PDF-XChange Viewer Click to buy NOW! www.docu-track.com

Chapter 2
On-Chip Instrumentation Components

In this chapter we examine a typical on-chip instrumentation environment and discuss some of the individual instruments used for system debug, their typical features, and their integration, both with each other and the systems being analyzed.

2.1 Trace and Event Triggering

Concepts of the tracing of data as it moves through the application or system are central to most other instrumentation capabilities. To address different debug requirements, instrumentation blocks must support different implementations of trace collection. Typical requirements include the ability to trace in cycle, branch, and timer modes. Cycle mode collects all bus cycles generated by the core(s). Branch mode collects all execution path changes, sometimes called branch trace messages. Timer trace mode records a frame with a timestamp each time an event is satisfied, providing basic performance analysis measurements.

Event recognition is widely used in conjunction with trace to capture information on events and operations in the SoC. Trace data values can be monitored and compared to provide real-time triggers to control event actions such as breakpoints and trace collection. Event recognizers can simultaneously look for bus address, data, and control values and be programmed to trigger on specific values or sequences such as address regions and data read or write cycle types. The event recognizers can control enable or disable of breakpoints and trace collection (Fig. 2.1).

Data tracing based on recognizable events opens doors to new capabilities in real-time SoC analysis. The data trace mode provides real-time information about the status and data of a system's internal signals, including, for example, analysis of cache performance and internal memory and data transfer operations that cannot otherwise effectively be extracted from a system. In-line or postprocessing trace information allows for analysis of data flow performance or measurement of system characteristics such as bus availability or cache hit/misses, which require long-term steady-state (measured over many cycles) system information. Additional detection of events in traced data allows the development environment to flag specific features in the trace data as it flows through the application.

N. Stollon, *On-Chip Instrumentation: Design and Debug for Systems on Chip*,
DOI 10.1007/978-1-4419-7563-8_2, © Springer Science+Business Media, LLC 2011

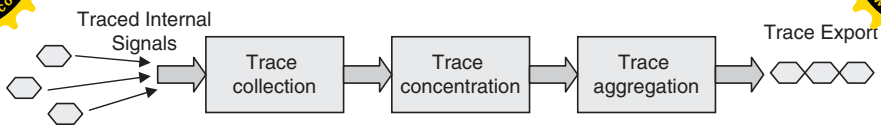

Fig. 2.1 On-chip trace formatting and export instrumentation

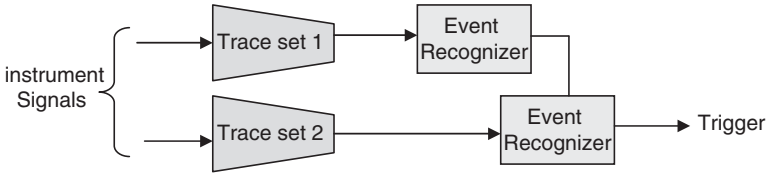

Fig. 2.2 Instrumented event recognition

As an example of a complex instrument event recognizer feature, four event recognizers can be combined in a 1–2 and 3–4 arrangement to produce two complex events. In this arrangement, the complex events can be configured so that first event of the event pair must be satisfied before the second event is enabled (Fig. 2.2).

2.2 External Interfaces for On-Chip Instrumentation

JTAG pin interfaces are the default interfaces of the most basic debug functions. Higher pin-out trace and probe ports are used with many on-chip instrumentation approaches. Even with these ports, however, the amount of debug information available can easily exceed the allocated debug interface of a SoC. To reduce the information being sent over the interface, approaches such as data compression increase the performance of the debug interface without significantly affecting system cost.

Obviously, the most useful approach to reducing the information from the debug port to the host development tool is to limit transmissions to new information and have inferred information derived by the development tools. For example, for required addresses to trace the instruction flow, it can be seen that not every instruction is required to construct an instruction trace. If the target processor does not have a change of flow, then the full address does not need to be transmitted. Only when a change of flow, such as an interrupt or branch, occurs would the system need to send the new beginning address. In addition, if the debugging session must be real time, then some information may be held in reserve. For instance, not all data values have to be visible at all times; only the data that the engineer is concerned with should be sent to the debug port during run time.

One of the major limiting factors on the use of instrumentation in SoC and multicore architectures is the ability to quickly export data as it is generated. On-chip instrumentation can address many of the operations associated with large amounts of

on-chip debug, including triggering and performance monitoring. There is, however, almost always a need to be able to view the debug signals such as instruction/data trace from a processor, which means data must be exported off chip. The ability to transmit debug signals, most notably trace, is a hard limited function of two parameters:

1. The number of IO pins that can be dedicated to export of debug information at any given time.
2. The speed at which these signals can transmit the data.

This problem of exporting debug data is compounded for multiple-core SoC architectures, with monitoring of internal address, data, and control signals for each core, with the addition of inter-core and peripheral bus signals. One basic instrumentation approach is to rely on on-chip memory to buffer between traced data and the export bandwidth available. Trace buffer must consider the differences between data being generated on chip and the throughput of the debug interface. If buffers are modest in size, they can be overloaded by a large amount of trace data, as example, from multiple IP blocks or internal buses.

Despite the increasing number of IO signals available in leading-edge packages, system designers must limit the number of IO signals dedicated to trace and debug to reduce system cost (with packaging becoming an increasingly dominant factor in system cost). Most current approaches to increasing the IO bandwidth for debug rely on increasing the effective number of IO pins available (by multiplexing debug mode information into other system pins) and using higher-speed IO to increase the throughput of each pin. Each of these approaches to increase debug throughput has advantages and disadvantages. Increasing the effective pin count by statically multiplexing pins is a well-proven and low-risk approach. It does, however, involve coordination over the entire operation of the SoC, because pins that are dedicated to extended clock cycles to debug operations are unavailable for use in other modes of operation. To support SoC core and internal bus speeds, bigger pin bandwidth is increasingly required for instrumentation interfaces of a SoC.

2.3 Performance Analysis Using On-Chip Instrumentation

Customized instrumentation can integrate performance analysis of SoC architectures as part of a debug solution. Performance analysis (PA) is an all-encompassing term that refers to many types of measurements that provide information on how a particular core is being used, both in context of other parts of the system and with regard to specific algorithms. Integrating instruments to allow processor characterization, software performance, and system performance metrics provides valuable and concise information, which is more simply gathered locally to the processor because the lack of IO signal visibility in individual processor operations limits tracking of embedded processor performance. Performance metrics can be distorted or obscured by the layers of system buses, peripherals, and limited IO access between an embedded processor and the external test environment.

Some common types of tests that are desirable in processor and SoC system performance analysis are:

- Find and profile hot spots in execution.
- Be able to measure loop times.
- Trace function calls, returns, and interrupts and measure the performance of this code.
- Measure the duration of ISR (interrupt service routine) and other events.
- Track interrupts and measure the maximum interrupt latency.
- Track RTOS context switches, measure task duration, and measure OS events such as semaphore waits.
- Measure the cache hit/miss ratio.
- Measure on-chip and off-chip memory access use.
- Count the number of processor stalls caused by (slow) bus accesses.
- Measure bus use and which master-slave transactions are using the bus the most.
- Count the number of processor stalls in a section of code.
- Count the number of instructions executed between two points in a program.

2.4 On-Chip Logic and Bus Analysis

Instrumentation-based logic trace allows analysis of bus architectures and related nonprocessor IP. Logic analysis instrumentation typically consists of debug blocks that are integrated into synthesizable logic files (typically VHLD or Verilog).

The bus analyzer collects a history of on-chip bus activity and exports it through the JTAG interface. Bus signal information is connected to the data inputs. A triggering system user starts and stops collection of data to an on-chip trace RAM. When collection stops, the most recent activity remains in the trace memory, from which it is unloaded through JTAG and displayed. The bus trace configuration includes a timestamp, which is stored with the data; to provide synchronization and interval information, on-chip counters for performance measurements; of the frequency of system events, and JTAG-controlled registers that hold parameters for input and output triggering of control operations that allow captured bus signals to interact on-chip with other debug components in the system.

- Bus fields include address bus, data bus, and user extension field and can track a number of bus masters in the system. More than one bus layer may be supported in a single instance. For more trace capability, or trace over different clock domains, more than one bus navigator instance can be implemented in a single JTAG chain.
- The trigger state (started, active, stopped, stalled) is recorded in the trace buffer. A multistate trigger allows triggering on sequential events. For example, a configuration that recognizes bus cycle A followed by bus cycle B is:

```
if(event A and state 0)then goto state 1
if(event B and state 1)then trigger
```

- Timestamps are used to indicate the distance between recorded samples when collecting trace using qualifications such as trace-on/trace-off, or collecting filtered trace that matches an event definition. Because bus measurements may be large numbers of cycles, the timestamp is set up to cover a large time range.

Being able to trigger from instrument data allows for both dynamic interactions with the target system and improved capture of the information of interest. Analyzers nominally support multiple triggers with multiple states per trigger (Fig. 2.3). Trigger conditions can be created as application-specific combinations of three components:

- Raw or processed data (filtered or aligned) compared to logic or edge events on each signal.
- Counter or times values matching a preprogrammed value.
- Trigger state (what trigger-related operations have occurred previously?).

When a trigger condition is satisfied, one or more actions can be taken, such as to mark the trigger frame, turn trace on or off, record a single frame, turn the counter on or off, increment or clear the counter, assert the external trigger out, or change the trigger state. The flexibility of this system under a wide variety of conditions and actions can improve visibility and monitor and tune system performance based on a range of operational parameters.

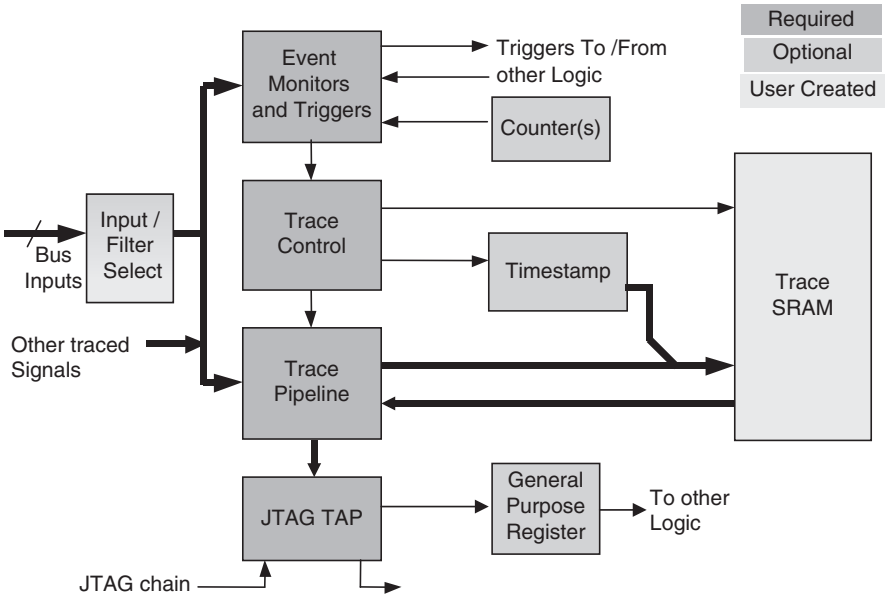

Fig. 2.3 Bus trace instrumentation block diagram

2.5 On-Chip Instrumentation Examples

In this section, we present several examples that illustrate the instrument features just discussed.

2.5.1 Trace Monitoring and Interfaces

Embedded processor instrumentation addresses embedded processor debugging and system validation features such as run control, trace history, memory and register visibility, and complex breakpoints.

The external trace monitor is an instrumentation block integrated into and supporting of processor core monitoring. Trace monitoring allows capture of both execution history and other real-time information from the core and allows either on-chip or off-chip trace storage. Trace monitors can also be configured to collect profiling data for performance analysis. The specific instance of the instrument interfaces a debug unit interface for a processor architecture that provides debug functions such as start/stop execution, single-step, breakpoints, and register/memory access (Fig. 2.4).

The trace monitor allows trace history to be captured in several modes (instruction and/or data full or compressed, etc.), depending on the available bandwidth and information desired. The block combines trace messages of various lengths into trace words of fixed width suitable for writing into memory, which are then sent to either on-chip memory or through a trace port to off-chip memory. Because the bandwidth of an external trace port is limited, the user must be selective about what information to collect. Typical choices include execution trace, data cycle trace, and profiling trace. The trace collection may also be enabled and disabled by hardware breakpoint registers set to generate trace actions (Fig. 2.5).

The trace monitor buffers trace words using a first-in-first-out (FIFO) memory, in order to compensate for the latency for outputting a trace word. The size of the FIFO is application dependent, and if the size is too modest, trace data can overflow

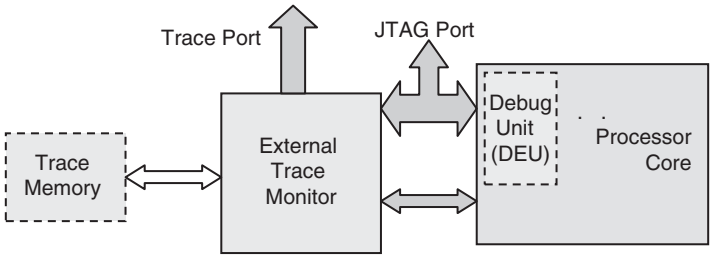

Fig. 2.4 Processor trace

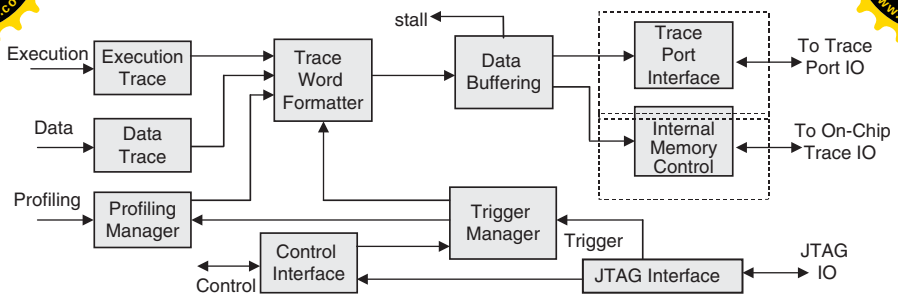

Fig. 2.5 Trace monitor block architecture

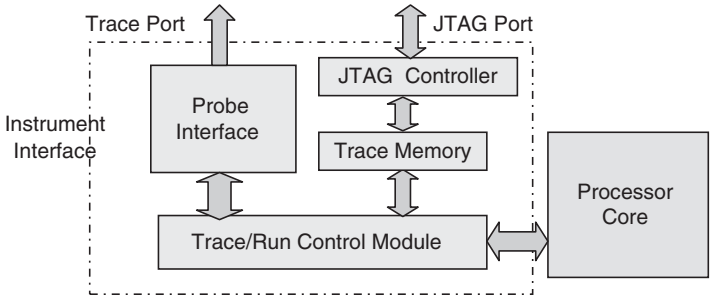

Fig. 2.6 A processor-instrument interface

and become corrupted. The trace monitor control logic allows requests that the processor pipeline be stalled so that no trace information is lost (Fig. 2.6).

The trace monitor allows internal or external trace memory. When data is available from the data buffer, it is written to the internal memory. For external trace, the off-chip trace port logic multiplexes the trace words from the data buffer onto the trace port pins. As in the previous examples, control of the instrument block is handled by a JTAG interface and can be configured for on-chip or off-chip trace storage.

2.5.2 Bus Logic Monitoring

With increasing core density and interconnection of blocks in modern SoC design, monitoring internal bus operations is an important capability to debug the entire SoC. OCP and AMBA AHB are leading on-chip buses in use by many SoC design architectures to communicate between cores. On-chip instrumentation applied to the AHB captures bus activity and allows system diagnostics in real time.

In this case, the instrument connects to the AHB address bus, data buses, and various control signals at the bus multiplexed outputs. In AHB, signals are driven from each master and multiplexed onto a common address/data/control bus

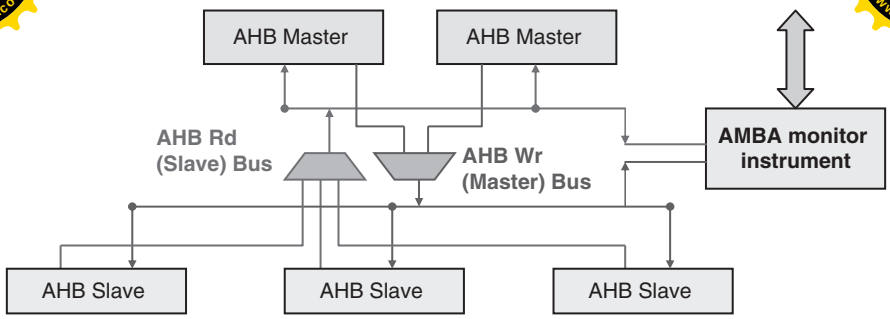

Fig. 2.7 Instrument interface into an AMBA bus

by a multiplexer controlled by the arbiter. The multiplexed output is fanned out to all the AHB slaves. Similarly, data from each slave is multiplexed onto the common bus and sent to the SoC masters. The instrumentation interfaces are configured to receive address/control and data bus data from the currently granted master and addressed slave. The instrument passively collects bus activity and transfers the collected data through a trace port to the external probe interface (Fig. 2.7).

The bus instrument was developed to support a range of single-master and multiple-master systems. Additional signals can be hooked up to any nodes in the SoC, such as interrupt requests, additional AHB status, and processor control signals. The additional signals can also be used to compare and recognize specific on-chip activity outside the AHB bus, and then are transmitted to the probe for triggering purposes. As an example of real-time processing for debug that the instrument enables, the bus monitor allows probing of data in different modes. In the AHB case, data can be probed in two modes. Bus-cycle mode captures all address/control and data signals exactly as they occur per clock on the bus. Bus-transfer mode reduces the delays and latencies between address and data cycles on the bus, by aligning to the same clock cycle, operations that occur in different cycles. This reduces the number of trace cycles that are stored and allows for efficient combination of address-data-control event triggering for trace and monitoring operations. Bus transfer mode is especially effective for bus read operations in which the master transfer operation providing addresses and control and slave response providing data back to the master may be separated by a large number clock cycles of the bus waiting for the operation to complete. As an additional example of trace in-line processing, the trace hardware can be configured to filter out idle, busy, and not-ready cycles where no active data is being transferred. This allows each trace frame to record the critical AHB signals along with additional user-selectable signals and a timestamp to aid in performance analysis.

The host software for AMBA monitoring provides a good example of a specialized debug interface to support bus operations. Bus values can be viewed either numerically or symbolically. The symbolic representation increases the visibility and comprehension of complex bus operation (Fig. 2.8).

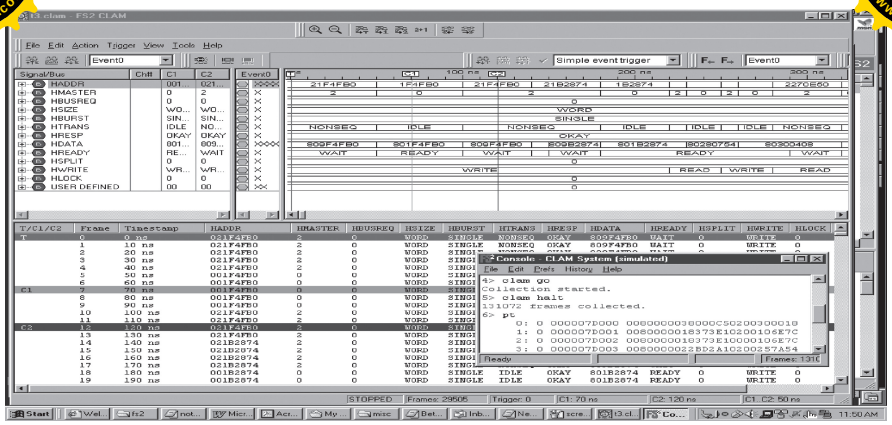

Fig. 2.8 A bus analyzer display

2.5.3 Real-Time Data Exchange

Real-time data exchange is the ability to "exchange commands and data with the application while it executes." This approach to "dynamic instrumentation," called "dynamic variable adjustments" or "dynamic data collection," was introduced by Texas Instruments and is becoming widely utilized. Dynamic data collection refers to the ability to capture specific address ranges of data from the SoC target and present them to the user on the host machine. The data can be "pulled" periodically by instrumentation or on-demand by the user using the JTAG and/or trace port. Pulled data-exchange methods of implementation can include a JTAG command that suspends the processor, reads a range of data values from the target, and passes them to the host via the probe interface.

Debug data can also be "pushed" from the target based on instrumented code to output variables or arrays periodically (i.e., timer interrupt) or from executing a specific location in code – such as when a variable is updated. Pushed data exchange can be implemented based on instructions in the target code, such that a range of data will be copied from memory to the instrumentation trace port. The hardware core and instrumentation block provide an instruction that can write memory to the trace port or a DMA channel configuration that can do a range transfer from memory to trace port. The data format can function as burst mode – first the start and end addresses are sent out (or start address and length,) followed by the data. If the trace port is not available, a breakpoint can be placed in the code and the run control unit fetches the string buffer via JTAG reads.

A third technique is the use of "shadow memory" – an external RAM that is interfaced to hold the same image of values as in the processor's main memory (or cache). Shadow-memory techniques include zero-overhead methods in which the instrumentation is set up with a range of addresses to shadow. When a read or

write occurs in the address range, the instrumentation captures the address and data value and sends it off-chip through a trace port. To a probe which includes a trace memory that allows a real time access to the data.

2.6 Multiprocessor Debug

As more processing elements, features, and functions are simultaneously embedded into the silicon, the emerging level of embedded complexity outstrips the capability of a stand-alone logic analyzer, a debugger, and an emulator-based diagnostic tool. While these tools allow the capture of data off the system data bus, they work only as long as every access (read or write) occurs over the external data bus. For embedded processors and buses with no direct external access, this points to a growing gap in effectively being able to provide the necessary controllability and, in particular, the visibility of the internal operations of a complex system.

The need for improved methods of observing and analyzing embedded processor and SoC operation has increased at a pace at least proportional to the explosive growth in SoC designs and new IP cores. This forces the analysis side of the SoC world into a constant process of catching-up to the designer's ability to add cores and integrate new resources on chip. With an ever-shortening development cycle, and often several generations of products being produced in parallel or rapid succession, standardized embedded tools and capabilities that enable quick analysis and debug of the embedded IP are a critical factor in keeping SoC verification a manageable part of the process.

Before we can implement an on-chip debug system suitable for multicore systems we have to ask the user requirements.

1. Each core and bus must be observable. We must be able to see or reconstruct the program flow of each single core independently as well as of the data flow on the system buses. Also important are signals allowing conclusions to be drawn about power modes, bus accessing modes, and others.
2. It is crucial for system analysis to recognize events that arise from interactions between the cores and buses. A single core on its own is no longer of interest. Rather, events coming from several cores have to be considered. To minimize this challenge, cross triggers must be used, which combine events from different sources and make them available systemwide.
3. The interactions between all SoC components during debug become more complex as more components are involved. A debug system with complex cross triggering is hard for the user to manage. The debugger as a user interface for the complex debug hardware must support the user in its work finding the mistakes or performance bottlenecks. It has to hide the complexity that comes with multicore debugging. We must not forget the user's task is to cope not with the debug hardware itself but with the faulty system.

On-chip instruments (and simulation) play an important part of SoC development and verification flows, providing the ability to analyze what is happening on the hardware itself, during both prototyping and system-level verification stages, and increasingly on the final products themselves. The problem in analyzing information like embedded buses in hardware in many cases hinges on a problem of visibility: *it is difficult to fix what you cannot see.* This visibility problem for the embedded SoC is more complex than can be addressed adequately by traditional on-chip test methods such as traditional JTAG scan, for several reasons:

- Bus operations are multicycle, with signals in a bus cycle becoming active at different times, requiring sequential tracing rather than as a single-cycle snapshot that scans typically provide.
- Bus operation problems are interrelated with the operations of at least two communicating blocks (a processor and memory peripheral, for example). Traditional debug methods, such as halting part of a system for testing, can introduce changes and new variables that interfere with the test scenario and process.
- If problems are intermittent or sparse, then trace operations need to operate in a triggered mode, so information for a given range of bus cycles of interest is captured in real time.

The problem is, to a large extent, a multicore extension of embedded processor analysis, with run control, instruction execution, and data trace as integral parts of processor support. For larger systems with multiple cores, the problem extends beyond processor execution to understanding system operation and communications (Fig. 2.9).

In formal terms, multicore embedded systems present an asymmetric functional test problem. Their controllability is high, because the systems are dominated by programmable processor cores. The observability is low, however, in terms of both the critical signals that are directly available and the amount of embedded logic and internal signals as a ratio of the available IO in which to observe them. Adding dedicated resources and structures that support functional analysis is necessary to increase system observability. This requires a hierarchical focus to the issue of system analysis, starting at the individual core level of debug instrumentation and resources and increasing to a more system-centric diagnostic capability to facilitate increased observability. While embedded debug instrumentation approaches are becoming increasingly common at the core level, system-level diagnostics and analysis at the multicore level have historically been a largely underaddressed area in complex embedded systems.

Single-core approaches for debug and trace often fall short when used with multiple cores and processor interfacing with complex application-specific IP. Increasingly, SoCs integrate multiple types of cores, either for DSP or other specialized processors or for other complex application-specific IP operations for a myriad of functions. These cores may be running asynchronously or with variable or indirect communications with each other, which makes debugging over multicore difficult to correlate. Complicating multicore debug issues further, in many cases,

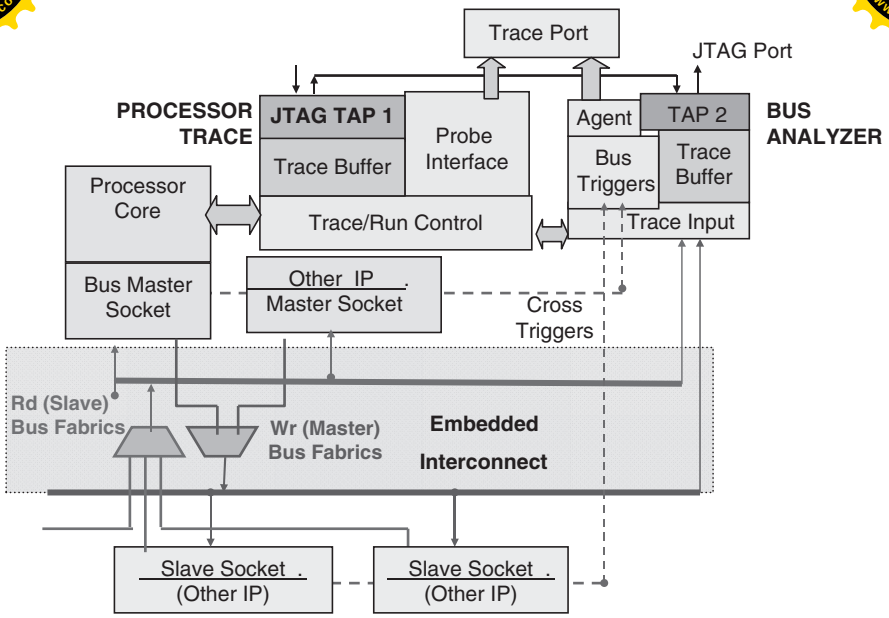

Fig. 2.9 SoC processor and bus trace instrumentsssss

different IP blocks come from a variety of vendors and have different compile and debug environments or levels of debug features. Tasks such as processor interfacing, interprocessor communications, run-time execution and coordination, and data presentation place a significant overhead on the debug requirements for heterogeneous and multicore chips. In these architectures, a range of the instrumentation block(s) must be customized to support the specific verification and debug requirements of both processors on a stand-alone basis and in a multiprocessing configuration. Among the basic requirements, instrumentation blocks must be diverse enough to effectively communicate debug data with their respective cores and have a sufficient common interface to coordinate all their activities. For example, synchronization of all processors in SoC is required in starting and stopping their operations.

Instrumentation solutions for on-chip buses provide a valuable resource for observation in multiprocessor debug. For systems that have multiple processors communicating over a standard bus such as AMBA AHB, access to information such as which processor owns the on-chip bus can provide valuable context as to what the relative communication and stages of processor execution are. With increasingly complex bus architectures being introduced, it is generally agreed that future generations of multiprocessor debug will rely on more extensive tracing and triggering of bus operations to address interprocessor communication issues in conjunction with more specific point solutions for processor-specific analysis.

Looking ahead to more complex systems, instrumentation must have sufficient "embedded intelligence" to interpret information passing between cores, determine what is needed to be extracted for debug, and perform other task-aware debug for

on-chip RTOS or network protocol analysis. Equally challenging is presentation of all the diverse debug information in a coherent, understandable way. As in many areas of complex SoC design, new classes of instrumentation are needed to address diverse debug and analysis requirements of emerging architecture.

Integrating instrumentation into design hardware enables on-chip debug capabilities by providing not only visibility but also control features such as breakpoints for developing and integrating SoC application code. On-chip instrumentation and debug are critical resources to aid in both processor function and performance assessment and effectively evaluating silicon prototypes (for example, programmable logic implementations) and first silicon debug and validation.

There are a wide range of approaches taken for embedded processor debug, several of which are discussed in later chapters. There is no magic bullet instrumentation approach, but rather a number of commonly required capabilities needed to provide a robust debug solution. On-chip instrumentation enables a range of widely used best practices in debugging and interfacing embedded information, including data tracing, triggering run control that has proven analysis benefits. Implementing an instrumented interface on SoC designs offers distinct advantages in efficiently implementing run control, real-time instruction and data trace information, RTOS support, memory subsystem, breakpoints, and watch-points, to name just a few.

An instrumentation implementation that is scalable and configurable to map to a range of instrumentation requirements on the SoC allows support of user-definable general-purpose or application-specific features. Instrumentation hardware should be a synthesizable solution, both to facilitate integration into a range of target platforms and to load instrumentation into hardware emulators to provide a synchronized method of loading and debugging code and functionality in a pre-silicon environment. Synthesizable instrumentation solutions also allow their integration into high-end FPGA parts. In many cases, programmable devices incorporating instruments through their system interfaces are ideal for pre-silicon verification.

One of the most important features of instrumentation capability is support of collection and streaming of data off-chip to a logic analyzer or other trace postprocessing environments, which integrates trace processing along with a low overhead control interface.

Support for complex event recognition and triggering capabilities is also required to provide a robust level of control and monitoring of operations. Complex address and data triggers, coupled with bus trace, can be used for a range of operations from multiprocessor synchronization to debugging device drivers. Having a source-level debug GUI coupled with the instrumentation complex triggers may rapidly uncover execution errors and problems such as improperly defined variables. By coupling timestamps to trace data, complex triggers can be used to provide a range of performance analysis information.

The ability to interpret debug information is essential. A documented API allows fast, efficient porting of instrumentation to customer-specific GUIs. Scripting of validation and manufacturing tests is a useful means for efficiently leveraging embedded instrumentation. Host debugger environments for an instrument solution benefit from command-line interfaces that allow effective script file usage.

PDF-XChange Viewer Click to buy NOW! www.docu-track.com

Instrumentation solutions for processors should support complete integration into a source-level debug interface to provide access to disassembly information needed to understand the context of application-related problems.

Instrumentation extensions can be customized in a range of areas for the system to debug application-specific IP. Their value in providing otherwise unavailable visibility in a range of internal system characteristics, including code coverage, RTOS task analysis, and protocol analysis, will only increase with larger, more complex, and increasingly deeply embedded next-generation architectures.

http://www.springer.com/978-1-4419-7562-1

On-Chip Instrumentation
Design and Debug for Systems on Chip
Stollon, N.
2011, X, 244 p., Hardcover
ISBN: 978-1-4419-7562-1