Hybrid Approach to Testbench and Software Driven Verification on Emulation

Debdutta Bhattacharya[1] Ayub Khan[2]

[1] debdutta_bhattacharya@mentor.com, Mentor, A Siemens Business, 46871 Bayside Pkwy, Fremont, CA-94538
Contact: 510-354-4841
[2]ayub_khan@mentor.com, Mentor, A Siemens Business, 46871 Bayside Pkwy, Fremont, CA-94538
Contact: 510-354-6748

Abstract **–** Configurable System-on-Chip (SoC) designs used to power Software Defined Networks and Datacenters have transitioned from fixed function topologies to highly configurable software-driven devices. With explosive growth of cloud services, ASIC vendors must verify chips that will work with a multitude of software profiles in such environments. They have used emulation as a way to start pre-silicon software development. However, it happens only after full software profile is available and typically after tape-out. We look at a methodology to run mixed software application based and testbench based profiles using a combination of software driver/application along with testbench driven verification for RTL. This could shorten the chip schedule and enhance the quality of the chip.

I. Introduction:

With shrinking time-to-market, ASIC vendors face the daunting task of verifying their ASICs that would work with a variety of applications with different software profiles. As an example, with the growth of cloud computing and services, network switch vendors are increasingly focused on Software Defined Networks (SDNs) to enable organizations accelerate application deployment and delivery and such ASICs must be verified against a host of software applications and profiles during the development process.

Traditional validation techniques that rely on silicon samples not only prolong time-to-market but also can result in expensive re-spins. ASIC vendors have successfully used emulation as a way to start software development before the silicon comes back. Typically in such applications, full software stack can be run on a virtual machine like Quick EMUlator (QEMU) with a Guest OS which interacts with synthesized ASIC running on the emulator via emulator host links. The communication can follow a protocol like the PCIe protocol. This enables driver and software development on emulation with real RTL as opposed to white models. Such a system also enables SoC debug with full RTL visibility as opposed to very limited visibility in post-Silicon lab software bringup.

In contrast to software based verification of SoCs, testbench driven verification is more common in early stages of the SoC development and verification cycle [1]. Abstractions of the software, often static software profiles are created and used in a verification setup with C/C++/SystemC or SystemVerilog testbenches which provide stimulus to the ASIC in form of configuration or traffic. Testbenches are written as abstractions for most parts of software functionality including performing resets, configuration of various blocks, driving traffic for verification and doing end-of-test checks. However, many software modules are often complex and require considerable effort to develop for each project and maintain till software based verification can take over later in the development cycle. Examples of this may include multi-threaded environments doing Dynamic Memory Accesses (DMA), maintaining their own queues and descriptors for sending and receiving packets etc.

In this paper, we look at a method which combines the strengths of both the above approaches while addressing their limitations. Software driven verification can happen only when software is ready or close to being complete and hence typically happens close-to or after tape-out. Testbench driven verification which is used to verify the RTL pre-tape-out risks leaving gaps in verification due to inability to model complex software scenarios using testbenches. We explore a methodology to run mixed profiles, namely software based, and testbench based profiles using a combination of software stack and testbench driven verification for RTL. This could shorten the chip schedule by months and result in better silicon quality.

II. Testbench Driven Verification in Emulation

Testbench based verification is one of the verification modes that enables SoC verification on a hardware emulator. The testbench runs on a server connected to the hardware emulator, and co-modeling software helps it communicate with synthesized SoC running on the Hardware emulator. Among other modes of verification on a hardware emulator is In-Circuit Emulation (ICE), where physical targets are connected to the hardware emulator for verification. Testbenches can also be fully synthesized along with the Design-Under-Verification and run on the hardware emulator. Of the three modes discussed, virtual testbench based verification provides the maximum flexibility in terms of running a variety of tests by changing testbench stimulus, altering stimulus to find specific bugs and uploading waves for debug based on signal triggers or time points.

Virtual testbench based verification again can be run in co-simulation mode, where design signal level activity can be accessed or driven by the testbench, or in transaction based mode where the testbench communicates with the synthesized DUT only via transactions. Transaction based communication between such a virtual testbench and the hardware emulator often provides 50x-1000x speed-up as compared to simulation verification flows. For the purposes of this paper, transaction based acceleration on emulation is discussed which provides the most acceleration along with flexibility in run and debug.

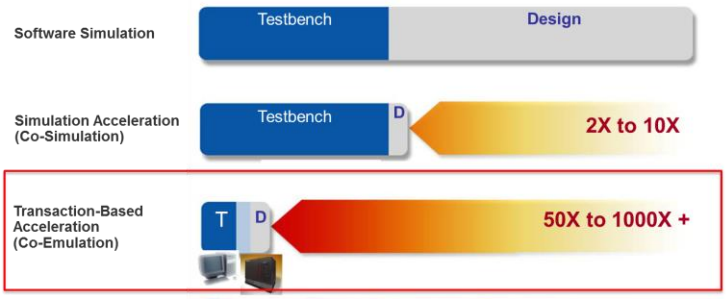

Figure 1. Comparison between Software simulation,
Simulation Acceleration and Transaction-based Emulation

The testbench can be written in C, C++, SystemC or SystemVerilog and can run as a standalone executable or in a simulator. Testbenches, combined with comodelling software libraries can communicate with the Design-under-test (DUT) running on the hardware emulator via transaction interfaces. Stimulus used to verify the DUT is applied through the testbench.

In a virtual verification environment, virtual devices can be used to verify peripherals connected to the synthesized SoC. Virtual devices are architected as a software stack running on a comodel-host server connected to the emulator and interacting via a communication-protocol IP with the design running on the emulator. As an example, virtual Ethernet Traffic generators can be connected to ethernet ports of a synthesized DUT being emulated, and traffic driven through them. As opposed to ICE targets connected to the emulator, the virtual verification methodology eliminates dependence on physical targets enabling multiple users run their tests simultaneously and enables flexible debugging (stopping clocks, peeking into register values, uploading waveforms as needed) by utilizing capabilities of the comodeling software running on the comodel-host server.

In this paper, we focus on virtual testbenches used to configure a synthesized DUT running on the emulator communicating via a specific protocol (eg. PCIe/USB). Applications such as Software Defined Networking require a SoC to work with a multitude of software profiles. Such an SoC is configured on the fly by software stack running on a processor interacting with the DUT following a communication protocol. Such a system enables changing switching and routing conditions in real-time and provides flexibility to change speeds to meet network requirements among others. Other applications can include enabling or disabling firewall rules at runtime, communicating with Network Attached Storage (NAS) for storing data based on incoming traffic etc. A full software stack communicating with, and configuring the SoC is used to achieve this.

Verifying such an SoC in pre-silicon comes with a host of challenges since the software is not ready early in the design cycle. Software profiles must then, be abstracted and used for chip charaterization in pre-silicon. The host of software profiles can be huge depending on the capabilities of the SoC being verified, but a virtual verification methodology can be successfully utilized to verify such an SoC in pre-Silicon.

An example SystemC based testbench is shown in Figure 2. The components of the testbench shown are representative of a testbench used to verify SDN switches discussed above. The main functions of such a testbench must include sequences for powering up the SoC, manage initialization based on software profiles, launch a virtual traffic generator like the Ethernet Packet Generator and Monitor (EPGM) to control traffic on various ports of the chip, manage DMA traffic to-and-from the chip, initiate end checks to verify results of traffic testing and finally managing a contextual database of the entire verification process to cover all software profiles and speeds.

Figure 2. Components of a sample SystemC testbench used to verify SDN switches

This verification methodology must include static software profiles for initializing the chip for a particular kind of traffic or stimulus, and the other controllers managing stimulus to-and-from the chip are configured to verify the chip functionality for said profile. This methodology is used extensively for chip characterization and ensuring verification coverage. Such a testbench mimics the functionality of real applications running on a processor connected to the SoC in post-silicon, and each software profile is an abstraction of a set of configurations driven by the software.

However, certain software features can be very complex to model in simplistic testbenches. One such example is Direct Memory Access (DMA). DMA enables CPU concurrency and boosts overall system performance by handling large memory operations through a DMA engine. DMA engines are often multi-threaded, handling multiple descriptors to write to or read large chunks of memory at a time. End-point intiated DMA further reduces load on CPU which only interacts with a local memory and transport latencies in read requests are avoided. The steps required for performing a memory read or write operation is shown.

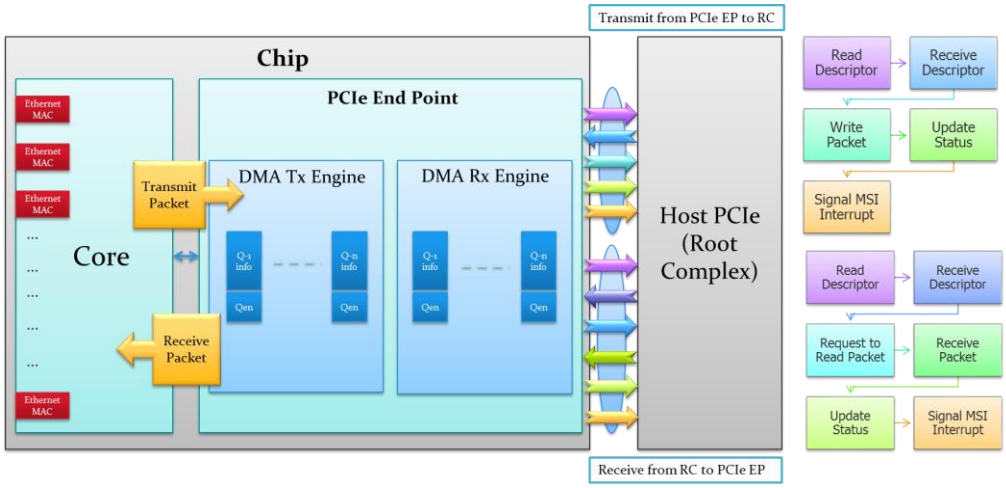

Figure 3. Steps in end-point initiated DMA.
For SDN switches, DMA is used to read and write packets to the CPU

As is evident, each read and write consists of multiple steps and in cases where multiple descriptors and multiple queues need to be supported, creating a multi-threaded system with shared memory to handle DMA in pre-silicon SoC verification is a complicated task. Real-life applications running on a software stack on a processor are well equipped to handle complex DMA transactions which is difficult to model in testbench based verification.

Thus, testbench based verification is well suited to chip characterization using an extensive set of verification profiles, but certain applications like DMA are hard to verify using this methodology. Reproducing validation tests from real-life applications on such a setup is either not possible or is complex to do. Emulation enables software based verification, discussed in the next section which deals with the drawbacks of testbench based verification discussed above.

III. Software Driven Verification in Emulation

Software verification in pre-silicon is enabled by hardware emulation through the use of virtual verification platforms. Each platform corresponding to a particular protocol or software is structured as a verilog hardware component connected to, and synthesized with the DUT, a testbench component which integrates with the user testbench, and a virtual interface which is similar to the platform that the SoC DUT would be verified with in post-silicon validation in a lab. Software based verification is not possible in simulation due to lower speed in simulation as compared to emulation, since real-life stimulus is applied to the DUT via software applications which often consists of millions of vectors. Simulation is not well equipped to handle such deep tests which emulation enables.

As an example, virtual ethernet traffic generators can be used to connect to a network switch and verify functionality for mesh or snake traffic by passing hundreds of millions of packets. Traffic generators can be connected to each port of the SoC for various network speeds and highly configurable static or constrained random traffic can be generated and applied to the SoC. Ethernet Packet Generator and Monitor (EPGM) is an example of such an application shown in Figure 4.

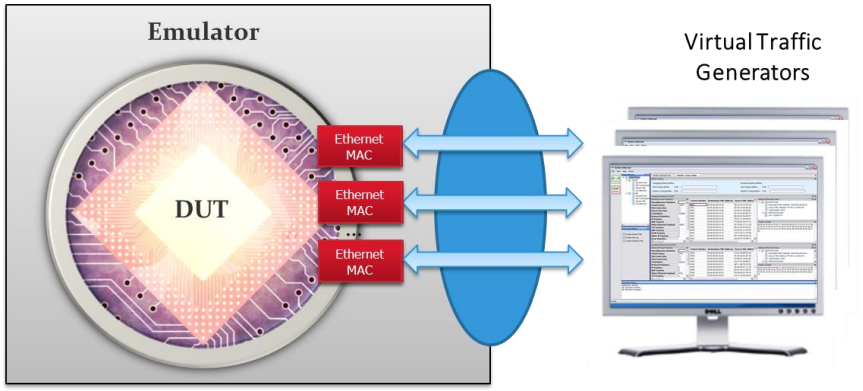

Figure 4. EPGM Virtual Traffic Generators used to verify a network switch SoC

In the case of SoCs which are configured by Software stack running on a Guest OS on a processor, the virtual platform provides a method to run the same software application that would be used to verify the SoC in post-silicon. The platform consists of a Guest OS running in an open source CPU emulator like Quick EMUlator (QEMU). QEMU runs on a host server interacting over a communication protocol like PCIe with the synthesized DUT running in the hardware emulator through the comodelling infrastructure. This unleashes the power of post-silicon lab-like development and debug in pre-silicon. Software applications can be verified with real RTL of the SoC running on the hardware emulator. Virtual methodology enables multiple users to verify, build and run their applications in parallel with designs running on the emulator as opposed to reliance on actual hardware in the lab. One such application of this methodology is its usage in SDN switch verification on emulation [2].

QEMU can run Guest OS such as Fedora or Ubuntu which enables a variety of applications that can be used to run and configure the SoC. Underlying protocol transactions are captured at the host controller inside QEMU (eg. PCIe or USB) and through comodelling infrastructure consisting of a systemC proxy, hardware Transactor connected to the DUT. VirtuaLAB Virtual PCIe platform is shown in Figure 5.

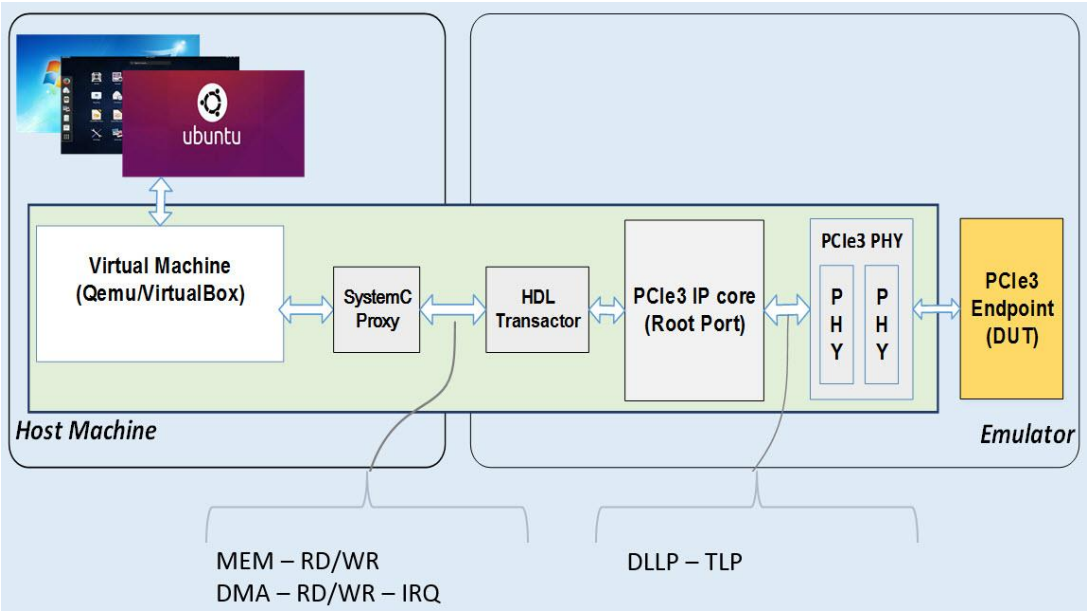

Figure 5. QEMU front-end with underlying comodelling PCIe infrastructure for HW emulation

Advanced debug infrastructure can be enabled like protocol tracers to monitor transactions between the virtual platform and the SoC. As an example, in case of Virtual PCIe device, Protocol Analyzer can monitor transactions, TLP/DLLP, link state, monitor traffic statistics, aid packet decoding on a per-packet basis etc. Transactions can also be recorded and post-processed for debug. This is shown in Figure 6. Such infrastructure can help flush out bugs issues in conjunction with kernel and software debugging tools.

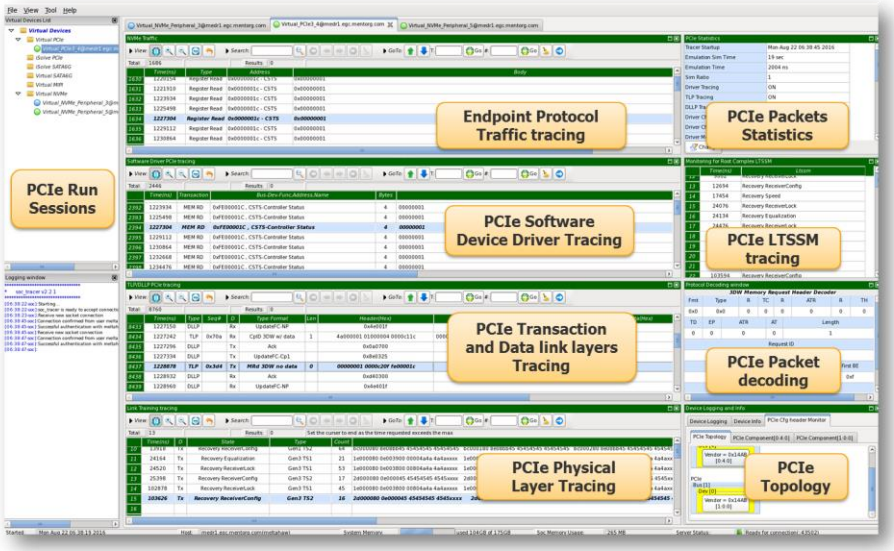

Figure 6. Protocol Tracer application to monitor PCIe link in Virtual PCIe

Advantage of such virtual platform based verification is that the full capabilities of SW stack can be leveraged and complex software can be written and run in Guest OS running in QEMU with the communication infrastructure (the PCIe host controller, comodelling software and PCIe interface on the chip) handling the transaction transfer to and from the software application(s). Additionally this setup enables simulatenous verification of hardware and software by enabling the full debug capabilities of a hardware emulator like waveform upload and viewing, enabling triggers to capture bugs, streaming live waveforms during tests in addition to software debug using gdb or kernel messages in QEMU (Figure 7).

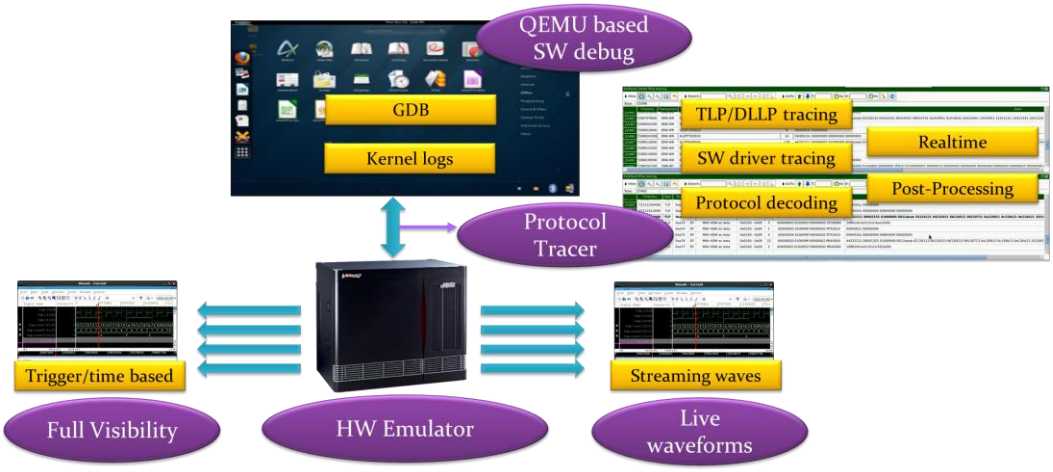

Figure 7. Concurrent HW and SW debug using virtual PCIe platform on emulation

Referring to the DMA example mentioned in the above section, a software stack enables multi-threaded, multi-queue, multi-descriptor DMA application to be written and executed in the Guest OS which acts as the root complex, is responsible for host memory management, interuppt polling etc.

Despite the obvious advantages of using a full software based validation flow in emulation, it is to be kept in mind that software readiness comes only late in the design cycle once the full software stack is ready, and the SoC can be fully configured based on a certain software profile for a particular scenario. Thus, even though emulation enables software development start in pre-silicon phase, this happens close to software and design readiness very close to tape-out. Prior to this, whitemodels are used to develop the software which are abstractions of the RTL.

IV. Hybrid Approach to Testbench and Software Driven Verification on Emulation

A hybrid method for testbench based and software based verification in emulation is proposed. One part of the setup consists of single or multiple threads of a testbench communicating to a synthesized DUT running on an emulator (Fig. 1). As mentioned in Section 1, the testbench can be written in C, C++, SystemVerilog or SystemC, and may run on a simulator or as an executable. This testbench comprises, among other components, infrastructure for driving static configurations to the DUT for verification. These configurations are abstractions of software and often based on common configuration profiles shared by RTL verification and software development teams. The testbench runs on a host machine on the network connected to the emulator. The second part of the setup consists of a virtual machine like QEMU, running a Guest OS in turn running software drivers and applications. This virtual machine also runs on the host network connected to the emulator and the software stack drives configuration to configure the DUT synthesized and running on the emulator.

Both parts of the setup communicate with the emulator via comodelling infrastructure. In this fashion, the testbench running on the host machine can drive configuration and stimulus to the DUT on the emulator simultaneously with the software running on the virtual machine.

Figure 8. Hybrid setup for Testbench and Software Driven Verification in Emulation

A sample infrastructure for the proposed method is discussed below. The testbench is written in SystemC and the communication protocol is PCIe. The portion of the setup which is synthesized and runs on the emulator consists of the DUT, and a synthesizeable emulation-friendly PCIe Root Complex Transactor connected to the DUT via null-PHY layer. The PIPE connection between the PCIe root complex Transactor and DUT are shown in Table 1.

An accelerated transaction based infrastructure handles the communication between the synthesized root complex Transactor and the PCIe host controller in QEMU running on the comodel host machine. The accompanying SystemC thread for QEMU (which in itself is a standalone GUI running the GuestOS) is integrated with the testbench and consists of an interface handle to the synthesized root complex on the emulator. Libraries integrated and compiled with the testbench provide the accelerated communication bridge between the testbench and the DUT. The PCIe host controller inside QEMU and the SystemC testbench can use this handle for routing verification vectors to the synthesized DUT.

Sample Root Complex of the Transactor
(Synthesized with the DUT – Hardware side)

```
pcie_transactor #(
   .LANES          (LANES )
  ) transactor_hdl (
   //-----------< clk / reset >-----------
   .clk            (clk ),
   .rst          ( rst),

   //-----------< Pipe Interface >-----------
   .tx_pipe        ( tx ),
   .tx_pipek       ( tx_k ),
   .tx_startblock    ( tx_startblock ),
   .tx_datavalid     ( tx_datavalid ),
   .tx_syncheader    ( tx_syncheader ),
   .rx_pipe        ( rx ),
   .rx_pipek       ( rx_k ),
   .rx_startblock    ( rx_startblock ),
   .rx_datavalid     ( rx_datavalid ),
   .rx_syncheader    ( rx_syncheader ),

   //-----------< Signals for debugging >-----------
   .ltssm            ( ltssm)
);
```

Sample SystemC Threads for Virtual PCIe Device
(Integrated with the testbench – Software side)

```
class Testbench : public sc_module {
 private:
   void UserThread1();
   void UserThread2();
   ..
   void UserThreadN();

   SC_HAS_PROCESS(Testbench);
 public:
   Testbench( sc_module_name name) :
            sc_module(name)
      {
            virtual_pcie_thread = new pcie_inf(
                  (const char *) <hierarchical handle to
pcie transactor>);
            SC_THREAD( UserThread1 );
            SC_THREAD( UserThread2 );
            ..
            SC_THREAD( UserThreadN );
      }
      void start_of_emulation();
};
```

Table 1. a) PCIe Root Complex Transactor synthesized with DUT
b) SystemC thread for PCIe providing an interface for PCIe accesses in testbench

The testbench configures various sub blocks in the synthesized design via APIs. Typical configuration commands in SystemC for a particular block might look as shown below. A block configuration function would consist of a multitude of such configuration calls, often reading and writing bursts of data.

```
int configure_block()
{
      pcie_inf -> write (addr1, data1);
      pcie_inf -> write (addr2, data2);
      ..
      pcie_inf -> write (addrN, dataN);
      read_dataN = pcie_inf -> read (addrN);
```

```
    compare_read_data(read_dataN);
    ..
}
```

The complete verification profile in the testbench would consist of functions which might configure, do end of test checks or generically drive stimulus for all blocks of the device under test in a particular run. A typical verification profile might look like this –

```
int initialization_proc()
{
    Block1_configure();
    Block2_configure();
    Block3_configure();
    …
    BlockN_configure();
 }
 ..
 int end_of_test_proc()
 {
    Block1_end_of_test();
    Block2_end_of_test();
    Block3_end_of_test();
    ..
    BlockN_end_of_test();
 }
```

Alongside above testbench, QEMU frontend would run on the comodel host on top of the virtual PCIe SystemC thread mentioned in Table 1 and communicate with the interface handle internally. Software based verification part of the proposed setup would rely on applications in the Guest OS configuring the chip using the PCIe or USB host controller.

Software running inside QEMU accesses the PCIe device in the OS through Input/Output Control (IOCTL) functions in the PCIe driver to perform accesses to the DUT. IOCTL functions [3] provide a file descriptor handle for the PCIe device in the software driver to communicate with the device. The driver module is loaded prior to software operations on the PCIe device by various applications. Typical PCIe read, write functions in the driver are shown below. The application layer on top of the driver depends on the intended application and industry, but the underlying methodology for device access would be through some similar device – PCIe, USB, I2C etc.

A sample C++ driver gets the device handle inside the Guest OS in below fashion. It opens the handle for read and write operation and stores the descriptor.

```
int get_fd(int devId)
{
    char deviceName[MAX_DEV_NAME_LEN];
    sprintf(deviceName, "%sdev%d", PCI_DEV_NAME, devId);
    return open(deviceName, O_RDWR);
}
```

Sample read and write operations on the device bus using the descriptor handle might look like below.

Write operation	**Read operation**
int regWrite(int fd, uint32_t regAddr, uint32_t size,	*int regRead(int fd, uint32_t regAddr, uint32_t size,*

```
uint8_t *value)
{
        int32_t ret;
        busReadWriteStruct busRW;

        if(fd < 0) // fd is the descriptor for PCIe device
        {
                LOG("Selected Bus not initialized\n");
                return -1;
        }

        busRW.regAddress = regAddr;
        busRW.value = (uint64_t)(size_t)value;
        busRW.size = size;
        busRW.direction = WRITE_OPERATION;

        ret = ioctl(fd, PCIE_REG_CMD, &busRW);

        return ret;
}
```

```
uint8_t *value)
{
        int32_t ret = -1;
        busReadWriteStruct busRW;

        if(fd < 0) // fd is the descriptor for PCIe device
        {
                LOG("Selected Bus not initialized\n");
                return -1;
        }

        busRW.regAddress = regAddr;
        busRW.value = (uint64_t)(size_t)value;
        busRW.size = size;
        busRW.direction = READ_OPERATION;

        ret = ioctl(fd, PCIE_REG_CMD, &busRW);
        return ret;
}
```

Table 2. Software driver functions for reading and writing to PCIe device

Applications running in the QEMU can be complex. The Guest OS provides the platform and infrastructure to write complicated multi-threaded applications eg. multi-threaded DMA applications accessing a DMA memory in the OS. The driver, host controller and the emulation infrastructure shown in Figure 8 provide a common pathway for communication between these applications and the synthesized SoC running in the DUT. These accesses would be in form of reads and writes.

Sample block level functions in a software application might look like below with each block consisting of a multitude of write and read calls.

```
int block_verification_routine( )
{
  block1_SW( );
  block2_SW( );
  ..
  blockN_SW( );
}
```

These routines are similar to the ones which are part of the SystemC testbench running on the comodel host machine. The testbench functions representing software profiles are developed and used for verification in early stages of the verification cycle when the software is not ready. Once the software matures, the testbench functions are replaced by functions or applications running in the Guest OS. Apart from the block level verification functions shown, some applications inside QEMU might have complex software blocks which may not have a counterpart in the testbench initially due to complexities and time-consuming nature of modelling in testbench based verification, eg. multi-threaded DMA applications.

Thus, this setup enables software development for sub-blocks in a modular fashion wherein parts of the testbench responsible for configuration and providing stimulus to a particular sub-block can be swapped with corresponding software application running on the virtual machine once that is ready. Testbench based verification for the full SoC on emulation provides a fast track to start the verification process as a logical next step to block level verification, which is typically started in simulation. The block level testbenches and verification profiles can be used to build the testbench verification functions and used for verification early-on in the project cycle. As the software gets

written, verification routine for blocks can be substituted one-by-one from the testbench to applications in the Guest OS. At the end of the verification process, the software (applications inside Guest OS) drives the verification fully and provides a seamless transition to post-Silicon debug in the lab which relies solely on software.

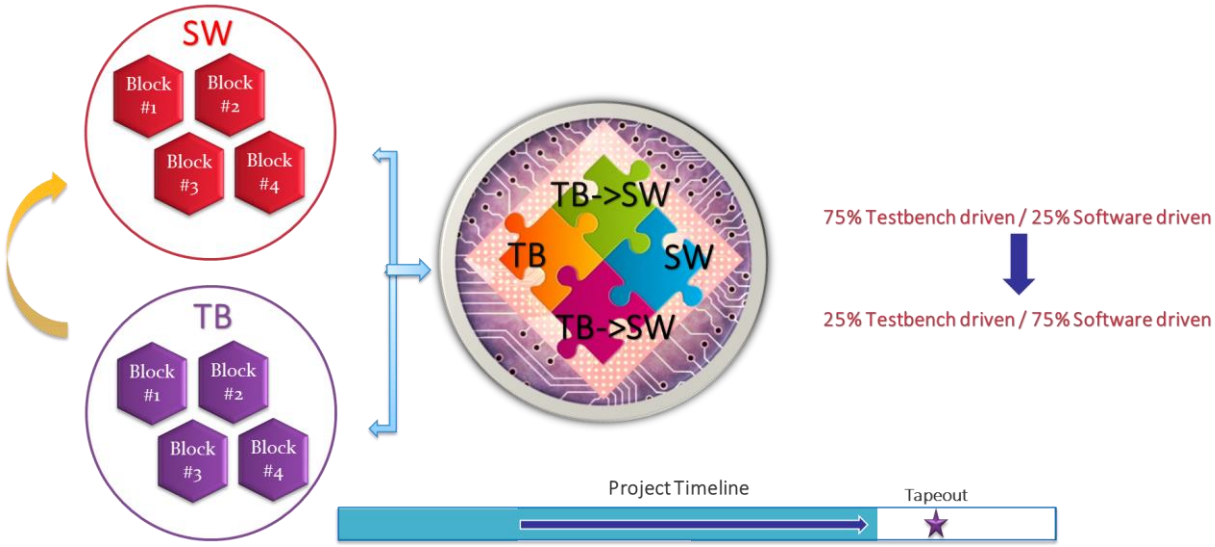

Figure 9. Modular development of software and stepwise migration
from testbench driven verification to software driven verification

As described, this setup also enables early development and testing of any software feature that are difficult to model or abstract to a testbench model used for chip configuration. These blocks being complex eg. DMA applications can solely be developed and run in QEMU, while software profiles are driven by the testbench. This enables best of both the methodologies.

V. Results

The hybrid method proposed above has been applied to verification of SDN switches being configured by external processor using the PCIe protocol, and using systemC testbenches for verification. Following results have been observed:

- Software driver and application development with real RTL started early, months in advance to post-silicon validation. As a result, the software was of better quality at tape-out and before start of validation
- Testbench driven verification happened in conjugation with software driven verification eliminating the need of modelling complex software scenarios like DMA in testbenches. This helped re-use software blocks and saved time and effort spent in modelling such software blocks in testbenches. This also helped catch bugs early on related to these scenarios which would otherwise be detected late in the project cycle when software validation typically starts.
- Improved corner-case testing was made possible with real software configurations. Validation test suites could be run in pre-silicon, since QEMU provided the same front-end as post-Silicon lab validation for developing and running software validation test suites
- Overall project cycle was shorter since software was in better shape in pre-silicon verification. Effect of starting software based verification hand in hand with testbench based verification is shown in the figure below.

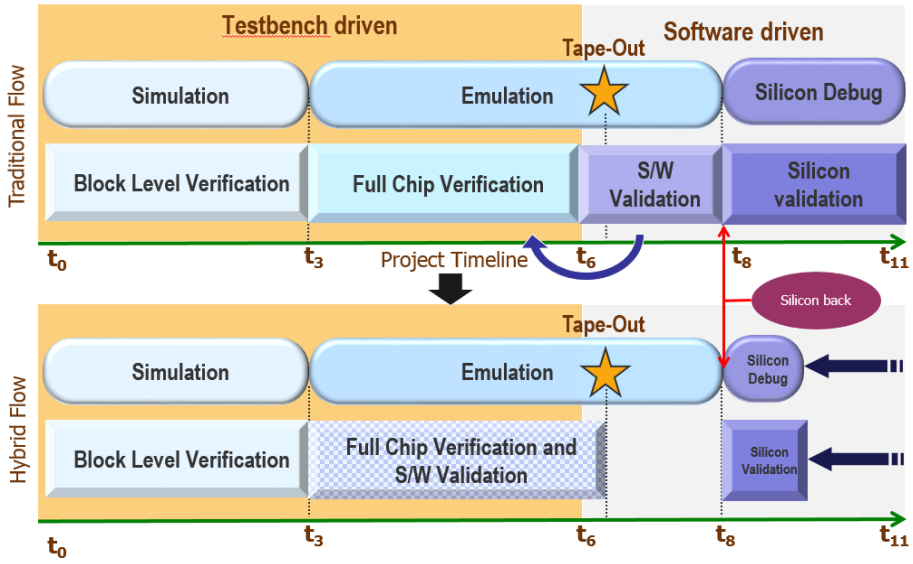

Figure 10. Reduction in Overall Project Cycle

- Quality of silicon was better, and costs potentially lower with bugs addressed early on in product cycle.

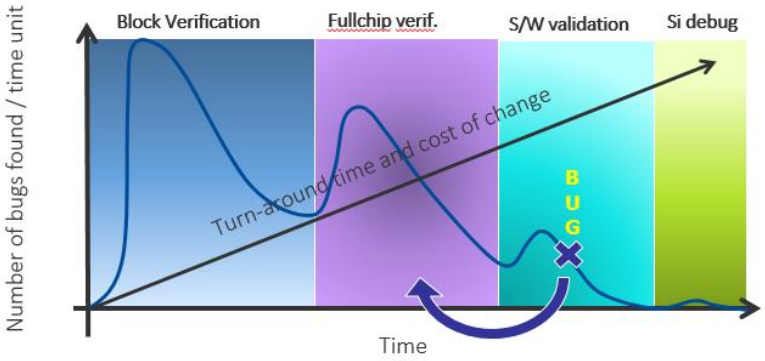

Figure 11. Better quality of Silicon and Lower Costs

VI. Future Work and Conclusion

The proposed method has been described for PCIe protocol but can be extended to any other protocol that can be used to configure an SoC by an external processor. The testbench and the software verification platform (QEMU) must be using the same protocol for verification. The proposed method pertains to early development of software profiles which are interchangeable between a testbench and software application. Overall this method enables a left-shift in verification timeline resulting in better silicon quality and lower costs

VII. References

1. Mohamed AbdElSalam, Ashraf Salem. “SoC Verification Platforms Using HW Emulation and Co-modeling Testbench Technologies”, 2015 10th International Design & Test Symposium (IDT)
2. Ronald Squiers, “Moving to virtual emulation for software-defined networking” [https://www.edn.com/electronics-blogs/absolute-eda/4458740/Product-how-to--Moving-to-virtual-emulation-for-software-defined-networking]
3. IOCTL routines for driver development[http://pubs.opengroup.org/onlinepubs/009695399/functions/ioctl.html]