BREKER™

TrekSoC-Si WHITE PAPER

Post-Silicon Validation of an x86 Server SoC

Summary

The engineers for the "Blender" system-on-chip (SoC) project were not sleeping well. Blender was the heart of a new server based on the Intel x86 architecture. It was a very complex chip and the team knew that it had not been verified well enough before the mandated tape-out date. They wanted to stress-test the silicon as soon as it came back from the foundry in order to find any lurking bugs and build confidence that it was ready for production.

The Blender team chose Breker's TrekSoC-Si™ product to automatically generate multi-threaded, multi-processor test cases to run on the actual chip in the lab. Figure 1 is a real-time display of one test case running with two threads on each of Blender's 12 processors. The colored lines show end-user scenarios migrating from processor to processor and thread to thread.

The team generated and ran hundreds of such test cases on the SoC using the lab setup shown in Figure 2. The test cases stressed Blender's interconnect fabric and memory subsystem, finding a few design issues that, fortunately, the team was able to fix in software. Once these fixes were made, it took much less time to get the production software running than on previous projects. The team approved Blender moving into production as part of a successful server product.

Project Overview

As shown in Figure 3, Blender contains 12 embedded processors (eight x86 central processing units (CPUs) and four applications processors), a highly sophisticated graphics processing unit (GPU), an I/O subsystem, and a multi-channel memory controller. These are all interconnected by a complex switching fabric.

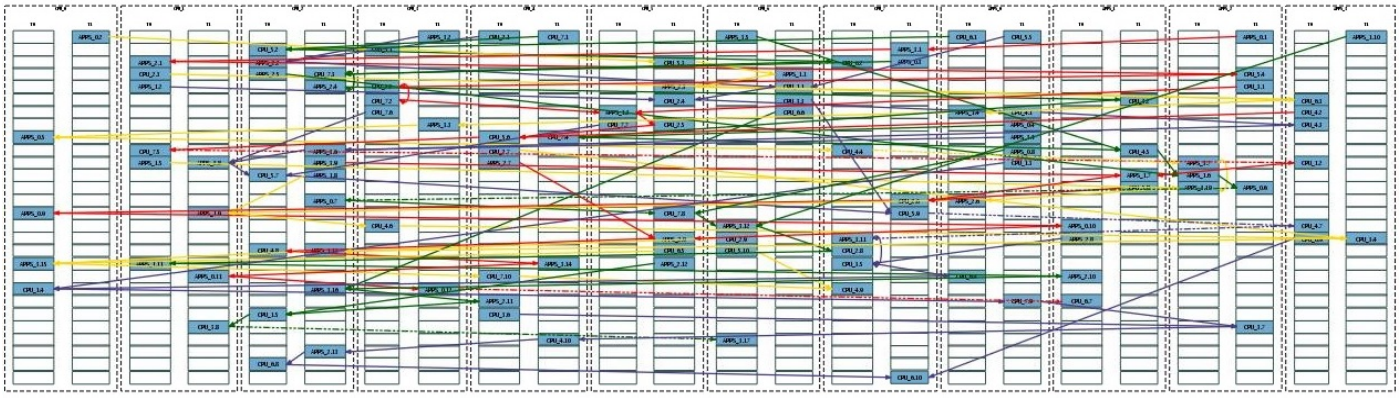


Figure 1: Test Case Map Showing Two Threads Running on Each of 12 Processors in Blender Chip

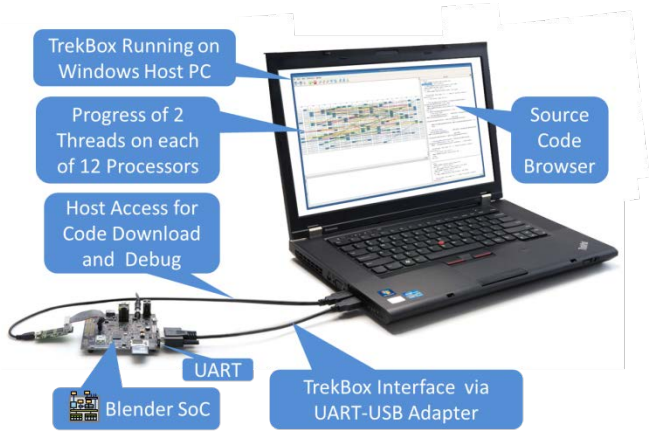


Figure 2: Blender Silicon Lab Setup

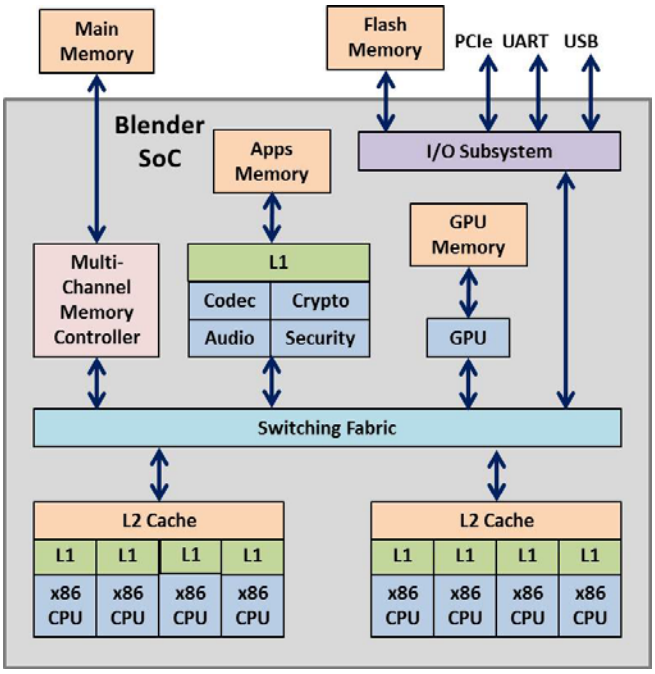


Figure 3: Blender SoC Block Diagram

The eight x86 CPUs are arranged in two clusters of four each. The identical CPUs contain 64-bit data paths and have an attached level one (L1) cache. In addition to the four CPUs, each cluster contains a level two (L2) cache. All CPUs share a common memory space so that all the L1 and L2 caches must remain coherent with main memory.

Access to main (off-chip) memory is provided by the multi-channel memory controller, which receives requests from the central switching fabric. The fabric, in turn, routes memory requests and responses, internal bus transfers, and other messages among main memory, the CPU clusters, the applications processors, the GPU, and the I/O subsystem.

The I/O subsystem provides connections to an off-chip flash memory and to several standard protocols. These include PCI Express (PCIe), USB 3.0, and a UART. The GPU has access to a dedicated on-chip memory used for intermediate results as it performs its computations.

The four applications processors handle audio, coding/decoding (codec), cryptographic, and security functions. They share an L1 cache, access to a dedicated on-chip memory, and an interface to the switching fabric.

Verification Process

In pre-silicon simulation, the Blender team developed a constrained-random testbench compliant with the Accellera Universal Verification Methodology (UVM) standard. This testbench was used initially to exercise Blender's standard I/O ports, which used third-party design IP. The focus was on ensuring proper connection of the IP rather than repeating the standalone verification performed by the IP providers.

Since the bulk of Blender's power lies in its embedded processors, the team hand-wrote some simple C tests to run on the processors in simulation. The focus of these tests was verifying that the processors and other IP components were instantiated and interconnected properly within the SoC. The UVM does not encompass code running on embedded processors, so the two simulation efforts were largely independent.

The Blender team wrote and ran as many simulations as they could before a mandated

tape-out date. They even continued to run tests as the chips were being fabricated. In parallel, the software engineers were writing the production code for end-user applications. The team planned to run the production code on the first chips as soon as they arrived from the foundry. The validation of hardware and software running together was considered an essential step.

Verification Challenges

As the Blender team analyzed the coverage achieved in pre-silicon simulation, they identified several troublesome verification holes. Some of these were due to the complex routing paths through the switching fabric. Although not a complete crossbar switch, the fabric supports multiple concurrent operations. This led to the following identified verification challenges:

- Multiple types of agents access the fabric
- Fabric has multiple paths to each agent
- Multi-level fabric supports aggressive transaction reordering for performance
- Fabric also supports split transactions
- Data routing and ordering must be correct even during periods of heavy concurrency

Since all the processors and the GPU share a common memory space, verifying coherency was the other major concern. The Blender verification engineers noted the following challenges related to memories, caches, and coherency:

- The CPUs, apps processors, and GPU access memory and caches with different protocols and coherency rules
- The multiple memory controllers and channels contain internal buffering
- The memory subsystem must remain cache-coherent even under system stress

The team knew that these concurrency and coherency challenges were not addressed well by the UVM testbench, which supported only limited operations in parallel. They tried to fill this gap when hand-writing C tests to run in simulation but found that it was impossible to code in parallel for twelve processors if there was any significant level of interaction among them.

Further, the team was concerned that running production code on the chips in the lab would not sufficiently address the shortfalls in pre-silicon verification. Production software is designed to perform end-user tasks and not to find SoC design bugs. It does not exercise corner cases well and it can require days or weeks to debug even if suspected problems are detected.

The engineers specifically wanted to stress the SoC design by running many parallel threads across its multiple processors, accessing the common memory and cache structures. They knew that they had been unable to accomplish this before the mandated tape-out date and did not believe that production code would stress the design well enough. Thus, the Blender team investigated several options to better exercise the SoC before committing it to production.

Selection of TrekSoC-Si

The engineers chose the TrekSoC-Si product from Breker Verification Systems to supplement the pre-silicon testbench simulation and the hardware-software co-validation in the bring-up lab. TrekSoC-Si automatically generates multi-threaded, multi-processor, self-verifying C test cases that run on the SoC's embedded processors in any hardware platform, including production silicon. A companion product, TrekSoC™, is available for users who want to run the test cases

in RTL simulation and system-level virtual prototypes as well.

TrekSoC-Si takes as input a graph-based scenario model™ describing the design's intended functionality and the test space desired. It looks much like a standard dataflow diagram that the architect of the SoC would draw. It is possible to create a scenario model graphically, but most users opt to describe it textually using standard C/C++. Figure 4 shows a snippet of the scenario model definition for the Blender SoC. Figure 5 shows the complete Blender top-level graph-based scenario model.

The blue boxes represent actions whose children are executed sequentially, while the red

```
////////////////////////////////////////////////////////////////////////
// CPU instantiation
Cpu cpu_0("CPU_0");
Cpu cpu_1("CPU_1");
Cpu cpu_2("CPU_2");
Cpu cpu_3("CPU_3");
Cpu cpu_4("CPU_4");
Cpu cpu_5("CPU_5");
Cpu cpu_6("CPU_6");
Cpu cpu_7("CPU_7");

goal trek_memory_address_t cpu_check (trek_memory_address_t dest, trek_unsigned_t size )
  := cpu_0.cpu_check |
     cpu_1.cpu_check |
     cpu_2.cpu_check |
     cpu_3.cpu_check |
     cpu_4.cpu_check |
     cpu_5.cpu_check |
     cpu_6.cpu_check |
     cpu_7.cpu_check
{}

goal trek_memory_address_t cpu_copy (trek_memory_address_t dest, trek_unsigned_t size, int recursion_depth )
  := cpu_0.cpu_copy |
     cpu_1.cpu_copy |
     cpu_2.cpu_copy |
     cpu_3.cpu_copy |
     cpu_4.cpu_copy |
     cpu_5.cpu_copy |
     cpu_6.cpu_copy |
     cpu_7.cpu_copy
{}

// can consume any memory
goal trek_memory_address_t cpu_producer ( trek_unsigned_t size, int recursion_depth )
  := fill_buffer
{}

////////////////////////////////////////////////////////////////////////
// APPS instantiation

apps apps_0("APPS_0");
apps apps_1("APPS_1");
apps apps_2("APPS_2");
apps apps_3("APPS_3");

goal trek_memory_address_t apps_check (trek_memory_address_t dest, trek_unsigned_t size )
  := apps_0.apps_check |
     apps_1.apps_check |
     apps_2.apps_check |
     apps_3.apps_check
{}
```

Figure 4: Blender Scenario Model Definition

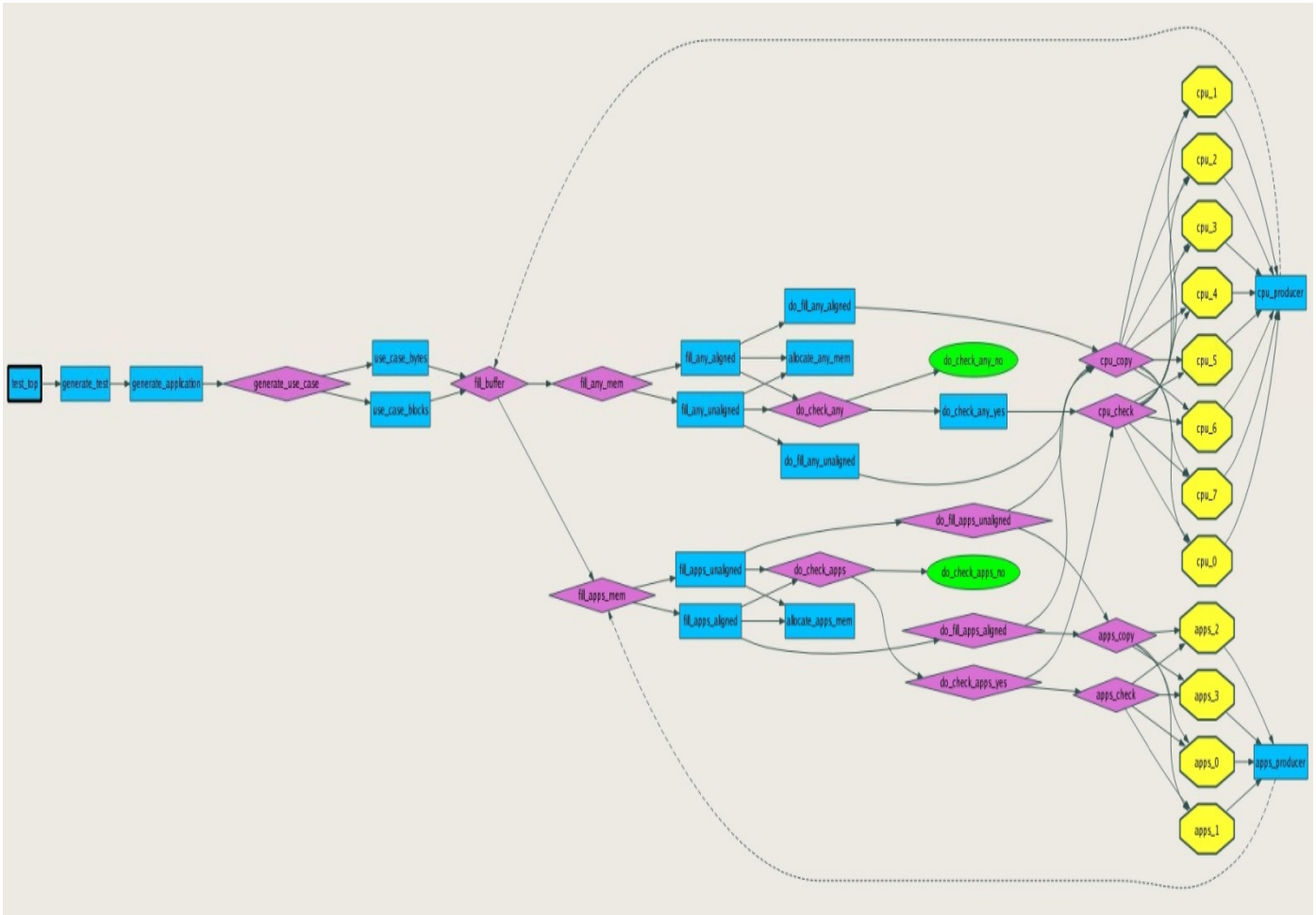


Figure 5: Top-Level Scenario Model for the Blender SoC

diamonds are decision points at which only one child is executed for each path through the graph. Breker scenario models are fully hierarchical so that lower-level graphs can easily be instantiated at the full-chip level. This provides a high degree of verification reuse. The yellow octagons represent sub-graphs that can be expanded when desired.

Scenario models are not inherently complex; any challenges in creation arise because they reflect the complexities of the SoC design itself. One example in the Blender design is asymmetry in the cache coherency requirements. If one of the applications processors reads a location in shared memory space, the CPU cache snooping logic must detect this, and the CPUs must flush any cache lines containing the same location so that the read gets the correct value.

In contrast, when one of the CPUs reads a shared memory location there is no corresponding flush in the cache for the applications processors. Graph-based scenario models have the flexibility to reflect design subtleties such as differing coherency rules. Developing a scenario model that can generate hundreds of test cases automatically takes about the same effort as hand-writing just a single test.

TrekSoC-Si Deployment

The usage of TrekSoC-Si began with a one-day training course for the Blender team, followed by roughly a week of on-site support by Breker applications engineers. By the end of the third day, the engineers were fleshing out the graph with enough detail that interesting test cases could be generated. These test cases were running on the Blender chips in the lab by the end of the first week.

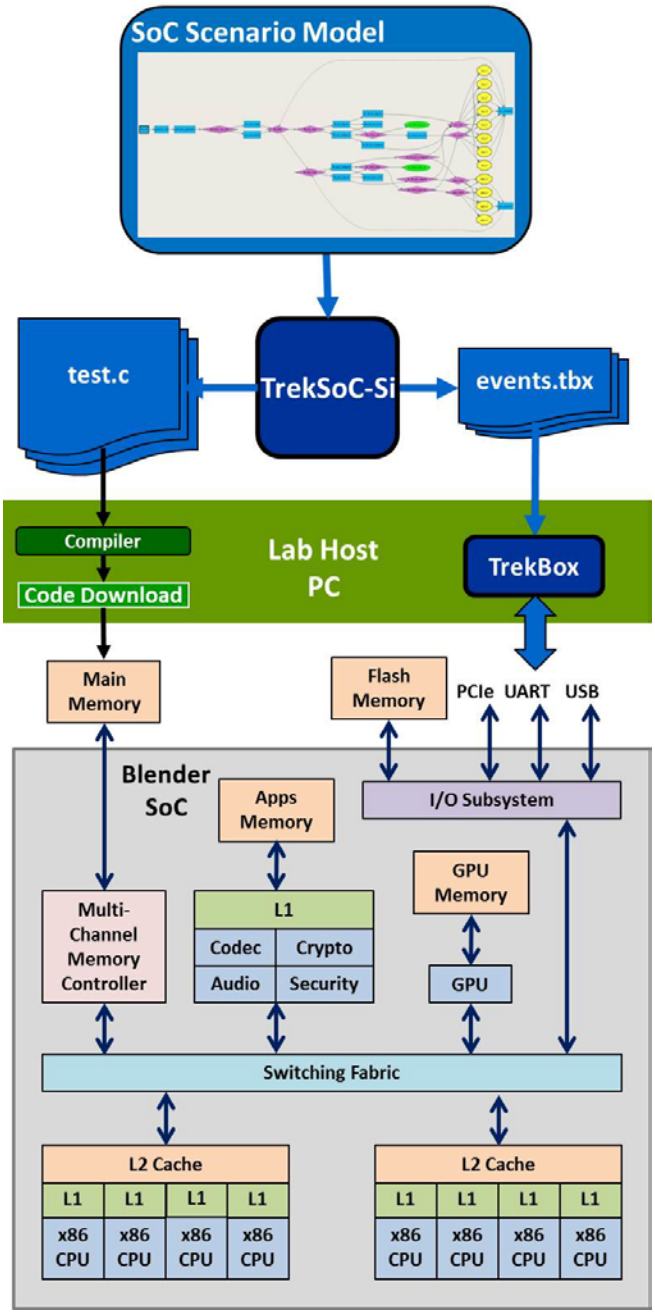
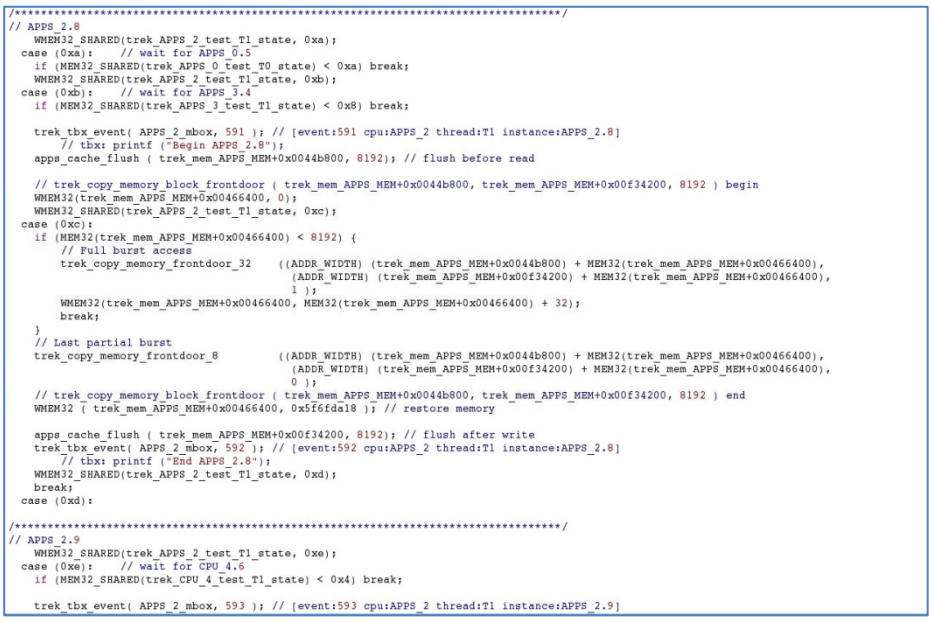


Figure 6: TrekSoC-Si Blender Connections

As shown in Figure 6, Breker also provides a runtime module called TrekBox™ that reports the progress of the test cases as they run on multiple threads over multiple processors. This module runs on the same host system used to download code to the Blender chip in the lab, and communicates with the embedded processors via a UART port in the I/O subsystem.



```
/*****************************************************************************/
// APPS_2.8
    WMEM32_SHARED(trek_APPS_2_test_T1_state, 0xa);
  case (0xa):    // wait for APPS_0.5
    if (MEM32_SHARED(trek_APPS_0_test_T0_state) < 0xa) break;
    WMEM32_SHARED(trek_APPS_2_test_T1_state, 0xb);
  case (0xb):    // wait for APPS_3.4
    if (MEM32_SHARED(trek_APPS_3_test_T1_state) < 0x8) break;

    trek_tbx_event( APPS_2_mbox, 591 ); // [event:591 cpu:APPS_2 thread:T1 instance:APPS_2.8]
        // tbx: printf ("Begin APPS_2.8");
    apps_cache_flush ( trek_mem_APPS_MEM+0x0044b800, 8192); // flush before read

    // trek_copy_memory_block_frontdoor ( trek_mem_APPS_MEM+0x0044b800, trek_mem_APPS_MEM+0x00f34200, 8192 ) begin
    WMEM32(trek_mem_APPS_MEM+0x00466400, 0);
    WMEM32_SHARED(trek_APPS_2_test_T1_state, 0xc);
  case (0xc):
    if (MEM32(trek_mem_APPS_MEM+0x00466400) < 8192) {
        // Full burst access
        trek_copy_memory_frontdoor_32    ((ADDR_WIDTH) (trek_mem_APPS_MEM+0x0044b800) + MEM32(trek_mem_APPS_MEM+0x00466400),
                                          (ADDR_WIDTH) (trek_mem_APPS_MEM+0x00f34200) + MEM32(trek_mem_APPS_MEM+0x00466400),
                                          1 );
        WMEM32(trek_mem_APPS_MEM+0x00466400, MEM32(trek_mem_APPS_MEM+0x00466400) + 32);
        break;
    }
    // Last partial burst
    trek_copy_memory_frontdoor_8         ((ADDR_WIDTH) (trek_mem_APPS_MEM+0x0044b800) + MEM32(trek_mem_APPS_MEM+0x00466400),
                                          (ADDR_WIDTH) (trek_mem_APPS_MEM+0x00f34200) + MEM32(trek_mem_APPS_MEM+0x00466400),
                                          0 );
    // trek_copy_memory_block_frontdoor ( trek_mem_APPS_MEM+0x0044b800, trek_mem_APPS_MEM+0x00f34200, 8192 ) end
    WMEM32 ( trek_mem_APPS_MEM+0x00466400, 0x5f6fda18 ); // restore memory

    apps_cache_flush ( trek_mem_APPS_MEM+0x00f34200, 8192); // flush after write
    trek_tbx_event( APPS_2_mbox, 592 ); // [event:592 cpu:APPS_2 thread:T1 instance:APPS_2.8]
        // tbx: printf ("End APPS_2.8");
    WMEM32_SHARED(trek_APPS_2_test_T1_state, 0xd);
    break;
  case (0xd):

/*****************************************************************************/
// APPS_2.9
    WMEM32_SHARED(trek_APPS_2_test_T1_state, 0xe);
  case (0xe):    // wait for CPU_4.6
    if (MEM32_SHARED(trek_CPU_4_test_T1_state) < 0x4) break;

    trek_tbx_event( APPS_2_mbox, 593 ); // [event:593 cpu:APPS_2 thread:T1 instance:APPS_2.9]
```

Figure 7: Blender Test Case Source Code

As shown in Figure 2, TrekBox includes a browser for the generated test case source code (also shown in Figure 7) and a map of the test case running across multiple threads and multiple processors (also shown in Figure 1). Both displays are animated and updated in real time as the test case runs on the chip. Periodic status updates are sent to TrekBox via the UART connection.

If a self-checking test case fails, TrekBox highlights the step in the test map display and the line of code in the source display. The Blender team found this capability a major asset in debugging the reason for the failure, especially when compared to production software that hangs or crashes with little debug information.

For each Blender test case, TrekSoC-Si generated 12 C/C++ files, one for each x86 CPU and applications processor. Figure 1 is an actual test map showing two threads running on each of the 12 processors. The Blender team expanded this to four threads per processor and ultimately to eight threads apiece, for a total of nearly 100 threads executing in parallel in each test case.

The many test case variations generated by TrekSoC-Si provided a level of exercise for the Blender design that could not have been accomplished in any other way. This included concurrent use cases from all system masters, concurrent access to all system memories, and a high degree of stress on the fabric and caches. The Blender team determined that the generated test cases covered the five verification challenges for switching fabric concurrency and the three challenges for memory/cache coherency that they identified as insufficiently addressed by pre-silicon simulation.

Results

The Blender team was relieved that the TrekSoC-Si test cases found no “killer bugs” that would have caused a chip “turn” (re-fabrication). The generated test cases did find some issues where the hardware implementation did not match the specification, a class of bug that the team was able to fix in software. If the team had run TrekSoC during pre-silicon verification, these issues could have been fixed in hardware.

The validation of the chip in the lab happened more quickly than expected. The team found that bringing up the operating system and production software was much easier once the test cases from TrekSoC-Si were passing. The final result was that the Blender SoC went to production with the team able to sleep at night, free from fears that a key customer’s server would someday crash due to a killer bug still lurking deep within the chip.

© 2013 Breker Verification Systems, Inc. 1879 Lundy Ave., Suite 126, San Jose, CA 95131
Call +1.650.336.8872 or visit us at www.brekersystems.com