3-2018

The Design of a Debugger Unit for a RISC Processor Core

Nikhil Velguenkar
nv8840@rit.edu

The Design of a Debugger Unit for a RISC Processor Core

by
Nikhil Velguenkar

Graduate Paper

Submitted in partial fulfillment
of the requirements for the degree of
Master of Science
in Electrical Engineering

Approved by:

Mr. Mark A. Indovina, Lecturer
Graduate Research Advisor, Department of Electrical and Microelectronic Engineering

---

Dr. Sohail A. Dianat, Professor
Department Head, Department of Electrical and Microelectronic Engineering

Department of Electrical and Microelectronic Engineering
Kate Gleason College of Engineering
Rochester Institute of Technology
Rochester, New York
March 2018

To my family and friends, for all of their endless love, support, and encouragement throughout my career at Rochester Institute of Technology

Abstract

Recently, there has been a significant increase in design complexity for Embedded Systems often referred to as Hardware Software Co-Design. Complexity in design is due to both hardware and firmware closely coupled together in-order to achieve features for low power, high performance and low area. Due to these demands, embedded systems consist of multiple interconnected hardware IPs with complex firmware algorithms running on the device. Often such designs are available in bare-metal form, i.e without an Operating System, which results in difficulty while debugging due to lack of insight into the system. As a result, development cycle and time to market are increased. One of the major challenges for bare-metal design is to capture internal data required during debugging or testing in the post silicon validation stage effectively and efficiently. Post-silicon validation can be performed by leveraging on different technologies such as hardware software co-verification using hardware accelerators, FPGA emulation, logic analyzers, and so on which reduces the complete development cycle time. This requires the hardware to be instrumented with certain features which support debugging capabilities. As there is no standard for debugging capabilities and debugging infrastructure, it completely depends on the manufacturer to manufacturer or designer to designer. This work aims to implement minimum required features for debugging a bare-metal core by instrumenting the hardware compatible for debugging. It takes into consideration the fact that for a single core bare-metal embedded

systems silicon area is also a constraint and there must be a trade-off between debugging capabilities which can be implemented in hardware and portions handled in software. The paper discusses various debugging approaches developed and implemented on various processor platforms and implements a new debugging infrastructure by instrumenting the Open-source AMBER 25 core with a set of debug features such as breakpoints, current state read, trace and memory access. Interface between hardware system and host system is designed using a JTAG standard TAP controller. The resulting design can be used in debugging and testing during post silicon verification and validation stages. The design is synthesized using Synopsys Design Compiler targeting a 65 nm technology node and results are compared for the instrumented and non-instrumented system.

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this paper are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other University. This paper is the result of my own work and includes nothing which is the outcome of work done in collaboration, except where specifically indicated in the text.

Nikhil Velguenkar

March 2018

Acknowledgements

I would like to thank my advisor, Mark Indovina for guiding me throughout the project, and also during the coursework at RIT which helped me build practical design concepts along with theoretical approach. Through his courses and labs I was able to convert complex design ideas into working implementation which I take with me as a valuable skill learned at RIT.

I would also like to thank all my friends at RIT who helped me during the project with their feedback.

Contents

List of Figures

List of Listings

List of Tables

Chapter 1

Introduction

Embedded Systems Design consists of both hardware and software, commonly known as firmware, that mostly operate and control in a standalone manner. The Embedded Systems market has seen a significant increase since almost every electronic device includes an microprocessor. According to the Zion market research report, "*Embedded Systems Market (Embedded Hardware and Embedded Software) Market for Healthcare, Industrial, Automotive, Telecommunication, Consumer Electronics, Defense, Aerospace and Others Applications: Global Industry Perspective, Comprehensive Analysis and Forecast, 2015 - 2021*" embedded systems market would generate a revenue almost double by 2021. This would bring new products for various applications from small sensor interface to high end low power machine learning based smart devices. With Internet of Things (IoT) expanding, many applications mostly use low power bare-metal sensor interfaced systems. This further adds to the growth in bare metal embedded systems. Even for high end systems like phones and tablets, some low power cores are commonly used for interfacing and controlling various sensors such as a gyroscope, accelerometer, and many more. These cores are always powered on, running bare-metal software for reduced complexity, and communicate

by interrupting the main Operating System (OS) or Real Time Operating System (RTOS). Increased market demand for devices providing complex, real-time performance resulted in an increase in hardware complexity. Hardware designs included the use of accelerators, co-processors, additional hardware for wake-up and sleep to control power consumption, and complex firmware design acting as a controlling mechanism. As the complexity increased, many iterations were required for fine tuning hardware cores and embedded software during development cycle at various stages. Hence to perform these tasks efficiently over many iterations there is a need to test and verify such designs. Design Instrumentation includes modifying hardware and firmware which could add some debugging capabilities that later can be used for debugging in post silicon verification and validation stages. For complex bare-metal hardware software embedded system design; debugging in post-silicon verification, and validation stages is the major challenge and to carry out these steps successfully hardware should provide a transparent observation and control of internal signals of the chip from an outside interface. This projects aims to design a Debug Module specifically targeted towards a bare-metal embedded systems that would provide a simple and correct view of internal signals, thus helping to analyze and debug both hardware as well as software. Bare-metal embedded systems do not use an OS; since there is no debugging support from the OS, debug support should be handled by instrumenting hardware in the manner that is compatible for debugging using a host computer interface. For implementation the design uses the Joint Test Action Group (JTAG) standard Test Access Port (TAP) for communication with the chip and open source AMBER 25 core (A25), available from the Open Cores Community, that is instrumented to read write debugging information. With many new micro-architecture implementations of various architectures, and System on Chip (SoC) designs targeting different applications, it is necessary to research and document new debugging infrastructures which are required for controlling, observing

and validating those features thus reducing the gap between debugging infrastructure and corresponding hardware designs. This in-turn would reduce bugs in silicon and effort put into re-spins. A proper and powerful debugging infrastructure will help designers to design first time right silicon. This chapter further explains the need for debugging instrumentation and evolution of debugging practices. Later, the paper covers a research literature survey and describes the Design for Debug methodology in Chapter 2. Chapter 3 goes through various debugging infrastructures. Chapter 4 discusses the design and implementation in this project. Chapter 5 discusses the results of the implementation and Chapter 6 concludes the project and lists its future scope. Finally, the Appendix includes listing of source code for designed Debug Module.

1.1 Need for Debugging Instrumentation

As complexity in SoC and Application Specific Integrated Circuit (ASIC) chips increases, hardware testing methodologies for observing and controlling internal state and signals are becoming more and more challenging. With addition of various new features and capabilities for low power, high performance, complex system integration, use of co-processors and so on, testing and validating these features while keeping the developmental cycle short can only be achieved by standardized debugging tools and capabilities which are capable of providing quick insight about the Design Under Test (DUT). This helps in validation of the complete system and identifying design faults that can be traced to the root cause. Today's Embedded Systems consists of more than one cores, co-processor, hardware Intellectual Property (IP), memories, sensors, all interfaced with one another via a bus or more than one buses, hence it is important to have good controllability and observability mechanisms of internal signals for testing and debugging purposes. Secondly, it is impor-

tant to understand that if there are bugs, we cannot fix anything that is not visible[2]. Graphical waveform based simulation is the best way to provide very high observability and controllability, but with increase in complexity simulation based approach is very time consuming and tedious. It is also not realistic to run only simulations for a large processor with intensive test cases and scenarios, since this would take a lot of time. Also for debugging software observing waveform is very tedious approach which will result in longer developmental cycles. Another major reason is post silicon validation where simulation of netlist is not only time consuming but also tiresome in resolving and tracing proper signals so as to obtain proper internal view. Core instrumentation for debugging is an approach which reads CPU (Central Processing Unit) state by controlling program flow based on breakpoints. This provides a good insight into the system for verification, testing and firmware debugging of the design. Downside of Core Instrumentation is it increases chip area of the hardware but to its advantage, it can be built generic and re-used at various stages of the design like pre-silicon debugging as well as post-silicon validation and debugging. Leveraging on other debugging and validation methods such as use of Logic Analyzers which captures signal trace based on triggers, FPGA (Field-Programmable Gate Array) based emulation for testing ASIC, validation and debugging using firmware in post-silicon stage, hardware-software co-simulation which uses hardware accelerator to speed up some simulation process to get a better and fast observability and controllability for internal states and signals, and so on to an great extend uses features instrumented in hardware. An ideal Embedded Systems debugging platform typically involves combination software debugging which can be handled using GNU Debugger (GDB) with software breakpoints and hardware debugging which can be achieved by core instrumentation for run stop approach, reading internal state and tracing signals using trigger events. Software debugging for Embedded Systems is dependent on IDE (Integrated Development Environment) which

to some extent uses features which are instrumented in the core for debugging purposes. Embedded Systems debugging can be divided in two types namely: stop and halt approach, and real time trace approach. For real-time Embedded Systems sometimes it not possible to halt and debug as this might cause the program to malfunction, due to the fact that serving deadlines is one of the important task for real-time Embedded Systems, hence real time trace dump data is read while debugging these systems. Trace dump is the data which captures trace for any configured signal and is stored while program is running without halting the processor. This stored trace data can be accessed over debugging port. This mechanism helps to find what was the CPU state or instructions that were executed over time. Overall these are major requirements for a debugging environment in real-time Embedded Systems which can be addressed via instrumentation of an on-chip debugger unit which communicate with any compatible external device using communication interface such as JTAG, Serial Wire Debug (SWD), etc.

1.2 Debugging Approach

Explained below are various debugging approaches which evolved over time and are currently used by different processor architectures. There are advantages and disadvantages of the various approaches depending upon requirements. Implementing any debugging methodology is not straight forward as it depends upon needs and requirements along with trade-off for performance and accessibility.

1.2.1 Hardware Triggering and Trace

Logic Analyzers are used to capture and trace signals based on trigger events. In this method internal signals are captured based on trigger events and stored in a trace buffer

memory which can then be read through debug port. In some systems a co-processor is used to capture events and collect trace data for required signals; typically, the trace data is sent to a host PC using a high-speed link. Collection of trace data does not stop the processor or deviate away from normal execution of the processor. It starts capturing trace data based on user defined trigger events such as a certain instruction or on a break-point or any other event of a defined signal, this data is then stored in a trace buffer depending upon size of memory available. As on-chip memories were readily available and easily fabricated, trace has become one of the more powerful debugging mechanism. Now-a-days software logic analyzers such as Xilinx Chipscope and Altera Signaltap are being used in FPGA which provides a soft IP core logic analyzer whose trigger can be configured on any event of an internal signal. Note however that instrumenting an FPGA with a soft IP core logic analyzer can significantly increase FPGA device utilization. Similarly, if hardware pins are accessible, external hardware logic analyzer such as Saleae can be used to trace and view signals on external host computer. SoC's and ASIC's many times have a trigger and trace unit internally which can be triggered based on the user defined input or an exception. Depending upon complexity of implementation this unit then captures all debugging data from instruction execution which time-stamped to bus-cycles, this data is stored in a trace buffer which is then compressed into packets and send out through the debugging port interface. Trace data can also be used for bus activity analysis which captures bus activity corresponding to clock cycles. This can then be analyzed for bus usage. This approach requires hardware intensive instrumentation and its complexity increases with an increase in the system complexity.

1.2.2 In-Circuit Emulator

An In-Circuit Emulator (ICE) has long been used for debugging hardware designs and has evolved over time. It typically uses an external hardware device connected to or in place the chip under test. This hardware emulates complete functionalities of the chip under test with many additional features to read its internal state and signals. In the early days of in-circuit emulation, the test processor was replaced with a special debugging version of the processor that has debug logic and many signals taken out of chip in-order to read using an external device. The special version of the processor was created due to the fact that the debugging instrumentation would take a large area in silicon and was to expensive to put into production silicon. These devices were typically very large as special hardware is required to emulate the device under test. This method was very powerful as it would provide a detailed view of internal signals for a chip. In fact in-circuit emulator would address nearly all aspects of the debugging requirements right from stop and read state to monitoring bus activity or trace for the processor. Now-a-days in-circuit emulators with limited features are being fabricated inside chip.

1.2.3 Communication-Centric Debug

This approach monitors the communication of the on-chip interconnect between various hardware modules. Communication between two modules is controlled by stalling communication protocols between two modules and reading the transaction activity on the bus. It can also be used to force particular request for accessing various shared resources or to replay a series of communication events over time. This is suitable for complex multicore network on chip based SoC designs as communication between various resources becomes a critical factor. This approach can also used to monitor bus activity for bus contention

and can be used as a bus analyzer.

1.2.4 Scan Chain Based Debug

Scan chains are connected to all flip flops which are used for manufacturing test for high observability and controllability of the design are inserted into a design as part of an overall Design For Test (DFT) methodology. These scan chains allow controlling internal state of the flip flops in the design during manufacturing test. This scan chain infrastructure can be re-used for accessing various hardware IPs through debugging port. Commonly used debugging ports such as JTAG based TAP or SWD. Scan chains can help extract complete state of an SoC by individually selecting IP through JTAG instructions that connects scan chain in and out path for reading state registers. It can also be used to modify data as required. This approach requires much less instrumentation of the hardware as it reuses the DFT scan chains. Implementation of TAP controller for JTAG instructions to access the scan chains in different modes of operation would consume some extra hardware overhead. This approach operates in debug mode and connects the scan chain to read the current state of the processor.

1.2.5 Run-Stop Based Debugging

This approach is similar to scan chain based with an additional module designed to control the operation of the core using break-points which stall the processor at a pre-defined point to read current state and then resumes operation. Break-points consists of program counter value which are defined by the user during debugging to observe the state of the processor or system during that instant. This approach is widely used but is not suitable in debugging real time embedded systems applications as halting the processor may result in

missed deadlines and would result in eventual malfunction. The Run-Stop method requires some hardware instrumentation to handle breakpoints and halt the processor. Individual IP for implementing control unit for debugging or a co-processor based control mechanism can be used for this approach. The hardware complexity is relatively less as compared to ICE and a slight bit more than scan-chain based approach.

Chapter 2

Design for Debug

As SoC designs are becoming more and more complex, there is need for an efficient means for identifying and debugging bugs. During the preliminary development phase, a design goes through a pre-silicon verification stage, which mainly uses simulation and formal verification tools. During this time visibility of the internal signals is maximum and thus identification and elimination of bugs can be performed easily. The internal state and signals are easily accessible to the designer, and thus during testing various scenarios many design errors and bugs can easily be identified and fixed. In the pre-silicon verification stage, verification is often done against a designed software model, hence the correctness of the Design Under Test (DUT) depends on the accuracy of the model. This is probably best known and widely adopted method for functional verification, but sometimes it might not take into consideration complex synthesized hardware scenarios. Hence this is not the final verification step for the chip to move to production due to limitations such as inaccurate design modeling that fails to take into consideration real hardware scenarios or misses any other relative dependencies. Hence to avoid this, prototypes are built which may represent actual hardware implementation of the synthesized design. This type of testing

is referred to as post silicon validation or testing. This can also be performed using gate level simulation, but simulating at the netlist level takes a lot of time; also locating internal signals is tedious. It is now-a-days critical to perform post silicon validation in order to avoid bugs passing into chips that are manufactured. In post-silicon validation errors or malfunction if occurred can be easily identified but debugging those seems critical as there is limited view for internal signals. To understand bugs, and also to perform root cause analysis to resolve them, a design should be instrumented with various debug points which at a given instance can read internal CPU state, signals or memory data to help designer understand what is going on internally. When designing using such a strategy there must be complete collaboration between the hardware design team and firmware team to identify important design features and means of testing those efficiently. Hence identifying which data to be visible or accessed during debugging is very important. Remote debugging of bare-metal embedded systems is generally categorized into two types: run-stop approach and real-time trace. Run stop approach is when a processor is halted at certain instruction commonly known as break-point and processor state is read through a JTAG port to understand was the operation performed as expected. Real-time trace approach is preferred for debugging real time systems as it doesn't halt time processor. This process consists of capturing debugging data in real time which can be read through a JTAG port or any other protocol. For real-time bare-metal systems with embedded processors parameters such as processor state, its internal register values, values in different memory locations, and logging program flow by performing instruction trace are some of the important points which are to be monitored while debugging. This is explained further in following sub-sections such as communication interface with the system, hardware instrumentation and firmware hooks, using which remote debugging for bare-metal real time embedded systems can be performed.

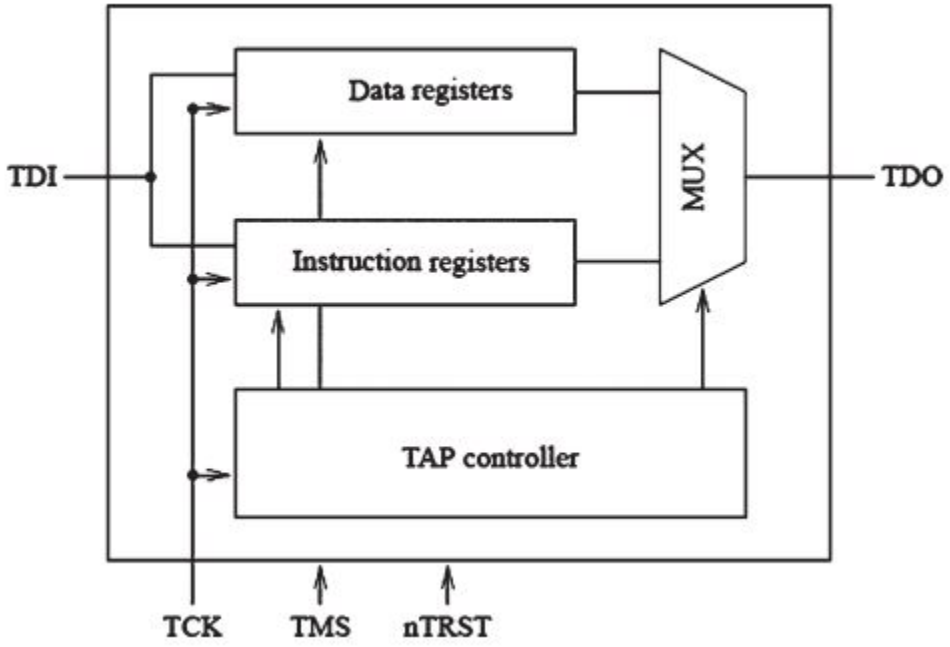

Figure 2.1: JTAG TAP Overview [2]

2.1 Communication Interface

The communication interface is a point of contact for an external device to talk with the internals of the hardware. Using this interface, required data inside the chip can be read by an external host for further debugging and analysis of currently executed operation or the one's executed previously. Widely accepted standard for such an interface is JTAG which uses a TAP controller defined by IEEE Std 1149.1-2013. A JTAG interface has five pins including optional negative edge triggered reset pin. The pins include clock (TCK), test mode signal (TMS) for changing the mode, test data in (TDI) for sending test data serially, test data out (TDO) for receiving output serially and negative edge triggered reset (nTRST). The TAP controller is implemented as a state machine as defined in the IEEE standard shown in Figure 2.1.

As shown, the JTAP TAP includes two main registers: instruction register and data register. For the instruction register and data register, the values are shifted serially in using shift IR or shift DR mode and latched into the register in update DR or update IR mode. There are also proprietary standards which are followed by various chip manufacturer such as ARM, Atmel and so on, for communication and debug interface. Some of the commonly known type are ARM's SWD and Atmel's Program and Debug Interface (PDI). Both of these are two wire interfaces with clock and data pins; as compared to JTAG which is a five wire interface, this indeed helps in reducing external pin count. ARM's CoreSight debugging infrastructures offers SWJ-DP which is a common interface for JTAG and SWD compatible target device. As only the JTAG interface is open and widely documented in the public domain this project uses the JTAG defined TAP controller as a communication interface with the internal system. Figure 2.2 shown below shows the standard TAP controller state machine as defined in IEEE Std 1149.1-2013.

2.2 Core Instrumentation

Core Instrumentation is a set of features designed into hardware in-order to modify it to read debug data from an external host device. This mainly addresses four items: processor specific controls, accessing processor internal state, read memory data, and signal trace which are required for debugging Real-Time Embedded Systems.

2.2.1 Processor Specific Controls

This method addresses how to control the execution of the processor, e.g when to halt the processor, when to resume the processor, how to change execution of the program. It is implemented by monitoring the program counter value and stalling all the pipeline stages

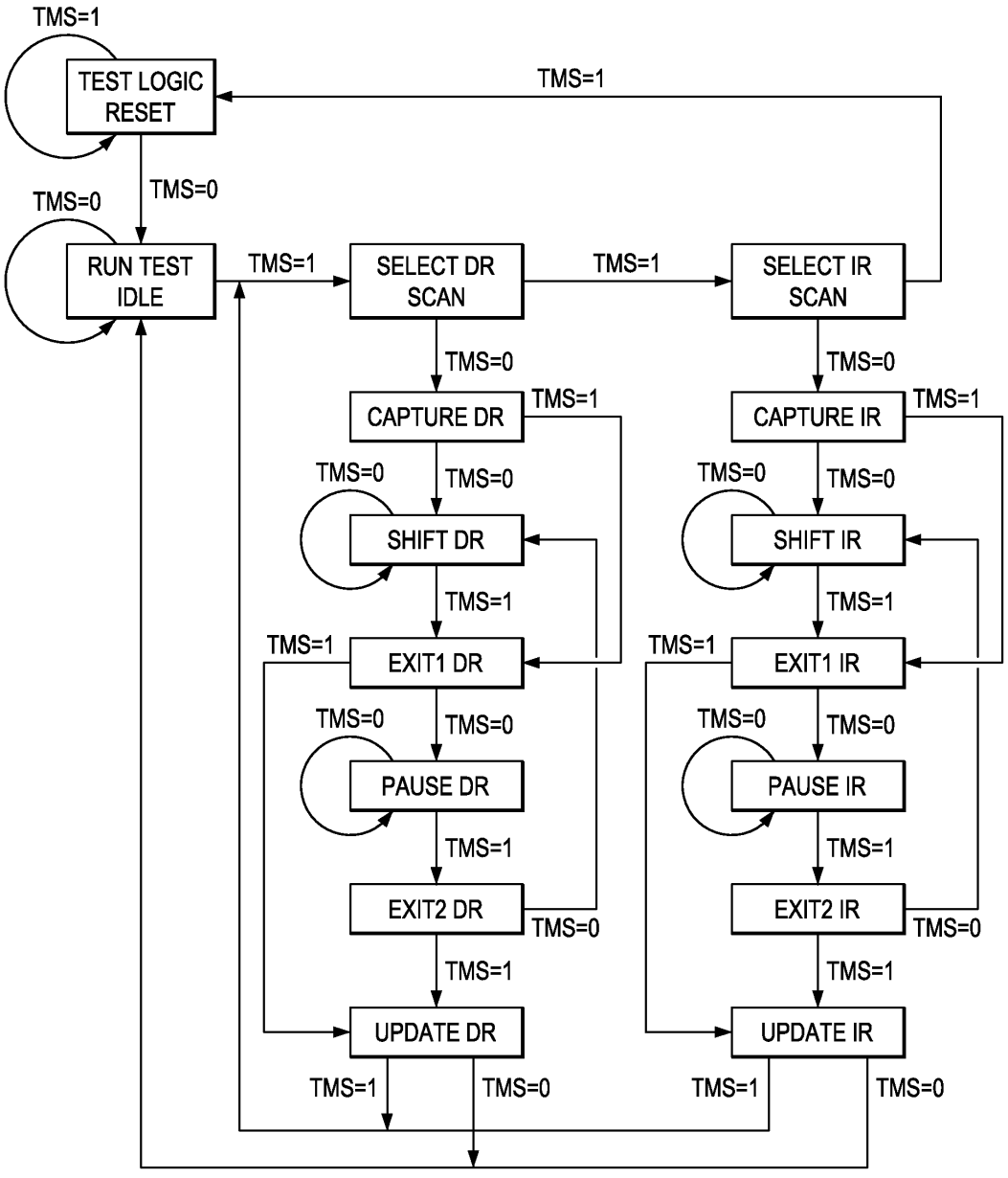

Figure 2.2: Test Access Port Controller State Machine[3]

which in turn stalls the core. Stalling the core can be performed by gating the clock, but there is high possibility that this would result in meta-stability if stalling logic is not performed properly. Alternatively processor can be stalled by forcing it in not ready state. For a pipelined processor it is required to stall all the stages of the processor in order to avoid malfunction.

2.2.2 Reading Processor Internal State

This procedure is targeted to identify what operations are performed by a processor and, are that the results are generated as desired. It is performed by accessing internal state, i.e values from internal registers, status registers and special purpose registers after every instruction execution or at a required program counter value. These results can be used to validate with expected values. This data can be read using an internal register file after halting the core so that no further instructions are executed. Reading values at that instant and then again resuming the processor further would help identify where the malfunction occurred. This type is often classified as run stop approach.

2.2.3 Memory Read-Write

Memory access is one of the major features for the debugger as reading and writing memory locations proves very helpful in debugging for forcing or validating a required value. It is performed after halting core and can be used to access single word or complete block of data. Implemented as a Direct Memory Access (DMA) in which a co-processor or other hardware IP takes control of the bus and communicates with the memory for reading and writing operations when CPU is in halt state. This helps a programmer to force some of the values in the memory to emulate changes or to read memory to check values against

accepted values. For processors using cache, debugging with memory access should make sure that the data in the memory is in coherence with the latest values which might be in cache. Some embedded processor provides a means to disable cache while debugging.

2.2.4 Trace Data

Trace is one of the most important feature used for debugging real-time applications. Many SoC's include a Debug and Trace Unit which handles various trigger events like exceptions from multiple clock domain signals, unhandled exceptions, branching instructions and so on. Based on this trigger, trace probe captures the signals at various clock cycles and stores it in the on-chip trace memory. Trace produces a lot of data in real-time, thus increasing bus activity, and due to this most of the complex SoC debugging infrastructures use a different bus for reading and writing trace information. As this results in huge data on chip, a compression unit is used along with Debug and Trace Unit so as to reduce the size of the data stored and transferred. This data can be read through a communication port such as JTAG, SWD, and so on. Lower end versions of embedded processors often have no or a limited version of the trace functionality such as the ability to only trace instruction execution.

2.3 Firmware Based Debugging

Firmware based debugging has the advantage of re-usability, by performing code modifications as per debug requirements with no impact on hardware units. This type of debugging requires hardware instrumentation in form of various debug registers and debug space in memory which are not accessible during normal operation but can be accessed based in debug mode. One of the widely used registers is the processor status register which con-

tinuously monitors output of instruction executed and updates according to the result. Firmware can be used to monitor status registers to check values from the execution stage and depending upon requirement raise an exception. Firmware accesses hardware resources mainly using two methods: Memory mapped I/O and I/O mapped I/O. Direct Memory Access (DMA) controllers are used for accessing debug memory address space. Timers can be used for counting bus utilization for checking bottle necks, keeping track of cycles for operations in various scenarios. Similar to hardware trace, firmware based Data Watching is a method to watch internal register changes. This is used for watching state changes or branching events which can be logged and later read from the memory. State data along with a counter for counting number of cycles for each state can be used for better observability. This approach has an overhead of more memory usage and consuming vital CPU resources, but requires less hardware space. The debugging data stored can be read through communication interface such as JTAG. Like all other approaches, this one also needs a good collaboration between hardware designers and test engineers to identify hooks for various cases and allotment of memory map.

Chapter 3

Hardware Instrumentation For Debugging

Hardware instrumentation for various micro-controller based platform such as x86 from Intel, Cortex series from ARM, AVR series from Atmel, MSP 430 from Texas Instruments, PIC from Microchip, and so on, have different implementations for various debugging features. Although not all the implementations are openly accessible, instrumentation with respect to features point view in ARM's CoreSight is very powerful, widely used, and is a well documented debugging infrastructure. This chapter discusses its features and relevancy with Design for Debug infrastructure. Note that Intel uses mostly a software based debugging approach since x86 platform.

3.1 ARM Debugging Instrumentation

ARM's debugging infrastructure is called CoreSight and often available in various versions such as Embedded Trace Macrocell (ETM) and System Trace Macrocell (STM), System

Trace Buffer (STB) and Embedded Trace Buffer (ETB) depending upon user requirement for debug functionalities for various hardware designs. CoreSight has a DAP (Debug Access Port) which provides an interface used by any debugging tool for communicating with SoC. The interface is compatible with both the JTAG and SWD standard. CoreSight ETM is a debug component that is used to reconstruct program execution by tracing all the branching instructions with a time-stamp. This is very basic trace module for CoreSight, but powerful for simple small footprint in real time embedded systems. It also supports output filtering and compression. CoreSight STM is an extended version of Instrumentation Trace Unit which provides high level debugging configurable using software. It provides a memory mapped area used by the software to communicate with the hardware. Messages written by the software are converted into hardware commands using CoreSight STM over a debug port. CoreSight ETB combines the features of STM and ETM. Although ARM has extensive debug options, bare-metal micro-controllers usually do not require such complex features as it can be redundant with increasing cost and area. Lower end versions of of the ARM Cortex family, such as Cortex M0+ have an Micro Trace Buffer (MTB) which consumes small area and less power. MTB uses portion of the system internal SRAM to store trace data. This data is accessible over the ARM Advanced Microcontroller Bus Architecture's (AMBA) High-performance Bus (AHB) Lite interface. The size of the SRAM Trace Buffer is configurable by software. External hardware can control trace start and stop using user input. When MTB is enabled it records executed branching instructions from Cortex M0+ in the trace buffer. It is a light weight trace unit which can ideally be used in small bare-metal embedded systems.

3.2 Intel Trace Instrumentation[1]

Intel's x86 architecture has been around for quite some time, but little used for light weight Embedded Systems. Debugging x86 systems normally used ICE based systems, and starting with 8086 architectures it uses firmware based debugging approach where debugging is done by setting and reading debugging registers[1]. Now-a-days modern x86 architectures may have advanced debugging support, but the debugging register approach is still carried out over time. It normally has two set of debug registers: Breakpoint Exception (BP) and Debug Exception (DB) using 8 to 16 registers depending upon the architecture implementation. These registers forces the core to generate an exception corresponding to breakpoint or data exception. Depending upon the exception data is handled using firmware.

Chapter 4

Design Implementation

The debug module designed and implemented in this project focuses mainly on hardware instrumentation for debugging purposes. Various concepts described in the previous chapters discussing about Design for Debug methodologies are taken into consideration while instrumenting the core. As noted earlier, the Open source IP AMBER 25 (A25) processor core from the Open Cores Community is used for this project. Some modifications have been made to the original A25 core as it was designed and optimized for FPGA implementation and this project targets an ASIC design methodology. The difference lies only in structuring hardware for DFT purposes, mainly required for manufacturing tests, which is a widely adopted approach. All the blocks of the IP are used as is and additional logic designed handles halting the core and reading its state. The debug module contains a main debug state machine that has been designed to address the features required for debugging. It communicates over a JTAG interface with a host computer which controls the operation of the processor in debug mode.

4.1 Design Components

The design components of the system are shown below in the Figure 4.1. The host can be computer or any other device which is able to send commands through JTAG interface on Debug Access Port. The JTAG interface is a standard TAP controller as defined in IEEE Std 1149.1-2013. The Figure 4.1 shows logical view of complete design implementation of the system.

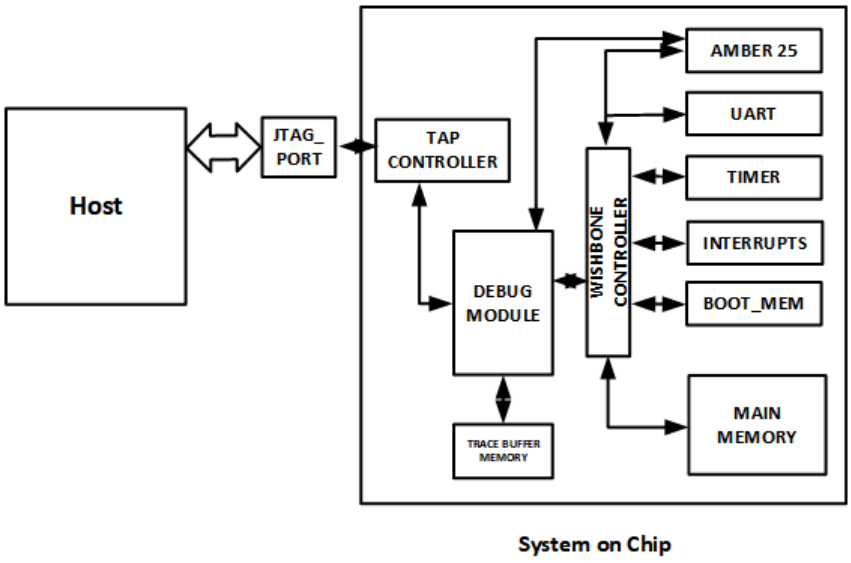

Figure 4.1: System Overview

The System on Chip shown in the Figure 4.1 is synthesizable HDL module containing the TAP controller, Debug Module, and Trace buffer memory that is used to trace operation of the A25 core. The Debug Module is connected to the A25 core and also to the system Wishbone bus as a Wishbone master along with A25 core. The Debug Module takes control over the bus when in debug mode or as instructed from the host over JTAG instructions. Main memory is non synthesizable memory and hence connected over wishbone as an off-chip slave peripheral. The SoC bus architecture uses the Wishbone bus which contains a

Wishbone arbiter to arbitrate between different masters and slaves. All the components of design are explained further in following sections.

4.1.1 AMBER 25 Core

The AMBER 25 Core is an ARM v2a compatible RISC processor core. It has two variants a 3-stage pipeline (A23) and 5-stage pipeline (A25), this project uses the A25 core. The A25 includes a 32-bit datapath, and separate instruction and data caches that can be configurable to either 2,3,4 or 8 way and each way of 8kB. As shown in the Figure 4.2, the A25 implementation has a five stage pipeline: *(i)* Instruction fetch - executed at first machine cycle used to load instruction from the main memory or cache; *(ii)* Decode - executed after fetch stage used by the core to decode the instruction; *(iii)* Execute - this stage process the instruction and performs operation defined in the instruction; *(iv)* Memory - this stage performs memory access if required; *(v)* write back - this stage writes the result to the destination location. Apart from the current design following modifications discussed in Section 4.1.1.1 are been made for instrumenting the IP for debug purpose.

4.1.1.1 IP Instrumentation

IP Instrumentation refers to the changes made in the IP in-order to communicate and control execution using the Debug Module. In order to halt the core, this design uses *i_system_rdy* signal from the core which when low, halts the system at fetch stage. *i_system_rdy* is used by the core for determining whether the core is ready start and all physical interfaces and memories connected are initialized. The stalling of the fetch stage further leads to stalling of other subsequent stages in the pipeline and thus halting the core. This signal is used by the debug module for controlling the program flow of the core, thus implementing run stop debugging methodology. Figure 4.3 mentions how a

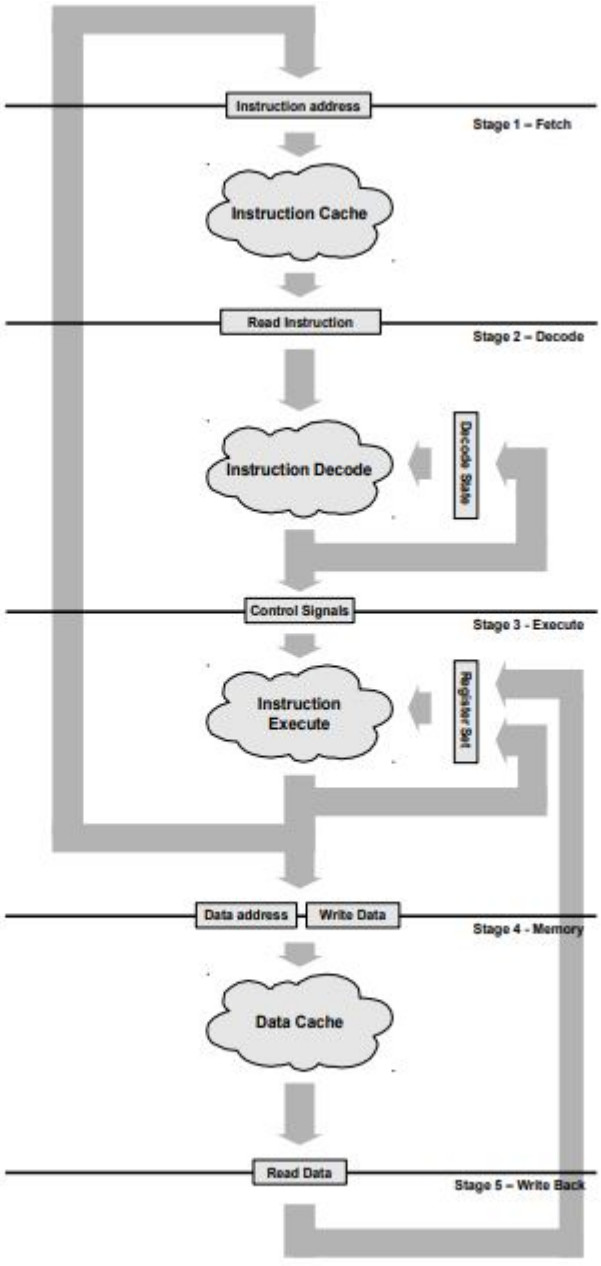

Figure 4.2: Amber Processor 5 Stage Pipeline[4]

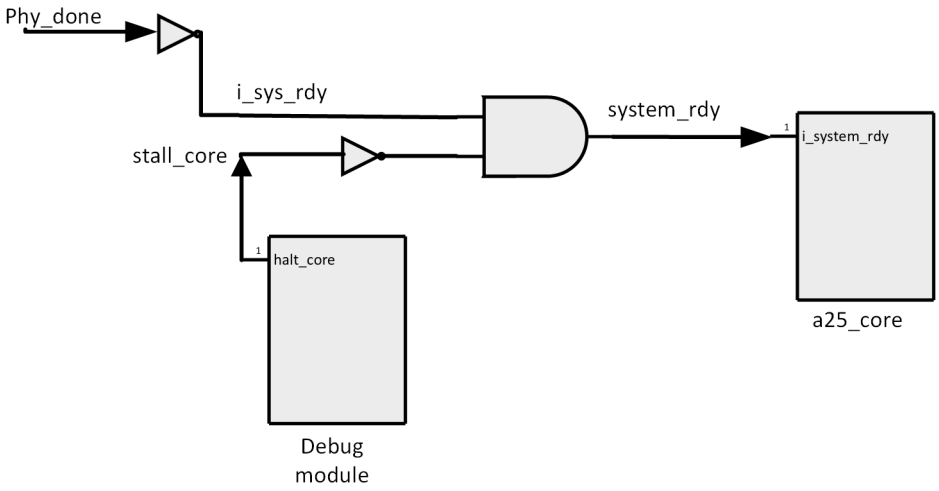

Figure 4.3: Halt Core Logic

processor is stalled in debug mode. Internal register values are read from the register file after halting the processor. The register values are multiplexed over 32 bit bus, based on 5 bit control signal. Figure 4.4 shows the reading process. The Debug Module controls the Multiplexer (MUX) output by *state_read_control_signal* and reads the corresponding register data over *o_debug_reg_bus.*

4.1.2 Test Access Port & JTAG

TAP controller is implemented as a state machine described in the IEEE Std 1149.1-2013 and in Figure 2.2 with instruction and data register. Instruction register holds the value of instruction which is captured serially over Test Data In pin in shift_ir state and latched to the IR register in update_ir state. Signals *out_data* and *out_ir* are 32 bit and 4 bit respectively which are used to communicate with Debug Module. As an instruction is latched it sends it over *out_ir* with *update_ir_o* which acts as a strobe for Debug Module to read the instruction. Similarly, data corresponding to the instruction is send over *out_data* and with a strobe *update_dr_o*.

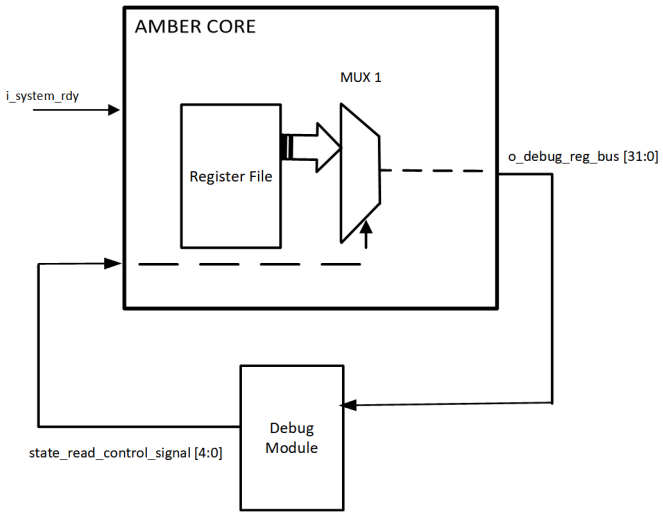

Figure 4.4: State Read Internal Register

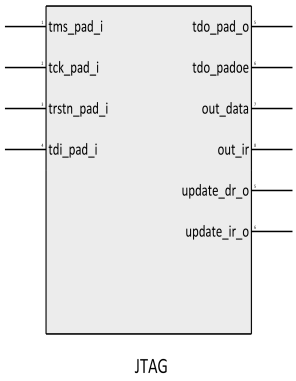

Figure 4.5: JTAG Module

4.1.3 Debug Module

Debug Module is mainly responsible for controlling the core according to instructions received over JTAG interface. Figure 4.6 shows state diagram of the state machine implementation and Figure 4.8 shows the pin-out of the debug module with left side are input pins and right side are output pins and Figure 4.7 shows logical view of the debug module. State machine acts as control unit which controls different units of the design.

instruction_in and *data_in* are 32 bit signals which receives 32 bit instruction and data value from JTAG module. *debug_state_bus* is an 32 bit input bus over which core sends value of the internal registers. As core sends multiplexed data on *debug_state_bus*, *state_read_control_signal* is 5 bit output used to send control signals that corresponds to the register value which are to be read over *debug_state_bus*. *halt_core* is the controlling signal to stall the processor. *serial_data_out* and *output_stb* are connected to *tdo_pad_o* and *tdo_padoe_o* respectively which sends serial data off-chip.

4.2 Design Implementation

Apart from A25 core, the system has peripherals like interrupt controller, Universal Asynchronous Transmitter Receiver (UART), timer, boot memory, and test module. All the modules communicate via Wishbone bus. The A25 core is a Wishbone master while remaining modules of the IP are configured as slaves. The Wishbone arbiter decides how a master selects and communicates with a slave. The complete system is designed as shown in Figure 4.9.

Note that the modules UART, Interrupt, timer and test module are used as is from the Open Cores Community as part of the AMBER system design. Debugging commands and data are transferred over the JTAG port, decoded by TAP controller, and further given to

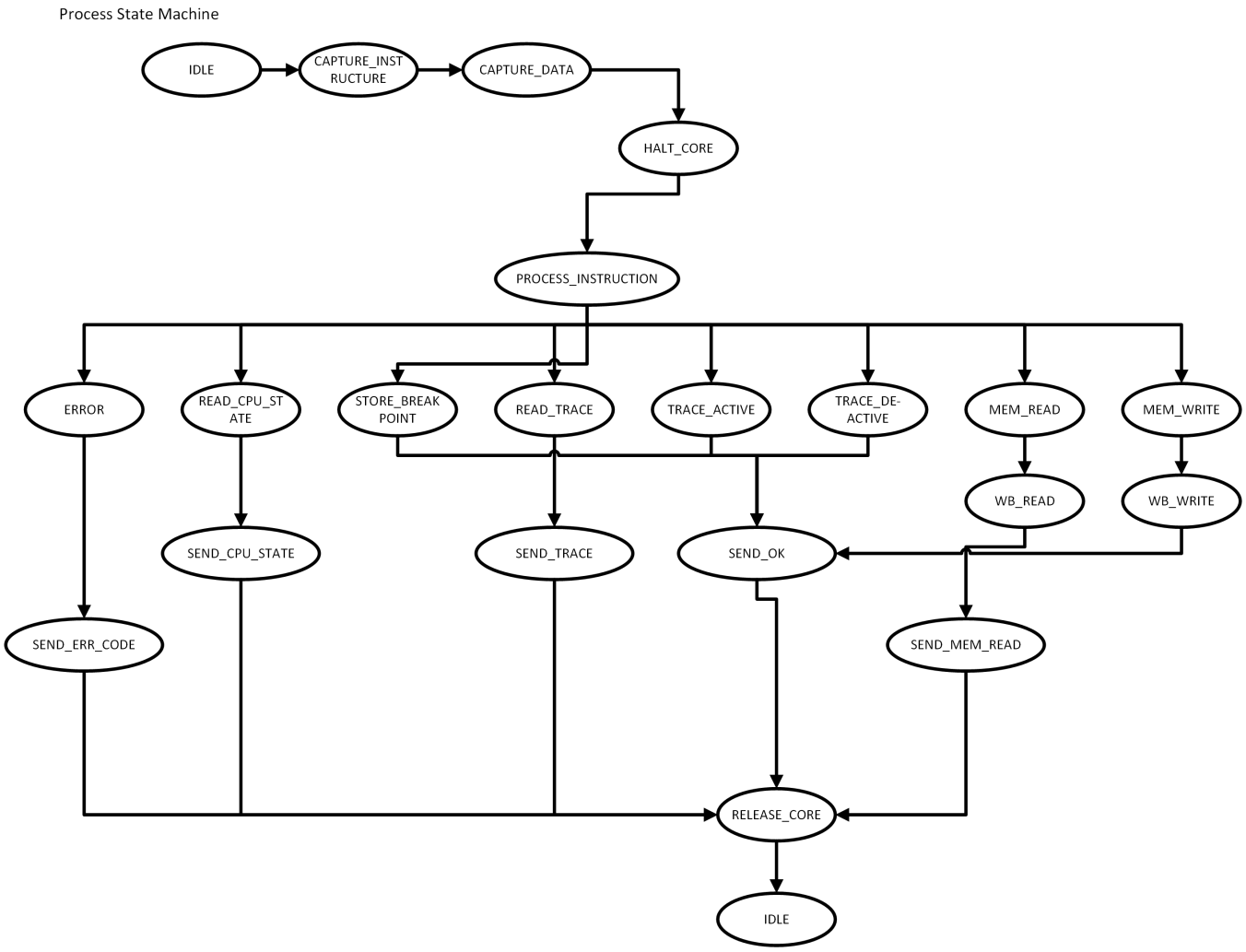

Figure 4.6: Debug State Machine

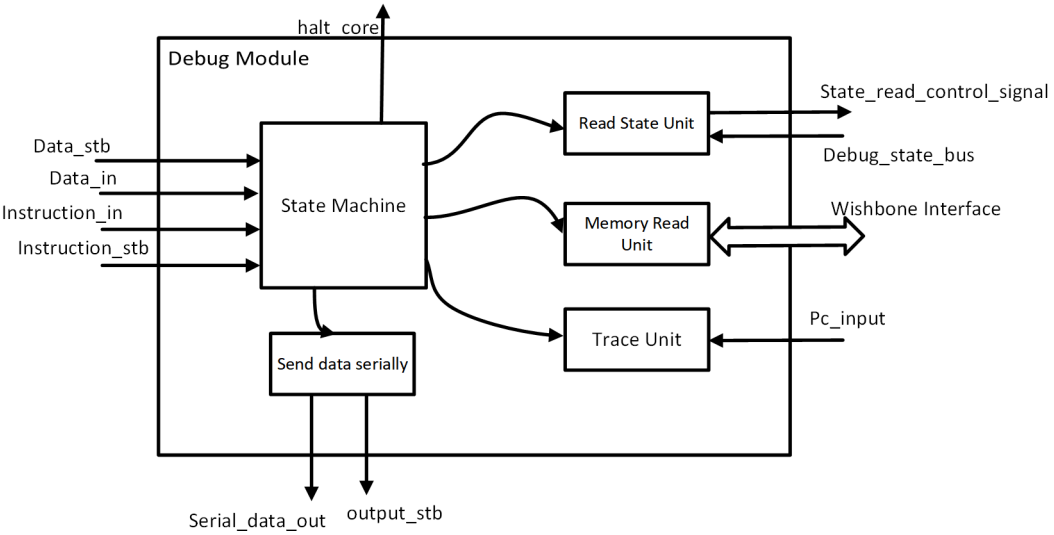

Figure 4.7: Debug Module Logical View

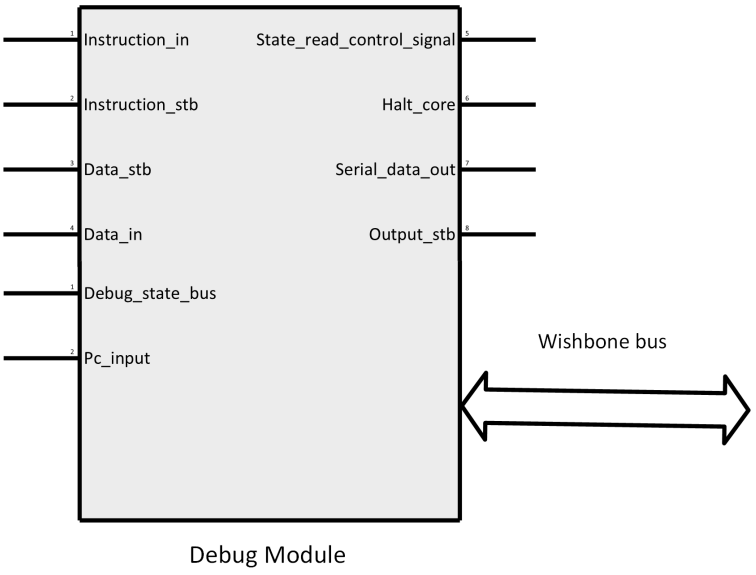

Figure 4.8: Debug Module

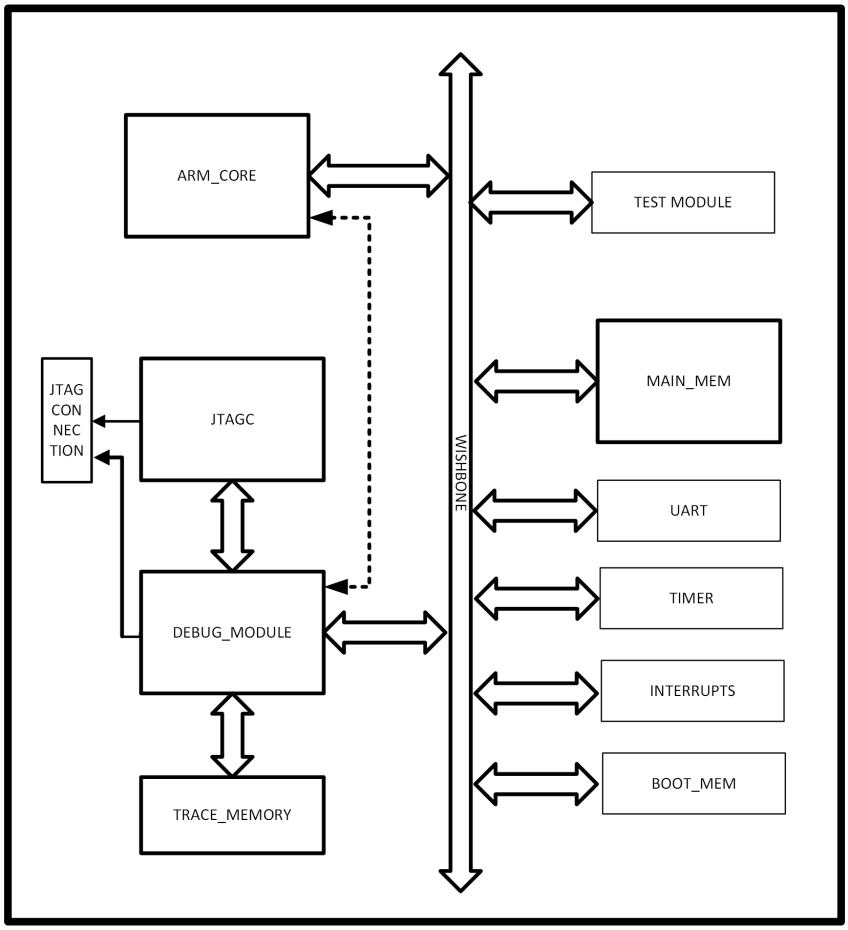

Figure 4.9: Complete System

Debug Module. These commands execute as shown in the state diagram in Figure 4.6. The design uses various aspects which were discussed in previous Chapters such as run stop debugging, memory access and trace. Along with real-time performance, area is also a very important factor when it comes to Embedded Systems, hence the added debugging logic should not take up more significant area as compared to processor logic and communication interface. Hence it is identified that run stop, trace and memory accesses are the minimum requirements for efficient debugging in this system.

4.2.1 JTAG Instructions Implementation

4.2.1.1 Read CPU State

Instruction (0101)b or (5)d can be used for read CPU state. This stage triggers a counter values from 0 to 32 which is send over *state_read_control_signal* and read back over *debug_state_bus* with the register values corresponding to the counter values. The data read over *debug_state_bus* is stored in array of registers. Later the stored register values in register array are send serially out on *serial_data_out*. *output_stb* remains high as long as valid data is available on *serial_data_out*.

4.2.1.2 Store Breakpoint

Instruction (0111)b or (7)d can be used for storing a breakpoint. Only one breakpoint is supported and this can be stored using (7)d instruction and data register which contains the program counter value for the breakpoint which is needed to be set. The debug module continuously compares current program counter input and stored breakpoint value, on hitting the stored breakpoint it generates *halt_core* which puts the processor into halt mode. Once the processor enters the halt mode, CPU state can be read using read CPU

state instruction. The return value received for this instruction is 32'h80000001 when everything is good, else 32'hff00ff00 for error condition.

4.2.1.3 Memory Access

Instruction (0011)b or (3)d can be used for reading memory locations where in 32 bit memory location address is provided by the data register. For memory access, debug module puts the processor in halt state and communicates directly acting as a wishbone master with the memory over wishbone bus. For read memory, data is send back serially through JTAG. This only supports single location for memory read and not block memory read.

4.2.1.4 Trace Data

There are three instructions that supports trace operation namely trace read, trace log active and trace log de-active. Trace read (0110)b or (6)d is used for reading the trace data from the memory and sending over *serial_data_out* signal. Trace memory is designed as an array of 32 bit register memory. Trace log when active stores the program counter value when there is branch in program flow. Branching can be detected by comparing current program counter with previous program counter. If the current program counter is greater than four as compared to the previous one, it can be concluded that it has taken a branch. The record of theses branching instruction is stored in the memory and are send out using *serial_data_out* signal on trace read command. Trace log active is set by instruction (1010)b and de-active by (1011)b. If the the memory is full trace log is automatically deactivated in order to prevent over written of the earlier values.

Chapter 5

Tests and Results

This section discusses results for the various tests performed on the debug module. Results are captured as two different types: testing JTAG instructions and identifying system overhead due to added debug module logic.

5.1 JTAG Instructions

5.1.1 Run and Stop

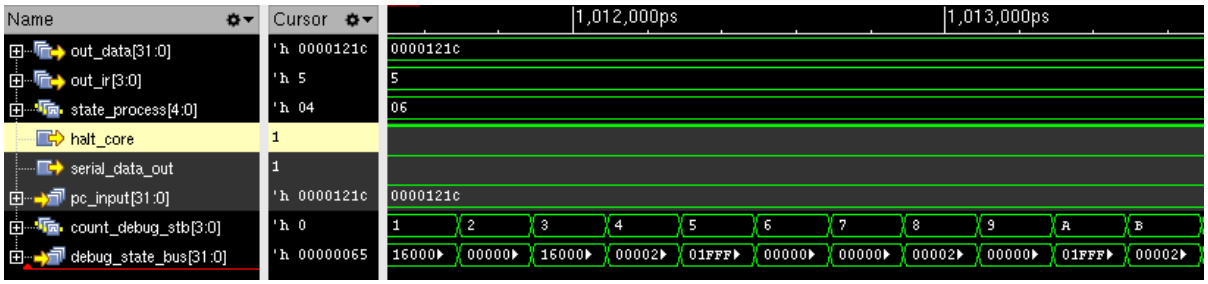

Figure 5.1: Stop and Read State

Figure 5.1 shows stop and read CPU state, *count_debug_stb* is the control signal to read register values from registers R0, R1, and so on and *debug_state_bus* is 32 bit data bus

which contains values for internal register that corresponding to *count_debug_stb*. These are stored in the array of memory and are send serially which can be seen in Figure 5.2.

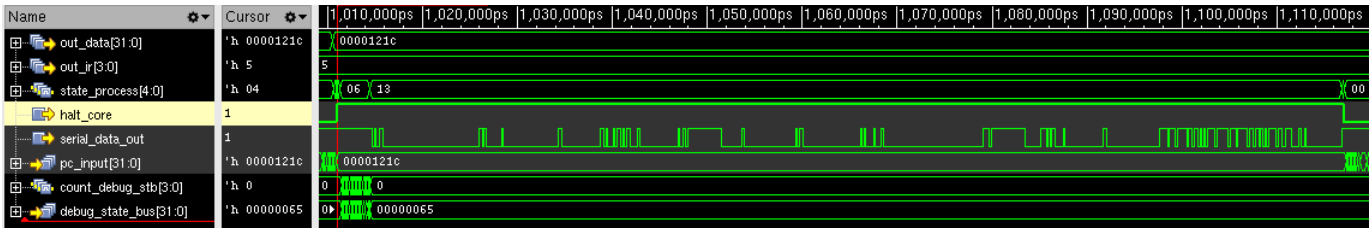

Figure 5.2: Send CPU State Date

5.1.2 Memory Access

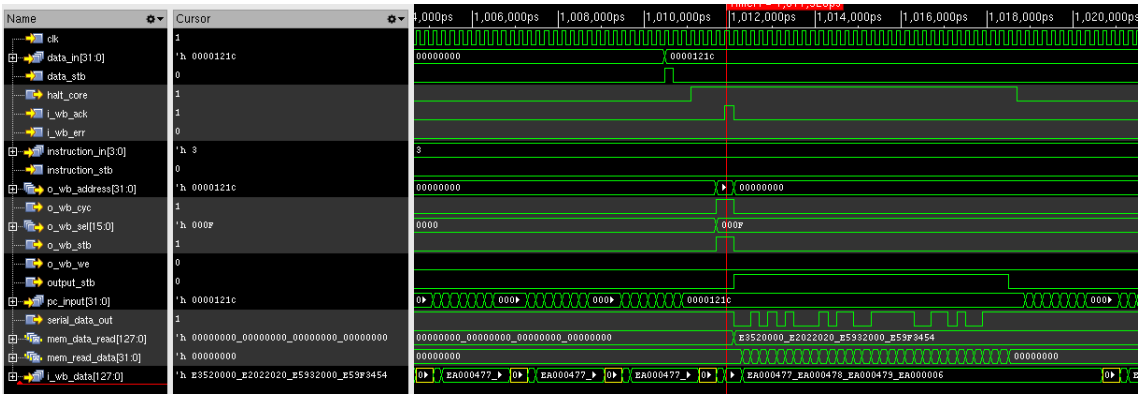

Figure 5.3: Memory Read

Figure 5.3 shows the memory read operation of the debugger. *out_data* is the breakpoint address received from JTAG interface. *pc_input* is the program counter value which is halted at the same point as stored breakpoint. *mem_data_read* is 128 bit value from the wishbone memory read and depending upon bank selected *mem_read_data* sends valid 32 bits over *serial_data_out* serially. halt_core is debug module output signal which halts the core. *out_ir* holds the data of current instruction which is 3 in this case.

5.1.3 Trace

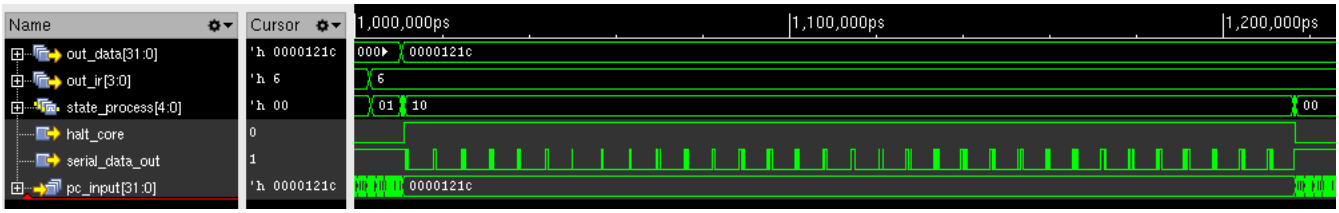

Figure 5.4: Read Trace Memory

The Figure 5.4 shows read trace memory. Read trace memory halts the core and sends out trace data, this is done to so that program counter address can be set at which instant back trace of the program execution flow can be done. Trace active and de-active does not halts the core, it starts logging program counter value if it changes greater than old program counter value with an offset of 4.

5.2 System Overhead

5.2.1 Area Overhead

The design is synthesized targeting a TSMC 65 nm library using Synopsys Design Compiler using a two pass methodology: RTL synthesis resulting in a pre-scan netlist, and Test synthesis resulting in a DFT instrumented post-scan netlist utilizing a full scan protocol. Both synthesized versions of the design for the complete system contain hard macro implementations of memories, this contributed to 1412070µm^2 of are used by memories macros in the design. The total area shown in the Table 5.1 are inclusive with area for the memory. The Debug Module is dominant in area consumption as a part of instrumentation of the core for debugging purposes. Table 5.1 shows difference in area with and without debugger unit.

Table 5.1: Area Overhead

Module	Pre-Scan (µm²)			
	Combinational	Non-Combinational	Buffer/Inv	Total
Debug Module	90,301.780	134,466.394	3,925.152	224,768.174
System w/o Debug	401,596.274	347,658.6994	44,128.023	2,161,324.974
System with Debug	491,834.853	482,125.094	47,979.994	2,386,029.947
	Post-Scan (µm²)			
	Combinational	Non-Combinational	Buffer/Inv	Total
Debug Module	90,371.634	160,478.848	3,805.401	250,850.483
System w/o Debug	401,416.649	417,280.262	43,951.723	2,161,324.974
System with Debug	491,751.693	577,918.777	47,743.819	2,481,740.471

Table 5.2: Area Distribution For Complete System With Debug Module

Module	Pre-Scan (µm²)	Percentage	Post-Scan (µm²)	Percentage
Debug Module Total Area	224,748.216	9.4	250,950.275	10.1
Debug Module Combinational	87780.369	3.67	87,810.306	3.53
Debug Module Non Combinational	134466.394	5.63	160,638.515	6.47

Table 5.2 shows the the Debug Module increases the area post addition of scan chains by 10.1 percent. In Table 5.1 the values for Debug Module are calculated for a standalone design, whereas Table 5.2 shows the distribution of the area for Debug Module synthesized with complete system. Percentage in Table 5.2 is calculated with respect to complete system area.

5.2.2 Power Overhead

Power overhead is shown in Table 5.3.

Table 5.3: Power Overhead, Post Scan Netlist

Module	Internal Power	Switching Power	Total Dynamic Power	Leakage Power
System w/o debug	9.6514 mW	6.9248 mW	16.5762 mW	3.3116 uW
System with debug	17.5114 mW	7.9742 mW	25.4856 mW	4.3778 uW

Note that the complete system with Debug Module consumes $8.90mW$ more than the one without the Debug Module. There is also a substantial increase in leakage power of $1.06uW$, for a total overall increase in power consumption of $8.91mW$. The results in the Table 5.4 shows pre-scan power consumption, its shows very little increase as compared to one with scan chains.

Table 5.4: Power Overhead, Pre-Scan Netlist

Module	Internal Power	Switching Power	Total Dynamic Power	Leakage Power
System w/o debug	11.4877 mW	1.8732 mW	13.3609 mW	2.9435 uW
System with debug	17.6651 mW	2.0396mW	19.7048mW	3.8850 uW

With an increase in area, internal power is increased considerably as compared to switching power. The Debug Module can be power gated when not in use to save power consumption.

Table 5.5 shows that the increase in power consumption is dominated by use of many registers in the Debug Module.

Table 5.5: Power Overhead for Standalone Debug Module

Group	Group	Internal	Switching	Leakage	Total	Percentage
Pre-Scan Netlist	Register	6.3024 mW	1.3717e-02 mW	664.5439 nW	6.3168 mW	96.70%
	Combinational	2.5073e-02 mW	0.1903 mW	277.2005 nW	0.2157 mW	3.30%
	Total	6.3275 mW	0.2040 mW	941.7444 nW	6.5325 mW	100%
Post-Scan Netlist	Register	8.2769 mW	0.4618 mW	788.8556 nW	8.7395 mW	88.80%
	Combinational	0.4357 mW	0.6660 mW	278.1273 nW	1.1020 mW	11.20%
	Total	8.7126 mW	1.1278 mW	1.0670e+03 nW	9.8415 mW	100%

Chapter 6

Conclusions

This study discussed the importance of Design for Debug approach for hardware design of Embedded Systems and then implements debug hardware blocks to verify overhead impact on the target system design. As this study was focused on bare-metal real time embedded systems, features such as breakpoints, memory access, and trace were identified as important and a minimum requirement for debugging embedded systems. It can be observed that for the features implemented the design with debugger increases area by about 10.1 percent as compared to one without debugger. There is a substantial increase in internal power by about $8.9mW$, mostly dominated by the use of registers. As the debugging unit is not active most of the time, it can be shut-off using power gating technique to save power. Although, active the power consumption would rise, this can be compensated by providing additional power through the debugging interface when used in debugging mode if power is a tight constraint. Instrumentation of the hardware provides access to the internal signals of the core. This can not only be used for debugging remote software on bare-metal embedded systems but also can be used for testing and analysis during chip development phase. When used during post silicon validation or FPGA emulation based

techniques, it would decrease time to verify and test for various cases. In case of bugs encountered, this can be helpful to provide with internal details required to understand the root cause easily and efficiently. This would reduce overall development cycle time and also minimizing chip re-spins. As embedded systems are designed and used for variety of reasons, always having software emulation based debugging is not advisable, true on-chip debugger should be required as this gives real view of working chip internals. Although trade offs can be made depending upon area required for the debugger with what features can be implemented in hardware or what can be implemented in firmware that can be triggered for debugging purposes. Most of the features can be implemented in firmware using interrupts and triggering appropriate ISR except for instruction trace. Firmware based debugging when used will often deviate from actual program execution and would not be able to give a complete out of the box system view to the designer or user. The implementation of the features still remains a trade-off between software and hardware giving the fact that hardware adds a 10% increase to the overall system area.

6.1 Future Work

Debugger design is always evolving as new micro-architecture implementations are evolving. Its capabilities will always change, update or become obsolete over time. Currently this project focused on hardware instrumentation of the core and its effect. Future scope would include compatible front end GUI for communicating with JTAG port. There are many open source implementations for front end GUI for communicating over JTAG. Open On-Chip Debugger is one of the open source projects which uses instrumented features for debugging of the core and communicates over JTAG or SWD interface and is integrated with GDB sever[5]. Hardware debugging capabilities are important and good to have fea-

tures in Embedded Systems hardware. Currently hardware debugging features are now a trade-off against area and performance as these corners are pretty compact in the Embedded Systems. Hence documenting new data with research for novel hardware debugging capabilities and its effects on embedded systems are also required with time.

References

[1] S. Bach, "Design and Implementation of a Debugging Unit for the OpenProcessor Platform," Master's thesis, University of Karlsruhe (TH) Institute of Operational and dialog systems, February 2008.

[2] N. Stollon, *On-Chip Instrumentation: Design and Debug for Systems on Chip*, 1st ed. Springer Publishing Company, Incorporated, 2010.

[3] IEEE, "IEEE Standard for Test Access Port and Boundary-Scan Architecture," *IEEE Std 1149.1-2013 (Revision of IEEE Std 1149.1-2001)*, pp. 1–444, May 2013.

[4] *Amber Open Source Project: Amber 2 Core Specification*, Mar. 2015.

[5] D. Rath, "Open On-Chip Debugger," Master's thesis, University of Applied Sciences Augsburg, 2005.

[6] P. Fogarty, "Minimising The Impact of Software Instrumentation Using On-Chip Debug and a Secondary CPU Core," in *Proceedings of the 2012 System, Software, SoC and Silicon Debug Conference*, Sept 2012, pp. 1–5.

[7] D. Leman, "Hardware-Assisted Breakpoints," *Dr.Dobb's Journal*, vol. 30, no. 6, pp. 87–90, 06 2005, copyright - Copyright CMP Media LLC Jun 2005; Document feature - Tables; Photographs; Last updated - 2014-05-25; CODEN - DDJOEB.

[Online]. Available: http://search.proquest.com.ezproxy.rit.edu/docview/202732033?accountid=108

[8] J. Backer, D. Hely, and R. Karri, "Secure and Flexible Trace-based Debugging of Systems-on-chip," *ACM Trans. Des. Autom. Electron. Syst.*, vol. 22, no. 2, pp. 31:1–31:25, Dec. 2016. [Online]. Available: http://doi.acm.org.ezproxy.rit.edu/10.1145/2994601

[9] N. Ohba and K. Takano, "Hardware Debugging Method Based on Signal Transitions and Transactions," in *Proceedings of the 2006 Asia and South Pacific Design Automation Conference*, ser. ASP-DAC '06. Piscataway, NJ, USA: IEEE Press, 2006, pp. 454–459. [Online]. Available: https://doi-org.ezproxy.rit.edu/10.1145/1118299.1118412

[10] N. C. Darounkola, "Source-Level Debugging Framework Design For FPGA High-Level Synthesis," Ph.D. dissertation, Department of Electrical and Computer Engineering University of Toronto, 2014. [Online]. Available: http://search.proquest.com.ezproxy.rit.edu/docview/1648982034?accountid=108

[11] M. Sami, M. Malek, U. Bondi, and F. Regazzoni, "Embedded Systems Education: Job Market Expectations," *SIGBED Rev.*, vol. 14, no. 1, pp. 22–28, Jan. 2017. [Online]. Available: http://doi.acm.org/10.1145/3036686.3036689

[12] G. Stringham, *Hardware/Firmware Interface Design, Best Practices for Improving Embedded Systems Development.* Elsevier Science, 2009. ProQuest Ebook Central, 2009. [Online]. Available: https://ebookcentral.proquest.com/lib/rit/detail.action?docID=566649.

[13] H. P. E. Vranken, M. P. J. Stevens, and M. T. M. Segers, "Design-For-Debug in Hardware/Software Co-Design," in *Hardware/Software Codesign, 1997. (CODES/CASHE '97), Proceedings of the Fifth International Workshop on*, Mar 1997, pp. 35–39.

[14] H. F. Ko, A. B. Kinsman, and N. Nicolici, "Design-for-Debug Architecture for Distributed Embedded Logic Analysis," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 19, no. 8, pp. 1380–1393, Aug 2011.

[15] K. G. Bart Vermeulen, *Debugging Systems-on-Chip*. Springer, Cham, 2014.

[16] E. A. A. Daoud, "On-Chip Debug Architectures For Improving Observability During Post-Silicon Validation," Ph.D. dissertation, McMaster University, 2008.

[17] A. Kourfali and D. Stroobandt, "Efficient Hardware Debugging Using Parameterized FPGA Reconfiguration," in *2016 IEEE International Parallel and Distributed Processing Symposium Workshops (IPDPSW)*, May 2016, pp. 277–282.

[18] Y.-H. Im, S.-J. Nam, B.-W. Kim, K.-G. Kang, D.-H. Lee, J.-H. Yang, Y.-S. Kwon, J.-H. Lee, and C.-M. Kyung, "Co-development of Media-processor and Source-level Debugger using Hardware Emulation-based Validation," in *VLSI and CAD, 1999. ICVC '99. 6th International Conference on*, 1999, pp. 95–98.

[19] C. R. Hill, "A Real-time Microprocessor Debugging Technique," in *Proceedings of the Symposium on High-level Debugging*, ser. SIGSOFT '83. New York, NY, USA: ACM, 1983, pp. 145–148. [Online]. Available: http://doi.acm.org/10.1145/1006147.1006179

[20] I.-J. Huang and T.-A. Lu, "ICEBERG: An Embedded In-Circuit Emulator Synthesizer for Microcontrollers," in *Proceedings 1999 Design Automation Conference (Cat. No. 99CH36361)*, 1999, pp. 580–585.

[21] M. Abramovici, P. Bradley, K. Dwarakanath, P. Levin, G. Memmi, and D. Miller, "A Reconfigurable Design-for-Debug Infrastructure for SoCs," in *2006 43rd ACM/IEEE Design Automation Conference*, July 2006, pp. 7–12.

[22] Y.-C. Hsu, F. Tsai, W. Jong, and Y.-T. Chang, "Visibility Enhancement for Silicon Debug," in *2006 43rd ACM/IEEE Design Automation Conference*, 2006, pp. 13–18.

[23] K. Holdbrook, S. Joshi, S. Mitra, J. Petolino, R. Raman, and M. Wong, "MicroSPARC: A Case-Study of Scan Based Debug," in *Proceedings., International Test Conference*, Oct 1994, pp. 70–75.

[24] A. B. T. Hopkins and K. D. McDonald-Maier, "Debug Support for Complex Systems On-Chip: A Review," *IEE Proceedings - Computers and Digital Techniques*, vol. 153, no. 4, pp. 197–207, July 2006.

[25] B. Vermeulen and S. K. Goel, "Design for Debug: Catching Design Errors in Digital Chips," *IEEE Design Test of Computers*, vol. 19, no. 3, pp. 35–43, May 2002.

[26] A. Verge, N. Ezzati Jivan, and M. R. Dagenais, "Hardware Assisted Software Event Tracing," *Concurrency and Computation: Practice and Experience*, vol. 29, no. 10, 5 2017. [Online]. Available: https://doi.org/10.1002/cpe.4069

[27] B. Scherer and G. Horvath, "Trace and Debug Port Based Watchdog Processor," in *2012 IEEE International Instrumentation and Measurement Technology Conference Proceedings*, May 2012, pp. 488–491.

[28] *White Paper CoreSight Technical Introduction*, Aug. 2013. [Online]. Available: http://infocenter.arm.com/help/topic/com.arm.doc.epm039795/coresight_technical_introduction_EPM_039795.pdf

Appendix I

Source Code

Listing I.1: Debug Module Header

```
/*
 *
 *    Author:   Nikhil Velguenkar
 *    Date:     1/1/2018
 *
 *
 *
 */

//-------------------------------------
//Process Data State Machine
//-------------------------------------

`define IDLE_PROCESS              5'd0
`define CAPTURE_INSTRUCTION       5'd1
`define CAPTURE_DATA              5'd2
`define HALT_CORE                 5'd3
`define PROCESS_INSTRUCTION       5'd4
`define ERROR                     5'd5
`define READ_CPU_STATE            5'd6
`define STORE_BREAKPOINT          5'd7
`define READ_TRACE                5'd8
`define READ_MEM                  5'd9
`define WB_READ                   5'd10
`define WRITE_MEM                 5'd11
```

```
`define WB_WRITE                5'd12
`define RELEASE_CORE            5'd13
`define TRACE_LOG_ACTIVE        5'd14
`define TRACE_LOG_DEACTIVE      5'd15
`define SEND_TRACE              5'd16
`define SEND_MEM_READ           5'd17
`define SEND_ERR_CODE           5'd18
`define SEND_CPU_STATE          5'd19
`define SEND_OK                 5'd20
```

Listing I.2: Debug Module

```
/*
 *
 *       Author:         Nikhil Velguenkar
 *       Date:           1/1/2018
 *       Description:    This module is for  debug control implemented as
    state
 *       machine
 *
 */


module DebugModule (
            reset,
            clk,
            scan_in0,
            scan_en,
            test_mode,
            scan_out0,
            instruction_stb,
            data_stb,
            instruction_in,
            data_in,
            debug_state_bus,
            state_read_control_signal,
            halt_core,
            serial_data_out,
            output_stb,
            pc_input,
            o_wb_address,
            o_wb_sel,
            o_wb_data,
            o_wb_we,
            o_wb_cyc,
            o_wb_stb,
            i_wb_data,
            i_wb_ack,
            i_wb_err
            //state
        );

input
```

```
    reset,                          // system reset
    clk;                            // system clock

input
    scan_in0,                       // test scan mode data input
    scan_en,                        // test scan mode enable
    test_mode;                      // test mode select

output
    scan_out0;                      // test scan mode data output
input
    instruction_stb,                    // instruction read control stb
    data_stb;                       // debug data read control stb

`include "./include/DebugModule.h"

input   [3:0]     instruction_in;   // instruction_reg JTAG
input   [31:0]     data_in;         // JTAG data reg
input   [31:0]     debug_state_bus;    // state read bus
input   [31:0]     pc_input;           // PC values from the core

output  reg [4:0]       state_read_control_signal;
output  reg             halt_core;
output  reg         output_stb;
output  reg         serial_data_out;
//output  [4:0]       state;


//--------------------------------
//wishbone interfacing signals
//--------------------------------

output reg  [31:0]  o_wb_address;
output reg  [15:0]  o_wb_sel;
output reg  [127:0] o_wb_data;
output reg          o_wb_we;
output reg          o_wb_cyc;
output reg          o_wb_stb;

input   [127:0] i_wb_data;
input           i_wb_ack;
input           i_wb_err;
```

```
//-----------------------------------
// internal register and wires
//-----------------------------------

reg serial_done;

reg [4:0] counter;

reg  [4:0]    state_process;
reg  [4:0]    next_state_process;

//control register

reg           halt_done;
reg           w_halt_core;

reg           wb_read_done, wb_write_done, wb_write, wb_read;
reg           store_breakpoint_done, store_breakpoint;
reg           step_by_step;
reg           read_cpu_state, trace_active, trace_deactive;
reg           cpu_read_done;

reg    [3:0]       count_debug_stb;
reg    [31:0]      breakpoint_reg;

reg    [31:0]     mem_read_data;

//-----------------------------------------------
//   Wiring
//-----------------------------------------------

//assign w_state = state;
//assign w_serial_done = serial_done;
//assign w_read_data = data_reg;
//assign state = state_process;
//assign halt_core = s_halt_core;

//-----------------------------------------------
//Other Logic
//-----------------------------------------------

//-----------------------------------------------------------------------
```

```
// Process State
//-----------------------------------------------------------------

/*
IDLE_PROCESS          :     default state
HALT_CORE             :     stall core
PROCESS_INSTRUCTION   :     process instruction
ERROR                 :     Error State
READ_CPU_STATE        :     read CPU registers
STORE_BREAKPOINT      :     hardware registers to store breakpoint
READ_TRACE            :     Read trace data
READ_MEM              :     Read memory through wb interface
WRITE_MEM             :     Write Memory through wb interface
WB_READ               :     Read protocol for wishbone interface
WB_WRITE              :     Write protocol for wishbone interface
SEND_CPU_STATE        :     Send CPU state over JTAG
SEND_TRACE            :     Send TRACE data over JTAG
SEND_ERR_CODE         :     send a code over Jtag
SEND_MEM_READ         :     Send Memory read Data
SEND_OK               :     Sends Ok error code
RELEASE_CORE          :     Releases core from halt state
*/

reg send_cpu_state;
reg send_err_code;
reg send_trace;
reg send_ok;
reg send_mem_read;
reg error;
reg send_trace_done;
reg send_mem_data_done;
reg send_ok_done;
reg capture_ins;
reg capture_data;
reg [3:0] instruction_reg;
reg [31:0] data_reg;
reg [4:0] send_count;
reg [4:0] send_err_count;
reg [4:0] index;
reg [5:0] index_trace;
reg [4:0] send_cpu_state_count;
reg [4:0] done_count;
reg [127:0] mem_data_read;
```

```
always @(posedge clk, posedge reset)
begin
    if(reset == 1'b1)
        begin
            state_process <= `IDLE_PROCESS  ; // reset condition
        end
    else
        begin
            state_process <= next_state_process;
        end
end

/*
JTAG INSTRUCTIONS:

0011 b --> MEMORY READ
0100 b --> MEMORY WRITE
0101 b --> READ CPU STATE
0110 b --> READ_TRACE
0111 b --> STORE BREAKPOINT
1010 b --> TRACE_LOG_ACTIVE
1011 b --> TRACE LOG DEACTIVE

*/

always @(state_process, instruction_stb, data_stb, instruction_reg,
   store_breakpoint_done, count_debug_stb, index, index_trace,
   send_count, wb_read_done, wb_write_done, done_count, i_wb_ack)
begin
    case (state_process)
        `IDLE_PROCESS: if (instruction_stb == 1'b1)
                           next_state_process = `CAPTURE_INSTRUCTION;
                       else
                           next_state_process = `IDLE_PROCESS;
        `CAPTURE_INSTRUCTION: if (data_stb == 1'b1)
                                  next_state_process = `CAPTURE_DATA;
                              else
                                  next_state_process =
                                     `CAPTURE_INSTRUCTION;
        `CAPTURE_DATA: next_state_process = `HALT_CORE;
        `HALT_CORE:          next_state_process = `PROCESS_INSTRUCTION;
```

```
`PROCESS_INSTRUCTION: if (instruction_reg == 4'b0111)
                          next_state_process =
                             `STORE_BREAKPOINT;
                      else if (instruction_reg == 4'b0101)
                          next_state_process = `READ_CPU_STATE
                             ;
                      else if (instruction_reg == 4'b0110)
                          next_state_process = `READ_TRACE;
                      else if (instruction_reg == 4'b0011)
                          next_state_process = `READ_MEM;
                      else if (instruction_reg == 4'b0100)
                          next_state_process = `WRITE_MEM;
                      else if (instruction_reg == 4'b1010)
                          next_state_process =
                             `TRACE_LOG_ACTIVE;
                      else if (instruction_reg == 4'b1011)
                          next_state_process =
                             `TRACE_LOG_DEACTIVE;
                      else
                          next_state_process = `ERROR;

 `ERROR: next_state_process = `SEND_ERR_CODE;

 `SEND_ERR_CODE: if (send_err_count == 5'b11111)
                    next_state_process = `RELEASE_CORE;
                 else
                    next_state_process = `SEND_ERR_CODE;

 `STORE_BREAKPOINT: if (store_breakpoint_done == 1'b1)
                        next_state_process = `SEND_OK;
                    else
                        next_state_process = `STORE_BREAKPOINT
                           ;

 `READ_CPU_STATE: if (count_debug_stb == 4'b1111)
                     next_state_process = `SEND_CPU_STATE;
                  else
                     next_state_process = `READ_CPU_STATE;

 `SEND_CPU_STATE: if (index == 5'b10000)
                     next_state_process = `RELEASE_CORE;
                  else
                     next_state_process = `SEND_CPU_STATE;
```

```
            `READ_TRACE: next_state_process = `SEND_TRACE;

            `SEND_TRACE: if (index_trace == 6'b100000)
                             next_state_process = `RELEASE_CORE;
                         else
                             next_state_process = `SEND_TRACE;

            `READ_MEM: next_state_process = `WB_READ;

            `WB_READ: if (i_wb_ack == 1'b1)
                          next_state_process = `SEND_MEM_READ;
                      else
                          next_state_process = `WB_READ;

            `SEND_MEM_READ: if (done_count == 5'b11111)
                                next_state_process = `RELEASE_CORE;
                            else
                                next_state_process = `SEND_MEM_READ;

            `WRITE_MEM: next_state_process = `WB_WRITE;

            `WB_WRITE: if (wb_write_done == 1'b1)
                           next_state_process = `RELEASE_CORE;
                       else
                           next_state_process = `WB_WRITE;

            `TRACE_LOG_ACTIVE: next_state_process = `SEND_OK;

            `TRACE_LOG_DEACTIVE: next_state_process = `SEND_OK;

            `SEND_OK: if (send_count == 5'b11111)
                          next_state_process = `RELEASE_CORE;
                      else
                          next_state_process = `SEND_OK;

            `RELEASE_CORE: next_state_process = `IDLE_PROCESS;

            default: next_state_process = `IDLE_PROCESS;
        endcase
    end

    //Control Signals
```

```
always @(state_process)
begin
    case (state_process)
        `IDLE_PROCESS: begin
                            w_halt_core      = 1'b0;
                            store_breakpoint = 1'b0;
                            read_cpu_state   = 1'b0;
                            error            = 1'b0;
                            step_by_step     = 1'b0;
                            wb_read          = 1'b0;
                            wb_write         = 1'b0;
                            trace_active     = 1'b0;
                            trace_deactive   = 1'b0;
                            serial_done      = 1'b0;
                            send_cpu_state   = 1'b0;
                            send_err_code    = 1'b0;
                            send_trace       = 1'b0;
                            send_ok          = 1'b0;
                            send_mem_read    = 1'b0;
                            capture_ins      = 1'b0;
                            capture_data     = 1'b0;
                       end
        `CAPTURE_INSTRUCTION: begin
                            w_halt_core      = 1'b0;
                            store_breakpoint = 1'b0;
                            read_cpu_state   = 1'b0;
                            error            = 1'b0;
                            step_by_step     = 1'b0;
                            wb_read          = 1'b0;
                            wb_write         = 1'b0;
                            trace_active     = 1'b0;
                            trace_deactive   = 1'b0;
                            serial_done      = 1'b0;
                            send_cpu_state   = 1'b0;
                            send_err_code    = 1'b0;
                            send_trace       = 1'b0;
                            send_ok          = 1'b0;
                            send_mem_read    = 1'b0;
                            capture_ins      = 1'b1;
                            capture_data     = 1'b0;
                      end
        `CAPTURE_DATA: begin
```

```
                    w_halt_core       = 1'b0;
                    store_breakpoint  = 1'b0;
                    read_cpu_state    = 1'b0;
                    error             = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done       = 1'b0;
                    send_cpu_state    = 1'b0;
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b1;
                    capture_data      = 1'b1;
                end
    `HALT_CORE: begin
                    w_halt_core       = 1'b1;
                    store_breakpoint  = 1'b0;
                    read_cpu_state    = 1'b0;
                    error             = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done       = 1'b0;
                    send_cpu_state    = 1'b0;
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
                    capture_data      = 1'b0;
                end
    `PROCESS_INSTRUCTION: begin
                    w_halt_core       = 1'b0;
                    store_breakpoint  = 1'b0;
                    read_cpu_state    = 1'b0;
                    error             = 1'b0;
                    step_by_step      = 1'b0;
```

```
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done       = 1'b0;
                    send_cpu_state    = 1'b0;
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
                    capture_data      = 1'b0;
                  end
`STORE_BREAKPOINT: begin
                    w_halt_core         = 1'b0;
                    store_breakpoint    = 1'b1;
                    read_cpu_state    = 1'b0;
                    error               = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done       = 1'b0;
                    send_cpu_state    = 1'b0;
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
                    capture_data      = 1'b0;
                    end
`READ_CPU_STATE:  begin
                    w_halt_core         = 1'b0;
                    store_breakpoint    = 1'b0;
                    read_cpu_state    = 1'b1;
                    error               = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done       = 1'b0;
```

```
                    send_cpu_state   = 1'b0;
                    send_err_code    = 1'b0;
                    send_trace       = 1'b0;
                    send_ok          = 1'b0;
                    send_mem_read    = 1'b0;
                    capture_ins      = 1'b0;
                    capture_data     = 1'b0;
                    end
`READ_TRACE:  begin
                    w_halt_core        = 1'b0;
                    store_breakpoint = 1'b0;
                    read_cpu_state   = 1'b0;
                    error              = 1'b0;
                    step_by_step     = 1'b1;
                    wb_read          = 1'b0;
                    wb_write         = 1'b0;
                    trace_active     = 1'b0;
                    trace_deactive   = 1'b0;
                    serial_done      = 1'b0;
                    send_cpu_state   = 1'b0;
                    send_err_code    = 1'b0;
                    send_trace       = 1'b0;
                    send_ok          = 1'b0;
                    send_mem_read    = 1'b0;
                    capture_ins      = 1'b0;
                    capture_data     = 1'b0;
                end
`ERROR:  begin
                    w_halt_core        = 1'b0;
                    store_breakpoint = 1'b0;
                    read_cpu_state   = 1'b0;
                    error              = 1'b1;
                    step_by_step     = 1'b0;
                    wb_read          = 1'b0;
                    wb_write         = 1'b0;
                    trace_active     = 1'b0;
                    trace_deactive   = 1'b0;
                    serial_done     = 1'b0;
                    send_cpu_state   = 1'b0;
                    send_err_code    = 1'b0;
                    send_trace       = 1'b0;
                    send_ok          = 1'b0;
                    send_mem_read    = 1'b0;
```

```
                capture_ins      = 1'b0;
                capture_data     = 1'b0;
        end
`READ_MEM:  begin
                w_halt_core        = 1'b0;
                store_breakpoint = 1'b0;
                read_cpu_state   = 1'b0;
                error              = 1'b0;
                step_by_step     = 1'b0;
                wb_read          = 1'b0;
                wb_write         = 1'b0;
                trace_active     = 1'b0;
                trace_deactive   = 1'b0;
                serial_done     = 1'b0;
                send_cpu_state   = 1'b0;
                send_err_code    = 1'b0;
                send_trace       = 1'b0;
                send_ok          = 1'b0;
                send_mem_read    = 1'b0;
                capture_ins      = 1'b0;
                capture_data     = 1'b0;
            end
`WB_READ:  begin
                w_halt_core        = 1'b0;
                store_breakpoint = 1'b0;
                read_cpu_state   = 1'b0;
                error              = 1'b0;
                step_by_step     = 1'b0;
                wb_read          = 1'b1;
                wb_write         = 1'b0;
                trace_active     = 1'b0;
                trace_deactive   = 1'b0;
                serial_done     = 1'b0;
                send_cpu_state   = 1'b0;
                send_err_code    = 1'b0;
                send_trace       = 1'b0;
                send_ok          = 1'b0;
                send_mem_read    = 1'b0;
                capture_ins      = 1'b0;
                capture_data     = 1'b0;
            end
`WRITE_MEM:  begin
                w_halt_core        = 1'b0;
```

```
                store_breakpoint = 1'b0;
                read_cpu_state   = 1'b0;
                error              = 1'b0;
                step_by_step     = 1'b0;
                wb_read          = 1'b0;
                wb_write         = 1'b0;
                trace_active     = 1'b0;
                trace_deactive   = 1'b0;
                serial_done     = 1'b0;
                send_cpu_state   = 1'b0;
                send_err_code    = 1'b0;
                send_trace       = 1'b0;
                send_ok          = 1'b0;
                send_mem_read    = 1'b0;
                capture_ins      = 1'b0;
                capture_data     = 1'b0;
            end
`WB_WRITE:  begin
                w_halt_core        = 1'b0;
                store_breakpoint = 1'b0;
                read_cpu_state   = 1'b0;
                error              = 1'b0;
                step_by_step     = 1'b0;
                wb_read          = 1'b0;
                wb_write         = 1'b1;
                trace_active     = 1'b0;
                trace_deactive   = 1'b0;
                serial_done     = 1'b0;
                send_cpu_state   = 1'b0;
                send_err_code    = 1'b0;
                send_trace       = 1'b0;
                send_ok          = 1'b0;
                send_mem_read    = 1'b0;
                capture_ins      = 1'b0;
                capture_data     = 1'b0;
            end
`TRACE_LOG_ACTIVE:  begin
                w_halt_core        = 1'b0;
                store_breakpoint = 1'b0;
                read_cpu_state   = 1'b0;
                error              = 1'b0;
                step_by_step     = 1'b0;
                wb_read          = 1'b0;
```

```
                    wb_write          = 1'b0;
                    trace_active      = 1'b1;
                    trace_deactive    = 1'b0;
                    serial_done       = 1'b0;
                    send_cpu_state    = 1'b0;
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
                    capture_data      = 1'b0;
                    end
`TRACE_LOG_DEACTIVE: begin
                    w_halt_core         = 1'b0;
                    store_breakpoint    = 1'b0;
                    read_cpu_state    = 1'b0;
                    error               = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b1;
                    serial_done       = 1'b0;
                    send_cpu_state    = 1'b0;
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
                    capture_data      = 1'b0;
                    end
`RELEASE_CORE: begin
                    w_halt_core         = 1'b0;
                    store_breakpoint    = 1'b0;
                    read_cpu_state    = 1'b0;
                    error               = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done       = 1'b1;
                    send_cpu_state    = 1'b0;
```

```
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
                    capture_data      = 1'b0;
                end
`SEND_CPU_STATE:  begin
                    w_halt_core         = 1'b0;
                    store_breakpoint = 1'b0;
                    read_cpu_state    = 1'b0;
                    error               = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done      = 1'b0;
                    send_cpu_state    = 1'b1;
                    send_err_code     = 1'b0;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
                    capture_data      = 1'b0;
        end
`SEND_ERR_CODE:  begin
                    w_halt_core         = 1'b0;
                    store_breakpoint = 1'b0;
                    read_cpu_state    = 1'b0;
                    error               = 1'b0;
                    step_by_step      = 1'b0;
                    wb_read           = 1'b0;
                    wb_write          = 1'b0;
                    trace_active      = 1'b0;
                    trace_deactive    = 1'b0;
                    serial_done      = 1'b0;
                    send_cpu_state    = 1'b0;
                    send_err_code     = 1'b1;
                    send_trace        = 1'b0;
                    send_ok           = 1'b0;
                    send_mem_read     = 1'b0;
                    capture_ins       = 1'b0;
```

```
                    capture_data     = 1'b0;
                end
`SEND_TRACE: begin
                w_halt_core         = 1'b0;
                    store_breakpoint = 1'b0;
                    read_cpu_state   = 1'b0;
                    error              = 1'b0;
                    step_by_step     = 1'b0;
                    wb_read          = 1'b0;
                    wb_write         = 1'b0;
                    trace_active     = 1'b0;
                    trace_deactive   = 1'b0;
                    serial_done     = 1'b0;
                    send_cpu_state   = 1'b0;
                    send_err_code    = 1'b0;
                    send_trace       = 1'b1;
                    send_ok          = 1'b0;
                    send_mem_read    = 1'b0;
                    capture_ins      = 1'b0;
                    capture_data     = 1'b0;
                end
`SEND_OK: begin
                w_halt_core         = 1'b0;
                    store_breakpoint = 1'b0;
                    read_cpu_state   = 1'b0;
                    error              = 1'b0;
                    step_by_step     = 1'b0;
                    wb_read          = 1'b0;
                    wb_write         = 1'b0;
                    trace_active     = 1'b0;
                    trace_deactive   = 1'b0;
                    serial_done     = 1'b0;
                    send_cpu_state   = 1'b0;
                    send_err_code    = 1'b0;
                    send_trace       = 1'b0;
                    send_ok          = 1'b1;
                    send_mem_read    = 1'b0;
                    capture_ins      = 1'b0;
                    capture_data     = 1'b0;
                end
`SEND_MEM_READ: begin
                    w_halt_core         = 1'b0;
                    store_breakpoint = 1'b0;
```

```
                        read_cpu_state   = 1'b0;
                        error             = 1'b0;
                        step_by_step     = 1'b0;
                        wb_read          = 1'b0;
                        wb_write         = 1'b0;
                        trace_active     = 1'b0;
                        trace_deactive   = 1'b0;
                        serial_done     = 1'b0;
                        send_cpu_state   = 1'b0;
                        send_err_code    = 1'b0;
                        send_trace       = 1'b0;
                        send_ok          = 1'b0;
                        send_mem_read    = 1'b1;
                        capture_ins      = 1'b0;
                        capture_data     = 1'b0;
                    end

        default: begin
                        w_halt_core      = 1'b0;
                        store_breakpoint = 1'b0;
                        read_cpu_state   = 1'b0;
                        error            = 1'b0;
                        step_by_step     = 1'b0;
                        wb_read          = 1'b0;
                        wb_write         = 1'b0;
                        trace_active     = 1'b0;
                        trace_deactive   = 1'b0;
                        serial_done      = 1'b0;
                        send_cpu_state   = 1'b0;
                        send_err_code    = 1'b0;
                        send_trace       = 1'b0;
                        send_ok          = 1'b0;
                        send_mem_read    = 1'b0;
                        capture_ins      = 1'b0;
                        capture_data     = 1'b0;
                end
    endcase
end

//Capture Instruction

always @(posedge clk, posedge reset)
begin
```

```
    if (reset == 1'b1)
        instruction_reg <= 4'd0;
    else  begin
        if (capture_ins == 1'b1)
            instruction_reg <= instruction_in;
        else
            instruction_reg <= instruction_reg;
    end
end

//Capture Data

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)
        data_reg <= 32'd0;
    else  begin
        if (capture_data == 1'b1)
            data_reg <= data_in;
        else
            data_reg <= data_reg;
    end
end

//---------------
//Halt
//---------------

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)
        halt_core <= 1'b0;
    else  begin
       if ((w_halt_core == 1'b1) || (breakpoint_reg == pc_input))
            halt_core <= 1'b1;
       else if ((serial_done == 1'b1))
            halt_core <= 1'b0;
       else
            halt_core <= halt_core;
    end
end

//Read CPU state
```

```
//control state signal

reg [31:0] register_file [15:0];

//assign
always @(posedge reset, posedge clk)
begin
    if (reset)
        state_read_control_signal = 5'd0;
    else
        state_read_control_signal = {1'b0, count_debug_stb};
end

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1) begin
        count_debug_stb <= 4'b0;
        register_file[0] <= 32'd0;
        register_file[1] <= 32'd0;
        register_file[2] <= 32'd0;
        register_file[3] <= 32'd0;
        register_file[4] <= 32'd0;
        register_file[5] <= 32'd0;
        register_file[6] <= 32'd0;
        register_file[7] <= 32'd0;
        register_file[8] <= 32'd0;
        register_file[9] <= 32'd0;
        register_file[10] <= 32'd0;
        register_file[11] <= 32'd0;
        register_file[12] <= 32'd0;
        register_file[13] <= 32'd0;
        register_file[14] <= 32'd0;
        register_file[15] <= 32'd0;
    end
    else
    begin
        if ((read_cpu_state == 1'b1) || (breakpoint_reg == pc_input))
           begin
            count_debug_stb <= count_debug_stb + 1'b1;
            register_file[count_debug_stb] <= debug_state_bus;
        end
        else  begin
            count_debug_stb <= 4'b0;
```

```
            register_file[count_debug_stb] <= register_file[
               count_debug_stb];
        end
    end
end

// Store breakpoint
// Data register is the breakpoint

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1) begin
        breakpoint_reg <= 32'hffffffff;
        store_breakpoint_done <= 1'b0;
    end
    else if (store_breakpoint == 1'b1) begin
        store_breakpoint_done <= 1'b1;
        breakpoint_reg <= data_reg;
    end
    else begin
        breakpoint_reg <= breakpoint_reg;
        store_breakpoint_done <= 1'b0;
    end
end


// pc trace
//lock in trace state
//
reg [4:0] trace_counter;
reg trace;

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)
        trace <= 1'b0;
    else if (trace_active == 1'b1)
        trace <= 1'b1;
    else if ((trace_deactive == 1'b1) || (trace_counter == 5'b11111))
        //the overflow condition also
        trace <= 1'b0;
    else
        trace <= trace;
```

```
end



// memory array
//

reg [31:0] trace_mem [31:0];
reg [31:0] old_pc, new_pc;


//capture change in PC
always @(posedge clk, posedge reset)
begin
    if (reset) begin
        new_pc <= 32'd0;
        old_pc <= 32'd0;
    end
    else  begin
        new_pc <= pc_input;
        old_pc <= new_pc;
    end
end


always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1) begin
        trace_mem['d0] <= 'd10;
        trace_mem['d1] <= 'd14;
        trace_mem['d2] <= 'd18;
        trace_mem['d3] <= 'd22;
        trace_mem['d4] <= 'd26;
        trace_mem['d5] <= 'd30;
        trace_mem['d6] <= 'd1;
        trace_mem['d7] <= 'd4;
        trace_mem['d8] <= 'd8;
        trace_mem['d9] <= 'd34;
        trace_mem['d10] <= 'd38;
        trace_mem['d11] <= 'd42;
        trace_mem['d12] <= 'd46;
        trace_mem['d13] <= 'd50;
        trace_mem['d14] <= 'd54;
```

```
            trace_mem['d15] <= 'd58;
            trace_mem['d16] <= 'd62;
            trace_mem['d17] <= 'd66;
            trace_mem['d18] <= 'd70;
            trace_mem['d19] <= 'd74;
            trace_mem['d20] <= 'd75;
            trace_mem['d21] <= 'd78;
            trace_mem['d22] <= 'd82;
            trace_mem['d23] <= 'd86;
            trace_mem['d24] <= 'd90;
            trace_mem['d25] <= 'd94;
            trace_mem['d26] <= 'd98;
            trace_mem['d27] <= 'd102;
            trace_mem['d28] <= 'd105;
            trace_mem['d29] <= 'd108;
            trace_mem['d30] <= 'd110;
            trace_mem['d31] <= 'd111;
        end
        else  begin
            if ((trace == 1'b1) && (new_pc > old_pc + 32'd4))
                trace_mem[trace_counter] <= pc_input;
            else
                trace_mem[trace_counter] <= trace_mem[trace_counter];
        end
    end

    //counter

    always @(posedge clk, posedge reset)
    begin
        if (reset == 1'b1)
            trace_counter <= 4'd0;
        else  begin
            if ((trace == 1'b1) && (new_pc > old_pc + 32'd4))
                trace_counter <= trace_counter + 1'b1;
            else
                trace_counter <= trace_counter;
        end
    end


    //Send ok
    //
```

```
reg [31:0] ok_reg;                  // Contains 0 code for ok
reg [31:0] error_reg;               // Contains FFFF_FFFF h for error
    code

//@ToDo MUX it for different signals

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1) begin
        send_count <= 5'd0;
        ok_reg <= 32'h80000001;
    end
    else   begin
        if (send_ok == 1'b1) begin
            send_count <= send_count + 1'd1;
            ok_reg <= ok_reg >> 1;
        end
        else  begin
            send_count <= 5'd0;
            ok_reg <= ok_reg;
        end
   end
end

//Send Error

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)  begin
        send_err_count <= 5'b0;
        error_reg <= 32'hff00ff00;
    end
    else  begin
        if (send_err_code == 1'b1) begin
            send_err_count <= send_err_count + 1'd1;
            error_reg <= error_reg >> 1;
        end
        else  begin
            send_err_count <= 5'b0;
            error_reg <= 32'hff00ff00;
        end
    end
```

```
end

//Send Cpu State
//
reg [31:0] send_buffer_state;


always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)
        index <= 5'd0;
    else  begin
       if (send_cpu_state_count == 5'b11110)
           index <= index + 1'b1;
       else if (index == 5'b10000)
           index <= 5'd0;
       else
           index <= index;
    end
end

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)  begin
        send_cpu_state_count <= 5'd0;
        send_buffer_state <= 32'd0;
    end
    else  begin
        if ((send_cpu_state == 1'b1) && (send_cpu_state_count != 5'
           b11111)) begin
            send_cpu_state_count <= send_cpu_state_count + 1'b1;
            send_buffer_state <= send_buffer_state >> 1;
        end
        else if ((send_cpu_state == 1'b1) && (send_cpu_state_count ==
           5'b11111))  begin
            send_cpu_state_count <= send_cpu_state_count + 1'b1;
            send_buffer_state <= register_file[index[3:0]];
        end
        else   begin
            send_cpu_state_count <= 5'd0;
            send_buffer_state <= register_file[0];
        end
    end
```

```
end

//Send Trace

reg [31:0] send_buffer_trace;
reg [4:0] send_trace_state_count;


always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)
        index_trace <= 6'd0;
    else  begin
       if (send_trace_state_count == 5'b11110)
           index_trace <= index_trace + 1'b1;
       else if (index_trace == 6'b100000)
           index_trace <= 6'd0;
       else
           index_trace <= index_trace;
    end
end

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)  begin
        send_trace_state_count <= 5'd0;
        send_buffer_trace <= 32'd0;
    end
    else  begin
        if ((send_trace == 1'b1) && (send_trace_state_count != 5'
           b11111)) begin
            send_trace_state_count <= send_trace_state_count + 1'b1;
            send_buffer_trace <= send_buffer_trace >> 1;
        end
        else if ((send_trace == 1'b1) && (send_trace_state_count == 5'
           b11111))  begin
            send_trace_state_count <= send_trace_state_count + 1'b1;
            send_buffer_trace <= trace_mem[index_trace[4:0]];
        end
        else   begin
            send_trace_state_count <= 5'd0;
            send_buffer_trace <= trace_mem[0];
        end
```

```
    end
end

// memory read through wishbone

// C1: put valid addr on adr_o; negates we_o; bank sel; cyc_o; stb_o
// C2: wait for ack
// C3: latches the data from DAT_I; negates stb_o and cyc_o; slave
    negates ack
//
// memory write through wiishbone
// //
// // C1: Valid address on adr_o; valid data on dat_o; asserts we_o;
    sel_o
// bank
// // select, cyc_o, stb_o
// // C2: waits for ack, prepares to terminate cycle
// // C3: master negates stb_o & cyc_o

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1) begin
        o_wb_address <= 32'd0;
        o_wb_data    <= 'd0;
        o_wb_we      <= 'd0;
        o_wb_cyc     <= 'd0;
        o_wb_stb     <= 'd0;
        o_wb_sel     <= 'd0;
    end
    else  begin
        if ((wb_read == 1'b1) && (i_wb_ack == 1'b0)) begin
            //valid addr and data
            o_wb_address <= data_reg;
            o_wb_data    <= 'd0;
            o_wb_we      <= 'd0;
            o_wb_cyc     <= 'd1;
            o_wb_stb     <= 'd1;
            o_wb_sel     <= 'hffff;    //have to confi
        end
        else if ((wb_read == 1'b1) && (i_wb_ack == 1'b1)) begin
            o_wb_address <= 'd0;
            o_wb_data    <= 'd0;
            o_wb_we      <= 'd0;
```

```
            o_wb_cyc      <= 'd0;
            o_wb_stb      <= 'd0;
            o_wb_sel      <= 'hffff;    //have to conform
        end
        else begin
            o_wb_address <= o_wb_address;
            o_wb_data     <= o_wb_data;
            o_wb_we       <= o_wb_we;
            o_wb_cyc      <= o_wb_cyc;
            o_wb_stb      <= o_wb_stb;
            o_wb_sel      <= o_wb_sel;    //have to conform
        end
    end
end

//Latch on ack
//
always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1) begin
        mem_data_read <= 'd0;
        wb_read_done  <= 'd0;
    end
    else begin
        if ((wb_read == 1'b1) && ((i_wb_ack == 1'b1) && (i_wb_err ==
            1'b0))) begin
            mem_data_read <= i_wb_data;
            wb_read_done <= 'd1;
        end
        else  begin
            mem_data_read <= mem_data_read;
            wb_read_done <= 'd0;
        end
    end
end

//to do read based on select bank only 32 bit
//Send memory read mem_read_data

reg  mem_out;

always @(posedge clk, posedge reset)
begin
```

```
    if (reset == 1'b1) begin
        //mem_out      <= 'd0;
        mem_read_data <= 'd0;
    end
    else begin
        if ((send_mem_read == 1'b1) && (done_count != 5'b11111)) begin
            mem_read_data <= (mem_data_read[31:0] >> (done_count + 1'
                b1));
            //mem_out  <= mem_read_data[0];
        end
        else begin
            //mem_out  <= 'd0;
            mem_read_data <= 'd0;
        end
    end
end

always @(posedge clk, posedge reset)
begin
    if (reset == 1'b1)
        done_count <= 'd0;
    else  begin
        if (send_mem_read == 1'b1)
            done_count <= done_count + 1'b1;
        else
            done_count <= 'd0;
    end
end

//have to mux out the pin for strobe
//

always @(*)
begin
    case (state_process)
        5'd18: output_stb = send_ok;
        5'd20: output_stb = send_err_code;
        5'd19: output_stb = send_cpu_state;
        5'd16: output_stb = send_trace;
        5'd17: output_stb = send_mem_read;
        default: output_stb = 1'b0;
    endcase
end
```

```
always @(*)
begin
    case (state_process)
        5'd18: serial_data_out = ok_reg[0];
        5'd20: serial_data_out = error_reg[0];
        5'd19: serial_data_out = send_buffer_state[0];
        5'd16: serial_data_out = send_buffer_trace[0];
        5'd17: serial_data_out = mem_read_data[0];
        default: serial_data_out = 1'b1;
    endcase
end

endmodule // DebugModule
```

Listing I.3: Debug Module Test Bench

```
module test;

wire  scan_out0;

reg  clk, reset;
reg  scan_in0, scan_en, test_mode;
wire  output_stb;
reg instruction_stb, data_stb;
reg [3:0] instruction_in;
reg [31:0] debug_state_bus;
wire [5:0] state_read_control_signal;
wire serial_data_out, halt_core;
reg [31:0] pc_input;
reg [31:0] data_in;
wire [4:0] state;
DebugModule top(
        .reset(reset),
        .clk(clk),
        .scan_in0(scan_in0),
        .scan_en(scan_en),
        .test_mode(test_mode),
        .scan_out0(scan_out0),
        .instruction_stb(instruction_stb),
        .data_stb(data_stb),
        .instruction_in(instruction_in),
        .data_in(data_in),
        .state_read_control_signal(state_read_control_signal),
        .debug_state_bus(debug_state_bus),
        .halt_core(halt_core),
        .serial_data_out(serial_data_out),
        .output_stb(output_stb),
        .pc_input(pc_input)
        //.state(state)
    );


initial
begin
    $timeformat(-9,2,"ns", 16);
`ifdef SDFSCAN
```

```
    $sdf_annotate("sdf/DebugModule_tsmc18_scan.sdf", test.top);
`endif
    clk = 1'b0;
    reset = 1'b1;
    scan_in0 = 1'b0;
    scan_en = 1'b0;
    test_mode = 1'b0;
    instruction_stb = 1'b0;
    data_stb = 1'b0;
    instruction_in = 32'd0;
    pc_input = 32'd0;
    #40
    reset = 1'b0;
    #10000
    instruction_in = 4'b1010;
    instruction_stb = 1'b1;
    #100
    instruction_stb = 1'b0;
    #100
    data_in = 32'h00008000;
    data_stb = 1'b1;
    #10
    data_stb = 1'b0;
end

always @(state_read_control_signal)
begin
    case (state_read_control_signal)
        6'b000000: debug_state_bus = 32'd1;
        6'b000001: debug_state_bus = 32'd2;
        6'b000010: debug_state_bus = 32'd3;
        6'b000011: debug_state_bus = 32'd4;
        6'b000100: debug_state_bus = 32'd5;
        6'b000101: debug_state_bus = 32'd6;
        6'b000110: debug_state_bus = 32'd7;
        6'b000111: debug_state_bus = 32'd8;
        6'b001000: debug_state_bus = 32'd9;
        6'b001001: debug_state_bus = 32'd10;
        6'b001010: debug_state_bus = 32'd11;
        6'b001011: debug_state_bus = 32'd12;
        6'b001100: debug_state_bus = 32'd13;
        6'b001101: debug_state_bus = 32'd14;
        6'b001110: debug_state_bus = 32'd15;
```

```
        default: debug_state_bus = 32'd0;
    endcase
end

// 50 MHz clock
initial
begin
   forever
    #10 clk = ~clk ;
end
/*
initial  begin
    $monitor ("time = %d state = %d", $time, state);
*/
always
    program_counter;


/*
initial begin
    $monitor ("time = %d state_process = %d", $time, w_state_process);
end
*/

/*
integer i;

task instruction;
input [31:0]  instruct;
input [31:0]  data;
begin
    @(posedge clk);
    instruction_reg = instruct;
    @(posedge clk);
    control_stb = 1'b1;
    @(posedge clk);
    control_stb = 1'b0;
    repeat(4)
        @(posedge clk);
    for (i = 0; i < 32; i = i+1) begin
        @(posedge clk);
        debug_stb = 1'b1;
        serial_data_in = data[31];
```

```
            data = data << 1;
        end
        @(posedge clk);
        debug_stb = 1'b0;
end
endtask
*/

//pc_input
reg [31:0] pc;
reg [2:0] p;
task program_counter;
begin
        pc = $random;
        forever
        begin
           // if (halt_core == 1'b1)
            //     pc_input = pc_input;
            //else begin
                p = $random;
                @(posedge clk);
                if (p > 3'b011)
                    pc_input = pc + 32'd64;
                else
                    pc_input = pc + 32'd4;
            //end
         end
end
endtask




task trace_data;
begin

end
endtask

endmodule
```