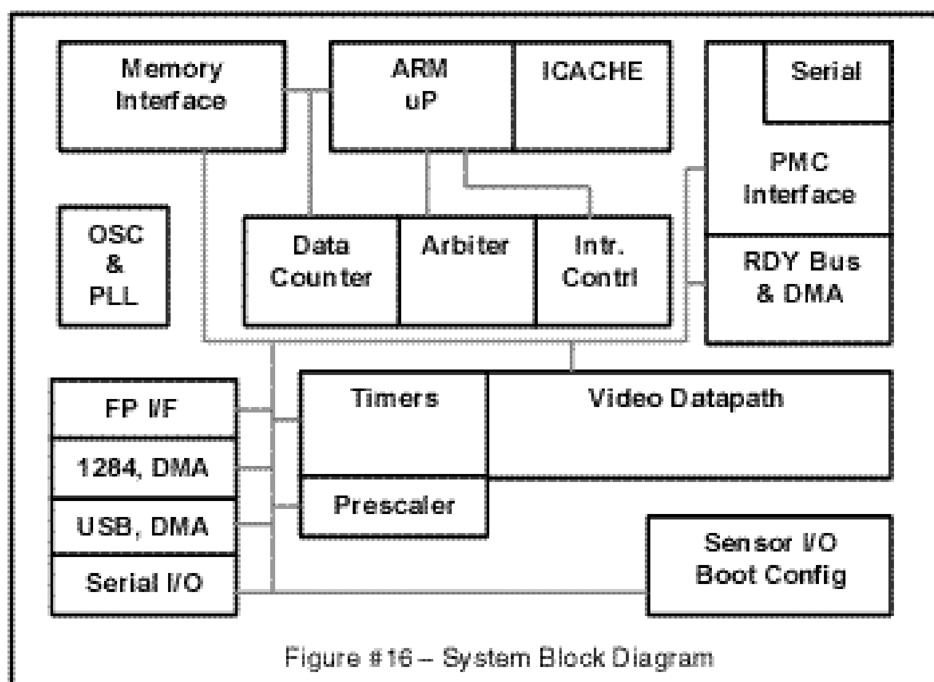


Figure #16 – System Block Diagram

The software run in the co-verification sessions was the Board Support Package (BSP) for the design, which included all of the drivers for the interfaces on the ASIC, and an API for the imaging application (which was developed by a 3rd party) and the VxWorks® operating system from WindRiver®. We created several small tasks to run in the RTOS to allow us to characterize the performance of the overall system and to see the effects of the co-verification tools. At the time this paper was submitted we had not yet run the printer applications using the co-verification tool. We expect to be able to present some results on running the application level code when this paper is presented.

Execution of RTOS

The goal of the co-verification sessions was to debug the board support package. We wanted to be able to completely verify that the system would boot once the physical prototype was created. If this is successful it means that there will be virtually no debugging of the BSP firmware once the physical prototype is in the lab. To do this we needed to prove the interrupt controller and its drivers, the serial port and the timers. We also wanted to see the task creation and scheduling take place within the context of the co-verification tool and determine if the performance was sufficient to run application level code.

Software Setup

Within the co-verification tool that we were using there were no changes to the software required to run the co-verification sessions. We should note that this is not the case for all co-verification tools. It was important to us that we be able to run the actual target system software in co-verification and not a modified version of the code. Although no changes were required, we did make some changes to the software build that we ran with our simulated design. First, there were several parts of the hardware system that were not included in the HDL description of the design. These were a non-volatile RAM and the Print Mechanism Controller (PMC). There was a software routine that performed a checksum on the NV-RAM, and this was omitted from the build, we also omitted the code that initialized and did a functional test of the PMC.

We also changed some of the software to improve the performance of the system. There were several memory tests, one that performed a complete test of RAM and some long tests on the ROM. These were either shortened or omitted to allow the system to boot faster. There is a short section of code that prints the VxWorks® banner to the UART, which took a couple of minutes of simulation time and was omitted from the build after running it once. We also modified the UART driver to output to an address that was easily snooped from the bus. We also omitted the UART test, which checked all possible baud rates. Finally, the C-run time library fills a large region of memory with zeros. We commented this routine out and used a debugger macro to initialize the memories instead. As we look forward to running application level code, we will probably remove all of the boot-up diagnostics, since those are now fully tested, to reduce boot-up time to a minimum.

Hardware Setup

The hardware did require some modifications to run in the co-verification tool. In the HDL description of the design we needed to replace the instances of the processor with the co-verification model provided by ARM. This was a simple addition of a new VHDL architecture and a two line configuration change. We also needed to replace several of the memory instances with co-verification memory models. This allowed the software debugger visibility into the memories of the simulated hardware without introducing bus cycles - which would be slow - into the logic simulation. This setup was fairly straightforward and was accomplished in less than a day.

To allow the timers in the HDL to be synchronized with the software, the co-verification tool provided several HDL function calls that we added to the description of the timers. These function calls allowed us to update the state timer with a software synchronized time value. While this did mean re-coding the timers to some degree, it involved changing a total of about 40 lines of HDL source code. This task was completed in a couple of hours.

Another change that was made to the hardware design was the addition of a bus monitor that watched for write cycles at address 0x31000000. This was the modified address of the UART, where the debug output from the RTOS would be written. This bus monitor was written in the TCL/TK extension language of the ModelSim logic simulator. The bus monitor opened a window on the Sun Workstation and echoed out the characters that were written to the UART. In this manner we were able to see the output from the RTOS as it executed.

Hardware/Software Synchronization Requirements

Earlier, we claimed that it was acceptable for the bus cycles to be hidden from the logic simulator because they did not affect the operation of the hardware. Then we showed you that the missing bus cycles did affect anything in the hardware design that was sensitive to the passage of time, since the time associated with the bus cycle is also omitted from the logic simulation. Finally we explained how we worked around this problem. Now we shall review some of the aspects of our design where it was necessary to work around this problem to get correct simulation results. The first and most obvious is the state of the hardware timers in the system. Another place where hardware and software needed to be kept completely in synchronization was a small section of code where we remap the memories, we call this the "atomic-swap" – as this must be performed as an atomic operation with respect to both hardware and software. We also have a number of DMA channels in our design that move data in and out of the system. We needed to make sure that the hardware had sufficient time (or clock cycles) to complete all DMA transfers. Finally, to do any type of performance analysis we needed to have the hardware and software synchronized. We were interested in measuring the maximum interrupt latency of the system.

System Timers

Within our system there are 3 main timers that are used to keep track of time in the hardware and generate the OS tick. The OS tick is configured to run at a 16 millisecond period. In our discussion of co-verification we showed how the state of the software advances faster than the state of the hardware as certain bus cycles are hidden. We also showed that the ratio of hidden cycles to non-hidden cycles is quite high. This means that if we did not compensate for this effect, the software would advance much more than 16 milliseconds before the OS tick arrived. Assuming a typical ratio of hidden to non-hidden cycles, the software would probably see about 16 seconds of time before the OS tick arrived. There is the temptation to reduce the hardware OS tick to 16 microseconds and hope all goes well. But, of course, this would fail if any of the tasks running performed a hardware operation that consumed more that 16 microseconds – which is quite probable. Leaving the OS tick at 16 milliseconds has the effect of forcing the task scheduling algorithm to be "run-to-completion". In our system, we cannot support a run to completion model, as many of the tasks simply run forever. To properly verify the system we need to have the tasks run for the correct amount of time. Our task switching model is Preemptive Priority Scheduling with a Round Robin scheduling for tasks of the same priority.

Atomic swap

When reset is released on our system, there is a ROM resident at location 0. This contains the reset code for booting the system. However, to run properly, VxWorks needs to be run with RAM at location 0. As part of the boot process we relocate the ROM from location 0 and move it to 0x800000, and move a RAM from 0x800000 to 0. While this is swap is taking place, we are executing code which is resident in ROM. For the underlying memory to be swapped correctly, without tripping the software, the hardware and software must be in complete synchronization. This also imposed a requirement on the co-verification tool that we be able to reconfigure the memory regions while the simulation is running.

DMA transfers

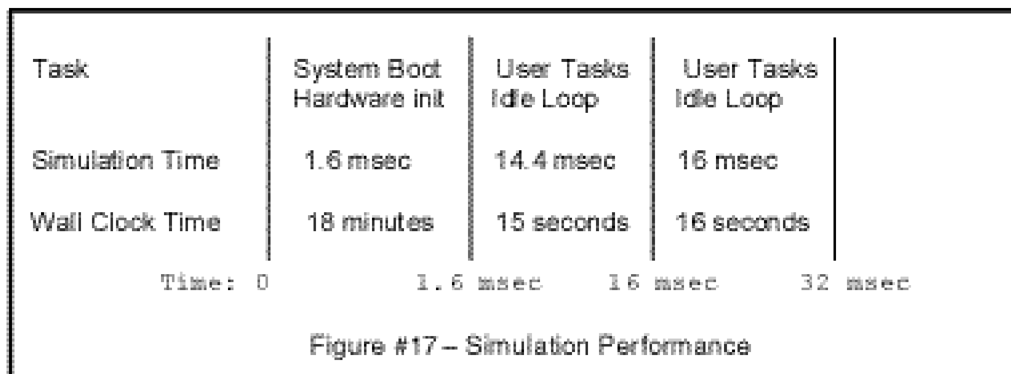
In our system there are a number of DMA channels. The DMA transfers are setup by the software by writing the appropriate pointers and counters to registers in the DMA controller. Once these are setup, the software moves on to other tasks and assumes that the hardware will complete the operation. For the operation to complete, the hardware must run some number of bus cycles. If the co-verification subsequently masks a large number of bus cycles from the hardware, the DMA may not complete. For this reason, we needed to co-verification tool to be able to suspend this cycle hiding for the duration of the DMA. We were able to accomplish this by setting breakpoints in the software that disabled the cycle hiding at the start of a DMA. In the hardware we

set breakpoints that watched for the completion of the DMA and would re-enable the cycle hiding. Once this was set-up the DMAs could be run correctly with no manual intervention.

Performance

With the design configured for co-verification, we proceeded to run a number of tests. To allow you to make performance comparisons, these are the tool that we were using. The computer was a Sun Ultra Sparc 60 with 2 processors running 360 MHz with 2 Gbytes of RAM running Solaris 2.6. The logic simulator was ModelSim version 5.4, the debugger was X-Ray version 4.4, the co-verification tool was Seamless version 4.0 – all of these software tools are from Mentor Graphics. The software was written in C, C++ and ARM assembler. The hardware design was represented in both Verilog and VHDL RTL code.

From the reset in the design until the creation of the first VxWorks task took approximately 18 minutes of wall clock time. Most of this time was consumed by the initialization of the hardware and the hardware diagnostics that are performed as part of the boot sequence. This advanced us to 1.6 milliseconds in the logic simulation. The OS tick for our system was configured to trigger every 16 milliseconds. By continuing to run the simulation – and not updating the timers – we estimated that it would take approximately 150 minutes (2 _ hours) to reach the first OS tick and almost 3 hours to reach the second OS tick. By accounting for the time of the hidden bus cycles, we were able to spin the idle loop of the RTOS without dragging along the slow logic simulator. When we did this we saw the OS tick trigger at the rate of 4 timer ticks per wall clock minute, see Figure 17.



Task	System Boot Hardware init	User Tasks Idle Loop	User Tasks Idle Loop
Simulation Time	1.6 msec	14.4 msec	16 msec
Wall Clock Time	18 minutes	15 seconds	16 seconds

Time: 0 1.6 msec 16 msec 32 msec

Figure #17 – Simulation Performance

We created 2 tasks and ran them in the operating system to allow us to see and measure the performance of the RTOS as it switched between tasks. The tasks were trivial, and simply wrote a character out to the UART. The UART had been modeled in the hardware in such a way as to print the characters output to a window on the Sun workstation. Each of the tasks relinquished its time by calling taskDelay(0) once the character had been delivered to the UART. By watching the characters being displayed we could determine the performance of the system as it switched between the tasks. We observed that we were able to get 16 task switches per wall clock second.

In another experiment we set the time slice of the operating system to 10 ticks, or for our design 160 milliseconds. We then configured the RTOS to switch between two simple tasks. The task went through a simple loop 220,000 times in one time slice. By disassembling the code, we found the inner-most loop of the task to contain 11 instructions – for a total of roughly 2.2 million instructions executed. We were able to complete the time slice on the simulated design in 170 seconds.

Conclusions

When we initially looked into co-verification tools, we were interested in verifying the interfaces between the hardware and software. To do this we expected to be able to run the boot and initialization code and the board support package. With the BSP and boot routines debugged and known good when the prototype arrived we felt confident that we could begin application debug immediately. By updating the timers in the HDL, we have reduced the simulation time between OS ticks from several hours to a fraction of a minute. This speed-up not only makes it possible to test our boot and initialization code before prototypes are available, but we can also exercise the RTOS and application code as well.

While the performance is much faster than logic simulation it is still quite slow when compared with a live target or an evaluation board. Being able to verify hardware/software interfaces in virtual hardware gives us several immediate advantages. First, we can overlap firmware and hardware designs by getting software designers onto a virtual version of the target hardware design as soon as RTL code becomes available, significantly reducing overall development schedules. Secondly, we reduce the risk of hardware/software incompatibilities. That "we can always fix it in software" requires either the addition of resources or a slip in schedule – or both. These always translate directly into additional dollars spent on the system design. Since to many programs any significant schedule slip will cause the product to completely miss its market window, co-simulation appears to be more of a program development requirement than simply an option.

References:

[ ] R. Klein, "Miami, a Hardware/Software Co-verification System," in Proc 7th IEEE Rapid Systems Prototyping Workshop, 1996, p. 173-177

[2] M. Stanbro, "Getting to the Bottom of HW/SW Co-verification Performance Claims," Computer Design, Vol 37 No 12, December 1998, p65-67

Copyright© 2000 by **CMPMedia Inc.,** 600 Community Drive, Manhasset, NY 11030. Reprinted from EMBEDDEDSYSTEMS CONFERENCE with permission.