

Chapter 2

Embedded Software Debug in Simulation and Emulation Environments for Interface IP

Cyprian Wronka and Jan Kotas

2.1 Firmware Debug Methods Overview

Present EDA environments [1, 2] provide various methods for firmware debug. Typically one can use one of the following:

- Simulation with a SystemC model of the hardware. This allows for a very early start of firmware development without any access to hardware and allows to test the functionality of the code assuming the model is accurate. The main limitations are lack of system view and (depending on the model accuracy) lack of hardware timing accuracy (behavioral models).
- Hardware simulation with firmware executing natively on the simulator CPU. This is the simplest method incorporating the actual RTL that allows to prototype the code. It requires some SystemC wrappers to get access to registers and interrupts. It lacks the system view and therefore cannot verify the behavior of the firmware in the presence of other system elements.
- Playback (with ability to play in both directions) of a recorded system simulation session.
- Hardware simulation with a full system model. (This is a synchronous hybrid, where RTL and software are run in the same simulation process). This can be divided into:
 - Using a fast model of the CPU [3]—this allows very fast execution of code (e.g., Linux boot in ~1 min) but lacks cycle accuracy due to TLM to RTL translation. It also slows down significantly when the full RTL simulation starts (all clocks enabled). Example of such a system is presented in Fig. 2.1.
 - Using a full system RTL—this is generally very slow and only allows to test simple operations (under 10k CPU instructions) in a reasonable time.

C. Wronka (✉)
Cadence® Design Systems, San Jose, CA, USA
e-mail: cwronka@cadence.com

J. Kotas
Cadence® Design Systems, Katowice, Poland
e-mail: jank@cadence.com

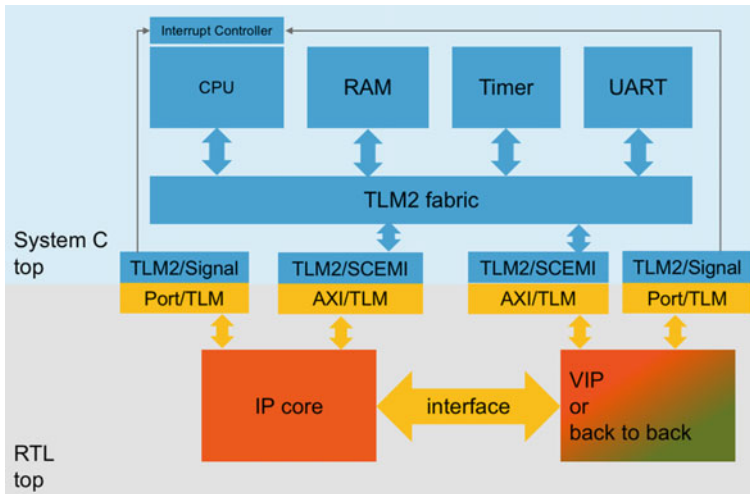


Fig. 2.1 Diagram showing a generic hybrid hardware simulation or emulation environment with a TLM part including the CPU fast model (*top*) and the RTL part including an interface IP core (*bottom*)

- Hardware emulation [4] of the full system. Again this can be divided into:
 - Hybrid mode consisting of a fast model CPU and emulation RTL. In the case of interface IP it provides very fast execution, but requires good management of memory between the fast model and the emulation device to assure that the data transfers (typically data written by CPU and later sent to interface) will be efficiently emulated. *NOTE: In this mode software is executing asynchronously to RTL and the two synchronize on cross domain transactions and on a set time interval. Effectively the software timing is not cycle accurate with the hardware and depending on setup would remove cache transactions and cache miss memory transactions.*
 - Full RTL mode where all system is cycle accurate. This is slower (Linux boot can take 10 min), however consistent performance is maintained through the emulation process. This mode allows to test the generic system use cases or replicate problems found during FPGA testing.
 - Emulation with no CPU—A PCIe SpeedBridge® Adapter can be used to connect an arbitrary interface IP device to a PC and develop a driver in the PCIe space. The emulation environment allows for access to all internal signals of the IP (captured at runtime, even using a very elaborate condition-based trigger) to debug the issues (whether originating from, software, hardware or the device connected at the other end of the interface).
- Hardware prototyping using FPGA. In this case the processor can run at 10–100 s of MHz (or even at GHz speeds if it is a silicon core connected to the FPGA logic).

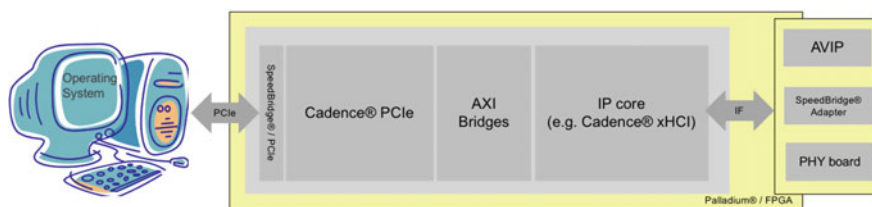


Fig. 2.2 Example setup of a PCIe device used for Linux driver debug with Cadence® Palladium® platform

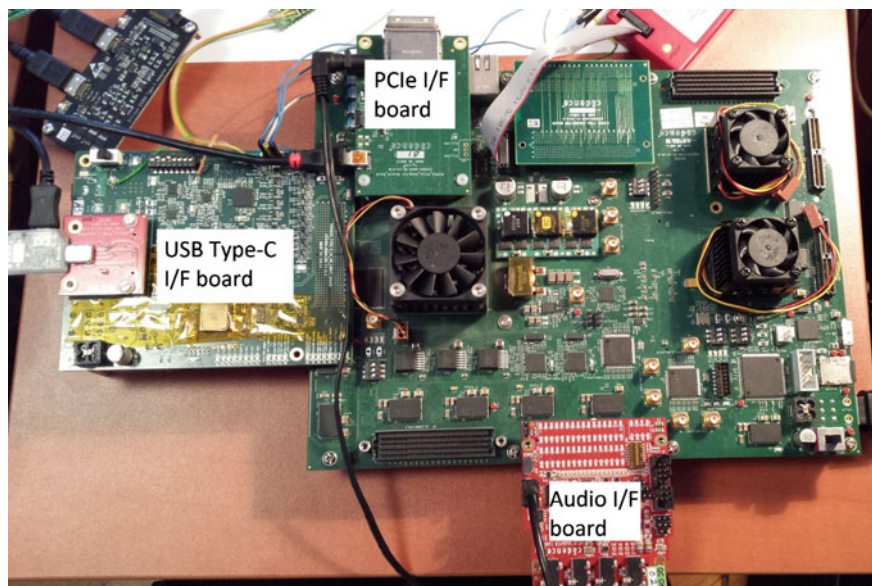


Fig. 2.3 Example Cadence® FPGA board setup used for interface IP bring-up. Please note the number of connectors and the attached boards. This is required for testing systems consisting of multiple interface IPs connected together (also with a CPU and system memory)

These environment are not very good at bringing up new (unproven) hardware, however they are great for:

- System tests in real time or near real time
- Performance tests
- Soak tests.

Example schematic used both for FPGA and Emulation or RTL interface IP connected to a standard PC is shown in Fig. 2.2. Alternatively it is possible to prototype a simple SoC in an FPGA. Such a system with PCIe EP (End Point) IP, USB Device IP, and an audio interface is presented in Fig. 2.3.

- Testing in silicon. This is generally considered an SoC bring-up, where all hardware issues should be ironed out, this does not prevent the fact that some tuning may still be done in firmware. The system is running at full speed and there is generally no access to the actual hardware (signals), however there should be a processor debug interface allowing to step through the code.

When debugging firmware for interface IP in simulation or emulation, it is required to connect the interface side of the IP to some entity that is compliant with the interface protocol. To achieve this one can use:

- In simulation:
 - Verification IP
 - Another instance of IP with the same interface
- In emulation:
 - Accelerated Verification IP
 - Another instance of IP with the same interface
 - A SpeedBridge® Adapter to connect to the real world.

2.2 Firmware Debuggability

In many cases the engineer looking after the firmware may not have been the creator of the code, and therefore it is important to provide as much information about the functionality of the code as possible.

The best practices include:

- Self-explaining register name structures or macros
- Self-explaining variable names (as opposed to a, b, c, d)
- Self-explaining function names
- Especially useful is information about the usage of pointers, as often firmware uses pointers to store routine addresses, it is important to provide sufficient information for a debugging engineer to be able to understand what the code is supposed to do.

There are new systems appearing currently on the market, where majority of IP driver code can be automatically generated from a higher level (behavioral) language [5]. In that case it is important to be able to regenerate the code and trace all changes back to the source (meta code) in order to propagate all debug

```

1 DESCRIPTION = ``Sequence for starting playback.``;
2 sequence start_playback
3 {
4     LOCAL unsigned int idx;
5
6     idx=0;
7     repeat (idx<8)

```

```

8  {
9      /* If a channel has been enabled, start playback on that
10     channel*/
11     if (DRV_I2S_CID_CTRL.I2S_STROBE[idx] == 0)
12     {
13         DRV_I2S_CTRL.ENB[idx] = 1;
14     }
15     idx = idx+1;
16 }
17 /* Enable clock to transmit sync */
18 DRV_I2S_CID_CTRL.STROBE_TS = 1;
19 DRV_I2S_CID_CTRL.STROBE_RS = 0;
20 DRV_I2S_CTRL.TFIFO_RST = 0;
21
22 /* Reset */
23 DRV_I2S_CTRL.TSYNC_RST = 1;
24 }

```

Listing 1 Example code using VayavyaLabs DPS language.

Generated code:

```

1  /*!
2  * \brief Sequence for starting playback.
3  * \return Success or Failure
4  * \retval 0 Success
5  * \retval -1 Failure
6  */
7
8  INT I2S_MC_IP_SLAVE_start_playback(void)
9  {
10     UINT idx;
11     UINT varDRV_I2S_CID_CTRL;
12
13     idx = 0;
14
15     while (idx < 8) {
16         DRV_I2S_CID_CTRL_RgRd(varDRV_I2S_CID_CTRL);
17         if (GET_VALUE(varDRV_I2S_CID_CTRL,
18             DRV_I2S_CID_CTRL_I2S_STROBE_LPOS,
19             DRV_I2S_CID_CTRL_I2S_STROBE_HPOS) == 0)
20             DRV_I2S_CTRL_ENB_UdfWr(idx, 1);
21         idx++;
22     }
23
24     DRV_I2S_CID_CTRL_STROBE_TS_UdfWr(1);
25     DRV_I2S_CID_CTRL_STROBE_RS_UdfWr(0);
26     DRV_I2S_CTRL_TFIFO_RST_UdfWr(0);
27     DRV_I2S_CTRL_TSYNC_RST_UdfWr(1);
28
29     return Y_SUCCESS;
30 }

```

Listing 2 Example code generated using VayavyaLabs DDGen

Using register abstraction as described above allows to detach the code from the register and filed addresses and if any register moves in the structure, the code is not affected.

2.3 Test-Driven Firmware Development for Interface IP

Working with a new piece of IP that is usually just being developed as the initial firmware is created, requires a constant closed-loop work mode in which new versions of code can be tested against new versions of RTL. A typical development flow within an interface IP firmware support team could look like this:

- Hardware team designs the IP
- Firmware engineers participate in the design to feed on the register interface design decisions
- Once a register model is designed and a first RTL implementation is in place with a functional bus connectivity, such IP can immediately be integrated into early firmware development.

One of the interesting methodologies to use in firmware development in test-driven development. This could be considered as a formalization of the hardware bring-up process, where some expectation of functionality is always well defined, and the development/bring-up aims at getting that feature enabled/supported.

The steps of test driven development are

- Design a test that fails
- Implement functionality to pass
- Refine for ease of integration.

2.3.1 *Starting Development*

The first cycle of development is typically hardware focused. The initial test is generally a register read/write operation on the address space of the IP to confirm that it has been properly integrated with the system Fig. 2.6.

Once this code is in place (and since it is always the same, it is easily reusable) the preparation of the test platform can start. In current hardware simulation environments, it is possible to use a Virtual Platform Environment (VPE) type environment (diagram) where the CPU and all system peripherals exist as SystemC fast models and the IP under development is being connected through a TLM to RTL wrapper. In a typical use case such connection requires the following steps:

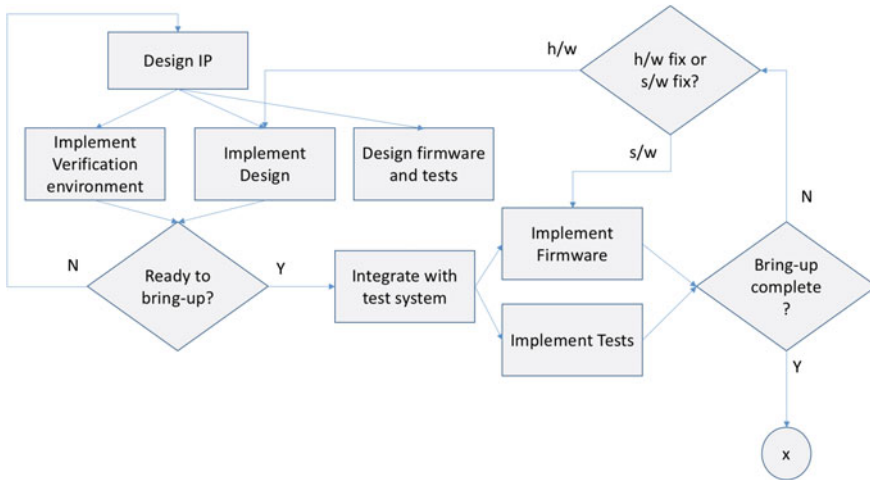


Fig. 2.4 Diagram showing cooperation between RTL (h/w) and C (s/w) teams to design and bring-up an IP block

- Preparation of a system-level wrapper for the new IP
- Integration of that wrapper with the existing system
 - Selecting base address
 - Providing any control signals that are required by that IP but not available on the available slave bus interfaces
 - In the case of interface IPs it is crucial to connect the actual ‘outside world’ interface of the IP (diagram) to a sensible transactor. These can be:
 - An instance of Verification IP
 - An instance of a IP core compatible with the interface.

In an ideal world such integration should be seamless as the IP comes with standard bus interfaces and only requires these connected to the systems. In reality the ‘transactor’ part of the test environment can be the most laborious element of the test environment preparation. In early stages this can be delayed until first contact with IP registers has been made, but this typically is a stage that can be passed very quickly (Fig. 2.4).

Once all is connected and a binary file is loaded into the system, a couple of software/hardware co-debug cycles encompassing:

- checking slave bus ports access,
- checking interrupt generation

lead to a first iteration of a working firmware debug environment.

Example C code of a register read and write operation:

```

1 log(“reading:\n”);
2 v=PAA_UncachedRead32((volatile uint32_t*)(IP_REGS_BASE+
3   offset));
4
5 log(“writing:\n”);
6 PAA_UncachedWrite32((volatile uint32_t*)(IP_REGS_BASE+i), 0
7   xFABEDDIE);

```

Listing 3 Example register read and write code

with bare-metal read and write functions:

```

1 uint32_t PAA_UncachedRead32(volatile uint32_t* address) {
2   return *address;
3 }
4
5 void PAA_UncachedWrite32(volatile uint32_t* address,
6   uint32_t value) {
7   *address = value;
8 }

```

Listing 4 Bare-metal implementation of platform abstraction API.

An example of a register read and write operation on a slave port is presented on Fig. 2.6.

At this stage the firmware debug process typically starts to cross its paths with the hardware verification team. If the IP comes with a machine-readable register description (such as IP-XACT or SysRDL) again it is straightforward to automatically generate system-level test sequences that can be executed immediately on the new design with the benefit of the test code being fully driven by the register design data (Fig. 2.5). This constitutes a very basic set of system sanity test cases such as:

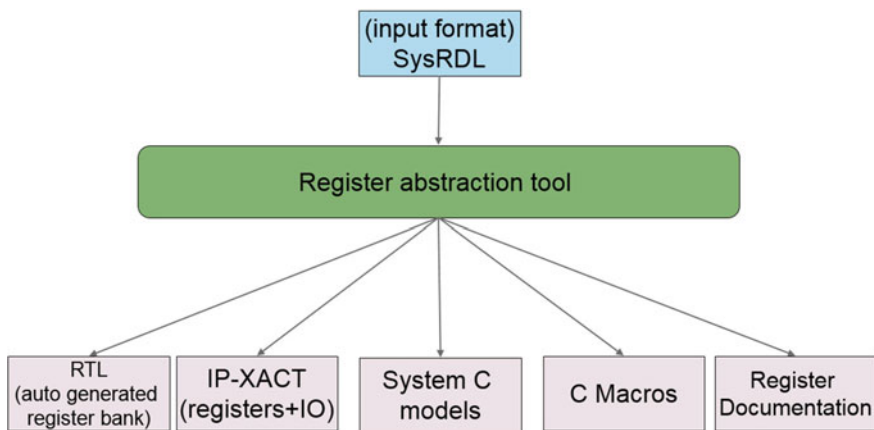


Fig. 2.5 Simplified register abstraction flow for interface IP

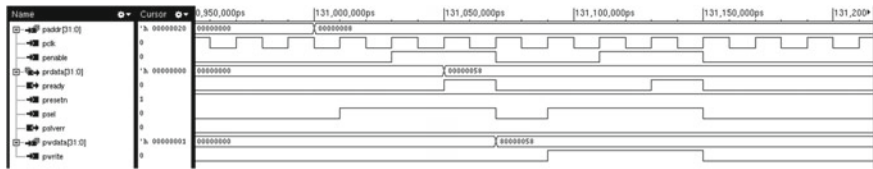


Fig. 2.6 Reading and writing register at address 8 (on APB)

- check register reset value
- confirm register writable bits are writable
- confirm register read only bits are read only.

These are generally covered by the hardware verification team, however the execution of these in a real system allows ironing out issues with bus access and any minor register bank problems.

At this stage a firmware developer can start running any functional test cases on the IP core.

2.3.2 First Functional Tests

In a typical situation of developing a low-level driver for a certain IP there is a number of ‘key features’ that need to be supported to allow for initial system-level integration. These typically are a subset of:

- Initialize IP,
- send data over interface using register interface,
- receive data over interface using register interface,
- use the master bus interface/trigger a DMA,

2.3.2.1 Initializing IP

Once a verification environment is in place, one can set up a simple routine to perform the initialization of the IP and confirm that it has been performed correctly. A test to confirm that may be

- receiving an interrupt that system is ready
- polling a register that system is ready
- in extreme cases it may require to peek into the actual IP signals to see that it has been initialized.

IP initialization may involve setting up elements connected to the IP to establish a working data channel, e.g., PHY initialization. At that stage it is important for all

components of the system to be connected properly and there may be cases where the initialization procedure executed by the firmware in a system environment uncovers a hardware connectivity problem. These integration problems can be easily avoided by using automated IP assembly tools [6]. In these cases it is also required to have a working ‘transactor’ at the other end of the ‘external interface’ to assure that link has been established. If the transactor is another instance of IP supporting the same protocol, it is desirable to have a unit test environment, where each IP can be tested in isolation against a known model (e.g., a Verification IP instance).

2.3.2.2 Sending and Receiving Data

Once the IP is initialized and a data channel is in place, it is possible to start transmitting data to confirm the system connectivity. In the case of interface IP it is required to have a ‘transactor’ instance able to fulfill the communication protocol requirements to assure that the IP can complete the data transmission. Let’s consider an example of a flash memory controller. The aim of the debug step is to confirm that we can create a working code able to write a page of data in the NAND memory by sending the data word by word through the register interface. To achieve this it is important to have the following prerequisites:

- the controller is initialized
- the memory is initialized
- it is required to know the protocol used by the memory controller (in the case of NAND memory it is a sequence of standardized commands (e.g., ONFI) interleaved with the desired data.

Let us consider an example operation where a memory page is written using register interface and later read to confirm consistency. Debugging this code in simulation, allows the following:

- stepping through the firmware code
- looking at the register interface to confirm the right data is sent to the flash controller
- looking at the memory interface to confirm that the right data is sent and received to/from the flash memory (model)
- looking at the register interface to confirm that the right data is read from the memory (model)
- accessing a memory (or having debug messages) to confirm the data integrity.

Once the code has been debugged and proven in simulation, it can be tested on an FPGA setup. Once all basic issues are resolved in simulation, the FPGA platform allows for performance analysis and profiling of the operations using real memory parts.

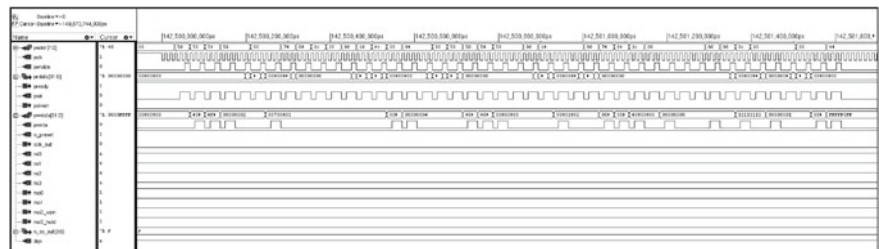


Fig. 2.7 Performing an initialization sequence on the APB bus

2.3.2.3 Testing the DMA

The simulation environment is a great platform to test the DMA transfers and to prototype the firmware to perform these. DMA transfers can show system problems (e.g., with memory caching) and therefore it is good to be able to see the system memory at the time of the transfer being initiated and the flow of data on the master bus interface to confirm the integrity of the data being transmitted. The diagram below shows the same memory write and read as in previous section, but this time a descriptor requesting each operation is built and after the initial trigger, the IP is expected to:

- read the descriptor
- start reading the memory (using a master bus interface)
- start writing the data to the device
- finish reading memory
- finish writing the data
- interrupt the processor to notify the operation has completed
- perform steps similar to above, but in the other direction.

Once a single DMA has been confirmed to work, it may be required to test a chained DMA operation, in which multiple descriptors are fetched (in a sequence, potentially with partial hardware buffering) to assure the firmware is capable of creating and handling these (Fig. 2.7).

Another route may be to test a multi-channel DMA where multiple chains of commands are created and executed by the interface IP on multiple connected devices.

2.3.2.4 Sending and Receiving Data on FPGA

In the case of an FPGA the debug capabilities are significantly reduced, the number of IP signals that can be traced is limited and having checked the code in simulation should significantly reduce the bring-up process. Having an embedded processor in the FPGA connected to the debugger still allows to step through the code, however looking at the interface signals can be difficult and may require the use of an oscilloscope or a logical analyser.

The earlier defined steps (already proven in simulation) can be repeated on FPGA to bring-up the basic functionality of the hardware.

The advantage of FPGA testing is speed and for example in the case of NAND flash testing it was possible to run over 2 orders of magnitude more data (500 MB) than in simulation (16 MB). This also allowed to detect issues with the driver with handling large number of data blocks. Further, due to the fact that testing was performed on real memory it was possible to discover that the memory model used was erased by default, which was not the case for the real memory die, and required an update in the memory erase procedure (Fig. 2.8).

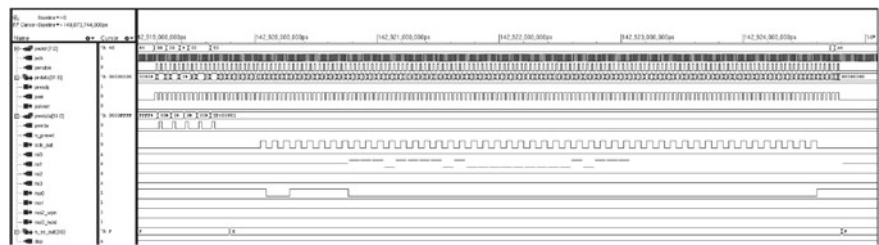


Fig. 2.8 Enabling the interface on a QSPI IP

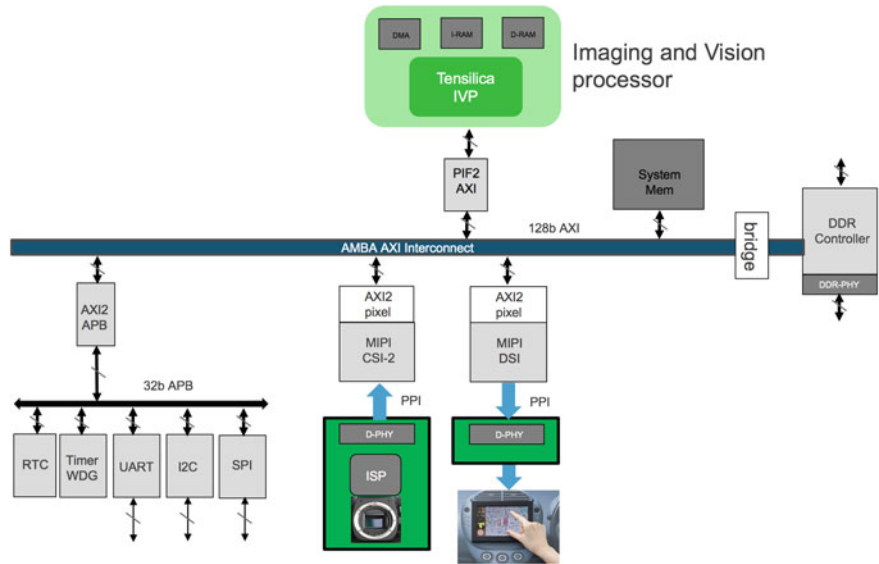


Fig. 2.9 Example system with an Image Processing Unit and a camera/display MIPI interface

2.3.3 *Debugging a System*

Once all components of the system have been brought up (at least in the positive path) it is possible to start debugging the system along with applications that use multiple components at the same time. The assumptions at this point should be:

- there is a debuggable CPU (allowing to step through code)
- all components can perform basic operations
- the CPU and all components have access to memory.

Once these steps are established one can proceed to system bring-up. In most cases such bring-up would start with some basic bare-metal tests, where multiple component would be used at the same time. For example the UART may need to be used as a debug output (or even input) while bringing up a camera interface (MIPI CSI) IP. Additionally it may be required to initialize the camera using another simple interface (e.g., I2C) while using the MIPI interface to obtain images and place them into the system memory. In that case it is important to understand the full system architecture Fig. 2.9 as multiple components will be interacting with each other and the CPU will be interacting with all components.

The debug steps in the example above could follow as:

- create a serial port connection for debug messages (run UART)
- create an I2C connection with camera and log connection results
- initialize camera and log the results
- initialize the MIPI CSI interface
- request the MIPI CSI interface to start transferring data from the camera to the system memory.
- access the memory using a debugger or dump its contents using the UART.

In this example, it would be desirable to have a controlled environment in which the camera would send known data that can be verified (e.g., test mode), whether the correct content has been written into the system memory. Once the system is confirmed to perform all operations in the right sequence, and the right data is copied into memory, it is also required to confirm that the data transfer is well synchronized with the rest of the system, i.e., that once the camera starts sending multiple frames per second, there is sufficient buffering and the frame data is not corrupted. This is a good opportunity to test the interrupts as multiple devices (UART, I2C, MIPI CSI) can trigger an interrupt and it is important to set up the right way of processing interrupts. Example of prioritized interrupt execution is shown in Fig. 2.10.

Interrupt handling should not be affecting the functionality of the system and therefore it may be required to create artificial conditions that trigger multiple interrupts at the same time to debug the scenario (Fig. 2.11).

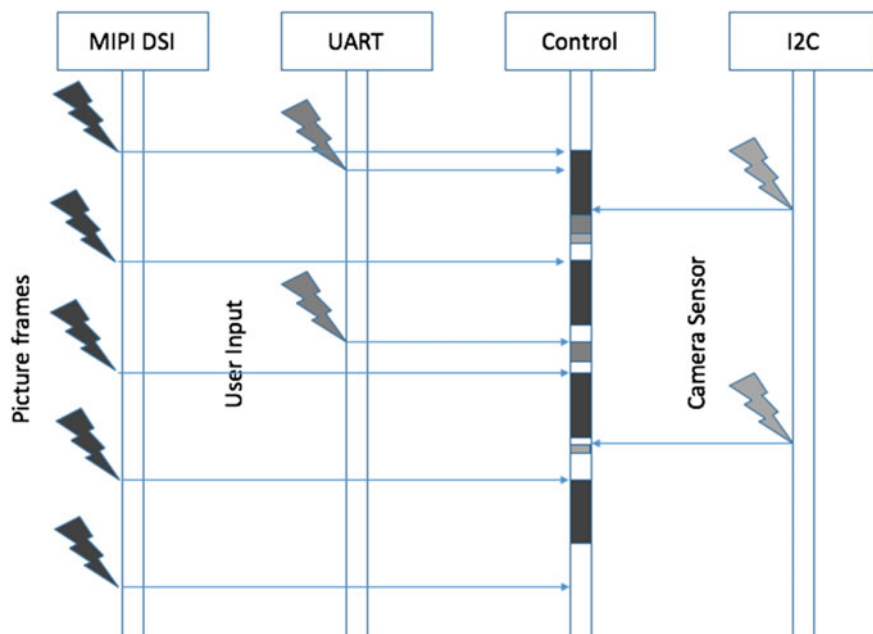


Fig. 2.10 Diagram showing multiple devices generating interrupts at various times. The interrupt priority needs to be set appropriately to assure no interruption in streaming data and user interaction can be compromised in this case, as it is only for set-up and debug purposes

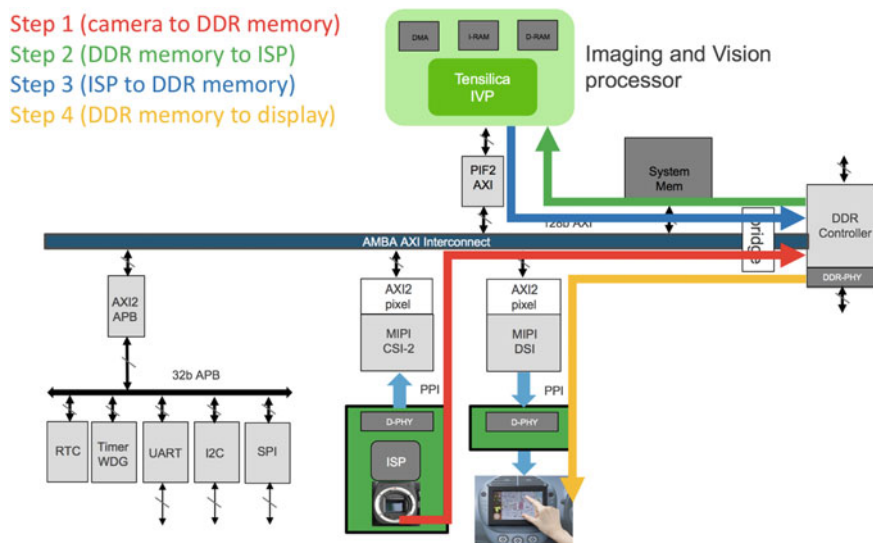


Fig. 2.11 Diagram showing memory transfers (DMA) in the system

2.3.4 *System Performance*

Once all components are working together it is possible to consider the system performance and look at code optimisations to achieve some desired benchmarks. Let's consider the system above with the addition of a screen interface and an image processing routine. In that case the image is

- fetched from the camera and stored into memory,
- read from that memory by an Image/Signal processing component,
- stored back to memory by the Image/Signal processing component
- read from memory by the display component (e.g., MIPI DSI).

Considering a VGA test image is processed in RGB mode the system will need to read 2 MB and write 2 MB of data per image. If that image is delivered at 30 fps, this is 60 MB/s read and 60 MB/s written. One has to consider that the system may need to access the memory at the same time for other purposes (e.g., logging/maintenance/reading program or other data related to image processing. If the system is running on an FPGA it is required to assure the system architecture enables the required bandwidth. And the FPGA platform itself is an ideal vehicle to prove that this can be achieved.

Once the system is performance proven in FPGA, the scalability to ASIC speeds should be much more simple and only minor software optimizations may be required. The issues to be considered here would be:

- bus access
- memory access
- memory bandwidth (bus, controller, module)
- memory caching.

Lab example

“In one of our demo designs, the team was presented with an existing FPGA board where the main (and biggest) FPGA chip did not have a direct connection to the DDR memory. The DDR controller was synthesized on a separate (smaller FPGA) that could be connected to the main one using a chip2chip connection. The system described above (with camera, image processor and display) was implemented on that board and the memory bandwidth turned out to be a limiting factor in terms of image resolution and frames per second. A special design using a Xilinx Chip2 AXI construct was used to achieve best performance. This needed further optimizations both on hardware and software level. On the hardware side the important tuning factors were:

- chip2chip clock speed (and synchronous/asynchronous implementation versus the system bus)
- image processor cache size
- image processor cache line size (tuned with the DDR line size).

On the software side it was important to limit memory access and assure that the image blocks (in each on the four data paths) would be sent using a ‘chained’ sequence of DMA operations to best utilize the fabric bandwidth.”

2.3.5 Interface IP Performance in a Full Featured OS Case

Another interesting example could be the Ethernet MAC implementation used as a NIC (network interface card) with a PCIe endpoint. Both cores are connected together and plugged into a PC (diagram below). There are two mirroring set-ups with:

- emulation (using a PCIe SpeedBridge[®] Adapter and Ethernet SpeedBridge[®] Adapter)
- FPGA with standard PCIe and Ethernet connectors.

Either of these implementations can be connected to a lab PC running Linux OS and a driver implemented for the interface IP as a PCIe device in Linux can be performance tested and debugged. (The same set-up is used for compliance testing).

In a particular case a feature was added to the Ethernet controller to allow offloading the software TCP/IP stack from performing ‘coalescing’ of received packets into one large packet. Support for this feature was added to the Linux driver and both implementations of the NIC platform were used to test the functionality in a system environment. The FPGA implementation was used to benchmark the CPU savings due to the offload feature and while running these a potential problem with the offloads was discovered. The following steps were used to track the problem with FPGA):

- Additional logging in the Linux driver
- Enabling additional kernel logging in Linux
- Using a TCP sniffer (wireshark) to confirm data transfer.

However all these methods did not allow to find the cause of the problem, and therefore the problem needed to be replicated in an emulation environment for full understanding. Moreover, it turned out that some of the problems are inconsistent from one PC to another. After replicating the problem in an emulation environment with full access to all hardware design signals it turned out that:

- The PCIe root-port (host) may reorder packets
- Some PCIe bus parameters influence the occurrence of this issue
- Further to all these, an actual hardware issue was detected with further refinement of the test, that was not related to the PCIe bus.

In the case of performing such lab tests, it is very important to have a very strict control over the test environment, to assure no random events or changes in the set-up cause the system to behave in a different way. The best practice is to set up a clean sandbox with a fully scripted path through the tests and a full control over external events (e.g., avoid connecting to live local networks).

2.3.6 *Low Level Firmware Debug in a State-of-the-Art Embedded System*

While bringing up a new SoC system in the lab used for compliance testing purposes, the hardware designers created a system consisting of a 64-bit CPU core and a contemporary interrupt controller, both used to test the Message Signal Interrupt (MSI) capability of an embedded xHCI controller 2.1 with xHCI controller as IP and a USB SuperSpeed device VIP instance. The MSI capability is typical and well supported for PCIe-based devices, however with the introduction of new interrupt controllers it is possible to receive these interrupts in embedded systems.

In the case of an embedded MSI interrupt the memory location is hijacked by the interrupt controller to generate an interrupt signal for the CPU. This has required to set the correct configuration of the bus (including protected access line) to work with the interrupt controller and allow devices on the bus to access the given memory location to generate an interrupt. An example device supporting this feature is an xHCI interface core. The Linux driver supports the MSI mode and with the full system running in the simulation environment it was possible to confirm that the IP generates MSI interrupts. After appropriate configuration of addresses, switching the xHCI core into the MSI mode and configuring the GIC to receive these messages.

2.4 Firmware Bring-up as a Hardware Verification Tool

With the current possibility to bring-up an IP quickly in a simulation/emulation in a full system environment, it is possible to use off-the-shelf system tests for early verification of newly developed IP.

This has been applied to two different cores recently in our lab setting.

2.4.1 *NAND Flash*

An MTD (Memory Technology Device) driver was created for a new NAND flash controller core to enable the usage of MTD-test framework in Linux to perform multiple (existing) system test on an IP that was still in development.

List of MTD tests available in the community¹

- **mtd_speedtest**: measures and reports read/write/erase speed of the MTD device.
- **mtd_stresstest**: performs random read/write/erase operations and validates the MTD device I/O capabilities.

¹<http://www.linux-mtd.infradead.org/doc/general.html>.

- **mtd_readtest**: this tests reads whole MTD device, one NAND page at a time including OOB (or 512 bytes at a time in case of flashes like NOR) and checks that reading works properly.
- **mtd_pagetest**: relevant only for NAND flashes, tests NAND page writing and reading in different sizes and order; this test was originally developed for testing the OneNAND driver, so it might be a little OneNAND-oriented, but must work on any NAND flash.
- **mtd_oobtest**: relevant only for NAND flashes, tests that the OOB area I/O works properly by writing data to different offsets and verifying it.
- **mtd_subpagetest**: relevant only for NAND flashes, tests I/O.
- **mtd_torturetest**: this test is designed to wear out flash eraseblocks. It repeatedly writes and erases the same group of eraseblocks until an I/O error happens, so be careful! The test supports a number of options (see `modinfo mtd_torturetest`) which allow you to set the amount of eraseblocks to torture and how the torturing is done. You may limit the amount of torturing cycles using the `cycles_count` module parameter. It may be very good idea to run this test for some time and validate your flash driver and HW, providing you have a spare device. For example, we caught a rather rare and nasty DMA issues on an OMAP2 board with OneNAND flash, just by running this tests for few hours.
- **mtd_nandecctest**: a simple test that checks correctness of the built-in software ECC for 256 and 512-byte buffers; this test is not driver-specific, but tests general NAND support code.

This allowed finding a number of issues with the Linux driver, as it was a ready set of ‘tested tests’ so a perfect environment for driver development with a well defined acceptance criteria (passing all tests).

The MTD framework allows to run file-systems (such as JFFS2) on top of the existing stack, giving further system and stress testing capabilities.

2.4.2 *xHCI*

In the case of an IP with a standard register interface, the Linux itself can be used as a first stage of system testing as soon as the IP can be integrated into a simulation/emulation environment. Furthermore, the USB-test Linux framework was used to perform multiple system tests to confirm the system-level stability of an IP (that was at the same time going through design updates and standard hardware verification process).

```

1 TEST 1:  write length bytes iteration times
2 TEST 2:  read length bytes iteration times
3 TEST 3:  write/length i*vary..length bytes iteration times
4 TEST 4:  read/length i*vary..length bytes iteration times
5 Queued bulk I/O tests
6 TEST 5:  write iteration sglists sglen entries of length
7         bytes

```

```

8 TEST 6: read iteration sglsts sglen entries of length
9     bytes
10 TEST 7: write/vary iteration sglsts sglen entries i*vary..
11     length bytes
12 TEST 8: read/vary iteration sglsts sglen entries i*vary..
13     length bytes
14 Non-queued sanity tests for control (chapter 9 subset)
15 TEST 9: ch9 (subset) control tests, iteration times
16 Queued control messaging
17 TEST 10: queue sglen control calls, iteration times
18 Simple non-queued unlinks (ring with one urb)
19 TEST 11: unlink iteration reads of length
20 TEST 12: unlink iteration writes of length
21 EP halt tests
22 TEST 13: set/clear iteration halts
23 Control write tests
24 TEST 14: iteration ep0out, 0..length vary vary
25 Iso write tests
26 TEST 15: write iteration iso, sglen entries of length bytes
27 Iso read tests
28 TEST 16: read iteration iso, sglen entries of length bytes
29 Tests for bulk I/O using DMA mapping by core and odd address
30 TEST 17: write odd addr length bytes iterations times core
31     map
32 TEST 18: read odd addr length bytes iterations times core
33     map
34 Tests for bulk I/O using premapped coherent buffer and odd
35     address
36 TEST 19: write odd addr length bytes iterations times
37     premapped
38 TEST 20: read odd addr length bytes iterations times
39     premapped
40 Control write tests with unaligned buffer
41 TEST 21: iterations ep0out add addr, 110..length vary vary
42 Unaligned iso tests
43 TEST 22: write iterations iso odd, sglen entries of length
44     bytes
45 TEST 23: read iterations iso odd, sglen entries of length
46     bytes
47 Unlink URBs from a bulk-OUT queue
48 TEST 24: unlink from iteration queues of sglen length-byte
49     writes
50 Simple non-queued interrupt I/O tests
51 TEST 25: write length bytes iterations times
52 TEST 26: read length bytes iterations times
53 Simple Bulk performance test
54 TEST 27: bulk write iterations*sglen*length /(1024*1204)
55     Mbytes
56 TEST 28: bulk read iterations*sglen*length /(1024*1204)
57     Mbytes

```

Listing 5 Non-queued bulk I/O tests from <http://www.linux-usb.org/usbtest/>

where:

- iterations—count iterations
- length—size of packet
- sglen—scatter/gather entries of
- vary—vary packet size by.
- i—current iteration

These tests allowed to find a number of issues with the driver, in particular:

- stall support
- device configuration.

The test suite was also used as a stress test for the IP using several system configurations (32/64 bit, legacy/MSI interrupt) allowing to assure system quality of the driver.

2.5 Playback Debugging with Cadence® Indago™ Embedded Software Debugger

Digital systems get more and more complex in time. The requirements for new functionality while preserving backwards compatibility makes each new system release more complicated than the previous one. To fully verify such a project, a tool is required that allows firmware and hardware co-verification. Hardware tests (whether UVM or functional/directed) may not suffice, in order to create more realistic use cases firmware driven test cases are required. This allows building a complete use case into a single test case, e.g., adding a new sound channel to a SoundWire device requires reconfiguring the IP and running all transmissions simultaneously. Therefore it is important to have a system with a microcontroller available as part of the verification platform.

On the other hand, increasing competition on the market enforces putting more consideration to the cost of designing new versions of devices. In particular the time-to-market perspective is very important. It is key to assure that the work is not held up by unexpected errors that are difficult to analyze. The occurrence of such issues can lead to project delays and introduction of ‘crunch time’—both significantly affecting the team and potentially leading to further problems. There is a lot of cases where the error is observable after it occurs and it is easy to set a breakpoint at the later execution time to see its effects, but it may be difficult to track the root cause (such as—unexpected input value or a bug in the code). In this case it is very important to be able to observe the past of the system (hardware and software including memory contents) to trace the actual root cause occurrence.

Indago™ Embedded Software Debugger is a solution allowing to debug the entire system (both hardware and software). It is a complete solution in the way that using a single tool allows to access all components of the project. It allows easy analysis

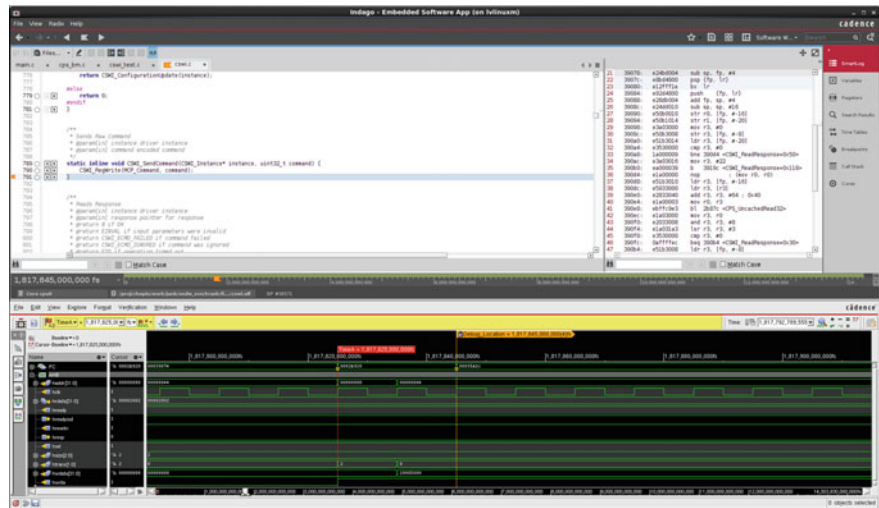


Fig. 2.13 Main Interface of Indago™ embedded software debugger application

- SoundWire Slave
- VIP:
 - SoundWire Slave

In order to use playback debugging with Indago™ Embedded Software Debugger, the ELF file and information about software execution flow are required. It can be either:

- Trace text file which allows to use all features of Indago™ Embedded Software Debugger,
- PC Counter trace on a waveform, which only provides basic debugging functionality.

A compatible waveform database containing information about hardware signals can also be loaded to the application, which allows hardware/software co-debugging.

Figure 2.13 shows the main interface of the Indago™ Embedded Software Debugger which contains two major parts, code debugger, and waveform window which can show information about both software and hardware.

This particular example shows the write operation to the hardware. CPU writes data to the FIFO which is shown at the hardware level as AHB write transaction.

The offline debugging feature allows to move forward and backward in code execution and setting unlimited breakpoints. This is especially useful when code fails in an interrupt handler. Indago™ Embedded Software Debugger allows to jump back to the last execution and see what went wrong without stepping through successful runs. Figure 2.14 presents an example function which was called by an interrupt handler. Call Stack and waveform can be used to quickly identify why it was called.

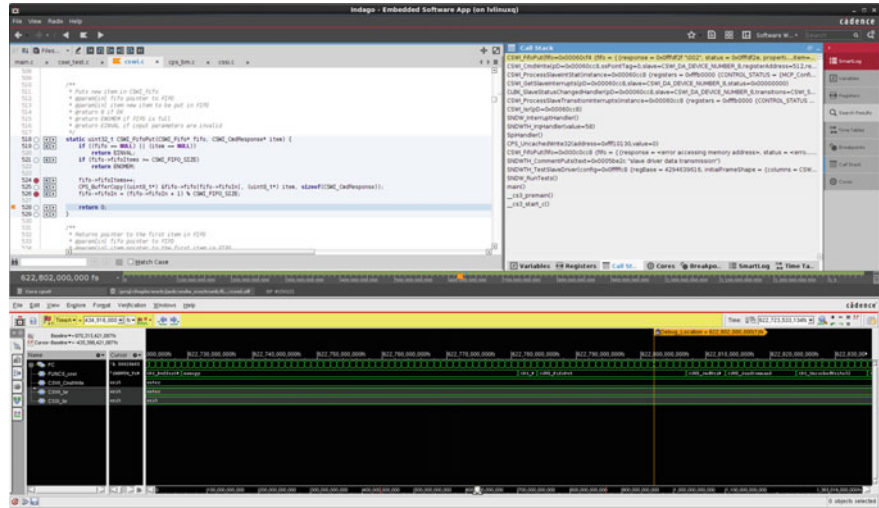


Fig. 2.14 Debugging of code executed during one of interrupts

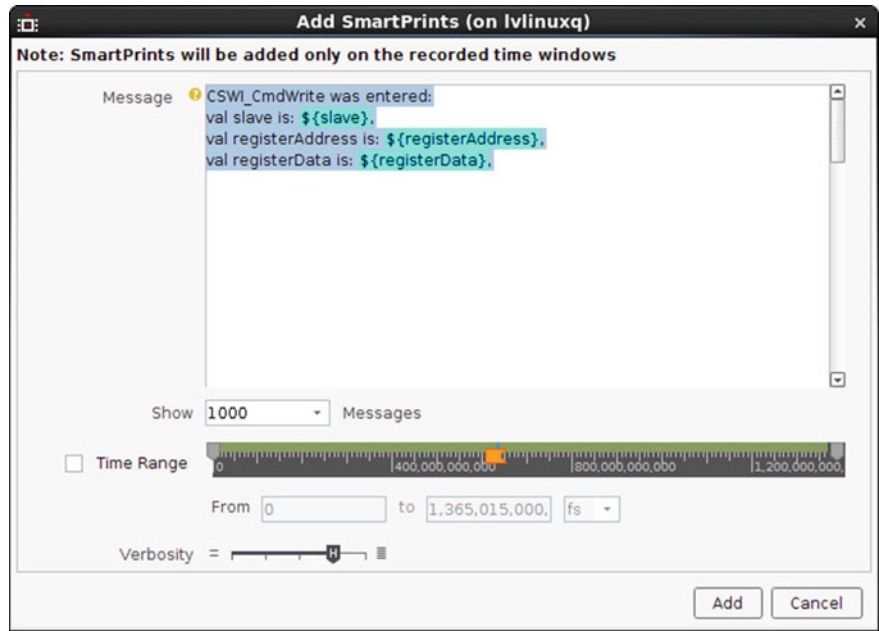


Fig. 2.15 SmartPrint settings

Offline debugging allows nonintrusive debugging methods. Using prints to debug code is quite common, but it affects code execution timing, so problematic code may start working after adding some prints in the code. Indago™ Embedded Software Debugger supports SmartPrints. In any line of code which was executed it is possible

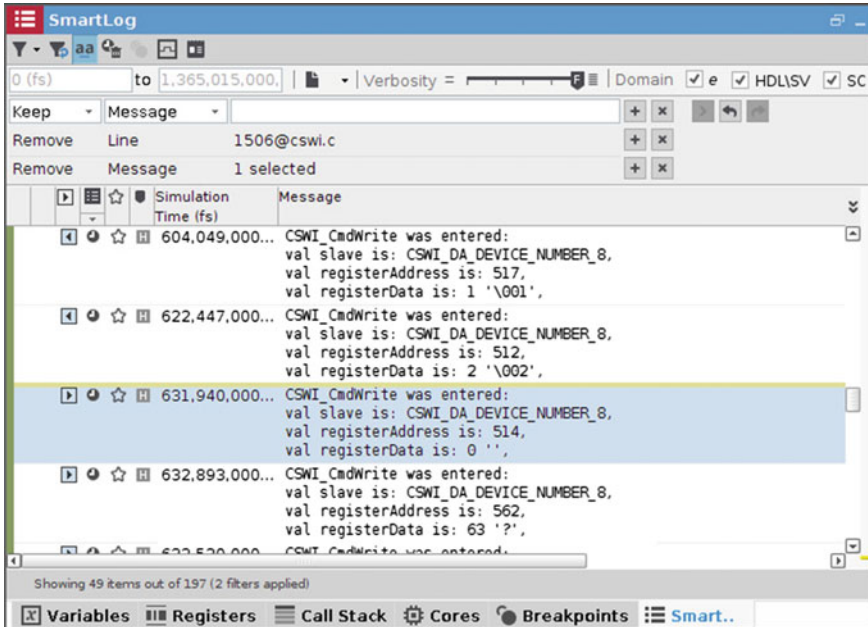


Fig. 2.16 SmartLog window

to add print message with custom information and available variables, as shown in Figs. 2.15 and 2.16. Because it uses offline database of software execution, it does not add any delays, which may affect the execution especially during interrupts. It also makes debugging much easier.

2.5.2 Coverage Measurement

IndagoTM Embedded Software Debugger along with Incisive[®] Enterprise Simulator Metrics Center allows for Code Coverage measurement.

Figure 2.17 shows the main window of the application. It provides a quick insight into code testing quality, and helps to identify and fix issues in the code verification. It is also possible to view code source code with indication which line was executed as shown in Fig. 2.18. Results from multiple test runs can be merged together.

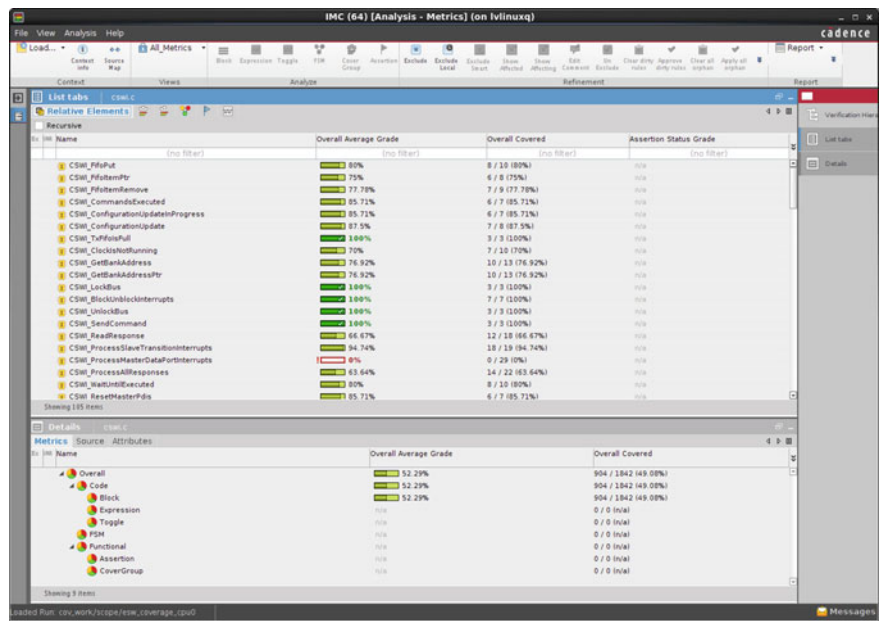


Fig. 2.17 Incisive® Enterprise simulator metrics center presenting example code coverage results

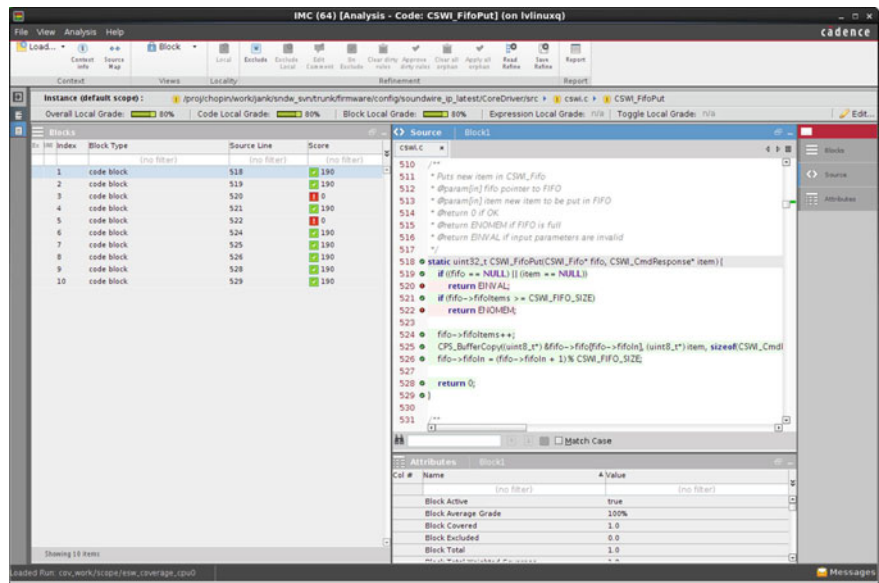


Fig. 2.18 Example function source coverage view

Table 2.1 Comparison of Cadence® runtime and playback debuggers

	Positives	Negatives
Indago™ embedded software debugger	Debug without affecting runtime. Unlimited rerun of prerecorded sequence. Forward and backward playback. Coverage measurement. SmartPrints	Cannot modify values during debug (need to rerun full simulation). Large recording file (need to provision disk space)
Virtual platform	Ability to modify variables/registers during debug. Much lower disk space requirements	Difficult debugging interrupts. Need to rerun simulation to step back. No coverage measurement

2.5.3 Drawbacks

The limitation is that modifications cannot be tested *in situ*. Therefore, it is important to have access to multiple debug tools to be able to reevaluate the new code as the recording of a new system run can be time consuming.

2.6 Conclusions

In this chapter, multiple examples of embedded systems with different debug approaches have been presented. The recent development in the EDA tools and the constant growth of computer speed and capacity allows to simulate or emulate very complex system on chip- type platforms with immediate access to both software debug symbols and stepping through code and tracing of the hardware signals, to assure easy bring-up of new devices (while being designed) in system environments and test against both industry standard verification IP or connectivity with real world devices. This in turn allows shortening the time to market for new devices as the software developers can start working on the drivers and system software as soon as the IP is *reasonably* stable or as soon as a TLM model is available.

With the new **playback** approach to system debug it is also possible to trace issues in the system without affecting the runtime. Table 2.1 shows a brief comparison of both techniques.

The most important elements in the embedded software debug are

- Platform that is set-up closest to the target system.
- Ability ot start working with the device or its model as early as possible in the development process.
- Cooperation of hardware and software engineers (especially in the early bring-up stages).
- The appropriate choice of tools.

Acknowledgements The authors would like to thank Rafał Ciepiela for his input into the contents and review of the chapter.

References

1. Cadence® (2016) Cadence Incisive® Enterprise simulator. Product website. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/simulation-and-testbench-verification/incisive-enterprise-simulator.html
2. Synopsys (2016) Synopsys virtualizer. Product website. <http://www.synopsys.com/Prototyping/VirtualPrototyping/Pages/virtualizer.aspx>
3. ARM (2016) ARM cpu fast models. Product website. <http://www.arm.com/products/tools/models/fast-models.php>
4. Cadence® (2016) Cadence Palladium® Z1 emulator. Product website. https://www.cadence.com/content/cadence-www/global/en_US/home/tools/system-design-and-verification/acceleration-and-emulation/palladium-z1.html
5. Vayavya Labs (2016) Vayavya device driver generator. Product website. <http://vayavyalabs.com/technology/ddgen/>
6. ARM (2016) ARM socrates DE tool. Product website. <https://www.arm.com/products/system-ip/ip-tooling/socrates-design-environment.php>

Embedded Software Verification and Debugging

Lettnin, D.; Winterholer, M. (Eds.)

2017, XVI, 208 p. 80 illus., 62 illus. in color., Hardcover

ISBN: 978-1-4614-2265-5