Single Core or Multi Core: Debug Made Easy With Nexus

Neha Jain, Debjit Roy Choudhury, Freescale


With the advent of newer technologies, chip technology is getting more and more complex. Nowadays, a large number of features are getting integrated in the same IC. And with increase in complexities of design it becomes highly important to test and debug all the critical features thoroughly. Considering the automobile industry, security and safety are two critical features that require zero-defect testing. The application program involving such features needs to be highly precise. Simulating such critical applications and debugging them in real-time environment helps in delivering quality product to customers. IEEE-ISTO 5001TM-2003, NEXUS, has become one of the most widely used debug standards, especially in the semiconductor industry. The NEXUS standard provides immense debug capabilities to the external debugger, making the whole debug process highly efficient. This article provides an overview of how NEXUS can be put to use to debug complex designs with complex embedded processors. The article starts with usage of NEXUS at processor level and gradually moves onto its integration and application at the SoC level.

NEXUS at a Module Level

The NEXUS standard categorizes debug functionalities into four classes, each higher class supporting features of its lower class. Any NEXUS-capable module supports features of one of the NEXUS classes. Additional features of higher classes can also be supported to add more debug functionality. Details of the NEXUS debug features are described in the IEEE-ISTO 5001TM - 2003 standard.

Figure 1 is the block level representation of a processor capable of debugging using NEXUS. It supports NEXUS class 3 and hence it supports features of NEXUS classes 1, 2 and 3. NEXUS1 provides the basic debug framework. It facilitates only static debugging where the core is halted when debug operation is carried out. It supports

different debug configuration and status registers, instruction/data compare registers and data value compare registers to configure a breakpoint against an instruction/data access or a scenario like change of flow due to branch instructions, instruction completion, return from an interrupt, etc. The debug register configurations determine what type of debugging is to be done – hardware or software. Both the mode of debugging is capable of configuring the debug registers.

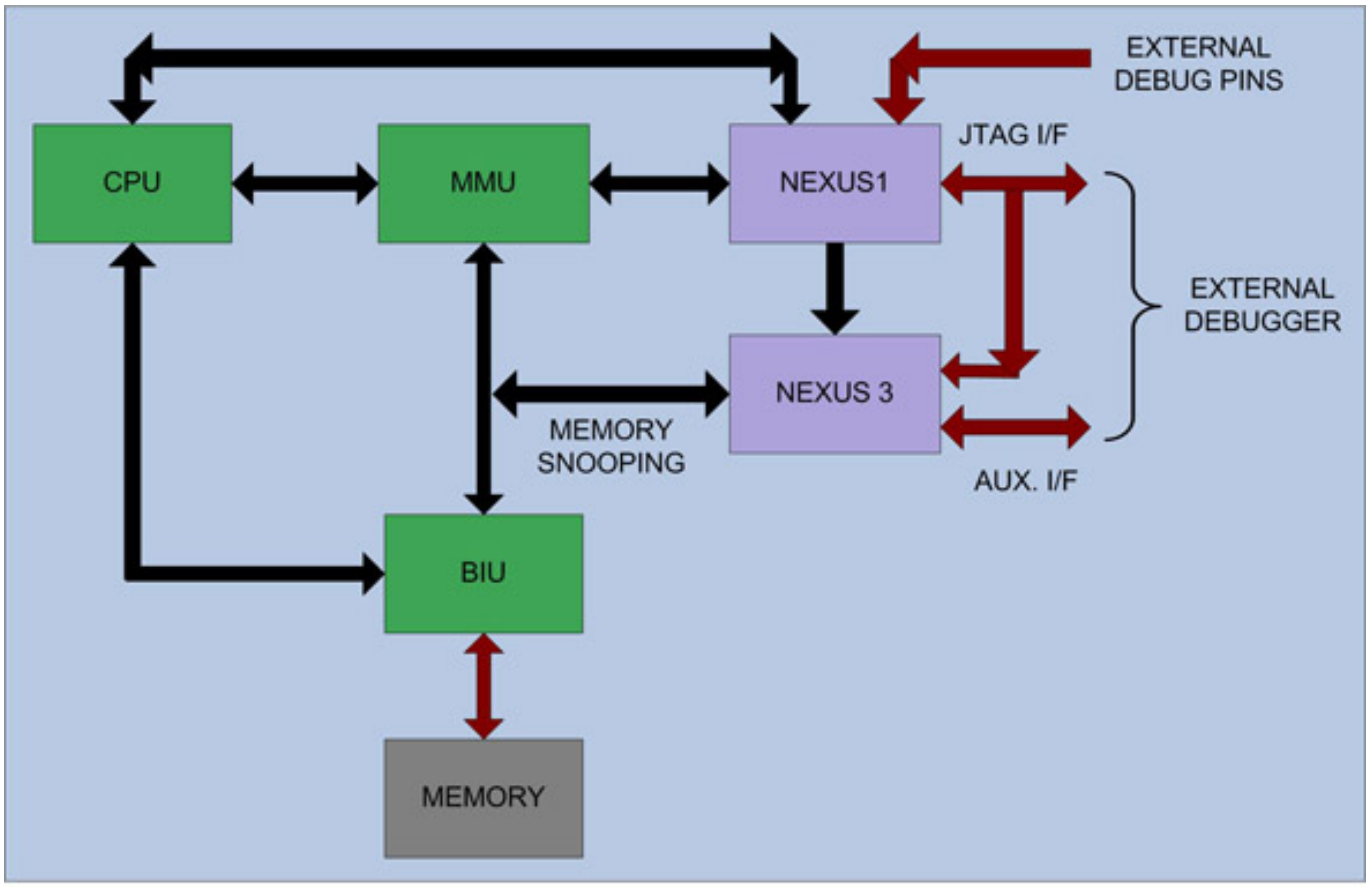

Fig. 1 : Processor supporting NEXUS class 3

Software debugging is the mode of operation where all the programming is done through software instead of any hardware interface (like JTAG). In this mode, generation of a debug event doesn't make the CPU (processor) to go into debug mode; instead, an interrupt gets generated, if enabled -- otherwise the user code keeps on running, with the related debug events' status getting logged in status registers. To enable such a debug operation, following steps are followed:

1. Configure a breakpoint by writing the instruction address into the instruction compare register, and enable the debug event in the debug configuration register through software.

2. An address match between the instruction address on the instruction bus and the instruction address compare register generates a debug event, and its status gets reflected in the status register. This address match happens once the user code that needs to be debugged starts executing.

3. If enabled, a debug interrupt is generated and the corresponding interrupt service routine is executed.

Similar is the case for placing a breakpoint against a data address or data value on the data bus (this happens when a load/store instruction is executed) or other types of debug event scenarios; only the programming of debug registers differs from one debug event to another. Debug interrupt can also be generated unconditionally by giving a trigger to external debug pins. As discussed later, these pins can be used for cross-triggering in a multi-core environment.

Since all configuration is software controlled in software debugging, it limits the role of the external debugger on program flow. External debugger can gain access to debug facilities only when external debug mode is enabled. This is the mode that a debugger generally uses to debug a user code.

Static Debugging Using Single Stepping

Once the external (hardware) debug mode is enabled, the external debugger gains access to the entire debug registers using the JTAG interface. It uses the IEEE 1149.1 Test Access Port (TAP) controller and pin interface to do all register configurations serially. To enable external debug mode, the following steps can be followed:

1. NEXUS TAP controller is selected via JTAG interface (TCK, TDI, TDO, TMS) and the Instruction Register (IR) of the TAP controller is configured to select the main debug register that will enable hardware debug mode. This constitutes the Instruction Register (IR) cycle of JTAG state machine.

2. The Data Register (DR) cycle of the state machine is then traversed to write a particular data into the selected data register that enables hardware debug mode. The Update_DR state of the JTAG state machine will then enable external debug capabilities. Steps 1 and 2 are then repeated several times to configure various debug registers as required.

3. After exiting from debug mode, the program is allowed to run and whenever a debug event is triggered (depending on debug register configurations from steps 1 and 2), the CPU enters debug mode and the corresponding flag is set in status register instead of generating an interrupt, as in case of software debugging. The status register needs to be cleared before exiting from the debug mode; otherwise the CPU will once again re-enter debug mode. Throughout the debug mode, the CPU is halted and no instruction execution happens. This is static debugging.

4. While in debug mode, a single instruction can also be executed through single step-in process. The IR register is loaded with an instruction opcode, and the CPU is allowed to come out of the debug mode, execute the instruction and re-enter the debug mode. This is typically used in accessing the memory or to check the status after the execution of a particular instruction while in debug mode.

As evident from the above steps, NEXUS class1 features lack real time debugging; debugging is done by configuring breakpoints or watchpoints which doesn’t provide

any information related to sequence of operations. It’s not possible to see how the instructions are executed and how the different registers are getting updated as and when the actual user code gets executed. To eliminate such problems, features from higher order NEXUS classes are used. As per the example (Figure 1), NEXUS3 is used, which includes features of class 2 and 3 on top of the basic debug framework provided by class1.

Real Time Debugging Using NEXUS Class 2 and Above

The most important feature that all NEXUS classes above class1 provide is real time debugging using message-based communication with external debugger. It has the capability to send out different trace messages in real time (i.e. CPU is not in debug mode) to external debugger using which the debugger can create the complete flow of the CPU operations and do the debugging much more efficiently. It has a message transmitter that formats messages (containing CPU related information) for transmission on the Nexus auxiliary port. The auxiliary port is configurable, and it comprises of input pins – MDI, MCKI, MSEI, EVTI, and output pins-MDO, MCKO, MSEO, and EVTO. Details of the pins and the registers can be found in the IEEE-5001 standard. NEXUS3 also has a separate set of registers to enable different trace messages.

The NEXUS3 operation is described below:

1. First, breakpoints are configured in the NEXUS1 debug registers either through software or hardware, as discussed previously.

2. NEXUS3 TAP is then selected by configuring the IR register (using the IR cycle of the TAP controller state machine) through JTAG.

3. By traversing the DR cycle of the TAP state machine twice, the particular register inside the NEXUS3 is selected and then it is configured with configuration data. In the whole process, CPU is not required to be in debug mode. Since there is only one TAP per module, any NEXUS3 register can be accessed as described in steps 2 and 3.

4. Any event of interest, as configured in NEXUS1 registers, generates a watchpoint to notify NEXUS3 about it. Depending on its type, NEXUS3 receives and embeds this watchpoint event into a watchpoint trace message (WPM) and sends out to an external debugger via the auxiliary port.

5. A watchpoint can also trigger other types of trace messaging (e.g Data and Program Trace, etc.). If multiple trace messages are enabled corresponding to a particular debug event, they are sent out according to a pre-defined messaging priority. Details about different message types can be found in the NEXUS standard.

With increasing frequency of CPU, the amount of debug information inside the CPU is much more than the bandwidth of auxiliary ports. A number of solutions are available to overcome this limitation. A few may be:

1. To have an internal message queue of pre-defined depth to store trace

messages. Since depth can be infinite, there are chances of overrun. In that case, some messages will be lost and the corresponding information is also sent to the debugger through error messages.

2. To halt the CPU and stop real-time debugging when the message queue reaches a particular limit. This prevents loss of any debug messages due to FIFO overflow.

3. By using a high speed bus to deliver debug information to an external debugger which matches the rate of generation of data inside CPU.

DMA Operation Using NEXUS

NEXUS3 can also snoop the memory bus, as shown in the Figure 1, and it can send out related information, whenever there is a memory access, through messages if tracing is enabled. In addition, NEXUS3 can act as a master to access the external memory. In such a case, the memory address is configured in a NEXUS3 register, and then a memory transaction can be made without the intervention of the CPU. It uses cycle stealing method to do the DMA transaction. This helps in accessing any memory mapped space, including the peripheral register space at system level.

As evident from the above description of the complete system, the CPU need not be put into debug mode to do any NEXUS3 configurations by the external debugger. Also, using the trace messages, generated by watchpoints or through memory snooping operation, the external debugger can easily create the sequence of events that the CPU is generating. That is how NEXUS3 helps in having a real time debugging environment.

This is how NEXUS, in general, is used for debugging the operation performed by a particular module. At system level, multiple modules can have NEXUS capabilities. Arbitration logic is required to initiate debugging capabilities in all these modules. Other than that, the basic operation remains the same. The following describes how NEXUS is used in a SoC (System on Chip) having a single processor (core).

NEXUS in a Single Core SoC Environment

At SoC level, multiple modules can have the NEXUS debug capabilities. Figure 2 is an example of a single core SoC environment where the core (processor) and the eTPU (Enhanced Time Processing Unit) have NEXUS support. Since the external debugger can talk to NEXUS enabled modules only one at a time via JTAG interface, there is a need for arbitration logic among the auxiliary ports of these modules. NPC (NEXUS Port Controller) holds the key to controlling different NEXUS modules.

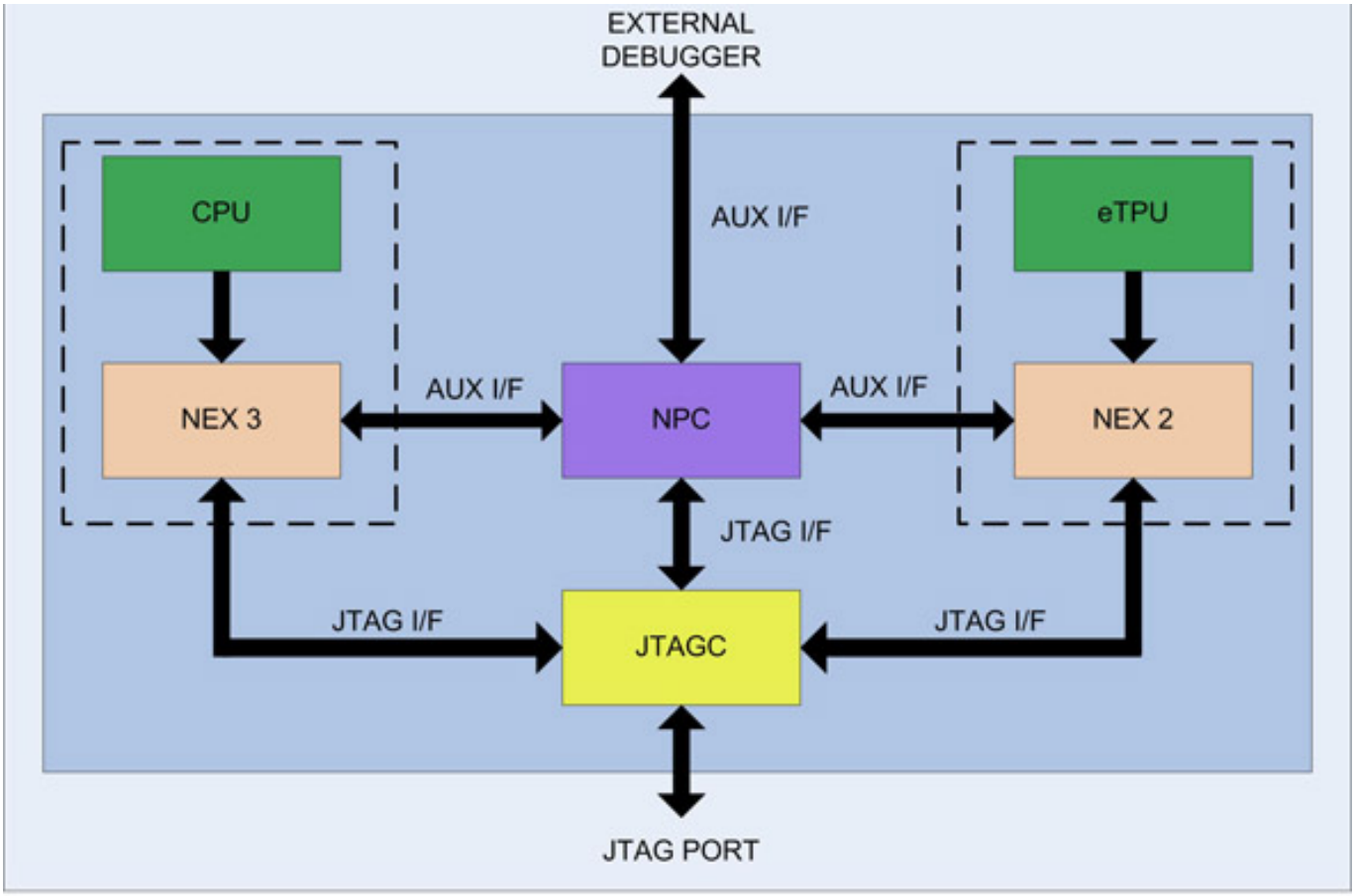

Fig.2 : SoC environment with different NEXUS modules

NPC allows any one auxiliary port of the NEXUS modules to interact with the outside world. The NEXUS blocks implement a request/grant scheme in order to avoid bus contention. All request signals are input to the NPC block. A grant signal is output to each of the blocks and asserted for the block with the highest priority request. Requests are only granted when no busy signals from higher priority NEXUS blocks are asserted. The block must start driving the auxiliary output pins and its busy signal on the clock following the grant. The block that is given the grant owns the port until it negates its busy signal. Devices without the grant negate the auxiliary output bus. Also, since all the NEXUS capable modules will be using the JTAG interface for configuration, as discussed previously, we need to have a mechanism to select different JTAG TAPs one at a time. A JTAG controller (JTAGC) can be used to select a particular TAP controller state machine. Each one of the NEXUS modules and the NPC has a TAP state machine.

In order to configure a particular register in any one of the NEXUS modules and make it active for debugging, follow these steps:

1. A register in JTAGC is configured to select the TAP of the module that needs to be configured. So, any serial operation on the JTAG interface now gets directed to/from the selected module. This allows the interface to all of these individual TAP controllers to appear to be a single port from outside the device. The JTAG input pins go directly to each TAP state machines, but the output pins are all multiplexed inside JTAGC and then sent out. The TAP state machines of JTAGC and the selected

NEXUS module are always synchronous to each other.

2. The NEXUS registers are then configured as described previously. The number of IR and DR cycles of the TAP state machine is dependent on the class of NEXUS that is getting configured. The rest of the NEXUS debugging remains the same as in module level. Whichever NEXUS is selected by the NPC, messages from that module will be sent out to debugger through auxiliary port.

3. Before NEXUS starts operating, the NPC needs to be configured and enabled so as to allow the transmission of messages from different NEXUS supporting modules. NPC also has a TAP and it needs to be configured after selecting it through JTAGC.

4. If the register configurations of any NEXUS need to be changed, the same procedure of TAP selection through JTAGC is followed. Other than port arbitration, NPC drives the clock output pin of the auxiliary port and also has some additional functionality, (it controls device-wide debug mode, controls sharing of auxiliary port pin, etc.). In a multi-core environment, the NPC configuration may change depending on the modes that it supports. Two of the widely used modes are described below:

NEXUS in Multi-core SoC Environment

In a multi-core (say, dual core) environment, core operations can be either synchronous or asynchronous to each other. In synchronous mode, both the cores are in a sort of lock-step mode where both execute the same code. The outputs of each core are compared to check proper functionality. Lock Step mode finds special application in safety related SoCs, especially in the automobile industry. In asynchronous mode, the cores operate independently; both of them are capable of running two completely different codes. In order to handle these two dual core configurations, two different types of debug arrangements are required.

Figure 3 shows a typical arrangement of two cores in synchronous or parallel mode. Besides Core0 and Core1, there can be other NEXUS enabled modules which are not shown in Figure 3 for the sake of simplicity, but the arrangement will be similar to Figure 2. The JTAGC is also omitted from Figure 3 because its operation is similar to any SoC.

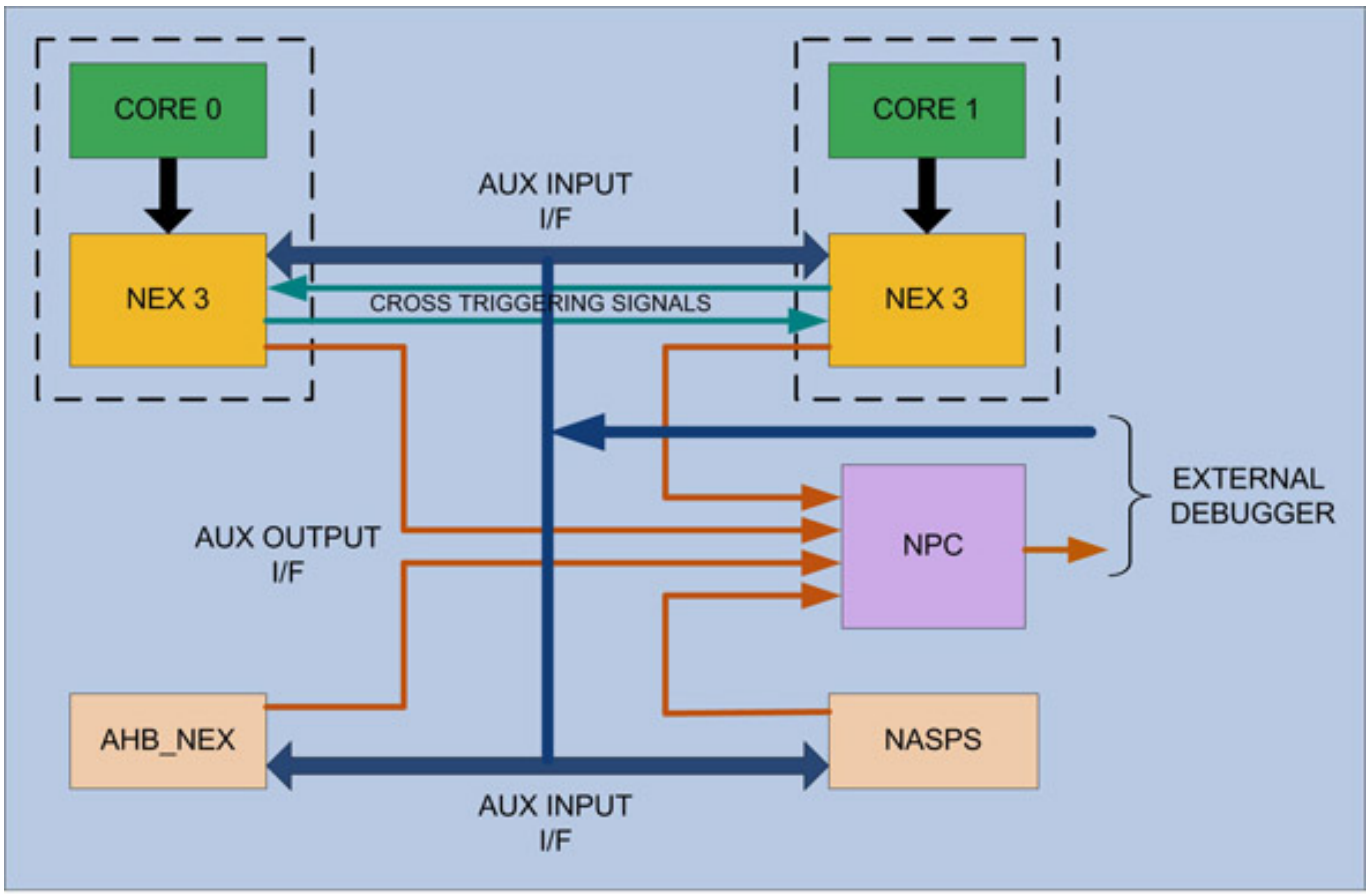

Fig. 3 : Dual core environment (parallel mode)

Since, in parallel mode, both cores can execute instructions independent of each other, their NEXUS configurations can be completely different from each other. Both of them may be configured to send trace messages over the single auxiliary port, at the chip level. So, the NPC needs to take both the cores' auxiliary outputs and then direct them to the external debugger only one at a time. NPC can use a priority algorithm to decide upon which core to be given grant when both of them are requesting for the bus access. The algorithm can also be a part of the core in which case the request from the cores will also assert the priority.

In parallel mode, it might also be required to trigger one of the cores to go into debug mode when the other core is either in debug mode or in run state. A similar requirement can also be that one core is able to trace the sequence of events in the other core and send out trace messages accordingly. In all these cases, cross-triggering can be used where auxiliary output pins of one core can be connected to auxiliary input pins of another core and vice versa. And, one of the external debug pins can be connected to the auxiliary output pin (EVTO) to put the other core in debug mode. This cross-triggering feature helps in debugging scenarios where both the cores interact with each other.

Sometimes, an arrangement can be made even at the system level where TAPs of both the cores are connected back-to-back. Such a feature is called multi-tap. It is used to shift data into the debug registers of both the cores in one go. This type of an arrangement helps in simultaneous debugging of both the cores without changing TAP selects in between debug operation. A completely separate TAP

select from the JTAGC can be used to tie multiple TAPs together.

In addition to the above modules, one can also have some extra NEXUS modules for the bus masters as well as for the bus slaves which don’t have in-built NEXUS capabilities. For example, one can have an AHB_NEXUS that snoops the AHB bus between the AHB masters, say DMA, and the crossbar, AXBS. Or, we can have a Nexus SRAM Port Snooper (NASPS) that sits on the slave side between the crossbar and the SRAM controller. These modules won’t be able to debug the internal functionality of the masters/slaves, but they can send out to the debugger some important trace messages related to bus accesses. Such Nexus modules generally have very limited functionalities like data trace message, watchpoint message and error message tracing. In such cases, NPC can be designed to play the role of a central controller for all NEXUS operations. The TAP controller might be present inside such modules or may be implemented externally. The requirement varies from one SoC to another. Figure 3 gives a basic arrangement of such modules in addition to the existing ones.

In the other type of dual core configuration (i.e. synchronous or lock step mode), both the cores are coherent to each other. If there is any difference in their execution, the error is flagged out. In such a case, there is no need to send out trace messages from both the cores; enabling a single core for the debug operation is sufficient.

As shown in Figure 4, the auxiliary ports of both the cores are multiplexed and then fed to the NPC. The select signal is configured through JTAGC. Generally it selects Core0 but it can be changed. Since both the cores are coherent to each other, there is no need of cross-triggering signals. Multi-tap feature is also not required for lock-step mode. The remaining configuration remains the same.

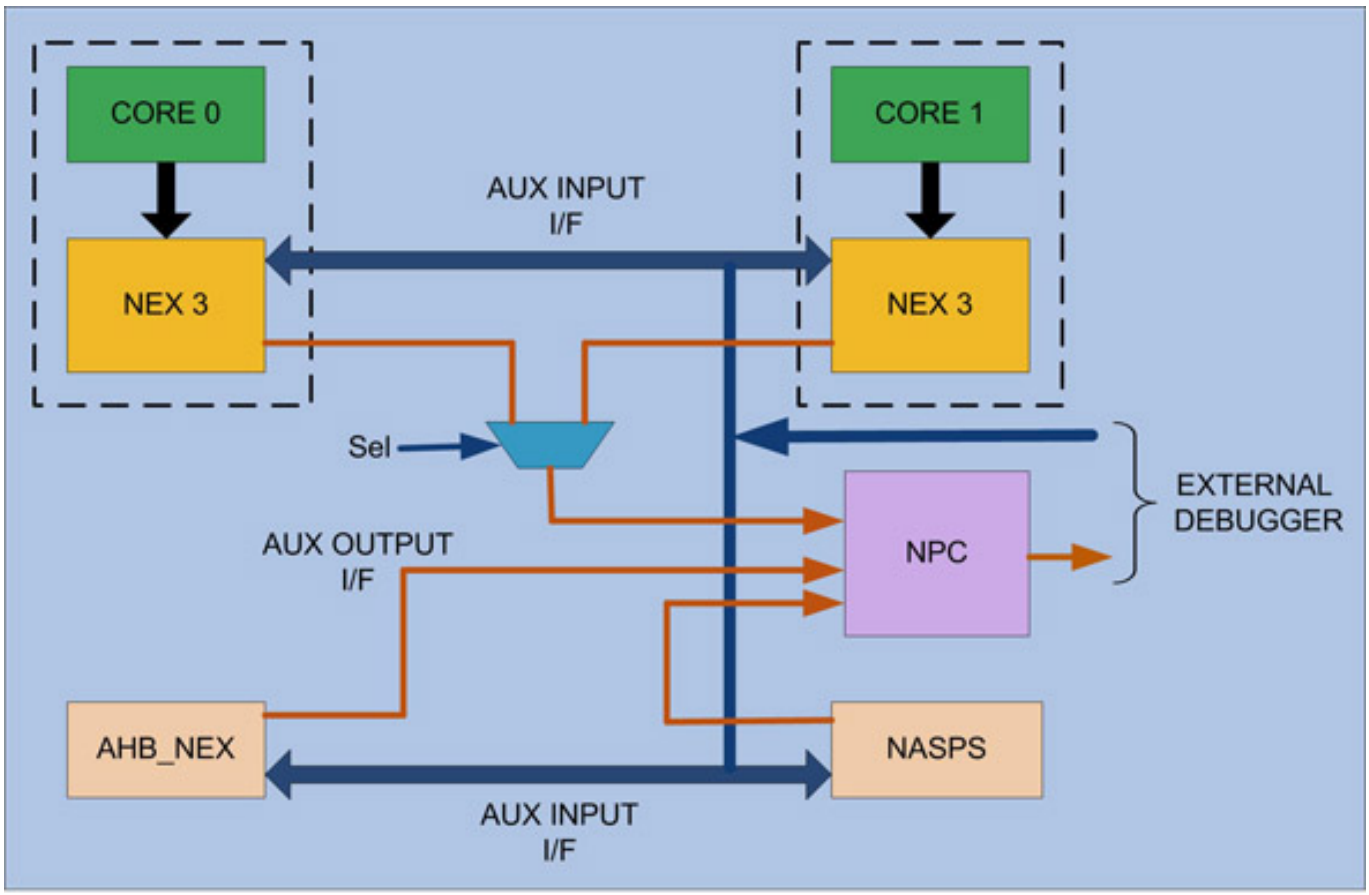

Fig. 4 : Dual core environment (lock - step mode)

With the increase in complexities of the SoCs, the debugging operation also gets complex and grows the need for increased visibility into the design. More and more NEXUS-driven modules are introduced to ensure the proper functioning of different modules. But on a broader scale, the overall NEXUS operation remains the same; it is only the configuration and the control of different modules that changes from one SoC to another.

References
1. http://www.nexus5001.org/ [1]

Source URL (retrieved on *02/01/2015 - 8:10pm*):
http://www.ecnmag.com/articles/2011/05/single-core-or-multi-core-debug-made-easy-nexus

Links:
[1] http://www.nexus5001.org/