

## Chapter 2

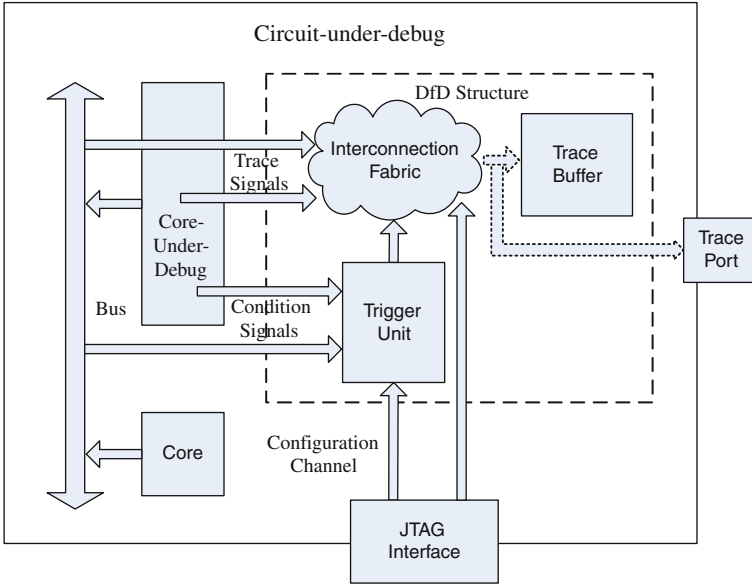
# State of the Art on Post-Silicon Validation

The first widely-adopted post-silicon validation technique utilized by the industry is to reuse the IEEE Std. 1149.1 (JTAG) test access port and existing design for test (DfT) structures in the circuit (e.g., scan chains) to run, halt and step the circuit under debug (CUD) to find bugs [71]. This technique is quite effective in identifying those easy-to-find bugs that leave “evidences” when the circuit halts, but fails to find those tricky bugs that manifest themselves only after a long period of operational time [65]. In addition, the behavior of many bugs is hard to repeat, making diagnosis with this run/stop debug methodology even more difficult. To mitigate the above problem, designers can add shadow flip-flops (FFs) to the CUD to increase its visibility during normal operation [32]. However, this method can only sample a few snapshots of the circuit’s operational states and it also involves nontrivial DfD overhead.

To be able to root-cause design bugs, post-silicon validation requires to increase controllability and observability of the CUD’s internal behavior to a much higher level than what manufacturing test generally needs [24]. A more effective silicon debug technique is to selectively monitor and trace internal signals of the circuit continuously during its normal operation [3]. The traced data can then be either stored in an on-chip trace buffer or transferred out of the chip via a trace port for later analysis.

The hardware infrastructure to facilitate trace-based silicon debug is shown in Fig. 2.1, wherein various DfD modules are introduced at design stage of the CUD for later debug purpose. Generally speaking, designers select to tap a number of signals in the CUD (typically thousands of signals in million-gate industrial designs [1]). However, only a subset of the tapped signals are traced concurrently during debug phase due to trace bandwidth limitation. This is achieved by an “interconnection fabric” (e.g., a MUX tree) that links the tapped signals to trace buffers or trace ports. In addition, trigger units are typically used to determine when to start and stop signal tracing so as to further reduce trace bandwidth requirement. In most cases, designers reuse JTAG test access port as the control interface for the debug phase.

When the first silicon is back with some bugs, in each debug run (see Fig. 2.2), designers first configure the DfD module in the CUD by selecting the to-be-traced



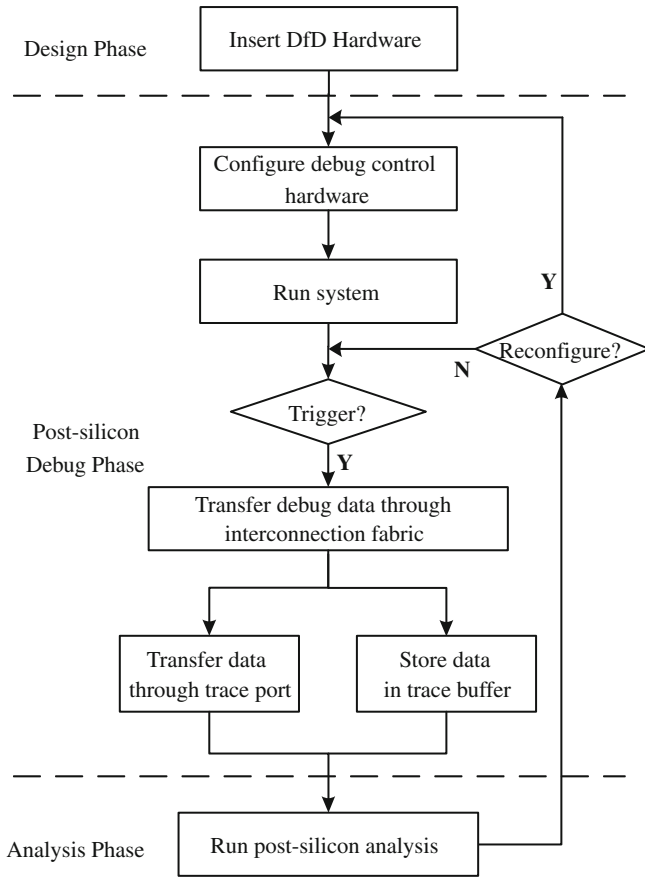
**Fig. 2.1** Trace-based debug infrastructure [75]

signals from the tapped ones and determining the trigger conditions for signal tracing, and then put the CUD into normal operational mode. If the pre-determined trace condition is met, the traced data is transferred through the interconnection fabric to on-chip trace buffers or off-chip trace ports. The collected data are then analyzed to root-cause the possible design bugs. The above process iterates a number of debug rounds or even a few re-spins (when unlucky), until all the bugs are eliminated.

Figure 2.3 depicts the ARM *CoreSight* trace-based debug solution [7], wherein each ARM core is equipped with an embedded trace macrocell (ETM) for capturing the processor's states. It also contains a cross trigger interface so that trigger events can be transferred between different ETMs to facilitate multi-core debug.

A huge volume of trace data, however, is difficult to analyze and results in high DfD overhead. Therefore, how to conduct signal tracing effectively for bug elimination while keeping the associated hardware cost manageable (usually required to be less than 10 % of the original design) is a challenging task for IC designers. In the following we provide in-depth discussion for trace-based debug strategy and review recent advancements in this important area. In particular, we discuss the following issues in trace-based debug solution:

- Out of the large amount of state elements in the circuit, which signals should we choose to monitor and trace to provide high visibility to the CUD?
- How do we design the trace data transfer module to provide enough observation flexibility while keeping the associated DfD overhead low?

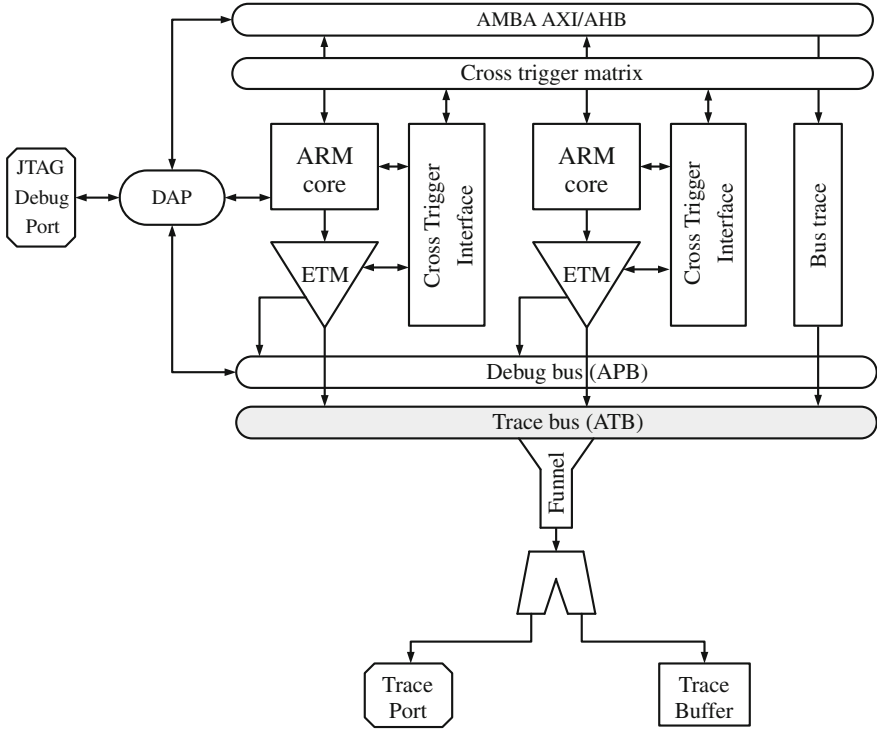


**Fig. 2.2** Trace-based silicon debug flow

- How can we compress the large volume of trace data effectively so as to make efficient use of the limited trace bandwidth provided by trace buffers and/or trace ports?
- How do we control the signal tracing effectively to obtain highly-qualified trace data?

## 2.1 Trace Signal Selection

Trace signal selection becomes the primary problem in trace-based debug. This is because, ideally we wish to “see any signal at any time” during post-silicon validation and clearly it is not achievable with the large amount of internal signals deeply embedded in the fabricated chip. As indicated in Fig. 2.1, with the help of constrained



**Fig. 2.3** ARM coreSight multi-core debug architecture [7]

DfD resources, we can only afford to tap a few internal state elements and use them to help designers root-cause the abnormal behaviors of the CUD. The objective is therefore to select those *essential* signals in the CUD so that bugs have a high chance to leave “evidences” on them, and the effectiveness of trace-based debug strategy highly relies on which signals are selected to be traced.

To debug errors on microprocessors and software running on them, naturally it is beneficial to observe the execution of the instructions. In [53], the authors proposed to trace the behavior of every execution stage of instructions to obtain more detailed information on how the microprocessor operates. In addition, several methods have been presented to monitor either the communication interface of the processor (data channel, address channel and control channel) [7, 39, 64], or the memory contents that store the execution results [38, 73].

For the increasingly popular network-on-chip (NoC) based designs, more visibilities are required to the communication among multiple cores, especially at transaction level to provide a globally consistent view of the system. Several trace selection methods were proposed for such type of designs in the literature to tackle this problem [17, 65, 72]. As an example, [17] proposed to attach dedicated monitoring probes on routers and provide transaction level observability of the NoC.

The above techniques are quite effective for the targeted types of circuits, but we are still facing the trace signal selection problem for general logic circuitries. In current practice, designers usually manually select those signals that are considered to be vulnerable to bugs or important for analysis to trace, based on their own design experience. This ad-hoc method, however, cannot guarantee the quality of the selected trace signals. More importantly, bugs often occur in unexpected scenarios and it is very difficult, if not impossible, to predict which signals will be related to them during the design phase. Therefore, we need to have at least some trace signals that are selected in an automated manner without designers' intervention.

## 2.2 Interconnection Fabric Design for Trace Data Transfer

As designers are not knowledgeable about which part of the design may contain bugs, a relatively large number of signals are selected to be traceable in the circuit, typically in the thousand range for million-gate designs [1]. Due to the associated DfD area cost and debug bandwidth requirement, however, it is impossible to *concurrently* monitor and trace all the tapped signals. Instead, only a small number of internal signals can be real-time observed together, and it is up to the designers to determine which signals to trace at a specific debug run, according to the system's erroneous behavior. These signals are then transferred to on-chip trace buffers and/or off-chip trace ports for diagnosis.

To reduce the DfD cost, industrial designs typically use MUX trees to select a subset of the tapped signals to trace in each debug run, in which the control signals to the multiplexers are configured through the JTAG interface (e.g., [1, 69]). To meet timing constraint for the tracing logic, the MUX trees can be pipelined. In addition, when the tapped signals come from multiple clock domains, first-in first-out (FIFO) buffers and/or flip-flop chains can be used to ensure data safety [1]. The above design methodology, however, limits this flexibility of observing any combinations of related tapped signals and reduces the visibility to the CUD, as any signals going through the same multiplexer cannot be traced concurrently. This problem can be easily solved by introducing non-blocking concentration network [48], which is able to select any  $m$  signals out of  $n$  inputs ( $m \leq n$ ) and output them to the trace buffers/ports, but such design is with prohibitive DfD cost.

## 2.3 Trace Data Compression

Due to the limited trace bandwidth provided by trace buffers and/or trace ports, storing the "raw" traced data is not quite economical. Various trace data compression methodologies were presented to tackle this problem effectively.

In [53], the authors utilized the locality feature of instruction sequence and redundant information in monitored data that can be easily identified with the executed

instruction to store the execution states of microprocessor. With the technique, a small amount of *footprints* are enough to observe the whole operational behavior of the microprocessor under debug. Recently, several works [38, 73] were presented for tracing the contents in cache. They both utilize the data locality feature when accessing cache and adopt dictionary-based compression to further improve the compression ratio. Different from each other, [73] observed the following features to enhance compression ratio: the similarity in tag field caused by spacial locality of memory reference and the unusual usage of high order bits for integer value; [38], on the other hand, proposed to reuse cache to compress instructions by inserting supporting module into it. The method is able to restore full information with a small amount of traced data combined with the contents remaining in cache.

Recently, a lossy compression method based on multiple-input signature register (MISR) was presented in [4]. With the assumption that the CUD behaves repeatable in different debug iterations, the method consecutively zooms-in the sampling intervals with compressed failure signatures generated from MISR to localize the error.

## 2.4 Trace-Based Debug Control

As shown in Fig. 2.1, trigger unit is designed for determining the signal tracing behavior. The behavior can be start and stop tracing so that unnecessary data can be filtered to reduce trace data volume (known as *trace qualification*). More importantly, the designers' capability of controlling the CUD directly affects the debug effectiveness. This is because, when designers can easily control the CUD into suspicious state and obtain relevant information, it becomes much earlier for them to root-cause the possible errors. The problem has been extensively studied in academia.

In [71], the authors described several basic DfD modules that can be used for debug control, including comparator to check if the condition signals are met with pre-configured value, and counter to facilitate the trigger control with temporal information. Later, [10] proposed to synthesize more complex control unit (i.e., assertion checker) to monitor complex behaviors of the CUD (e.g., ATB communication protocol). The unit is a state machine that can be generated from the description with formal languages for assertion-based verification. Later, the same authors [11] introduced several enhanced features to localize the errors that are buried as internal states in sophisticated assertions, in addition to reduce the associated hardware cost in unit generation in [10].

Multi-core debug control for complex SoC devices is a challenging task since we need to be able to control related cores simultaneously [72]. This problem becomes particularly difficult when the data transfer among cores is not deterministic, in which case, it is rather ineffective to configure all the required trigger conditions before running the system. To tackle this problem, [65] proposed a so-called in-band cross-trigger event transmission infrastructure. By inserting the cross-trigger events into the messages, designers are able to trace the desired messages more easily.

Trace-Based Post-Silicon Validation for VLSI Circuits

Liu, X.; Xu, Q.

2014, XV, 108 p. 59 illus., 38 illus. in color., Hardcover

ISBN: 978-3-319-00532-4