

External Debug Support For Shakti Processor

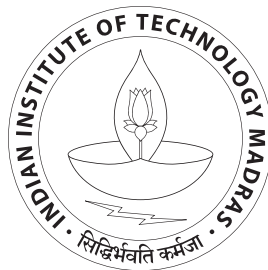
A Project Report

submitted by

Y.RAMA MUNI REDDY

*in partial fulfilment of the requirements
for the award of the degree of*

MASTER OF TECHNOLOGY



**DEPARTMENT OF ELECTRICAL ENGINEERING
INDIAN INSTITUTE OF TECHNOLOGY MADRAS.**

MAY 2018

THESIS CERTIFICATE

This is to certify that the report titled **External Debug Support For Shakti Processor**, submitted by **Y.RAMA MUNI REDDY**, to the Indian Institute of Technology, Madras, for the award of **Master of Technology**, is a bonafide record of the work done by him under my supervision. The contents of this report, in full or in parts, have not been submitted to any other Institute or University for the award of any degree or diploma.

Dr. V Kamakoti
Project Guide
Professor
Dept. of Computer Science and
Engineering
IIT-Madras, 600 036

Place: Chennai

Date:

ACKNOWLEDGEMENTS

I would like to express my sincere gratitude to my project guide Prof. V. Kamakoti sir for his guidance and support and for giving me opportunity to work for this project.

I would like to thank my co-guide Prof. Saurabh Saxena sir who has motivated me and supported me to work outside electrical department for this project.

I would also like to thank Dr. Neel Gala, Mr. Arjun, Mr. Rahul, Mr. Vinod and all my RISE lab mates who were always supportive and helped me throughout my project and cleared my doubts.

Lastly I would like to thank my parents for always standing by me and encouraging me to pursue higher studies at IIT Madras.

Y.RAMA MUNI REDDY

ABSTRACT

When a design progresses from simulation to hardware implementation, a users control and understanding of the systems current state drops dramatically. To help bring up and debug low level software and hardware, it is critical to have good debugging support built into the hardware.

This thesis outlines a standard architecture for external debug support on SHAKTI platforms. This architecture allows a variety of implementations like C-class, I-class, E-class. At the same time, this architecture defines common interfaces to allow debugging tools and components to target a variety of platforms. It can connect "blind" to any SHAKTI platform, and discover everything it needs to know without depending on the quality of a vendor's debugging toolchain.

It is regarded as the best friend of a software programmer as it externally allows us to run the program up to a certain point, then stop and print out the values of certain variables at that point, or step through the program one line at a time and print out the values of each variable after executing each line.

It consists of basic features like selecting harts, halt-resume, abstract commands, program buffer, single stepping.

TABLE OF CONTENTS

| | |
|--|-------------|
| ACKNOWLEDGEMENTS | i |
| ABSTRACT | ii |
| LIST OF TABLES | vi |
| LIST OF FIGURES | viii |
| ABBREVIATIONS | ix |
| 1 Introduction | 1 |
| 1.1 Supported Features | 1 |
| 1.2 Terminology | 1 |
| 2 System Overview | 2 |
| 3 Bluespec System Verilog | 4 |
| 3.1 Limitation of Verilog | 4 |
| 3.2 Bluespec | 5 |
| 3.3 Features of Bluespec | 6 |
| 3.3.1 Modules and Interfaces | 6 |
| 3.3.2 Rules | 7 |
| 3.3.3 Methods | 8 |
| 3.3.4 TLM Library | 8 |
| 4 JTAG TAP | 10 |
| 4.1 JTAG Debug Transport Module | 10 |
| 4.1.1 JTAG Background | 10 |
| 4.1.2 TAP controller state diagram | 10 |

| | | |
|----------|---|-----------|
| 4.1.3 | JTAG DTM Registers | 12 |
| 4.1.4 | IDCODE (at 0x01) | 12 |
| 4.1.5 | DTM Control and Status (dtmcs, at 0x10) | 14 |
| 4.1.6 | Debug Module Interface Access (dmi, at 0x11) | 16 |
| 4.1.7 | BYPASS (at 0x1f) | 19 |
| 5 | OpenOCD | 20 |
| 5.1 | What is OpenOCD? | 20 |
| 5.2 | Debug Adapter Configuration | 21 |
| 5.3 | Interface Drivers | 21 |
| 5.4 | Remote_bitbang Jtag driver | 22 |
| 5.5 | Server Configuration | 24 |
| 5.5.1 | Configuration Stage | 24 |
| 5.5.2 | Entering the Run Stage | 24 |
| 5.6 | Connecting to GDB | 25 |
| 6 | Debug Module | 26 |
| 6.1 | Selecting Harts | 26 |
| 6.2 | Run Control | 27 |
| 6.3 | Abstract Commands | 27 |
| 6.4 | Abstract Command Listing | 28 |
| 6.5 | Program Buffer | 31 |
| 6.6 | Overview of States | 33 |
| 6.7 | System Bus Access | 33 |
| 6.8 | Debug Module DMI Registers | 33 |
| 6.8.1 | Debug Module Status (dmstatus, at 0x11) | 35 |
| 6.8.2 | Debug Module Control (dmcontrol, at 0x10) | 38 |
| 6.8.3 | Hart Info (hartinfo, at 0x12) | 43 |
| 6.8.4 | Halt Summary (haltsum, at 0x13) | 45 |
| 6.8.5 | Abstract Control and Status (abstractcs, at 0x16) | 46 |
| 6.8.6 | Abstract Command (command, at 0x17) | 49 |

| | | |
|----------|---|-----------|
| 6.8.7 | Abstract Command Autoexec (abstractauto, at 0x18) . . | 50 |
| 6.8.8 | Abstract Data 0 (data0, at 0x04) | 50 |
| 6.8.9 | Program Buffer 0 (progbuf0, at 0x20) | 51 |
| 6.8.10 | Authentication Data (authdata, at 0x30) | 51 |
| 7 | Implementation and Design Challenges | 52 |
| 7.1 | JTAG TAP | 52 |
| 7.2 | Debug Module | 53 |
| 7.3 | Servers and Clients | 54 |
| 7.4 | Halting and Resuming | 55 |
| 7.5 | Abstract Commands | 56 |
| 7.6 | Program Buffer | 57 |
| 7.7 | Single stepping | 59 |
| 8 | RESULTS | 61 |
| 8.1 | Servers and Clients | 61 |
| 8.2 | Reading Idcode register | 62 |
| 8.3 | OpenOCD examining DTM Control register | 62 |
| 8.4 | Getting hart information | 62 |
| 8.5 | Halting and Resuming | 63 |
| 8.6 | Reading General Purpose registers | 64 |
| 8.7 | Program Buffer | 65 |
| 8.8 | Setting PC value | 66 |
| 8.9 | Loading instructions and data into memory | 68 |
| 8.10 | Setting breakpoints | 69 |
| 8.11 | stepping | 69 |
| 9 | CONCLUSION AND FUTURE WORK | 73 |

LIST OF TABLES

| | | |
|-----|--|----|
| 4.1 | JTAG DTM TAP Registers | 13 |
| 6.1 | Meaning of cmdtype | 28 |
| 6.4 | Debug Module Debug Bus Registers | 33 |

LIST OF FIGURES

| | | |
|------|--|----|
| 2.1 | Debug System Overview | 3 |
| 3.1 | Block Diagram:Representation of Methods,Interfaces,Rules in a module hierarchy | 6 |
| 3.2 | Block Diagram: Representation of TLMSendIFC and TLMRecvIFC | 9 |
| 4.1 | Top level view of TAP Controller | 11 |
| 4.2 | State transition diagram of TAP controller | 12 |
| 4.3 | Bypass register | 19 |
| 6.1 | Run/Halt Debug State Machine | 34 |
| 7.1 | Approach-1 | 52 |
| 7.2 | Approach-2 Overview | 53 |
| 7.3 | Approach-2 in brief | 53 |
| 7.4 | Final Approach | 54 |
| 7.5 | Servers and Clients | 54 |
| 7.6 | Halting the processor | 55 |
| 7.7 | Resuming the processor | 56 |
| 7.8 | Use of abstract commands | 56 |
| 7.9 | Program Buffer as Slave to the Syatem Bus | 58 |
| 7.10 | Program Buffer Execution process | 58 |
| 7.11 | Single Stepping | 60 |
| 8.1 | Remote Process as server | 61 |
| 8.2 | GDB connecting to OpenOCD | 61 |
| 8.3 | Reading IDCODE register | 62 |
| 8.4 | Reading DTM control register | 62 |
| 8.5 | Getting hart information | 62 |

| | | |
|------|---|----|
| 8.6 | Sending halt request | 63 |
| 8.7 | Sending resume request | 63 |
| 8.8 | Reading General Purpose Registers | 64 |
| 8.9 | Setting postexec bit value | 65 |
| 8.10 | Executing Program Buffer registers | 66 |
| 8.11 | GDB terminal:Setting PC value | 67 |
| 8.12 | OpenOCD Terminal:Setting PC value | 68 |
| 8.13 | Loading instructions and data into memory | 68 |
| 8.14 | Setting breakpoints and listing them | 69 |
| 8.15 | Setting step bit in dcsr | 70 |
| 8.16 | Stepping | 71 |
| 8.17 | Stepping after setting PC value | 71 |
| 8.18 | Reading dpc value | 72 |

ABBREVIATIONS

| | |
|----------------|---------------------------------------|
| IITM | Indian Institute of Technology Madras |
| DTM | Debug Transport Module |
| DM | Debug Module |
| OpenOCD | Open On Chip Debugger |
| GDB | GNU Debugger |
| DPC | Debug Program Counter |
| DCSR | Debug Control and Status Register |
| GPRs | General Purpose Register's |
| CSRs | Control and Status Register's |

CHAPTER 1

Introduction

1.1 Supported Features

The debug interface described in this thesis supports the following features:

1. Processor 32,64, and future 128 are all supported.
2. Any hart in the platform can be independently debugged.
3. A debugger can discover almost everything it needs to know itself, without user configuration.
4. Each hart can be debugged from the very first instruction executed.
5. The hart can be halted when a software breakpoint instruction is executed.
6. Hardware single-step can execute one instruction at a time.
7. Debug functionality is independent of the debug transport used.
8. The debugger does not need to know anything about the microarchitecture of the cores it is debugging.
9. Arbitrary subsets of harts can be halted and resumed simultaneously. (Optional)
10. Registers can be accessed without halting. (Optional)

1.2 Terminology

A platform is a single integrated circuit consisting of one or more components. Some components may be processor cores, while others may have a different function. Typically they will all be connected to a single system bus. A single core contains one or more hardware threads, called harts.

CHAPTER 2

System Overview

The figure shown below 2.1 shows the main components of External Debug Support. The user interacts with the Debug Host (eg. laptop), which is running a debugger (eg. gdb). The debugger communicates with a Debug Translator (eg. OpenOCD, which may include a hardware driver) to communicate with Debug Transport Hardware (eg. Olimex USB-JTAG adapter). The Debug Transport Hardware connects the Debug Host to the Platform's Debug Transport Module (DTM). The DTM provides access to the Debug Module (DM).

The DM allows the debugger to halt any hart in the platform. Abstract commands provide access to GPRs. Additional registers are accessible through abstract commands or by writing programs to the optional Program Buffer. The Program Buffer allows the debugger to execute arbitrary instructions on a hart. This mechanism can be used to access memory. An optional system bus access block allows memory accesses without using a hart to perform the access.

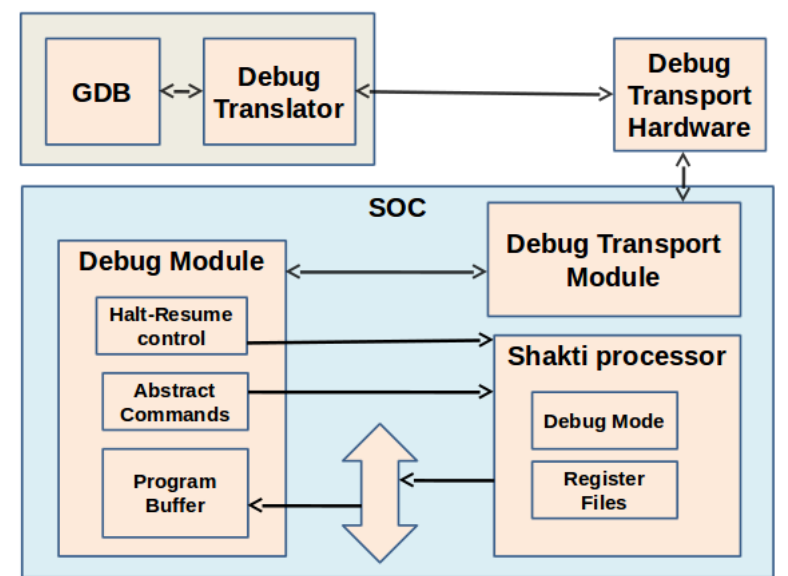


Figure 2.1: Debug System Overview

CHAPTER 3

Bluespec System Verilog

Bluespec System Verilog is a Hardware Description Language (HDL), which is used for specification, synthesis, modeling and verification of ASIC and FPGA design. With a radically different approach to highlevel synthesis, bluespec offers significantly higher productivity. It allows designers to express intended hardware through high-level constructs, where all behavior is described as a set of guarded atomic actions.

3.1 Limitation of Verilog

Verilog focusses more on simulation than logic synthesis. The source text of verilog often explicitly contains aspects of circuit that could be readily determined by the compiler, such as size of registers, width of busses etc. This makes the design less portable. Handling concurrency in hardware is relatively difficult in verilog as the designer should manage all the aspects of handshaking between combinational circuits. Shared use of register and other memory resources should also be elaborated. The behavioral specification of design in verilog often consumes multiple clock cycles. Attempts to resolve this problem results in a highly unreadable code with possible bugs. In practice, this problem is solved by separating the combinational and sequential parts of the circuit. Due to these shortcomings, the synthesis and verification of hardware in verilog is slowed down. This is a huge problem during the design of SOC.

3.2 Bluespec

Bluespec is based on atomic transactions, which increases the level of concurrency abstraction above SystemC and RTL without compromising the control over hardware design. It enables automatic synthesis of complex control logic, which is the source of many bugs. This results in highly adaptable, reusable and reconfigurable designs. Control adaptive parametrization in bluespec provides flexibility, where a significantly different micro-architecture can be generated by changing the parameters in the design with the associated control structures generated automatically. Bluespec allows user defined data types and static type checking. It provides several features of the modern high level languages and all of them can be synthesized.

In recent times, several attempts have been made to move the hardware design language towards a more software like specification of the circuit behaviour. Languages like C, C++ are used to express designs as sequential programs. However, the semantic gap between the software model and the hardware results in suboptimal designs with unpredictable speed and area. Bluespec System Verilog tackles this problem by building upon the traditional hardware semantics. It exploits advanced concepts from software only for static elaboration and static verification. It uses the standard hardware structure model of verilog such as modules, module instances, hierarchy etc. For communication between modules it uses the System verilog model of interfaces and interface instances. These added with the advanced features of the high level languages, makes designing and verification in bluespec much faster.

3.3 Features of Bluespec

3.3.1 Modules and Interfaces

Module is the basic element of the hardware design hierarchy in bluespec. A module can be instantiated multiple times, and also different parameters can be passed during every instantiation. Unlike verilog, bluespec does not have input, output and in-out pins as interface to modules. Methods are used to drive signals and busses in and out of modules. These methods are grouped together into interfaces. Modules contain rules, which use methods in other modules.

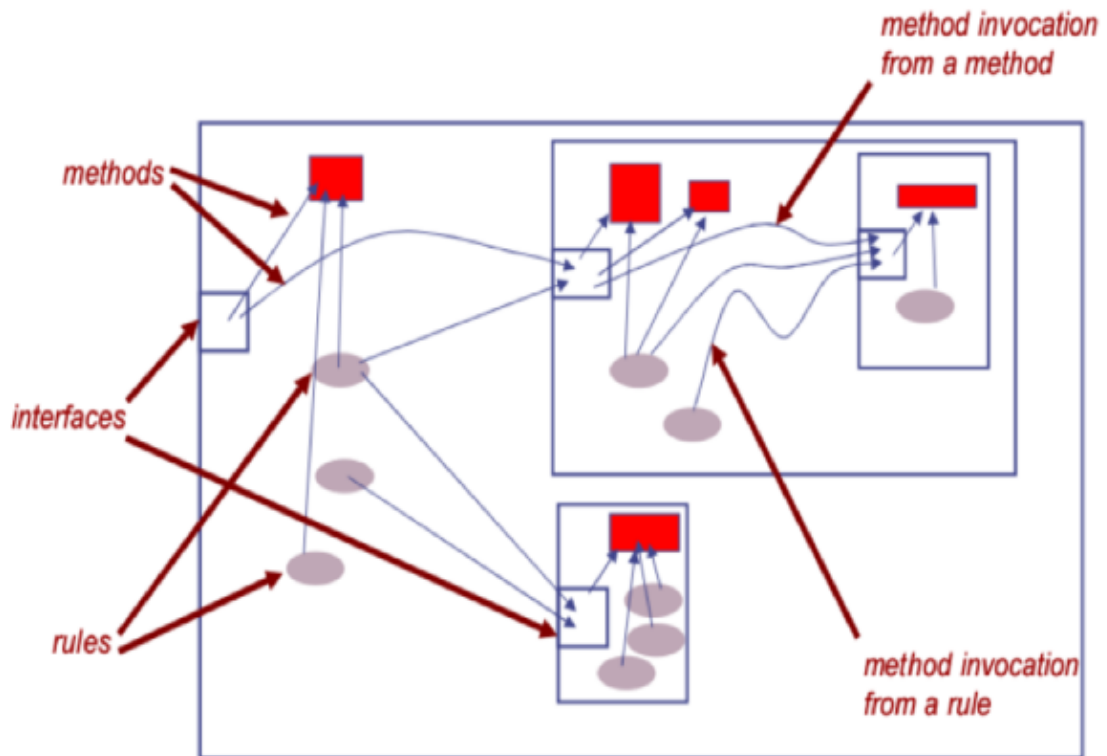


Figure 3.1: Block Diagram:Representation of Methods,Interfaces,Rules in a module hierarchy

In BSV, the interface declaration is done separately, outside the module definition. This allows declaration of common interfaces which can be used in multiple

modules, without having to methods and therefore share same number and type of inputs and outputs.

3.3.2 Rules

Rules manage the movement of data from one state to another, within the module. It consists of two parts: rule conditions and rule body. Rule conditions are boolean expressions which decide whether the rule can be fired. Rule body is a set of actions for state transitions. Rules in BSV are atomic. The actions within the rule completely describes the state transition. The process of determining the functional correctness of a design is greatly simplified by one-rule-at-a-time semantics. That is, because of the atomic property of rules, each rule can be looked at declare them repeatedly. All the modules which share the same interfaces also share same in isolation, without considering the actions of the other rules to determine functional correctness. Multiple rules can be executed concurrently in the hardware implementation.

The actions in a rule are executed simultaneously. This can be thought of as similar to the execution of non-blocking statements in always blocks of verilog. Also, as the rule has atomic property, the entire body of rule is executed and there is no partial execution of a rule. When there are several rules within a module, the execution of rules is ordered by the compiler. No two rules can execute simultaneously. The ordering of the rules by the compiler is called scheduling.

3.3.3 Methods

Method is a procedure which takes arguments and returns a value. It could also return a value without taking any arguments. It becomes a bundle of wires when translated into RTL. The method definition is written within the definition of the interface and it can be different in different modules sharing a common interface. A method also contains implicit conditions which are handshaking signals and logic automatically generated by the compiler. Methods are of three types: Value Methods, Action Methods and Action Value Methods. Value methods return a value. They do not alter any state within the module. Action methods cause actions to occur. They create state changes within the module. Action value methods are a combination of value methods and action methods. They cause state changes and also return values.

3.3.4 TLM Library

The TLM package includes definitions of interfaces, data structures, and module constructors which allow users to create and modify bus-based designs in a manner that is independent of any one specific bus protocol. Designs created using the TLM package are thus more portable as it allows the core design to be easily applied to multiple bus protocols.

The TLM interfaces define how TLM blocks interconnect and communicate. The TLM package includes two basic interfaces: The TLMSendIFC interface and the TLMRecvIFC interface. These interfaces use basic Get and Put sub-interfaces as the requests and responses. The TLMSendIFC interface generates (Get) requests and receives (Put) responses. The TLMRecvIFC interface receives (Put) requests

and generates (Get) responses. Additional TLM interfaces are built up from these basic blocks. The TLMSendIFC interface transmits the requests and receives the responses. The TLMRecvIFC interface receives the requests and transmits the responses.

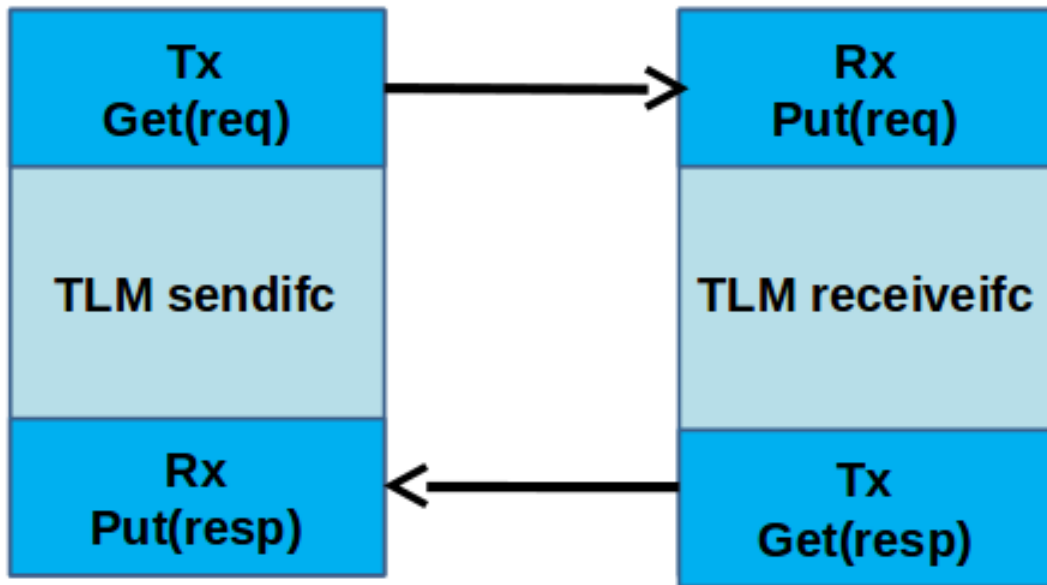


Figure 3.2: Block Diagram: Representation of TLMSendIFC and TLMRecvIFC

The two basic data structures defined in the TLM package are TLMRequest and TLMResponse. By using these types in a design, the underlying bus protocol can be changed without having to modify the interactions with the TLM objects. A TLM request contains either control information and data, or data alone. A TLMRequest is tagged as either a RequestDescriptor or RequestData. A RequestDescriptor contains control information and data while a RequestData contains only data.

CHAPTER 4

JTAG TAP

4.1 JTAG Debug Transport Module

This Debug Transport Module is based around a normal JTAG Test Access Port (TAP). The JTAG TAP allows access to arbitrary JTAG registers by first selecting one using the JTAG instruction register (IR), and then accessing it through the JTAG data register (DR).

4.1.1 JTAG Background

JTAG refers to IEEE Std 1149.1. It is a standard that defines test logic that can be included in an integrated circuit to test the interconnections between integrated circuits, test the integrated circuit itself, and observe or modify circuit activity during the components normal operation. The JTAG standard defines a Test Access Port (TAP) that can be used to read and write a few custom registers, which can be used to communicate with debug hardware in a component.

4.1.2 TAP controller state diagram

The operation of the test interface is controlled by the Test Access Port (TAP) controller. This is a 16-state finite state-machine whose state transitions are controlled by the TMS signal. The state transition diagram is shown in Figure 4.2. The TAP

controller can change state only at the rising edge of TCK and the next state is determined by the logic level of TMS. The TAP consists of four mandatory terminals plus one optional terminal as shown in Figure 4.1.

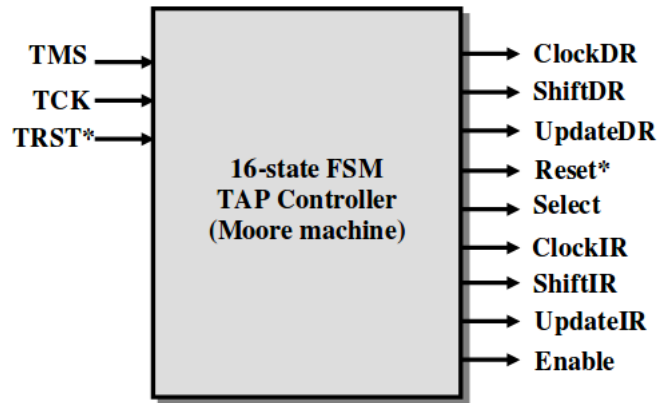
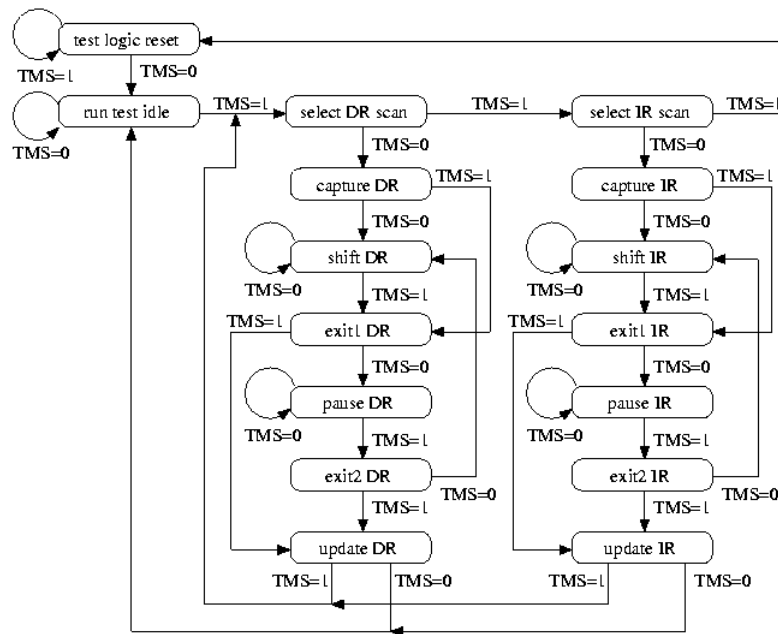


Figure 4.1: Top level view of TAP Controller

The main functions of the TAP controller are:

- To select the output of instruction or test data to shift out to TDO.
- To provide control signals to load instructions into Instruction Register.
- To provide signals to shift test data from TDI and test response to TDO.
- To provide signals to perform test functions such as capture and application of test data.



JTAG Test Access Port (TAP) controller state transition diagram

Figure 4.2: State transition diagram of TAP controller

4.1.3 JTAG DTM Registers

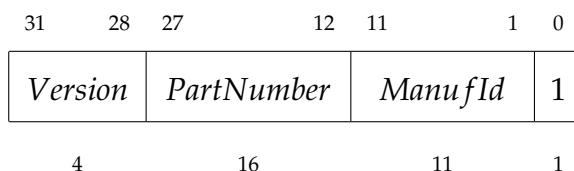
JTAG TAPs used as a DTM must have an IR of at least 5 bits. When the TAP is reset, IR must default to 00001, selecting the IDCODE instruction. A full list of JTAG registers along with their encoding is in Table 4.1. The only regular JTAG registers a debugger might use are BYPASS and IDCODE. Unimplemented instructions must select the BYPASS register.

4.1.4 IDCODE (at 0x01)

This register is selected (in IR) when the TAP state machine is reset. Its definition is exactly as defined in IEEE Std 1149.1. This entire register is read-only.

Table 4.1: JTAG DTM TAP Registers

| Address | Name | Description | Page |
|---------|-------------------------------|--------------------------------------|----------|
| 0x00 | BYPASS | JTAG recommends this encoding | 14 16 |
| 0x01 | IDCODE | JTAG recommends this encoding | |
| 0x10 | DTM Control and Status | For Debugging | |
| 0x11 | Debug Module Interface Access | For Debugging | |
| 0x12 | Reserved (BYPASS) | Reserved for future RISC-V debugging | |
| 0x13 | Reserved (BYPASS) | Reserved for future RISC-V debugging | |
| 0x14 | Reserved (BYPASS) | Reserved for future RISC-V debugging | |
| 0x15 | Reserved (BYPASS) | Reserved for future RISC-V standards | |
| 0x16 | Reserved (BYPASS) | Reserved for future RISC-V standards | |
| 0x17 | Reserved (BYPASS) | Reserved for future RISC-V standards | |
| 0x1f | BYPASS | JTAG requires this encoding | |



| Field | Description | Access | Reset |
|------------|---|--------|--------|
| Version | Identifies the release version of this part. | R | Preset |
| PartNumber | Identifies the designer's part number of this part. | R | Preset |
| Manu fId | Identifies the designer/manufacture of this part. Bits 6:0 must be bits 6:0 of the designer/manufacture's Identification Code as assigned by JEDEC Standard JEP106. Bits 10:7 contain the modulo-16 count of the number of continuation characters (0x7f) in that same Identification Code. | R | Preset |

4.1.5 DTM Control and Status (dtmcs, at 0x10)

The size of this register will remain constant in future versions so that a debugger can always determine the version of the DTM.



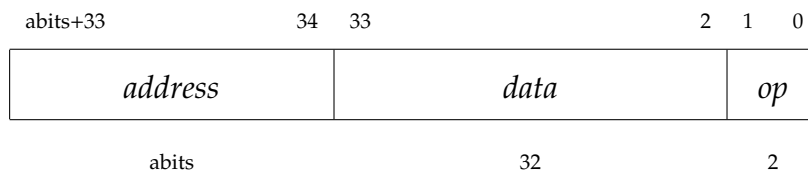
| Field | Description | Access | Reset |
|--------------|--|--------|-------|
| dmihardreset | Writing 1 to this bit does a hard reset of the DTM, causing the DTM to forget about any outstanding DMI transactions. In general this should only be used when the Debugger has reason to expect that the outstanding DMI transaction will never complete (e.g. a reset condition caused an inflight DMI transaction to be cancelled). | W1 | 0 |
| dmireset | Writing 1 to this bit clears the sticky error state and allows the DTM to retry or complete the previous transaction. | W1 | 0 |

| | | | |
|---------|---|---|--------|
| idle | <p>This is a hint to the debugger of the minimum number of cycles a debugger should spend in Run-Test/Idle after every DMI scan to avoid a 'busy' return code dmistat of 3. A debugger must still check dmistat when necessary.</p> <p>0: It is not necessary to enter Run-Test/Idle at all.</p> <p>1: Enter Run-Test/Idle and leave it immediately.</p> <p>2: Enter Run-Test/Idle and stay there for 1 cycle before leaving.</p> <p>And so on.</p> | R | Preset |
| dmistat | <p>0: No error.</p> <p>1: Reserved. Interpret the same as 2.</p> <p>2: An operation failed (resulted in op of 2).</p> <p>3: An operation was attempted while a DMI access was still in progress (resulted in op of 3).</p> | R | 0 |
| abits | The size of address in dmi. | R | Preset |

| | | | |
|---------|--|---|---|
| version | 0: Version described in spec version 0.11. 1: Version described in spec version 0.13 (and later?), which reduces the DMI data width to 32 bits. 15: Version not described in any available version of this spec. | R | 1 |
|---------|--|---|---|

4.1.6 Debug Module Interface Access (dmi, at 0x11)

This register allows access to the Debug Module Interface (DMI). In Update-DR, the DTM starts the operation specified in *op* unless the current status reported in *op* is sticky. In Capture-DR, the DTM updates data with the result from that operation, updating *op* if the current *op* isn't sticky.



| Field | Description | Access | Reset |
|---------|--|--------|-------|
| address | Address used for DMI access. In Update-DR this value is used to access the DM over the DMI. | R/W | 0 |
| data | The data to send to the DM over the DMI during Update-DR, and the data returned from the DM as a result of the previous operation. | R/W | 0 |

| | | | |
|----|--|-----|---|
| op | <p>When the debugger writes this field, it has the following meaning:</p> <p>0: Ignore data and address. (nop)</p> <p>Don't send anything over the DMI during Update-DR. This operation should never result in a busy or error response. The address and data reported in the following Capture-DR are undefined.</p> <p>1: Read from address. (read)</p> <p>2: Write data to address. (write)</p> <p>3: Reserved.</p> <p>When the debugger reads this field, it means the following:</p> <p>0: The previous operation completed successfully.</p> <p>1: Reserved.</p> <p>2: A previous operation failed.</p> <p>3: An operation was attempted while a DMI request is still in progress. The data scanned into dmi in this access will be ignored.</p> | R/W | 2 |
|----|--|-----|---|

4.1.7 BYPASS (at 0x1f)

1-bit register that has no effect. It is used when a debugger does not want to communicate with this TAP. This entire register is read-only.

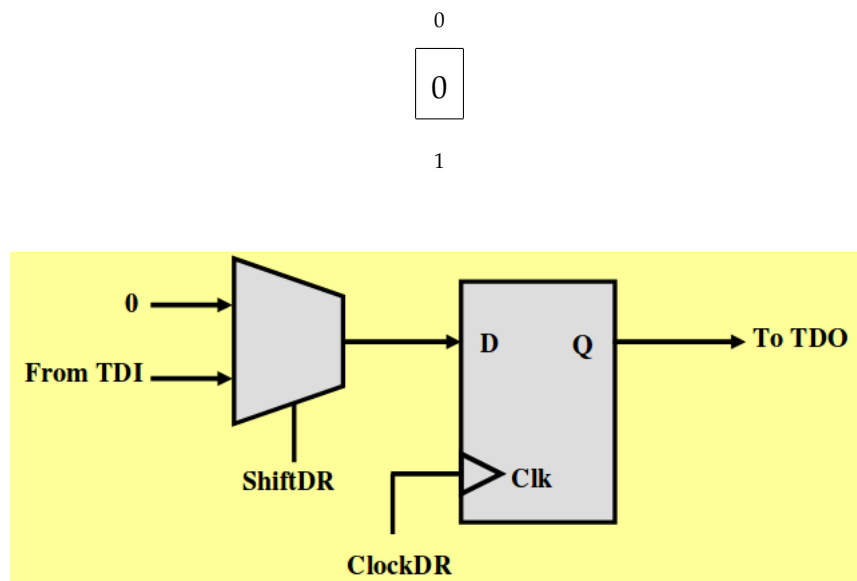


Figure 4.3: Bypass register

CHAPTER 5

OpenOCD

5.1 What is OpenOCD?

The Open On-Chip Debugger (OpenOCD) aims to provide debugging, in-system programming and boundary-scan testing for embedded target devices.

It does so with the assistance of a debug adapter, which is a small hardware module which helps in providing the right kind of electrical signal to the target being debugged. These are required since the debug host (on which OpenOCD runs) won't usually have native support for such signaling, or the connector needed to hook up to the target.

Such debug adapters support one or more transport protocols, each of which involves different electrical signaling (and uses different messaging protocols on top of that signaling). There are many types of debug adapters, and little uniformity in what they are called.

For example, a JTAG Adapter supports JTAG signaling, and is used to communicate with JTAG (IEEE 1149.1) compliant TAPs on your target board. A TAP is a "Test Access Port", a module which processes special instructions and data. JTAG supports debugging and boundary scan operations.

5.2 Debug Adapter Configuration

Correctly installing OpenOCD includes making your operating system give OpenOCD access to debug adapters. Once that has been done, Tcl commands are used to select which one is used, and to configure how it is used.

Debug Adapters/Interfaces are normally configured through commands in an interface configuration file which is sourced by your `openocd.cfg` file, or through a command line `-f interface/....cfg` option.

5.3 Interface Drivers

Interface driver explicitly enabled when OpenOCD is configured, in order to be made available at run time. Interface Driver: `remote_bitbang` Drive JTAG from a remote process. This sets up a UNIX or TCP socket connection with a remote process and sends ASCII encoded bitbang requests to that process instead of directly driving JTAG. The `remote_bitbang` driver is useful for debugging software running on processors which are being simulated.

Config Command: `remote_bitbang_port` number Specifies the TCP port of the remote process to connect to or 0 to use UNIX sockets instead of TCP.

Config Command: `remote_bitbang_host` hostname Specifies the hostname of the remote process to connect to using TCP, or the name of the UNIX socket to use if `remote_bitbang_port` is 0.

For example, to connect remotely via TCP to the localhost you might have something like:

```
interface remote_bitbang
```


remote_bitbang_port 10000

remote_bitbang_host localhost

To connect to another process running locally via UNIX sockets with socket named mysocket:

interface remote_bitbang

remote_bitbang_port 0

remote_bitbang_host mysocket

5.4 Remote_bitbang Jtag driver

The remote_bitbang JTAG driver is used to drive JTAG from a remote process. The remote_bitbang driver communicates via TCP or UNIX sockets with some remote process using an ASCII encoding of the bitbang interface. The remote process presumably then drives the JTAG however it pleases. The remote process should act as a server, listening for connections from the openocd remote_bitbang driver. The remote bitbang driver is useful for debugging software running on processors which are being simulated. The bitbang interface consists of the following functions.

blink on

Blink a light somewhere. The argument on is either 1 or 0.

read

Sample the value of tdo.

write tck tms tdi

Set the value of tck, tms, and tdi.

reset trst srst

Set the value of trst, srst.

An additional function, quit, is added to the remote_bitbang interface to indicate there will be no more requests and the connection with the remote driver should be closed. These five functions are encoded in ascii by assigning a single character to each possible request. The assignments are:

B - Blink on

b - Blink off

R - Read request

Q - Quit request

0 - Write 0 0 0

1 - Write 0 0 1

2 - Write 0 1 0

3 - Write 0 1 1

4 - Write 1 0 0

5 - Write 1 0 1

6 - Write 1 1 0

7 - Write 1 1 1

r - Reset 0 0

s - Reset 0 1

t - Reset 1 0

u - Reset 1 1

The read response is encoded in ascii as either digit 0 or 1.

5.5 Server Configuration

The commands here are commonly found in the `openocd.cfg` file and are used to specify what TCP/IP ports are used, and how GDB should be supported.

5.5.1 Configuration Stage

When the OpenOCD server process starts up, it enters a configuration stage which is the only time that certain commands, configuration commands, may be issued. Normally, configuration commands are only available inside startup scripts.

Those configuration commands include declaration of TAPs, flash banks, the interface used for JTAG communication, and other basic setup. The server must leave the configuration stage before it may access or activate TAPs. After it leaves this stage, configuration commands may no longer be issued.

5.5.2 Entering the Run Stage

The first thing OpenOCD does after leaving the configuration stage is to verify that it can talk to the scan chain (list of TAPs) which has been configured. It will warn if it doesn't find TAPs it expects to find, or finds TAPs that aren't supposed to be there. You should see no errors at this point. If you see errors, resolve them by correcting the commands you used to configure the server. Common errors include using an initial JTAG speed that's too fast, and not providing the right IDCODE values for the TAPs on the scan chain.

Config Command: `init`

This command terminates the configuration stage and enters the run stage. This

command normally occurs at or near the end of your openocd.cfg file to force OpenOCD to initialize and make the targets ready.

5.6 Connecting to GDB

OpenOCD can communicate with GDB in two ways:

- A socket (TCP/IP) connection is typically started as follows:
target remote localhost:3333
This would cause GDB to connect to the gdbserver on the local pc using port 3333.
- A pipe connection is typically started as follows:
target remote — openocd -c "gdb_port pipe; log_output openocd.log "
This would cause GDB to run OpenOCD and communicate using pipes (stdin/stdout).

CHAPTER 6

Debug Module

The Debug Module implements a translation interface between abstract debug operations and their specific implementation. It might support the following operations:

1. Give the debugger necessary information about the implementation.
2. Allow any individual hart to be halted and resumed.
3. Provide status on which harts are halted.
4. Provide read and write access to a halted harts GPRs.
5. Provide access to a reset signal that allows debugging from the very first instruction after reset.
6. Provide a Program Buffer to force the hart to execute arbitrary instructions.
7. Allow multiple harts to be halted, resumed, and or reset at the same time.
8. Allow direct System Bus Access. In order to implement memory access, a target must implement either the Program Buffer or System Bus Access.

6.1 Selecting Harts

All debug modules must support selecting a single hart. The debugger can select a hart by writing its index to hartsel. Hart indexes start at 0 and are contiguous until the final index. Up to 1024 harts can be connected to a single DM. The debugger selects a hart, and then subsequent halt, resume, reset, and debugging commands are specific to that hart.

6.2 Run Control

For every hart, the Debug Module contains 3 conceptual bits of state: halt request, resume request, and hart reset. (The hart reset bit is optional.) These bits all reset to 0. A debugger can write them for the currently selected harts through `haltreq`, `resumereq`, and `hartreset` in `dmcontrol`. In addition the DM receives `halted`, `running`, and `resume ack` signals from each hart.

When a running hart receives a halt request, it responds by halting and asserting its `halted` signal. The `halted` signals of all selected harts are reflected in the `allhalted` and `anyhalted` bits. `haltreq` is ignored by halted harts. When a halted hart receives a resume request, it responds by resuming, clearing its `halted` signal, and asserting its `running` signal and `resume ack` signals. The `resume ack` signal is lowered when the resume request is deasserted. These status signals of all selected harts are reflected in `allresumeack`, `anyresumeack`, `allrunning`, and `anyrunning`. `resumereq` is ignored by running harts.

6.3 Abstract Commands

The DM supports a set of abstract commands, most of which are optional. Depending on the implementation, the debugger may be able to perform some abstract commands even when the selected hart is not halted. Debugger can only determine which abstract commands are supported by a given hart in a given state by attempting them and then looking at `cmderr` in `abstractcs` to see if they were successful.

Debugger execute abstract commands by writing them to `command`. Debug-

Table 6.1: Meaning of cmdtype

| cmdtype | command |
|---------|-------------------------|
| 0 | Access Register Command |
| 1 | Quick Access |

gers can determine whether an abstract command is complete by reading busy in abstractcs. If the command takes arguments, the debugger must write them to the data registers before writing to command. If a command returns results, the Debug Module must ensure they are placed in the data registers before busy is cleared.

6.4 Abstract Command Listing

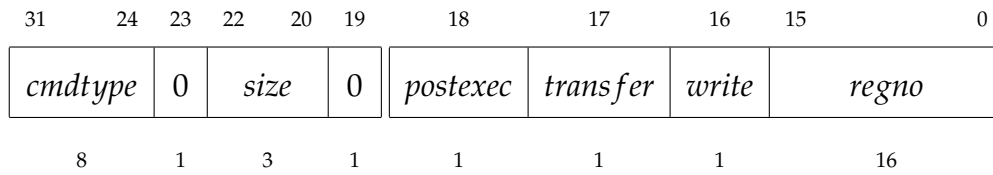
This section describes each of the different abstract commands and how their fields should be interpreted when they are written to command. Each abstract command is a 32-bit value. The top 8 bits contain cmdtype which determines the kind of command. The below table lists all commands.

Access Register

This command gives the debugger access to CPU registers and program buffer. It performs the following sequence of operations:

- Copy data from the register specified by regno into the arg0 region of data, if write is clear and transfer is set.
- Copy data from the arg0 region of data into the register specified by regno, if write is set and transfer is set.
- Execute the Program Buffer, if postexec is set.

If any of these operations fail, `cmderr` is set and none of the remaining steps are executed. An implementation may detect an upcoming failure early, and fail the overall command before it reaches the step that would cause failure. Debug Modules must implement this command and must support read and write access to all GPRs when the selected hart is halted.



| Field | Description |
|----------|--|
| cmdtype | This is 0 to indicate Access Register Command. |
| size | <p>2: Access the lowest 32 bits of the register.</p> <p>3: Access the lowest 64 bits of the register.</p> <p>4: Access the lowest 128 bits of the register.</p> <p>If size specifies a size larger than the register's actual size, then the access must fail. If a register is accessible, then reads of size less than or equal to the register's actual size must be supported.</p> |
| postexec | When 1, execute the program in the Program Buffer exactly once after performing the transfer, if any. |

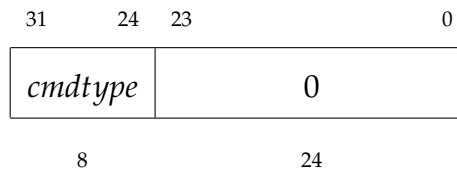
| Field | Description |
|----------|---|
| transfer | <p>0: Don't do the operation specified by write.</p> <p>1: Do the operation specified by write.</p> <p>This bit can be used to just execute the Program Buffer without having to worry about placing valid values into size or regno.</p> |
| write | <p>When transfer is set: 0: Copy data from the specified register into arg0 portion of data.</p> <p>1: Copy data from arg0 portion of data into the specified register.</p> |
| regno | <p>Number of the register to access, as described in Table ?? . dpc may be used as an alias for PC if this command is supported on a non-halted hart.</p> |

Quick Access

Perform the following sequence of operations:

- If the hart is halted, the command sets cmderr to halt/resume and does not continue.
- Halt the hart. If the hart halts for some other reason (e.g. breakpoint), the command sets cmderr to halt/resume and does not continue.
- Execute the Program Buffer. If an exception occurs, cmderr is set to exception and the program buffer execution ends, but the quick access command continues.
- Resume the hart.

Implementing this command is optional.



| Field | Description |
|---------|---|
| cmdtype | This is 1 to indicate Quick Access command. |

6.5 Program Buffer

To support executing arbitrary instructions on a halted hart, a Debug Module can include a Program Buffer that a debugger can write small programs to. Systems

that support all necessary functionality using abstract commands only may choose to omit the Program Buffer.

A debugger can write a small program to the Program Buffer, and then execute it exactly once with the Access Register Abstract Command, setting the postexec bit in command. The debugger can write whatever program it like, but the program must end with ebreak or c.ebreak.

If the debugger executes a program that does not terminate with an ebreak instruction, the hart will remain in Debug Mode until it is reset.

While these programs are executed, the hart does not leave Debug Mode. If an exception is encountered during execution of the Program Buffer, no more instructions are executed, the hart remains in Debug Mode, and cmderr is set to 3 (exception error). If the debugger executes a program that doesn't terminate, then it loses control of the hart.

Executing the Program Buffer may clobber dpc. If that is the case, it must be possible to read/write dpc using an abstract command with postexec not set. The debugger must attempt to save dpc between halting and executing a Program Buffer, and then restore dpc before leaving Debug Mode.

The Program Buffer may be implemented as RAM which is accessible to the hart. A debugger can determine if this is the case by executing small programs that attempt to write and read back relative to pc while executing from the Program Buffer. If so, the debugger has more flexibility in what it can do with the program buffer.

6.6 Overview of States

Figure 6.1 shows a conceptual view of the states passed through by a hart during run/halt debugging as influenced by the different fields of `dmcontrol`, `abstractcs`, `abstractauto`, and `command`.

6.7 System Bus Access

When a Program Buffer is present, a debugger can access the system bus by having a RISC-V hart perform the accesses it requires. A Debug Module may also include a System Bus Access block to provide memory access without involving a hart, regardless of whether Program Buffer is implemented. The System Bus Access block uses physical addresses.

6.8 Debug Module DMI Registers

Table 6.4: Debug Module Debug Bus Registers

| Address | Name | Page |
|---------|-----------------------------|------|
| 0x04 | Abstract Data 0 | 50 |
| 0x0f | Abstract Data 11 | |
| 0x10 | Debug Module Control | 38 |
| 0x11 | Debug Module Status | 35 |
| 0x12 | Hart Info | 43 |
| 0x13 | Halt Summary | 45 |
| 0x16 | Abstract Control and Status | 46 |
| 0x17 | Abstract Command | 49 |
| 0x18 | Abstract Command Autoexec | 50 |
| 0x20 | Program Buffer 0 | 51 |
| 0x2f | Program Buffer 15 | |
| 0x30 | Authentication Data | 51 |

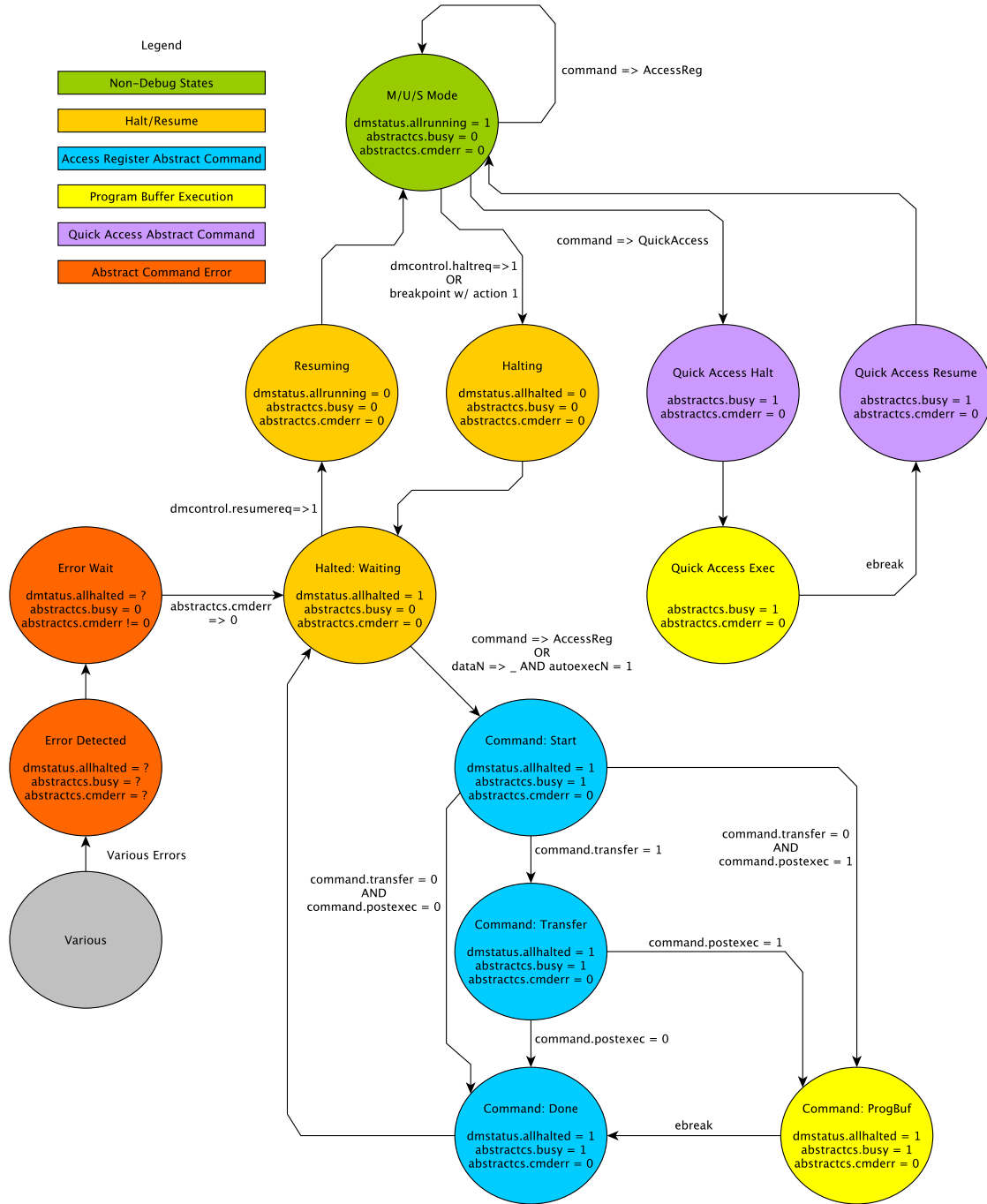
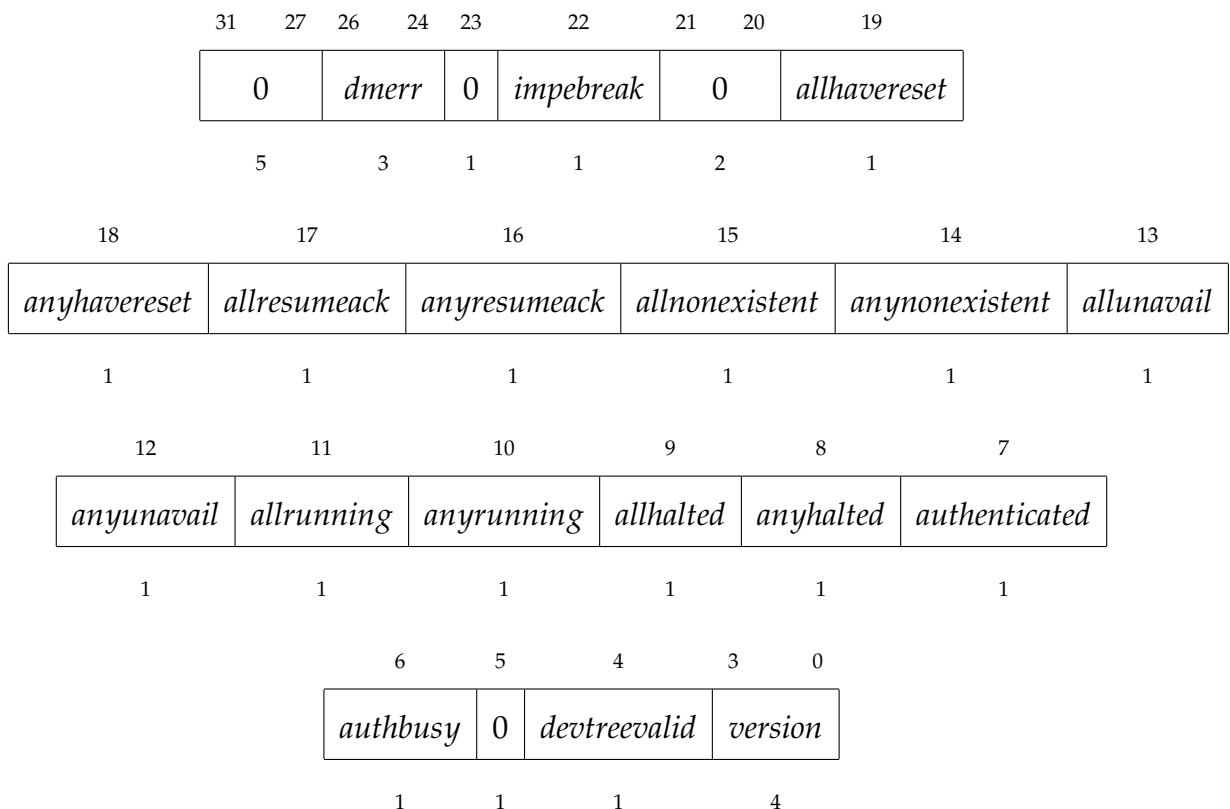


Figure 6.1: Run/Halt Debug State Machine. As only a small amount of state is visible to the debugger, the states and transitions are conceptual.

6.8.1 Debug Module Status (dmstatus, at 0x11)

This register reports status for the overall debug module as well as the currently selected harts, as defined in `hasel`.



| Field | Description | Access | Reset |
|-------|--|--------|-------|
| dmerr | Gets set if the Debug Module was accessed incorrectly. 0 (none): No error. 1 (badaddr): There was an access to an unimplemented Debug Module address. 7 (other): An access failed for another reason. | R/W1C | 0 |

| | | | |
|----------------|---|---|--------|
| impebreak | <p>If 1, then there is an implicit ebreak instruction at the non-existent word immediately after the Program Buffer. This saves the debugger from having to write the ebreak itself, and allows the Program Buffer to be one word smaller.</p> <p>This must be 1 when progbufsize is 1.</p> | R | Preset |
| allhavereset | This field is 1 when all currently selected harts have been reset but the reset has not been acknowledged. | R | - |
| anyhavereset | This field is 1 when any currently selected hart has been reset but the reset has not been acknowledged. | R | - |
| allresumeack | This field is 1 when all currently selected harts have acknowledged the previous resume request. | R | - |
| anyresumeack | This field is 1 when any currently selected hart has acknowledged the previous resume request. | R | - |
| allnonexistent | This field is 1 when all currently selected harts do not exist in this system. | R | - |
| anynonexistent | This field is 1 when any currently selected hart does not exist in this system. | R | - |
| allunavail | This field is 1 when all currently selected harts are unavailable. | R | - |

| | | | |
|---------------|---|---|--------|
| anyunavail | This field is 1 when any currently selected hart is unavailable. | R | - |
| allrunning | This field is 1 when all currently selected harts are running. | R | - |
| anyrunning | This field is 1 when any currently selected hart is running. | R | - |
| allhalted | This field is 1 when all currently selected harts are halted. | R | - |
| anyhalted | This field is 1 when any currently selected hart is halted. | R | - |
| authenticated | 0 when authentication is required before using the DM. 1 when the authentication check has passed. On components that don't implement authentication, this bit must be preset as 1. | R | Preset |
| authbusy | 0: The authentication module is ready to process the next read/write to authdata. 1: The authentication module is busy. Accessing authdata results in unspecified behavior. authbusy only becomes set in immediate response to an access to authdata. | R | 0 |

| | | | |
|--------------|--|---|--------|
| devtreevalid | 0: devtreeaddrzero–devtreeaddrthree hold information which is not relevant to the Device Tree. 1: devtreeaddrzero–devtreeaddrthree registers hold the address of the Device Tree. | R | Preset |
| version | 0: There is no Debug Module present. 1: There is a Debug Module and it con- forms to version 0.11 of this specification. 2: There is a Debug Module and it con- forms to version 0.13 of this specification. 15: There is a Debug Module but it does not conform to any available version of this spec. | R | 2 |

6.8.2 Debug Module Control (dmcontrol, at 0x10)

This register controls the overall debug module as well as the currently selected harts, as defined in hasel.



| Field | Description | Access | Reset |
|-----------|--|--------|-------|
| haltreq | <p>Writes the halt request bit for all currently selected harts. When set to 1, each selected hart will halt if it is not currently halted.</p> <p>Writing 1 or 0 has no effect on a hart which is already halted, but the bit must be cleared to 0 before the hart is resumed.</p> <p>Writes apply to the new value of hartsel and hasel.</p> | W | - |
| resumereq | <p>Writes the resume request bit for all currently selected harts. When set to 1, each selected hart will resume if it is currently halted.</p> <p>The resume request bit is ignored while the halt request bit is set.</p> <p>Writes apply to the new value of hartsel and hasel.</p> | W | - |

| | | | |
|--------------|--|-----|---|
| hartreset | <p>This optional field writes the reset bit for all the currently selected harts. To perform a reset the debugger writes 1, and then writes 0 to deassert the reset signal.</p> <p>If this feature is not implemented, the bit always stays 0, so after writing 1 the debugger can read the register back to see if the feature is supported.</p> <p>Writes apply to the new value of hartsel and hasel.</p> | R/W | 0 |
| ackhavereset | <p>Writing 1 to this bit clears the havereset bits for any selected harts.</p> <p>Writes apply to the new value of hartsel and hasel.</p> | W | - |

| | | | |
|---------|--|-----|---|
| hasel | <p>Selects the definition of currently selected harts.</p> <p>0: There is a single currently selected hart, that selected by hartsel.</p> <p>1: There may be multiple currently selected harts – that selected by hartsel, plus those selected by the hart array mask register.</p> <p>An implementation which does not implement the hart array mask register should tie this field to 0. A debugger which wishes to use the hart array mask register feature should set this bit and read back to see if the functionality is supported.</p> | R/W | 0 |
| hartsel | The DM-specific index of the hart to select. This hart is always part of the currently selected harts. | R/W | 0 |

| | | | |
|----------|---|-----|---|
| ndmreset | <p>This bit controls the reset signal from the DM to the rest of the system. The signal should reset every part of the system, including every hart, except for the DM and any logic required to access the DM. To perform a system reset the debugger writes 1, and then writes 0 to deassert the reset.</p> | R/W | 0 |
|----------|---|-----|---|

| | | | |
|----------|---|-----|---|
| dmactive | <p>This bit serves as a reset signal for the Debug Module itself.</p> <p>0: The module's state, including authentication mechanism, takes its reset values (the dmactive bit is the only bit which can be written to something other than its reset value).</p> <p>1: The module functions normally.</p> <p>No other mechanism should exist that may result in resetting the Debug Module after power up, including the platform's system reset or Debug Transport reset signals.</p> <p>A debugger may pulse this bit low to get the debug module into a known state.</p> <p>Implementations may use this bit to aid debugging, for example by preventing the Debug Module from being power gated while debugging is active.</p> | R/W | 0 |
|----------|---|-----|---|

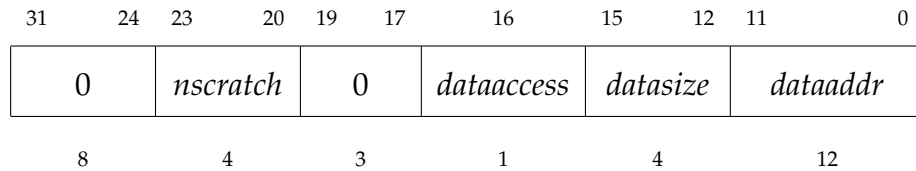
6.8.3 Hart Info (hartinfo, at 0x12)

This register gives information about the hart currently selected by hartsel.

This register is optional. If it is not present it should read all-zero.

If this register is included, the debugger can do more with the Program Buffer by writing programs which explicitly access the data and/or dscratch registers.

This entire register is read-only.



| Field | Description | Access | Reset |
|------------|--|--------|--------|
| nscratch | Number of dscratch registers available for the debugger to use during program buffer execution, starting from dscratchzero. The debugger can make no assumptions about the contents of these registers between commands. | R | Preset |
| dataaccess | 0: The data registers are shadowed in the hart by CSR registers. Each CSR register is XLEN bits in size, and corresponds to a single argument, per Table ??. 1: The data registers are shadowed in the hart's memory map. Each register takes up 4 bytes in the memory map. | R | Preset |

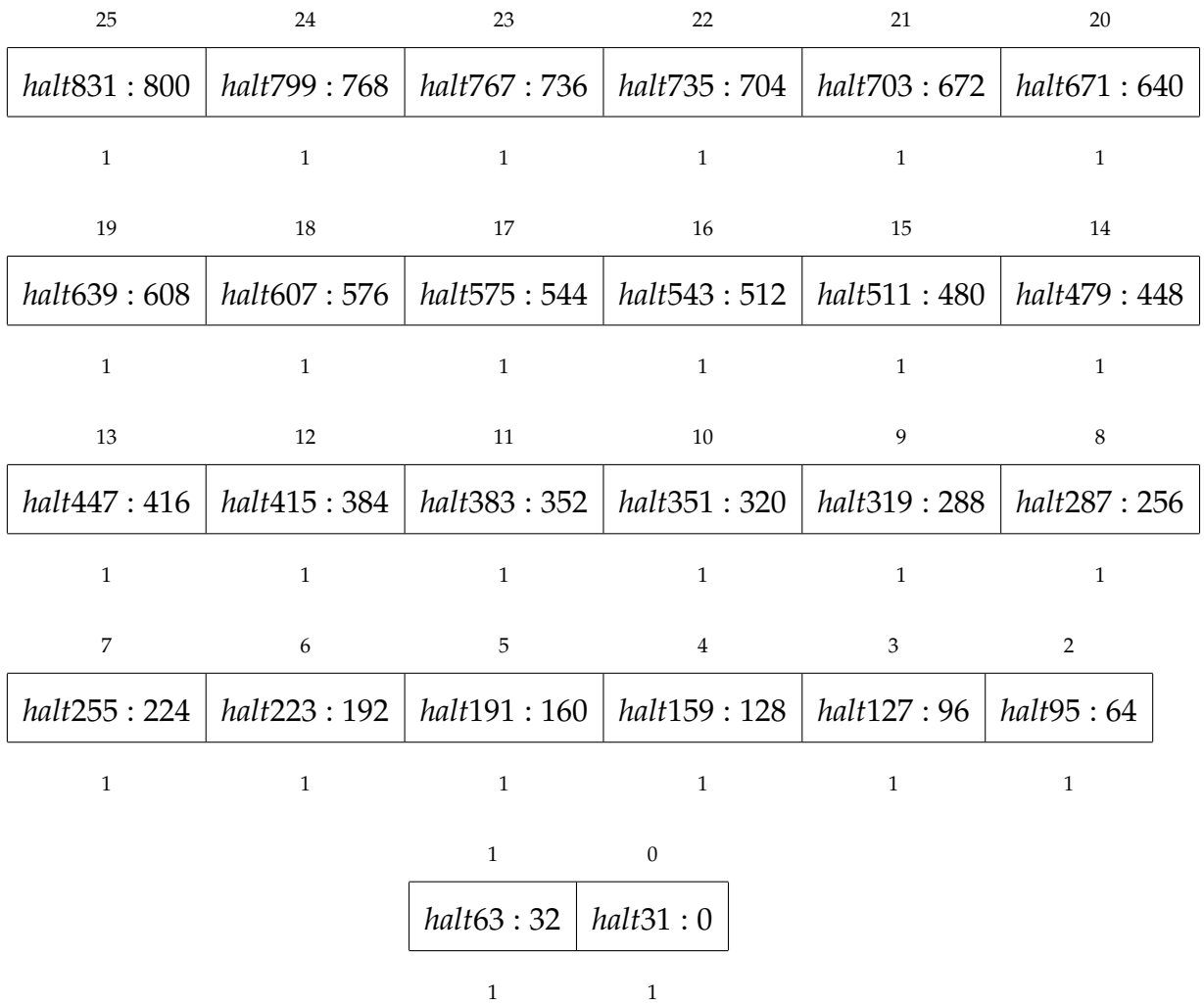
| | | | |
|----------|--|---|--------|
| datasize | <p>If dataaccess is 0: Number of CSR registers dedicated to shadowing the data registers.</p> <p>If dataaccess is 1: Number of 32-bit words in the memory map dedicated to shadowing the data registers.</p> | R | Preset |
| dataaddr | <p>If dataaccess is 0: The number of the first CSR dedicated to shadowing the data registers.</p> <p>If dataaccess is 1: Signed address of RAM where the data registers are shadowed, to be used to access relative to zero.</p> | R | Preset |

6.8.4 Halt Summary (haltsum, at 0x13)

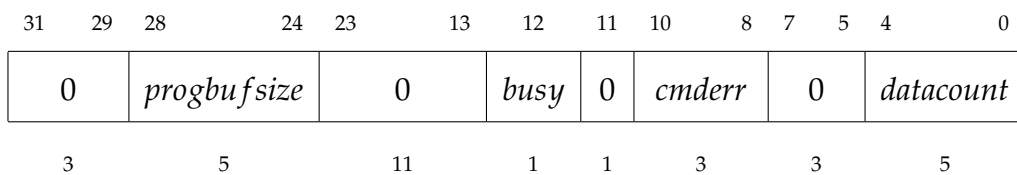
This register contains a summary of which harts are halted. Each bit contains the logical OR of 32 halt bits. When there are a large number of harts in the system, the debugger can first read this register, and then read from the halt region (0x40–0x5f) to determine which hart is the one that is halted.

This entire register is read-only.

| | | | | | |
|-----------------------|----------------------|----------------------|----------------------|----------------------|----------------------|
| 31 | 30 | 29 | 28 | 27 | 26 |
| <i>halt1023 : 992</i> | <i>halt991 : 960</i> | <i>halt959 : 928</i> | <i>halt927 : 896</i> | <i>halt895 : 864</i> | <i>halt863 : 832</i> |
| 1 | 1 | 1 | 1 | 1 | 1 |



6.8.5 Abstract Control and Status (*abstractcs*, at 0x16)



| Field | Description | Access | Reset |
|-------------|--|--------|--------|
| progbuFSIZE | Size of the Program Buffer, in 32-bit words. Valid sizes are 0 - 16. | R | Preset |

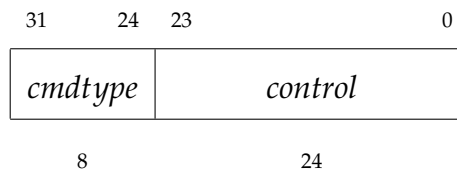
| | | | |
|------|---|---|---|
| busy | <p>1: An abstract command is currently being executed.</p> <p>This bit is set as soon as command is written, and is not cleared until that command has completed.</p> | R | 0 |
|------|---|---|---|

| | | | |
|--------|--|-------|---|
| cmderr | <p>Gets set if an abstract command fails. The bits in this field remain set until they are cleared by writing 1 to them. No abstract command is started until the value is reset to 0.</p> <p>0 (none): No error.</p> <p>1 (busy): An abstract command was executing while command, abstractcs,abstractauto was written, or when one of the data or progbuf registers was read or written.</p> <p>2 (not supported): The requested command is not supported. A command that is not supported while the hart is running may be supported when it is halted.</p> <p>3 (exception): An exception occurred while executing the command (eg. while executing the Program Buffer).</p> <p>4 (halt/resume): An abstract command couldn't execute because the hart wasn't in the expected state (running/halted).</p> <p>7 (other): The command failed for another reason.</p> | R/W1C | 0 |
|--------|--|-------|---|

| | | | |
|-----------|--|---|--------|
| datacount | Number of data registers that are implemented as part of the abstract command interface. Valid sizes are 0 - 12. | R | Preset |
|-----------|--|---|--------|

6.8.6 Abstract Command (command, at 0x17)

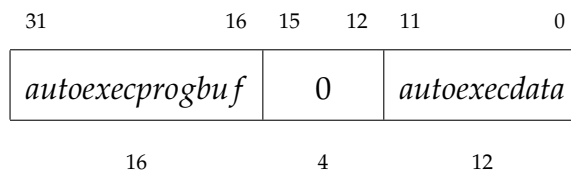
Writes to this register cause the corresponding abstract command to be executed. Writing while an abstract command is executing causes cmderr to be set. If cmderr is non-zero, writes to this register are ignored.



| Field | Description | Access | Reset |
|---------|--|--------|-------|
| cmdtype | The type determines the overall functionality of this abstract command. | W | 0 |
| control | This field is interpreted in a command-specific manner, described for each abstract command. | W | 0 |

6.8.7 Abstract Command Autoexec (abstractauto, at 0x18)

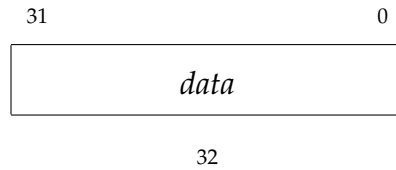
This register is optional. Including it allows more efficient burst accesses. Debugger can attempt to set bits and read them back to determine if the functionality is supported.



| Field | Description | Access | Reset |
|-----------------|---|--------|-------|
| autoexecprogbuf | When a bit in this field is 1, read or write accesses the corresponding progbuf word cause the command in command to be executed again. | R/W | 0 |
| autoexecdata | When a bit in this field is 1, read or write accesses the corresponding data word cause the command in command to be executed again. | R/W | 0 |

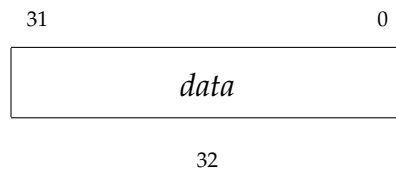
6.8.8 Abstract Data 0 (data0, at 0x04)

Basic read/write registers that may be read or changed by abstract commands. Accessing them while an abstract command is executing causes cmderr to be set. Attempts to write them while busy is set does not change their value.



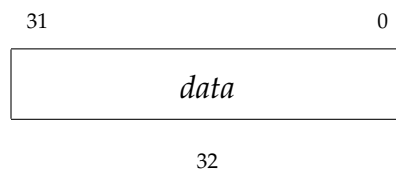
6.8.9 Program Buffer 0 (progbuf0, at 0x20)

The progbuf registers provide read/write access to the optional program buffer. Accessing them while an abstract command is executing causes cmderr to be set. Attempts to write them while busy is set does not change their value.



6.8.10 Authentication Data (authdata, at 0x30)

This register serves as a 32-bit serial port to the authentication module. When authbusy is clear, the debugger can communicate with the authentication module by reading or writing this register. There is no separate mechanism to signal overflow/underflow.



CHAPTER 7

Implementation and Design Challenges

7.1 JTAG TAP

Approach-1 is the initial stage of my project started with Jtag TAP code. I wrote a testbench for that code. This is my first approach of implementing the Jtag TAP. In this approach I followed clock domain crossing. The shifting of input data TDI takes place at rising edge of the clock TCK. The shifting of output data TDO takes place simultaneously at the falling edge of TCK. In bluespec the clock is default. In order to get the falling edge of the clock, I introduced another clock which is inverted form of TCK.

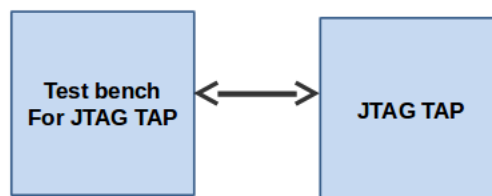


Figure 7.1: Approach-1

In approach-2, getting inputs TCK, TMS, TDI and sending output TDO to the OpenOCD is done. Here one driver is used to interact with OpenOCD which is called Remote_Bitbang driver.

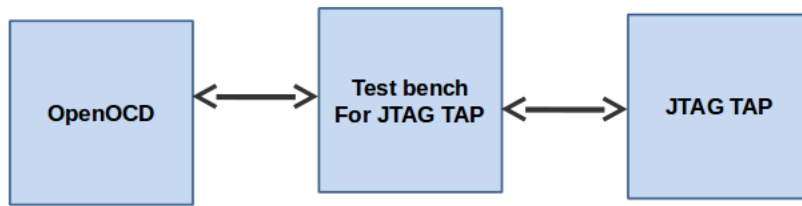


Figure 7.2: Approach-2 Overview

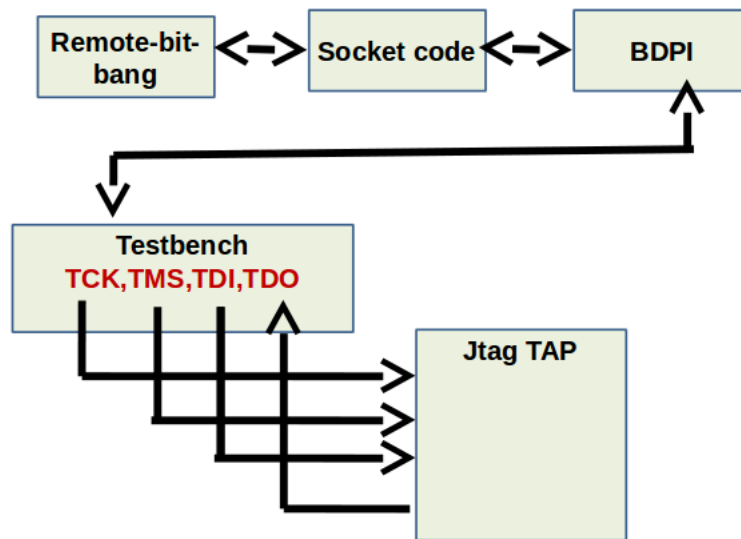


Figure 7.3: Approach-2 in brief

7.2 Debug Module

The debug module is created on the basis of FSM shown in the figure 6.1. The debug module is instantiated into the soc as shown in the figure 7.4. The updating of dmiaccess and capturing of it is done here.

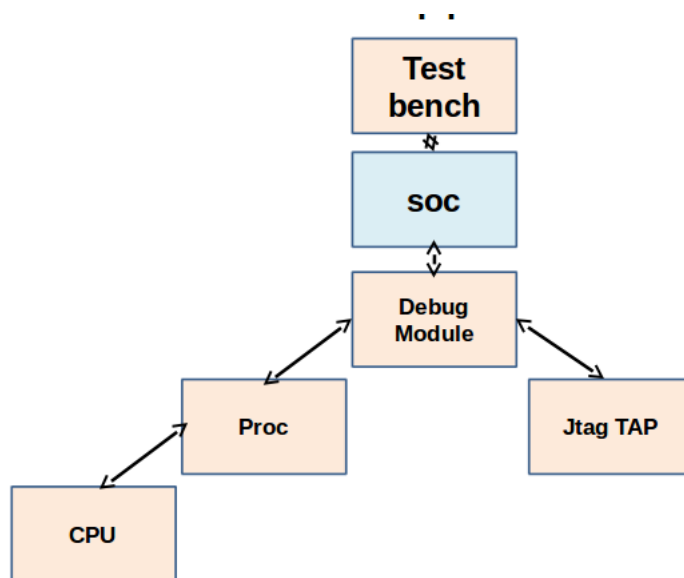


Figure 7.4: Final Approach

7.3 Servers and Clients

The OpenOCD acts as both server and client. First the debugger which is having remote process waits for the OpenOCD to connect. The OpenOCD listens to the request of remote process acting as client. After this, the OpenOCD acting as a server waits for the gdb to connect. The gdb acts as client to the OpenOCD.

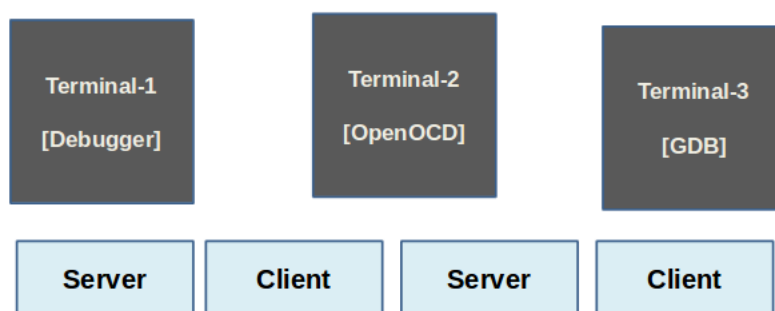


Figure 7.5: Servers and Clients

7.4 Halting and Resuming

Initially the processor will be in running stage and it will be in NORMAL mode. The OpenOCD sends halt request to the processor using debug module control register (0x10). When the processor receives halt request, the cpu state will go into stop state and cpu mode will go into DEBUG mode. If the processor completely halts then it sends halt acknowledgement to the debug module. After this, the debug module status register is updated with the values like allhalted to 1 and allrunning to 0. The OpenOCD will come to know about the processor halt state by reading debug module status register. During this things happening, the pc value is stored into dpc.

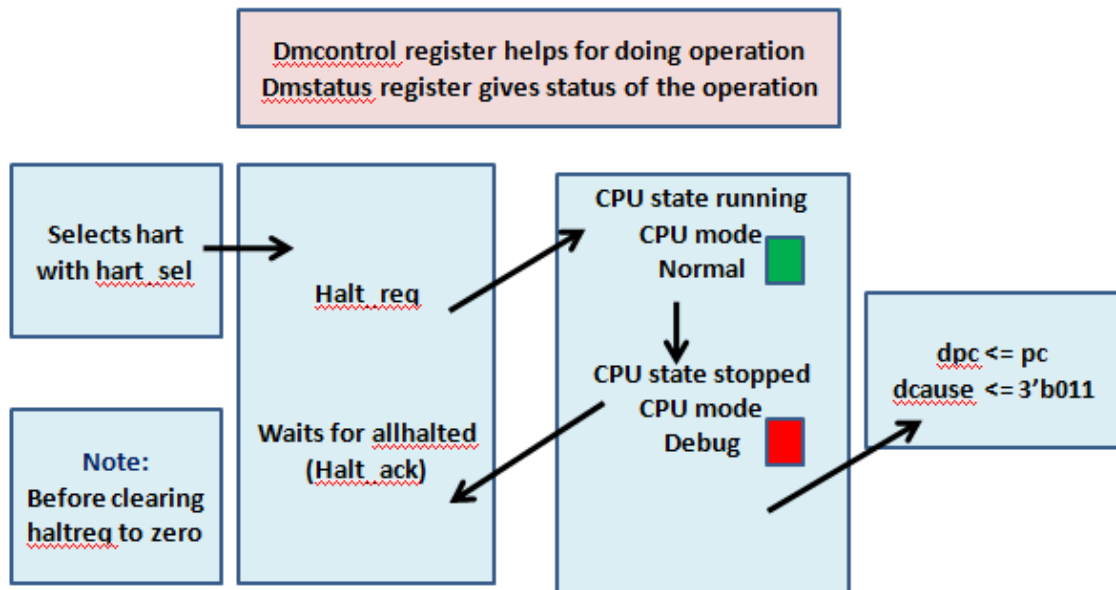


Figure 7.6: Halting the processor

When the processor is in stopped state and in DEBUG mode, the OpenOCD sends resume request and waits for resume acknowledgement. The processor gets into running state and enters into NORMAL mode. Here before resuming, the pc value gets updated with dpc value. After this the debug module status register is updated with the values like allhalted to 0 and allrunning to 1.

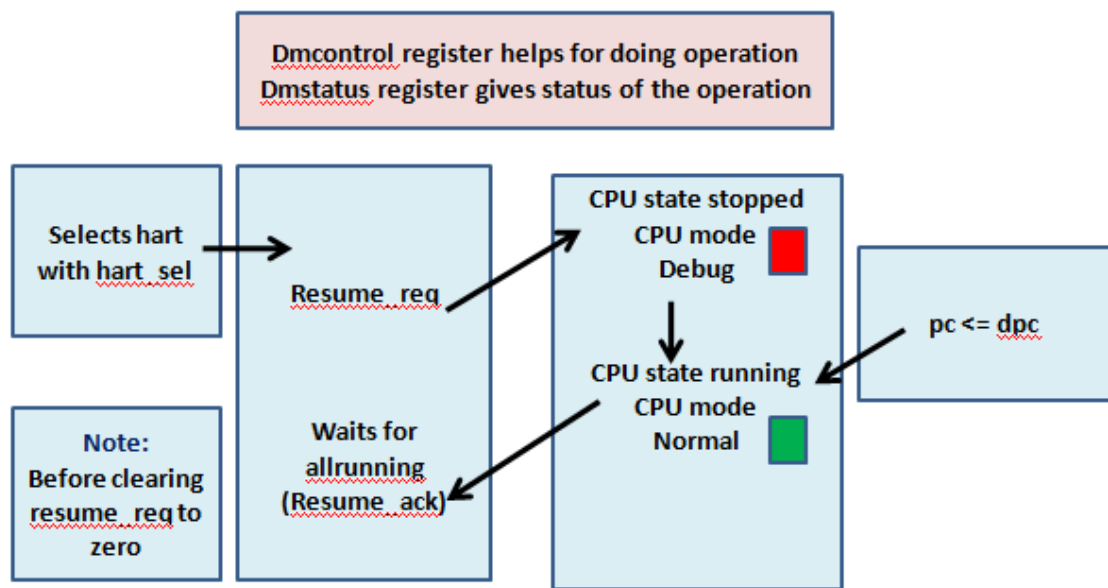


Figure 7.7: Resuming the processor

7.5 Abstract Commands

The abstract command gives the debugger access to CPU registers and program buffer. The sequence of operation is shown in below figure 7.8.

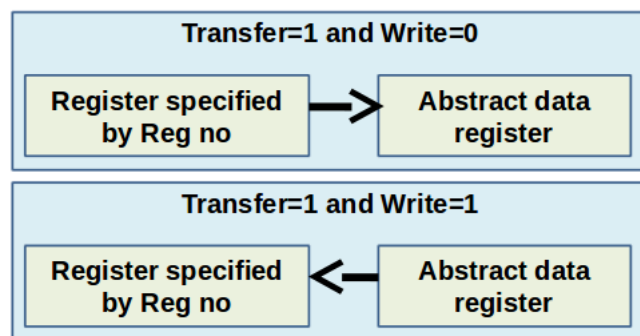


Figure 7.8: Use of abstract commands

7.6 Program Buffer

As mentioned in chapter 5 about program buffer, the debugger can write a small program to the Program Buffer, and then execute it exactly once with the Access Register Abstract Command, setting the postexec bit in command. The debugger can write whatever program it likes (including jumps out of the Program Buffer), but the program must end with ebreak. Here ebreak instruction is represented with 32 bits 0x00010073. Whenever ebreak occurs, the processor stops executing and jumps back to its previous halt state.

Here program buffer register values should be made as memory mapped, so that it acts as slave to the processor. The program buffer registers are represented as slave to the system bus. When postexec bit becomes 1, the debugger writes pc value with 0x00000020, so that the processor starts fetching the instruction from 0x00000020. Before making the processor to fetch, we have to flush all the pipeline stages. Here flushing the pipeline stages in the sense clearing all the pipeline FIFOs.

When the processor runs the ebreak instruction 0x00010073, it again goes to halt state. When the openocd send resume request, then dpc value is stored into pc value and the processor starts fetching the instructions as per pc value.

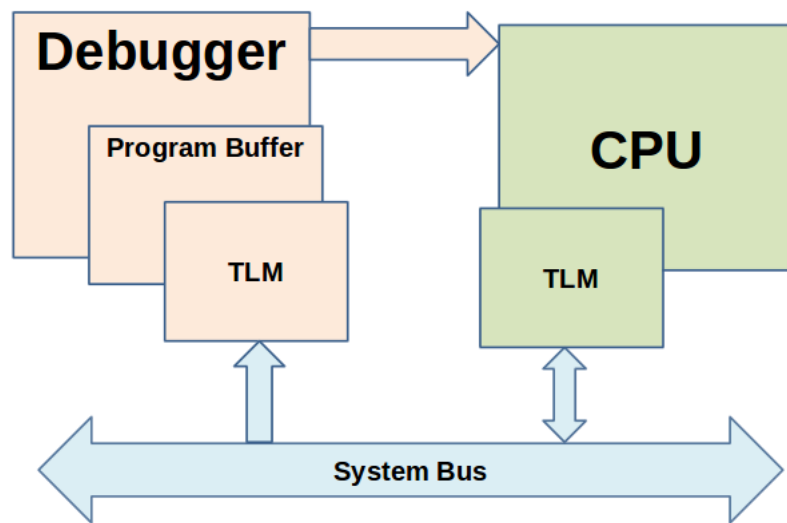


Figure 7.9: Program Buffer as Slave to the Syatem Bus

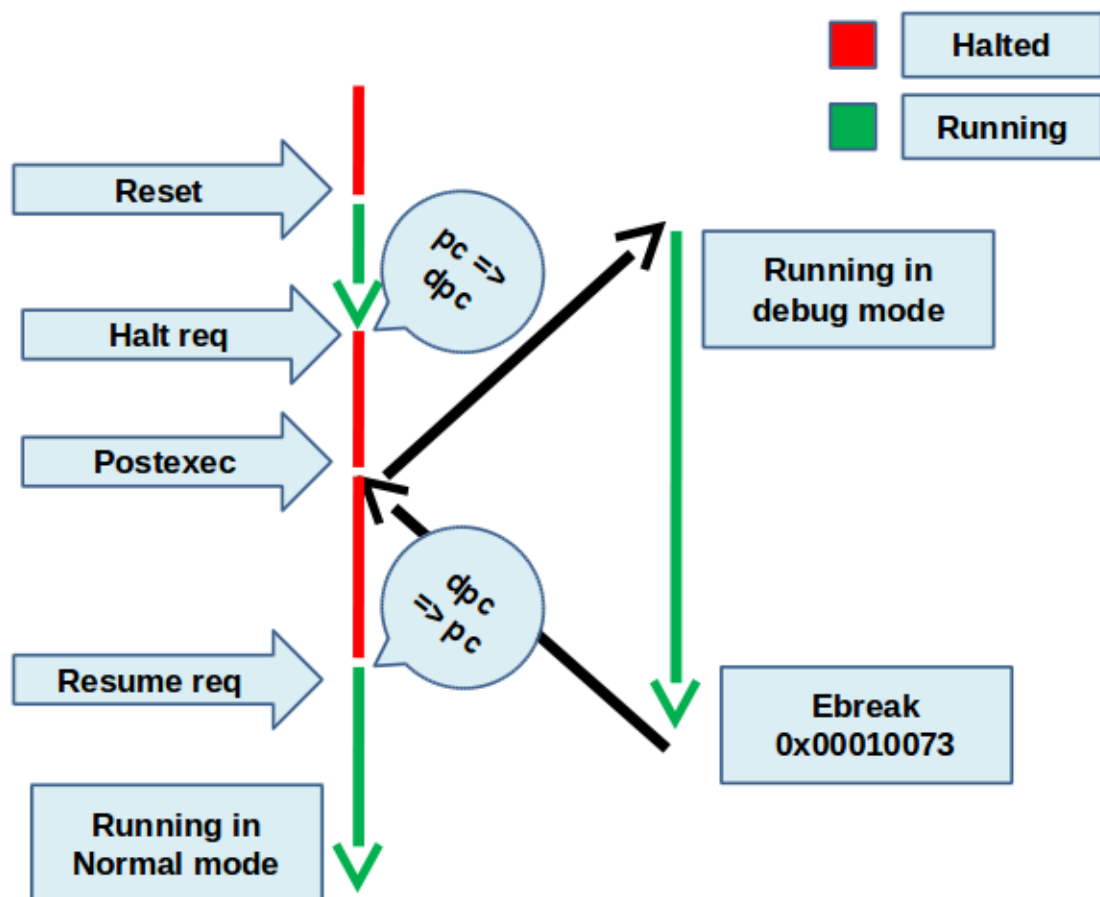


Figure 7.10: Program Buffer Execution process

7.7 Single stepping

A debugger can cause a halted hart to execute a single instruction and then re-enter Debug Mode by setting step before setting resumereq. If executing or fetching that instruction causes an exception, Debug Mode is re-entered immediately after the PC is changed to the exception handler and the appropriate tval and cause registers are updated.

After executing single instruction, the hart re-enter into debug mode. When re-entering into debug mode, the pc value is stored into dpc.

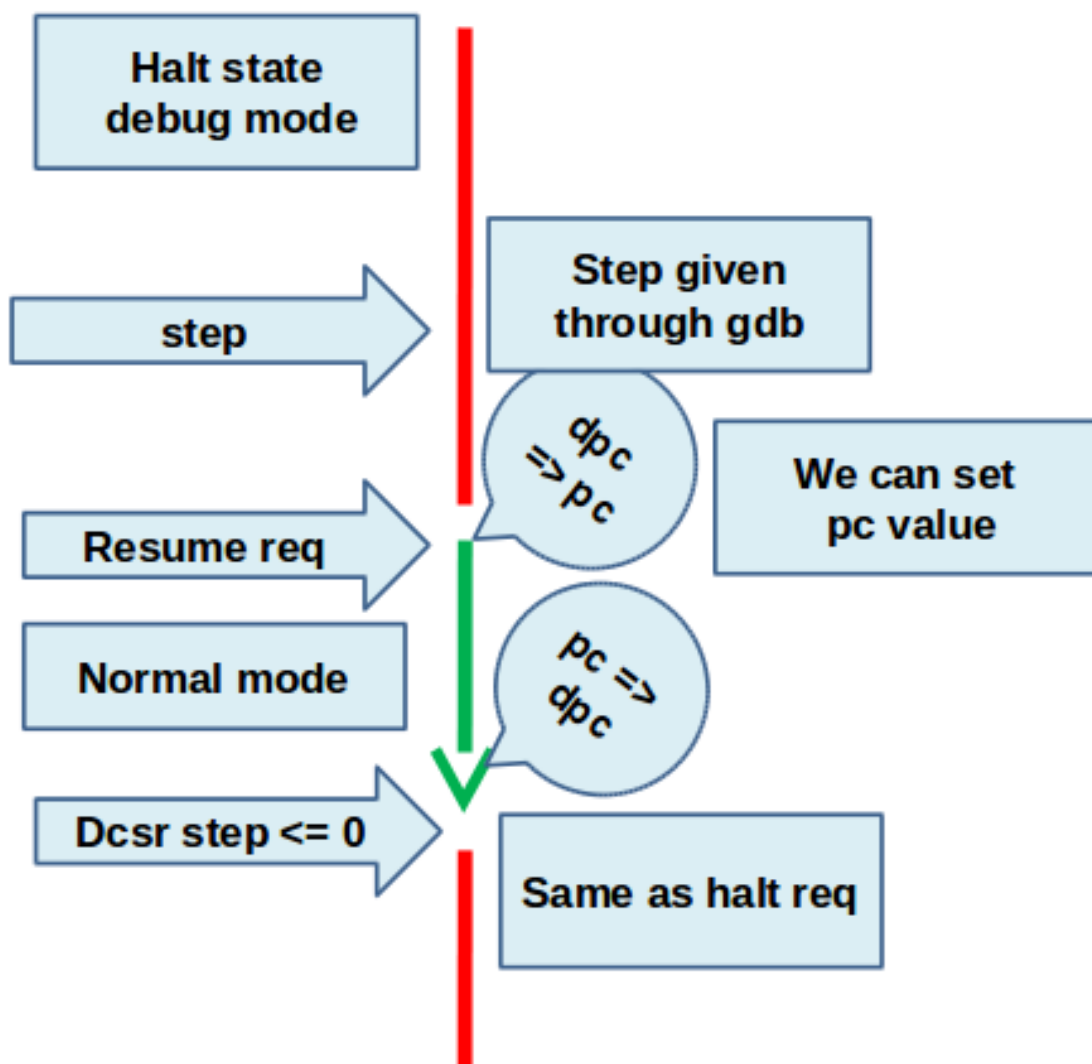


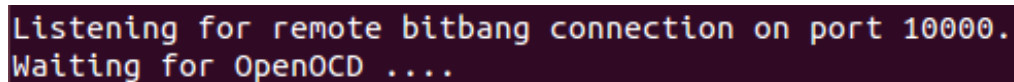
Figure 7.11: Single Stepping

CHAPTER 8

RESULTS

8.1 Servers and Clients

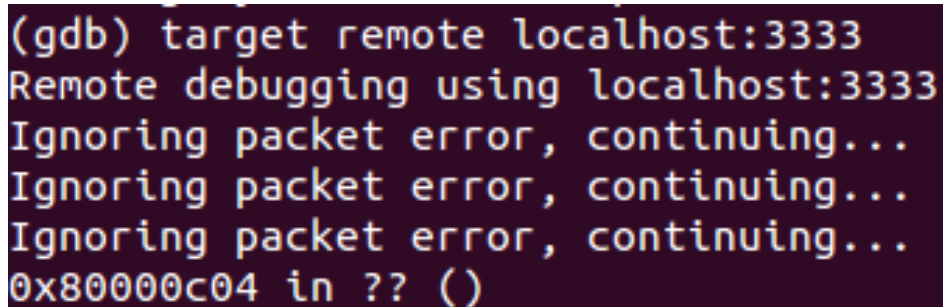
As shown in figure 7.5 in implementation chapter, three terminals are required. Initially in one terminal the remote process acting as server waits for remote_bitbang connection on port 10000. It is shown in figure 8.1.



```
Listening for remote bitbang connection on port 10000.  
Waiting for OpenOCD ....
```

Figure 8.1: Remote Process as server

Figure 8.2 shows that gdb acting as a client and connecting to the server OpenOCD.



```
(gdb) target remote localhost:3333  
Remote debugging using localhost:3333  
Ignoring packet error, continuing...  
Ignoring packet error, continuing...  
Ignoring packet error, continuing...  
0x80000c04 in ?? ()
```

Figure 8.2: GDB connecting to OpenOCD

8.2 Reading Idcode register

Figure 8.3 shows how the OpenOCD examining the IDCODE register values.

```
Info : Initializing remote_bitbang driver
Info : Connecting to localhost:10000
Info : remote_bitbang driver initialized
Info : This adapter doesn't support configurable speed
Info : JTAG tap: riscv.cpu tap/device found: 0x10e31913 (mfg: 0x489 (SiFive, Inc
.), part: 0x0e31, ver: 0x1)
Info : JTAG tap: auto0.tap tap/device found: 0x10e31913 (mfg: 0x489 (SiFive, Inc
.), part: 0x0e31, ver: 0x1)
Info : JTAG tap: auto1.tap tap/device found: 0x10e31913 (mfg: 0x489 (SiFive, Inc
.), part: 0x0e31, ver: 0x1)
```

Figure 8.3: Reading IDCODE register

8.3 OpenOCD examining DTM Control register

```
Debug: 219 5222 riscv.c:652 riscv_examine(): riscv_examine()
Debug: 220 5499 riscv.c:224 dtmcontrol_scan(): DTMCONTROL: 0x0 -> 0x2081
Debug: 221 5499 riscv.c:662 riscv_examine(): dtmcontrol=0x2081
Debug: 222 5499 riscv.c:664 riscv_examine(): version=0x1
Debug: 223 5499 riscv-013.c:1075 init_target(): init
Debug: 224 5713 riscv-013.c:329 dtmcontrol_scan(): DTMCS: 0x0 -> 0x2081
Debug: 225 5713 riscv-013.c:1132 examine(): dtmcontrol=0x2081
Debug: 226 5713 riscv-013.c:1133 examine(): dmireset=0
Debug: 227 5713 riscv-013.c:1134 examine(): idle=2
Debug: 228 5713 riscv-013.c:1135 examine(): dmistat=0
Debug: 229 5713 riscv-013.c:1136 examine(): abits=8
Debug: 230 5713 riscv-013.c:1137 examine(): version=1
Debug: 231 5940 riscv-013.c:266 scan(): 42b r 00000000 @11 -> + 00000000 @00
Debug: 232 6127 riscv-013.c:266 scan(): 42b - 00000000 @11 -> + 00000c82 @11
```

Figure 8.4: Reading DTM control register

8.4 Getting hart information

```
Debug: 245 7723 riscv-013.c:1166 examine(): dmcontrol: 0x00000001
Debug: 246 7723 riscv-013.c:1167 examine(): dmstatus: 0x00000c82
Debug: 247 7723 riscv-013.c:1168 examine(): hartinfo: 0x00000000
Debug: 248 7918 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @12
Debug: 249 8103 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
```

Figure 8.5: Getting hart information

8.5 Halting and Resuming

As per the operation mentioned in figures 7.6 and 7.7, the results are shown in figures 8.6 and 8.7 respectively

```
Debug: 268 10125 riscv-013.c:266 scan(): 42b r 00000000 @10 -> + 00000000 @11
Debug: 269 10319 riscv-013.c:266 scan(): 42b - 00000000 @10 -> + 00000001 @10
Debug: 270 10319 riscv-013.c:277 scan(): -> dmactive
Debug: 271 10525 riscv-013.c:266 scan(): 42b w 80000001 @10 -> + 00000000 @10
Debug: 272 10525 riscv-013.c:277 scan(): haltreq dmactive ->
Debug: 273 10585 riscv-013.c:266 scan(): 42b - 00000000 @10 -> + 80000001 @10
Debug: 274 10585 riscv-013.c:277 scan(): -> haltreq dmactive
Debug: 275 10642 riscv-013.c:266 scan(): 42b r 00000000 @11 -> + 00000000 @10
Debug: 276 10703 riscv-013.c:266 scan(): 42b - 00000000 @11 -> + 00000182 @11
```

Figure 8.6: Sending halt request

```
Debug: 11633 340799 riscv-013.c:266 scan(): 42b w 40000001 @10 -> + 00000000 @10
Debug: 11634 340799 riscv-013.c:277 scan(): resumereq dmactive ->
Debug: 11635 340949 riscv-013.c:266 scan(): 42b - 00000000 @10 -> + 40000001 @10
Debug: 11636 340949 riscv-013.c:277 scan(): -> resumereq dmactive
Debug: 11637 341000 riscv-013.c:266 scan(): 42b r 00000000 @11 -> + 00000000 @10
Debug: 11638 341042 riscv-013.c:266 scan(): 42b - 00000000 @11 -> + 00030382 @11
```

Figure 8.7: Sending resume request

8.6 Reading General Purpose registers

To read the general purpose registers, the command `i r` (info registers) is used in gdb terminal. All the 32 general purpose registers are read.

```
0x80000be4 in ?? ()
(gdb) i r
ra          0x00000000fffff83c          4294965308
sp          0x0000000000000000          0
gp          0x00000000fffff878          4294965368
tp          0x0000000000003603          13827
t0          0x0000000000000000          0
t1          0x00000000ffffa000          4294945450
t2          0x0000000000003402          13314
fp          0x0000000000003606          13830
s1          0x00000000000000fb          251
a0          0x00000000000007b7          1975
a1          0x00000000fffffffffa          4294967290
a2          0x00000000000000aa          170
a3          0x0000000000000040          64
a4          0x0000000000000000          0
a5          0x000000000000000a          10
a6          0x0000000000000000          0
a7          0x0000000000000000          0
s2          0x0000000000000004          4
s3          0x00000000000007e0          2016
s4          0x0000000000000000          0
s5          0x00000000fffff8aa          4294965418
s6          0x0000000000000000          0
s7          0x00000000ffffffffff          4294967295
---Type <return> to continue, or q <return> to quit---return
s8          0x0000000000000000          0
s9          0x0000000000000001          1
s10         0x00000000fffff83c          4294965308
s11         0x00000000ffffc9ff          4294953471
t3          0x0000000000000000          0
t4          0x0000000000000000          0
t5          0x0000000000000000          0
t6          0x0000000000000000          0
pc          0x80000be480000be4          -9223358960826840092
priv        0x80000b00          prv:0 [User/Application]
```

Figure 8.8: Reading General Purpose Registers

8.7 Program Buffer

The figure 8.9 shows how OpenOCD writes instructions into the program buffer registers and how it sets postexecution bit to 1. As per the FSM, the debugger will be in command start state and when it sets postexec bit to 1, the debugger changes to command programbuffer state. After reaching there, it waits for ebreak instruction. Whenever ebreak instruction occurs while the processor running it jumps to command done state.

```
Debug: 494 15809 riscv-013.c:266 scan(): 42b w 0000100f @20 -> + 00000000 @11
Debug: 495 15852 riscv-013.c:266 scan(): 42b - 00000000 @20 -> + 0000100f @20
Debug: 496 15853 program.c:33 riscv_program_write(): 0x7ffcba6c6d00: debug_buffer[01] = DASM(0x00100073)
Debug: 497 15903 riscv-013.c:266 scan(): 42b w 00100073 @21 -> + 00000000 @20
Debug: 498 15946 riscv-013.c:266 scan(): 42b - 00000000 @21 -> + 00100073 @21
Debug: 499 15946 riscv-013.c:565 execute_abstract_command(): command=0x241000
Debug: 500 16004 riscv-013.c:266 scan(): 42b w 00241000 @17 -> + 00000000 @21
Debug: 501 16049 riscv-013.c:266 scan(): 42b - 00000000 @17 -> + 00241000 @17
Debug: 502 16098 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @17
Debug: 503 16141 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 504 16141 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 505 16202 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @16
Debug: 506 16245 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 507 16245 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 508 16245 riscv-013.c:565 execute_abstract_command(): command=0x3207b0
Debug: 509 16294 riscv-013.c:266 scan(): 42b w 003207b0 @17 -> + 00000000 @16
Debug: 510 16337 riscv-013.c:266 scan(): 42b - 00000000 @17 -> + 003207b0 @17
Debug: 511 16385 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @17
Debug: 512 16429 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 513 16429 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 514 16479 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @16
Debug: 515 16521 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 516 16521 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 517 16569 riscv-013.c:266 scan(): 42b r 00000000 @05 -> + 00000000 @16
Debug: 518 16613 riscv-013.c:266 scan(): 42b - 00000000 @05 -> + 40000040 @05
Debug: 519 16665 riscv-013.c:266 scan(): 42b r 00000000 @04 -> + 00000000 @05
Debug: 520 16707 riscv-013.c:266 scan(): 42b - 00000000 @04 -> + 40000040 @04
```

Figure 8.9: Setting postexec bit value

```

2560030 CPU: EXECUTE: Setting up EXECUTE unit. PC: 00000020
2560030 CPU: EXECUTE: 2560030/////--yes--pc-///-///--
yes-pc-///--yes-pc-///--yes--pc-00000024
2560040 CPU: FETCH: Request to Memory. Addr: 00000024, burst_length
: 1
2560040 CPU: WRITE-BACK: Instruction (PC: 00000020) performed an ar
ithmetic or logical operation. ARF Updated. REG: 0 Value: 00000000
2560040/////-----yes--R-/////
shift dr
2560060data1 is 00100073
2560060cpu_rsp_value-----/////----- 180163741944381
80
2560060 CPU: FETCH: Response from Memory. Addr: 00000024, data: 001
00073
2560070 CPU: FETCH: Response received from Instruction Memory for P
C: 00000024. Recived Data: 00100073
2560070Fetch response came 000000121500100073
2560070 CPU: FETCH: Transfer initiated for PC = 00000028

2560080 CPU: DECODE: Instruction decoded. PC: 00000024, Instruction
: 00100073
2560080 CPU: DECODE: Opcode: 1110011, rs1: 0, rs2: 1, rd: 0, f3:
000, f7: 0000000
2560080 CPU: EXECUTE: Setting up EXECUTE unit. PC: 00000024
2560080 CPU: EXECUTE: 2560080/////--yes--pc-///-///--
yes-pc-///--yes-pc-///--yes--pc-00000028
shift dr
2560090 CPU: FETCH: Request to Memory. Addr: 00000028, burst_length
: 1
2560090///--yes--br-///--yes--br-/////--yes--br-//-----yes--br---/
//--yes--br-///--yes--br-///--yes--br-/////
2560090/////-----yes--R-/////
2560090///---HI scpu-///---HI scpu-///---HI scpu-///---HI scpu-
-///---HI scpu-//
2560110data1 is 00000000

```

Figure 8.10: Executing Program Buffer registers

8.8 Setting PC value

As per the requirements, we can set PC value with any address. Here I am setting PC value with 0x80000004.


```

0x80000e04 in ?? ()
(gdb) set $pc=0x80000004
Ignoring packet error, continuing...
Ignoring packet error, continuing...
Ignoring packet error, continuing...
(gdb) i r
ra          0x0000000000000000      0
sp          0x0000000000000000      0
gp          0x0000000000000000      0
tp          0x0000000000000000      0
t0          0x0000000080000e08    2147487240
t1          0x0000000000000000      0
t2          0x0000000000000000      0
fp          0x0000000000000000      0
s1          0x0000000000000000      0
a0          0x0000000000000000      0
a1          0x0000000000000000      0
a2          0x0000000000000000      0
a3          0x0000000000000000      0
a4          0x0000000000000000      0
a5          0x0000000000000000      0
a6          0x0000000000000000      0
a7          0x0000000000000000      0
s2          0x0000000000000000      0
s3          0x0000000000000000      0
s4          0x0000000000000000      0
s5          0x0000000000000007      7
s6          0x0000000000000000      0
s7          0x0000000000000000      0
---Type <return> to continue, or q <return> to quit---return
s8          0x0000000000000000      0
s9          0x0000000000000000      0
s10         0x0000000000000000      0
s11         0x0000000000000000      0
t3          0x0000000000000000      0
t4          0x0000000000000000      0
t5          0x0000000000000000      0
t6          0x0000000000000000      0
pc          0x0000000080000004    2147483652
priv        0x80000000      prv:0 [User/Application]

```

Figure 8.11: GDB terminal:Setting PC value

```

Debug: 10294 300890 riscv-013.c:266 scan(): 42b w 00000000 @05 -> + 00000000 @10
Debug: 10295 300934 riscv-013.c:266 scan(): 42b - 00000000 @05 -> + 00000000 @05
Debug: 10296 300986 riscv-013.c:266 scan(): 42b w 80000004 @04 -> + 00000000 @05
Debug: 10297 301030 riscv-013.c:266 scan(): 42b - 00000000 @04 -> + 80000004 @04
Debug: 10298 301030 riscv-013.c:565 execute_abstract_command(): command=0x3307b1
Debug: 10299 301097 riscv-013.c:266 scan(): 42b w 003307b1 @17 -> + 00000000 @04
Debug: 10300 301151 riscv-013.c:266 scan(): 42b - 00000000 @17 -> + 003307b1 @17
Debug: 10301 301214 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @17
Debug: 10302 301256 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 10303 301256 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 10304 301304 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @16
Debug: 10305 301346 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 10306 301346 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 10307 301346 riscv-013.c:565 execute_abstract_command(): command=0x3207b1
Debug: 10308 301394 riscv-013.c:266 scan(): 42b w 003207b1 @17 -> + 00000000 @16
Debug: 10309 301439 riscv-013.c:266 scan(): 42b - 00000000 @17 -> + 003207b1 @17
Debug: 10310 301493 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @17
Debug: 10311 301541 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 10312 301541 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 10313 301596 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @16
Debug: 10314 301641 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 10315 301641 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 10316 301690 riscv-013.c:266 scan(): 42b r 00000000 @05 -> + 00000000 @16
Debug: 10317 301734 riscv-013.c:266 scan(): 42b - 00000000 @05 -> + 80000004 @05
Debug: 10318 301786 riscv-013.c:266 scan(): 42b r 00000000 @04 -> + 00000000 @05
Debug: 10319 301829 riscv-013.c:266 scan(): 42b - 00000000 @04 -> + 80000004 @04

```

Figure 8.12: OpenOCD Terminal:Setting PC value

8.9 Loading instructions and data into memory

With the help of gdb , loading instructions and data into memory is done by using load command.

```

(gdb) load
Loading section .text, size 0x34 lma 0x80000000
Loading section .srodata.cst8, size 0x10 lma 0x80000038
Start address 0x80000000, load size 68
Transfer rate: 11 bytes/sec, 34 bytes/write.
(gdb) info files
Symbols from "/home/yanamala/Mount/simple".
Remote serial target in gdb-specific protocol:
Debugging a target over a serial line.
While running this, GDB does not access memory from...
Local exec file:
`/home/yanamala/Mount/simple', file type elf32-littleriscv.
Entry point: 0x80000000
0x80000000 - 0x80000034 is .text
0x80000038 - 0x80000048 is .srodata.cst8

```

Figure 8.13: Loading instructions and data into memory

8.10 Setting breakpoints

A breakpoint makes your program stop whenever a certain point in the program is reached. For each breakpoint, you can add conditions to control in finer detail whether your program stops. You can set breakpoints with the break command.

GDB assigns a number to each breakpoint when you create it; these numbers are successive integers starting with one. In many of the commands for controlling various features of breakpoints you use the breakpoint number to say which breakpoint you want to change. In order to get list of breakpoints, use the info breakpoints command as shown in figure 8.14.

```
(gdb) b 1
Breakpoint 1 at 0x80000000: file simple.c, line 1.
(gdb) b 5
Breakpoint 2 at 0x80000008: file simple.c, line 5.
(gdb) b 8
Breakpoint 3 at 0x80000024: file simple.c, line 8.
(gdb) info b
Num      Type           Disp Enb Address      What
1        breakpoint      keep y   0x80000000 in dmul at simple.c:1
2        breakpoint      keep y   0x80000008 in main at simple.c:5
3        breakpoint      keep y   0x80000024 in main at simple.c:8
```

Figure 8.14: Setting breakpoints and listing them

8.11 stepping

In order to perform stepping operation, si command is given through gdb terminal. After that the OpenOCD will write step bit in dcsr to 1 and dcause bits to 3'b100 as shown in figure 8.15. During this, the dpc value is stored into pc value. The OpenOCD sends resume request and the hart enters into NORMAL mode. After executing some instructions, the step bit in dcsr becomes 0 and hart again enters

into DEBUG mode. During this operation, the pc value is stored into dpc. The si command is given everytime through gdb for performing step as shown in figure 8.16.

In order to perform stepping operation from particular address, then the PC value is set to that address as shown in figure 8.17. After this, the stepping command is given through gdb.

```
Debug: 3918 118672 riscv-013.c:266 scan(): 42b w 00000000 @05 -> + 00000000 @10
Debug: 3919 118716 riscv-013.c:266 scan(): 42b - 00000000 @05 -> + 00000000 @05
Debug: 3920 118764 riscv-013.c:266 scan(): 42b w 4000b044 @04 -> + 00000000 @05
Debug: 3921 118808 riscv-013.c:266 scan(): 42b - 00000000 @04 -> + 4000b044 @04
Debug: 3922 118808 riscv-013.c:565 execute_abstract_command(): command=0x3307b0
Debug: 3923 118857 riscv-013.c:266 scan(): 42b w 003307b0 @17 -> + 00000000 @04
Debug: 3924 118912 riscv-013.c:266 scan(): 42b - 00000000 @17 -> + 003307b0 @17
```

Figure 8.15: Setting step bit in dcsr

```

(gdb) si
keep_alive() was not invoked in the 1000ms
(2093). Workaround: increase "set remotet
Ignoring packet error, continuing...
Ignoring packet error, continuing...
Ignoring packet error, continuing...
0x80000c6c in ?? ()
(gdb) si
keep_alive() was not invoked in the 1000ms
(2134). Workaround: increase "set remotet
Ignoring packet error, continuing...
Ignoring packet error, continuing...
Ignoring packet error, continuing...
0x80000cf0 in ?? ()
(gdb) si
keep_alive() was not invoked in the 1000ms
(2138). Workaround: increase "set remotet
Ignoring packet error, continuing...
Ignoring packet error, continuing...
Ignoring packet error, continuing...
0x80000dbc in ?? ()
(gdb) si
keep_alive() was not invoked in the 1000ms
(2200). Workaround: increase "set remotet
Ignoring packet error, continuing...
Ignoring packet error, continuing...
Ignoring packet error, continuing...
0x80000e08 in ?? ()

```

Figure 8.16: Stepping

```

pc          0x0000000080000004      2147483652
priv        0x80000000      prv:0 [User/Application]
(gdb) si
keep_alive() was not invoked in the 1000ms timelimit. GDB alive packet not sent!
(1993). Workaround: increase "set remotetimeout" in GDB
Ignoring packet error, continuing...
Ignoring packet error, continuing...
Ignoring packet error, continuing...
0x80000058 in ?? ()

```

Figure 8.17: Stepping after setting PC value

```

Debug: 1774 56397 riscv-013.c:565 execute_abstract_command(): command=0x3207b1
Debug: 1775 56449 riscv-013.c:266 scan(): 42b w 003207b1 @17 -> + 00000000 @10
Debug: 1776 56493 riscv-013.c:266 scan(): 42b - 00000000 @17 -> + 003207b1 @17
Debug: 1777 56544 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @17
Debug: 1778 56598 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 1779 56598 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 1780 56657 riscv-013.c:266 scan(): 42b r 00000000 @16 -> + 00000000 @16
Debug: 1781 56701 riscv-013.c:266 scan(): 42b - 00000000 @16 -> + 0f00000b @16
Debug: 1782 56701 riscv-013.c:277 scan(): -> progbufsize=15 datacount=11
Debug: 1783 56755 riscv-013.c:266 scan(): 42b r 00000000 @05 -> + 00000000 @16
Debug: 1784 56800 riscv-013.c:266 scan(): 42b - 00000000 @05 -> + 80000be8 @05
Debug: 1785 56849 riscv-013.c:266 scan(): 42b r 00000000 @04 -> + 00000000 @05
Debug: 1786 56895 riscv-013.c:266 scan(): 42b - 00000000 @04 -> + 80000be8 @04

```

Figure 8.18: Reading dpc value

CHAPTER 9

CONCLUSION AND FUTURE WORK

This thesis details how an external debugger might use the described debug interface to perform some common operations on Shakti cores using the JTAG DTM. This debugger architecture allows a variety of implementations like 32,64 and future 128 bit processors.

The future direction of this work is to implement a System Bus Access block to provide memory access without involving a hart, regardless of whether Program Buffer is implemented.

REFERENCES

1. **Bluespec-Inc**, *Bluespec System Verilog BSV by Example*. Bluespec Inc, 2010.
2. **Bluespec-Inc**, *Bluespec System Verilog Reference Guide*. Bluespec Inc, 2014.
3. **Newsome, T.**, *RISC-V External Debug Support Version 0.13*. SiFive Inc, 2017.
4. **SiFive-Inc**, *The RISC-V Instruction Set Manual*, volume I: User-Level ISA. SiFive Inc, 2017*a*.
5. **SiFive-Inc**, *The RISC-V Instruction Set Manual*, volume II: Privileged Architecture. SiFive Inc, 2017*b*.