Exploring Protolink: Effective Debugging from Firmware to Hardware

Owen Chang
Kevin Tung

Sonix
Hsinchu, Taiwan

www.sonix.com.tw

ABSTRACT

Traditional FPGA debugging is very time-consuming and with plenty of limitations such as: probe number, signal frequency and capture length. As digital design and firmware is getting more complex, it is increasingly difficult to identify whether a problem arises in the firmware or the hardware, or even to locate the waveform for analysis.

This paper reviews the key benefits by adopting Protolink during development of a USB3.0 project. Protolink has been proven to be an effective debugging tool for both firmware and hardware. More importantly, the information it provides would allow engineers to further analyze problems at the early stage of debugging, and shorten the development cycle. We will first describe how to use Protolink to trace firmware flow. Secondly, we will demonstrate how to combine other debugging tools (Protocol Analyzer) with Protolink for co-verification. Finally, we will describe how it has been used for analyzing PIPE 125MHz interface.

Table of Contents

Table of Figures

Table of Tables

1. Introduction

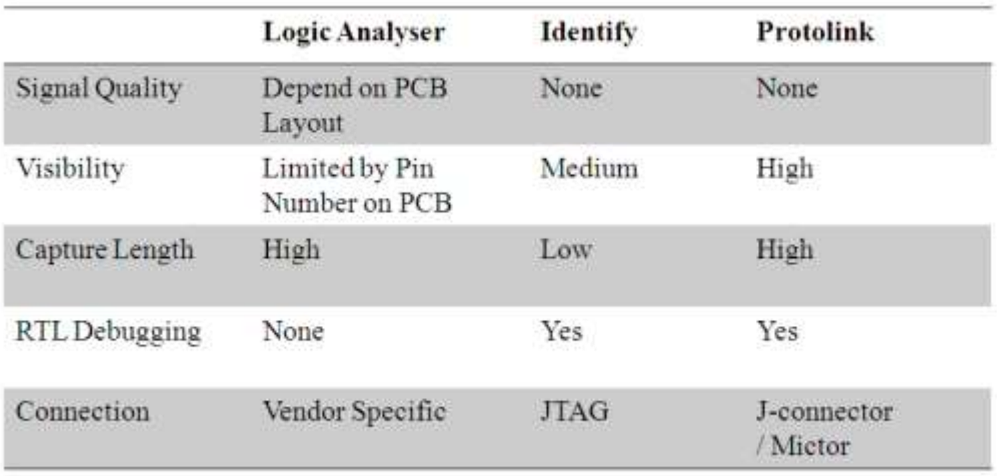
There are several phases involved in an SOC design: RTL coding, Simulation, FPGA verification, and back-end routing. The bottleneck of this development cycle is usually the FPGA verification. Once the verification team have found an issue, it will take a lot of time to figure out whether it is a hardware bug or not. Traditional FPGA debugging methods are either probing design signals to external pins on the FPGA board and capturing these signals by Logic Analyser, or using Identify [1] to extract certain signals for analysis. Both methods have their limitations and require long iteration time.

Here we will describe how Protolink is going to shorten debugging time; with it's larger memory for capturing signals, and the ability of co-verification. Three examples we choose here are based on our experiences while developing the USB3.0 project. You will see how Protolink benefits this project from the firmware development to the hardware verification.

2. Smart Debugging

Debugging hardware during FPGA prototyping is usually done by two approaches. Probing signals to PCB pins and capturing these signals by Logic Analyser (LA) is the first, and given the large memory within the LA, capture length could be very deep. Secondly, if you wish to have RTL source level debugging, Identify is another approach. However, it might require multiple triggers and captures until you locate the problem, since the capture length is not enough to observe the entire event.

Protolink combines the strength of LA and Identify - with its large capture length and RTL source level debugging ability, so the hardware problem could be located in a short period of time. By comparing the three different debugging methods, it's easy to realize the benefit the Protolink could bring on the table.



	Logic Analyser	Identify	Protolink
Signal Quality	Depend on PCB Layout	None	None
Visibility	Limited by Pin Number on PCB	Medium	High
Capture Length	High	Low	High
RTL Debugging	None	Yes	Yes
Connection	Vendor Specific	JTAG	J-connector / Mictor

Table 2-1: Debugging technique comparison

However, building such an effective debugging environment from scratch could be intimidating. Therefore here we will describe the details in preparation, and the few limitations you should be aware of in order to leverage the full capability of Protolink.

The Protolink comes with two parts: PCIe Card, and Protolink Interface Card. PCIe Card will be installed in workstation, requiring **x8 lane** for better user experience.

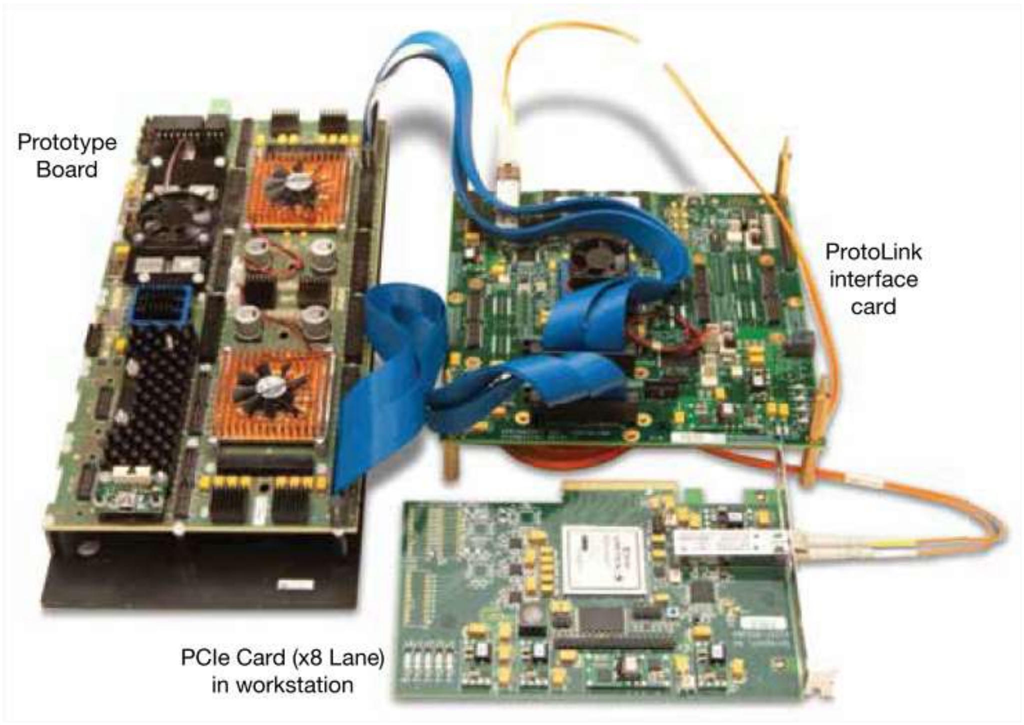


Figure 2-1: Protolink hardware setup [2]

As for the target prototyping board, you will need to reserve at least **22pins** (differential pad) for one J-connector (QTH-060), which includes one probe bus that allows you to capture more than 64 signals of your RTL design. If single-ended signals are all you have to connect with Protolink, it will at least require **69 pins** for capturing 64 signals.

The maximum capacity of signals captured for single FPGA is 3*64*2*16(probe bus number * 64)*(j-connector number)*(TDM) = **6144**.

Note: Reserve differential pad for Protolink while designing new FPGA board
Note: J-connector is better than Mictor in terms of probed signals. If you are new to Protolink, choose J-connector.

In our first Protolink project, we reserved two j-connectors for connecting Protolink, which gave us more than 384 probed signals, enough for performing RTL debugging. And in order to add reusability, we built two PCB boards and maintained high-speed signals on the top PCB, including FPGA, Protolink connector and USB PHY interface (Fig. 2-2). The rest of FPGA signals are routed to the bottom PCB board for specific project usage.

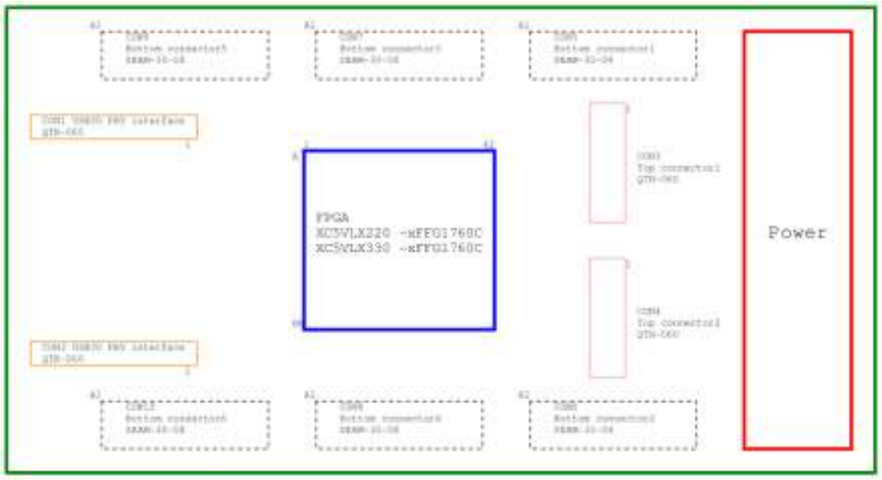


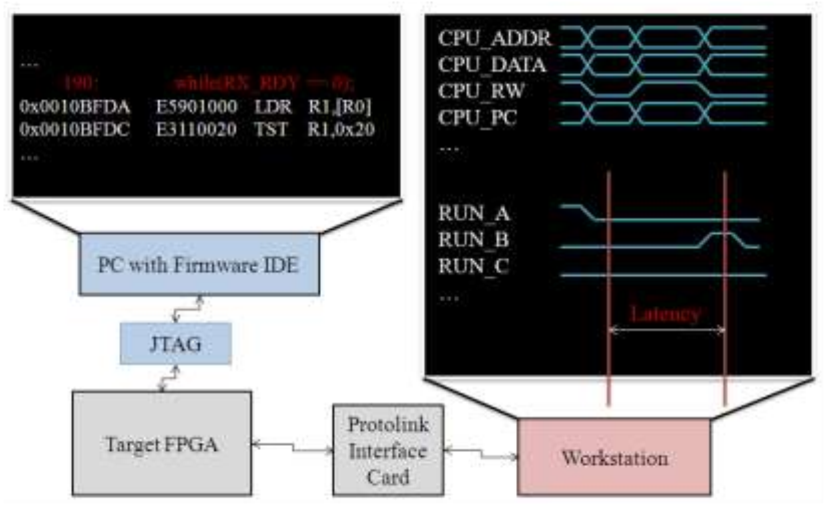
Figure 2-2: FPGA layout example for Protolink

3. Firmware Flow Tracer

3.1 Performance monitor

Firmware engineers usually use printf() for outputting messages to console for debugging. The messages include current program status, error message or user menu for handshaking with user. Despite the benefit that printf() brings to the firmware developer, it will also create a huge impact on the performance and change the program timing due to latency on UART protocol.

By using Protolink, it serves as a system monitor without any performance suffering. You could probe any active signal of your design (ex. RUN_A and RUN_B), and monitor their behaviour to determine if the **latency between this two tasks is longer than expected (Fig. 3-2)**. A system timer usually triggers the task switching in RTOS, and it occurs regularly over several milliseconds. Therefore capability of long capture length is required for such application.



```
...
     190:          while(RX_RDY == 0);
0x0010BFDA    E5901000   LDR   R1,[R0]
0x0010BFDC    E3110020   TST   R1,0x20
...
```

Figure 3-1: Performance bottleneck finder

3.2 Firmware execution and hardware status mapping

Many firmware developers will encounter a case in which an error occurs while in the “freerun” mode, especially when the system is based on an RTOS, with all the interruptions and scheduling process in the background, making it very hard to predict the firmware executing flow. Under this “freerun” mode, step by step debugging ability; system memory data and the variable monitoring capability are all unavailable. The root cause might be a firmware scheduling error, interrupting process sequence error or hardware error triggered by a fast task switching, but they are very difficult to clarify due to the lack of system information.

By using Protolink, you could probe CPU bus and all relative information, along with hardware signals, and then **based on firmware disassembly machine code to trigger signal capturing (Fig. 3-3).** Given the power of long capture length, you should be able to search back and forth to see if your firmware is executed as you expected. Through comparing with correspondent hardware behaviour, it should give you enough information to distinguish whether it is a firmware multi-task scheduling issue or an actual hardware bug.

It is worth mentioning that the user interface regarding the trigger setting (choose signal, setting trigger mode and position) is very easy, as simple as a regular logic analyzer.

Note: If there is an antique CPU without JTAG interface, the option is using hardware ice, or using Protolink to help with program flow analysis.

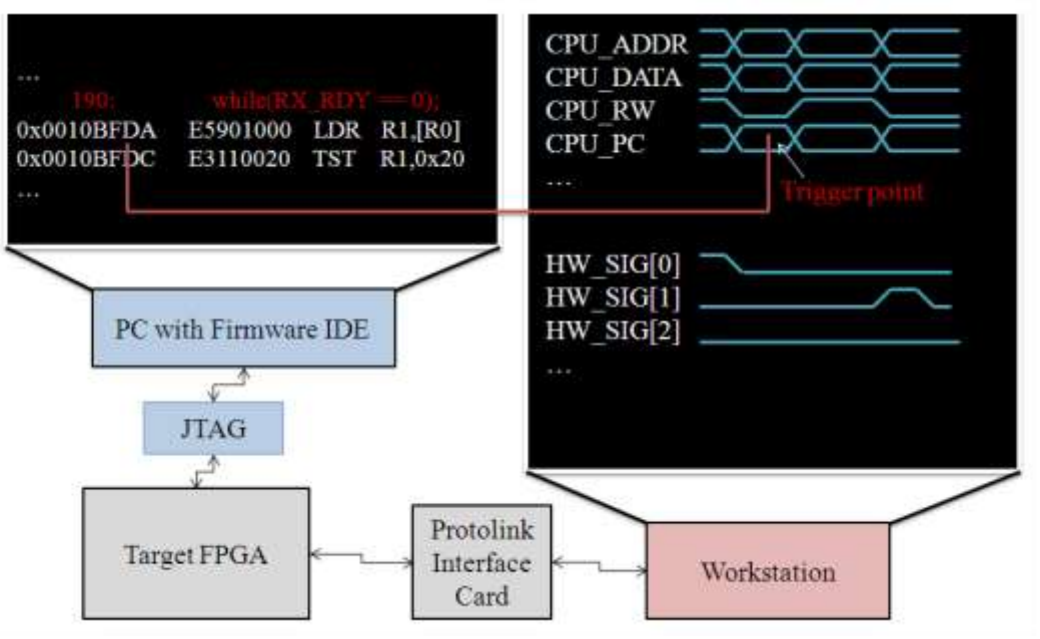


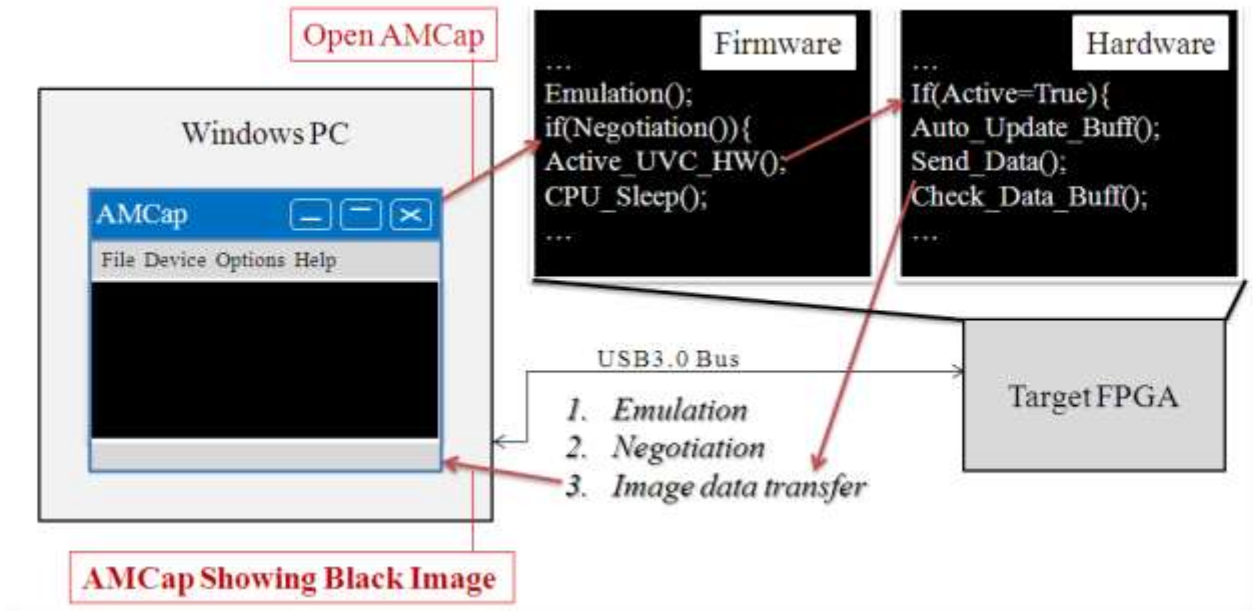
Figure 3-2: Firmware/hardware event synchronizer

4. Synchronize with Other Debug Tool for Quick Hardware Debugging

This USB3.0 project is aimed to develop a USB3.0 Camera (UVC Class). As we know, one frame is composited of many payloads, and the AMCap[3] will translate these payloads into live video based on information provided by the device (Format, Width, Height, PayloadSize…etc)

We have built a smart UVC controller which will automatically feed image data on USB bus without CPU intervention (CPU will then enter into power saving mode). Once the firmware has finished the procedure of bandwidth negotiation with AMCap, UVC controller will be activated. However, AMCap shows black image with no frame rate. And if we monitor the USB bus, there is image data transferred after negotiation is complete.

As showed in Fig. 4-1: After attaching FPGA into PC via usb3.0 bus, the emulation process will begin. After emulation completes we will see an image device shown in the device manager window. Then we open AMCAp, which will start a bandwidth negotiation process according to UVC protocol. Once it is completed, the firmware will activate UVC controller, then force CPU entering sleep mode. The UVC controller will handle all the image data transfer and handshake once it's activated. Despite sending image data to the Host continually, there is no image showed on the screen.



```
…
Emulation();
if(Negotiation()){
Active_UVC_HW();
CPU_Sleep();
…
```

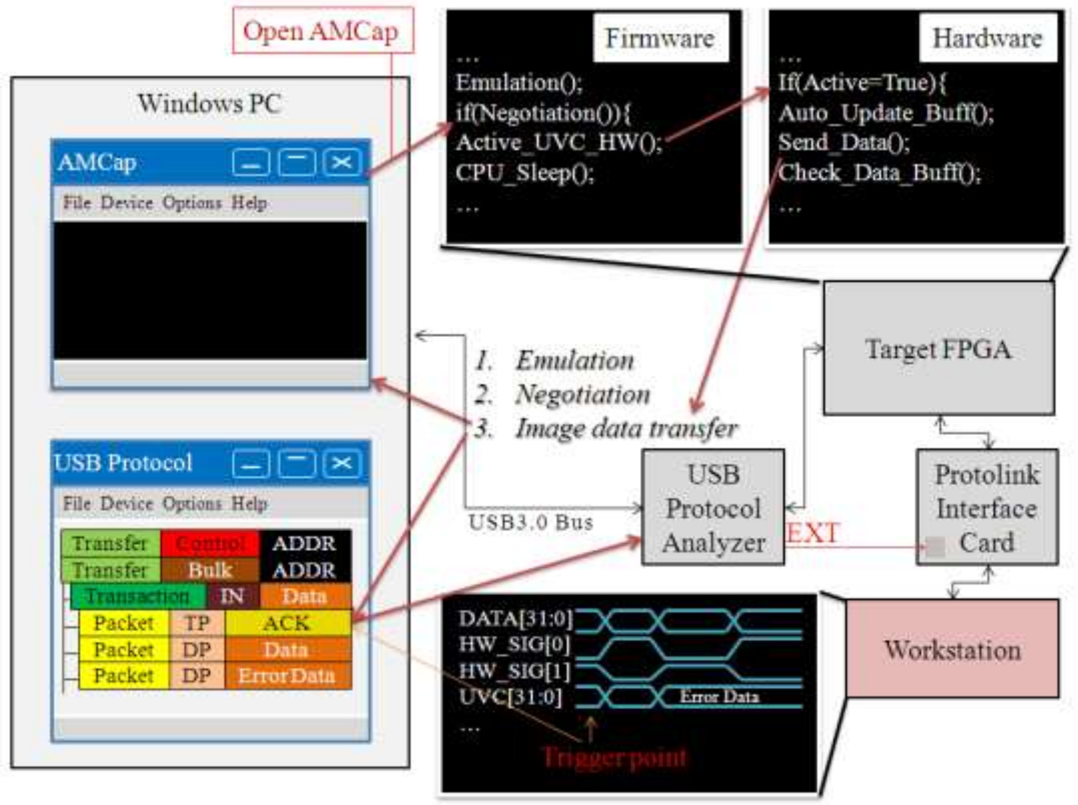
```
…
If(Active=True){
Auto_Update_Buff();
Send_Data();
Check_Data_Buff();
…
```

Figure 4-1: Problem scenario

Now we are facing a situation that doesn't have any possible clues on where the problem might be. The hardware engineers do not know how to debug because the UVC controller is sending data as expected. The firmware engineers also think the setting is correct and the image data is now handled by hardware alone, and should not have anything to do with firmware. The only information we have to analysis is the USB protocol data acquired by Protocol analyzer. After analyzing the bus traffic log by firmware engineer, we have found that the UVC controller will somehow transfer one oversized payload randomly, which will cause the total amount (frame size) of image data to not match the firmware setting and will confuse the AMCap. This is a hardware bug.

Since we have found the hardware bug, the next step is to see how the hardware processes the image data and the state machine at the moment to further clarify the root cause. Traditionally we will use Identify to trigger and capture hardware signals, but this error occurs randomly, and **there is no hardware trigger point**. Another traditional approach will be trying to duplicate in simulation. However this requires massive CPU time to complete transfer of one single frame, and still no hardware trigger point for locating the position of error payload.

Here we leverage the ability of external trigger input of Protolink, and setting trigger conditions from USB Protocol analyzer to the start of the frame (Fig. 4-2). Then we set the capture length of Protolink to 53ms (over a frame interval). After we open AMCap, the firmware and hardware will perform their tasks and UVC controller will send oversized payload in one frame randomly. The procedure is the same, but the difference is our ability to **synchronize the log of USB protocol and hardware behaviour**. Once we have the protocol log, the firmware engineers could analyze and find this random oversize payload, and record its data (running number). The hardware engineers could base on the trigger point, search this particular number in image data output buffer, and observe the relative signal behaviours once this oversized payload is located in the Protolink debug window. Under this powerful debugging environment, it only took us 2 days to locate, fix and confirm the bug.



```
...
Emulation();
if(Negotiation()){
Active_UVC_HW();
CPU_Sleep();
...
```

```
...
If(Active=True){
Auto_Update_Buff();
Send_Data();
Check_Data_Buff();
...
```

Figure 4-2: Protolink solution

This case has showed the potential usage of Protolink. By synchronizing other debugging tools with Protolink, the hardware engineers are more likely to locate the design flaw with the help of firmware or testing engineers. Both firmware engineers and hardware engineers could contribute to the debugging process, and this is in fact the most straightforward way to map the error status to the hardware status.

Note: external trigger signal quality should be taken care of, to avoid false triggers.

5. 125MHz Interface Debug Capability for USB3.0 PHY

In order to test system reliability, firmware engineers will usually prepare a so-called “burn-in” test, which is performing massive data read / write / compare over a long period of time. While performing burn-in test, we notice the USB bus will enter into Recovery mode randomly and then resume to normal data transfer. The data is intact, and no compassion error was found, but the system performance would decrease due to the extra latency caused by the Recovery cycle. We know the Recovery is triggered by an incorrect link packet, [4], but we don’t know whether this is caused by PHY or by the USB Controller itself (Fig. 5-1).

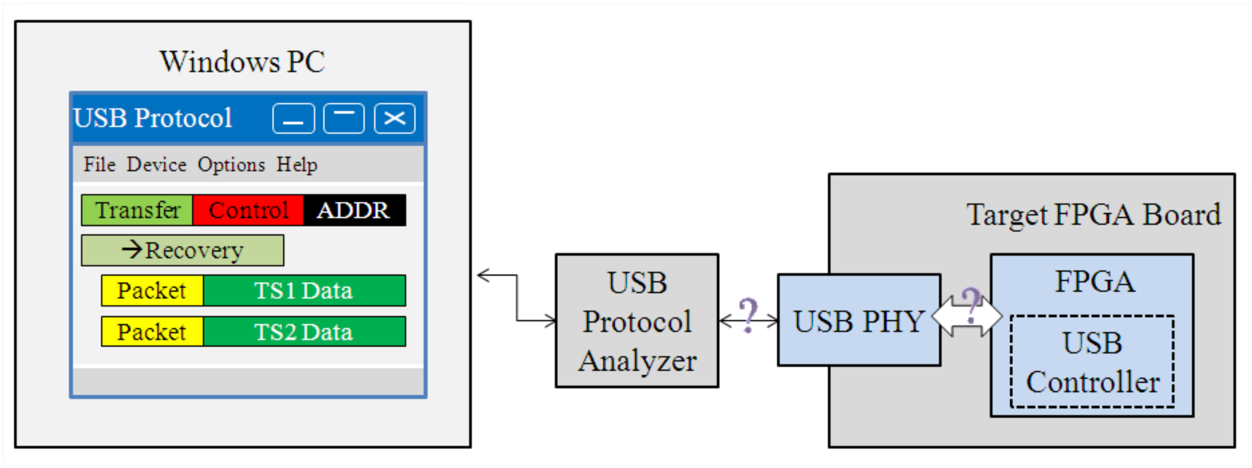


Figure 5-1: Problem scenario

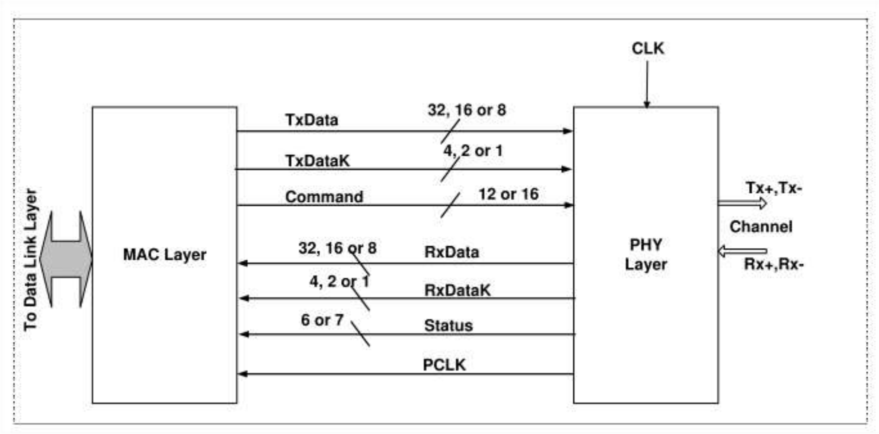


Figure 5-2: Phy interface[5]

The PIPE interface between USB3.0 Controller and PHY is running at 125MHz (Fig. 5-2), which is the maximum user clock for Protolink in Synch Probe mode (TDM=1). Therefore we use the 125MHz from PHY as a Synch Probe clock in order to capture the signal on PIPE interface.

Again, we use Protocol Analyzer as a trigger source, **triggering Protocol Analyzer and Protolink at the same time when spot Recovery on the bus**. Since we are interested in the state before entering Recovery, we set the trigger position to 20%. We cross compared each packet on the bus between USB Protocol Analyzer and Protolink, and finally we found a mismatch, it turned out there was **one bit different**. The data feed for USB Controller was incorrect, and caused USB Controller decode fail, then entered Recovery mode (Fig. 5-3).

For this issue, we were only concerned with the quality of the USB Controller, so we handed over the results to the PHY team for further investigation. In this case we are not only shown once again the useful techniques to synchronize with another debugging tools, but have also proven the ability of 125MHz Synch Probe mode could be adopted on any other high-speed interface.

Note: Only 64*6 probe signals are allowed at 125MHz Synch Probe mode.

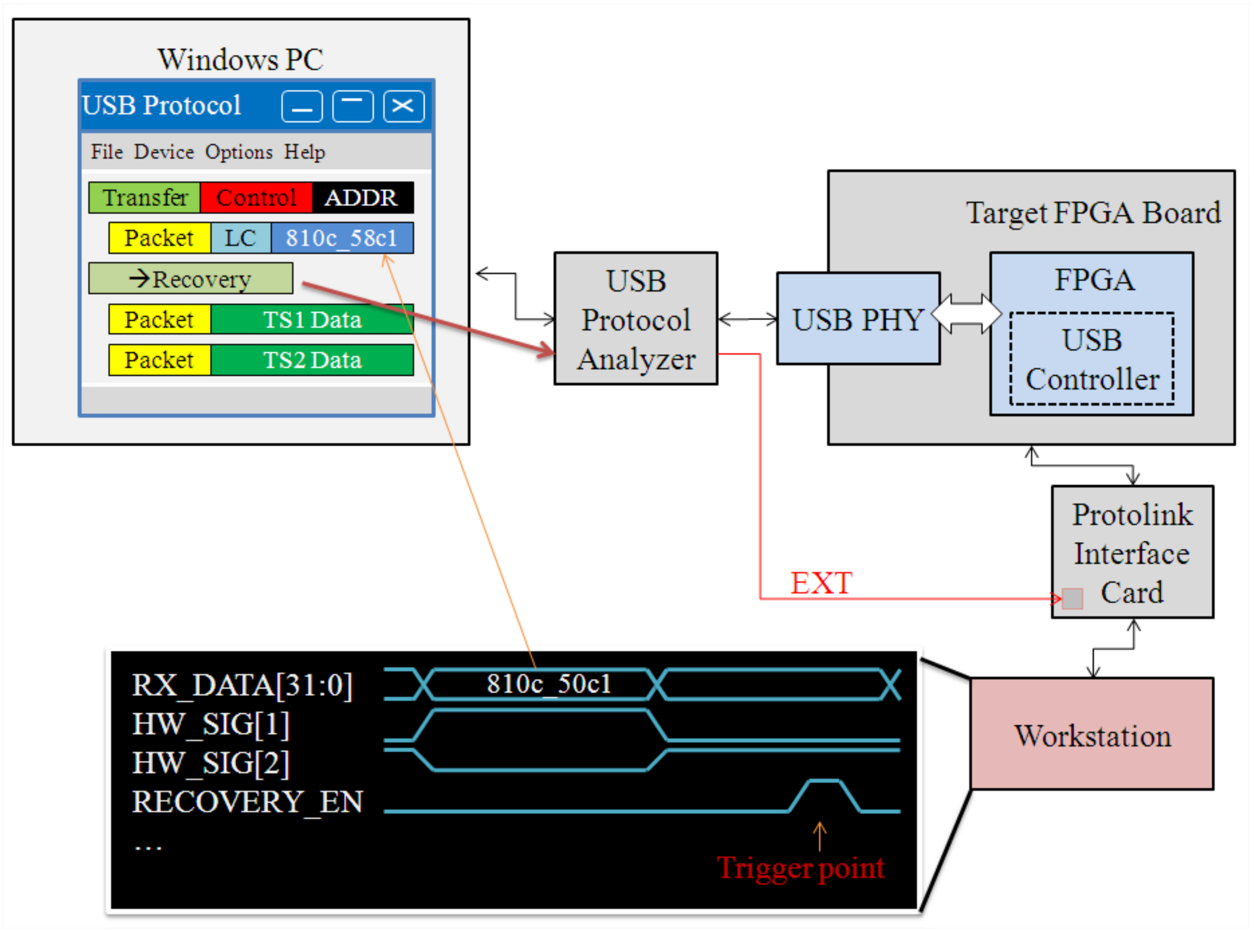


Figure 5-3: Protolink Solution

6. Summary and Results

The iteration of redesign and debugging is usually the most time-consuming portion within the entire development cycle. USB3.0 project is the first project applied with Protolink, starting from PCB design to RTL implementation. During the development process, Protolink shows its ability to benefit the project from firmware coding to hardware debugging. The user interface is also clean and easy to use.

By leveraging its long capture length and external trigger features, firmware / hardware co-verification is finally achieved. Remove the burden of finding a proper hardware signal trigger position. Using triggers from firmware or system events to further locate and analyze the problem is a better way to pinpoint the root cause. The collaboration of firmware engineers and hardware engineers would be the best and fastest way to solve any unclear issues.

7. Conclusions

We aim to provide a clear path to any user who is still struggling with the tradition debugging methodology. We have listed a few things, which should be taken into consideration even before designing PCB Board **(probe number and PCB reusability)**, and also described several useful tips and solutions for your references **(background performance analysis, synchronize with other debugging tools and high speed interface analysis)**.

The transition from old debugging methodology to new debugging methodology (Protolink) would certainly need extra engineering efforts, but it is the necessary move and worth taking. In our case, Protolink did help on shortening development cycle, and making the product more robust at the same time, ultimately contributing to the annual revenue (saving cost from ECO and re-tape out).

8. References

[1] Identify: Simulator-like Visibility into Hardware Debug. http://www.synopsys.com/Tools/Implementation/FPGAImplementation/FPGASynthesis/Pages/Identify.aspx

[2] ProtoLink: Multi-FPGA Debug Solution for SoC Prototype Systems. http://www.synopsys.com/Tools/Verification/debug/Pages/protolink-ds.aspx

[3] http://amcap.en.softonic.com/

[4] Universal Serial Bus 3.0 Specification, reversion 1.0, June 6, 2011

[5] PHY Interface For the PCI Express and USB3.0 Architectures, version3.0