Zilla Debug Unit Specification

Architecture Version 1.0.0

Table of Contents

Preface

About this Specification

This document outlines implementation level details of the Debug Unit based on the RISC-V Debug Specification.

This architecture has a variety of implementations and tradeoffs while retaining some common interfaces to allow debugging tools and components suitable for RISC-V ISA.

Using this Specification

Introduction

This chapter gives an overview of Debug and information and the terminology used in this document. It contains the following sections:

- About the Debug Architecture
- Terminology

About the Debug Architecture

There are several use cases for dedicated debugging hardware, both internal to a CPU core and with an external connection. This specification addresses the use cases listed below.

- Debugging low-level software in the absence of an OS or other software.
- Debugging issues in the OS itself.
- Bootstrapping a hardware platform to test, configure, and program components before there is any executable code path in the hardware platform.
- Accessing hardware on a hardware platform without a working CPU.

In addition, even without a hardware debugging interface, architectural support in a RISC-V CPU aids software debugging and performance analysis by allowing hardware triggers and breakpoints.

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

1. All hart registers (including CSRs) can be read/written when hart is halted.
2. Memory access from the hart's point of view.
3. RV64 support.
4. A hart can be debugged from the very first instruction executed.
5. A RISC-V 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 hart it is debugging.

Terminology

1. AMO - Atomic Memory Operation.
2. BYPASS - JTAG instruction that selects a single-bit data register, also called BYPASS.
3. Component - A RISC-V core, or other parts of a hardware platform. Typically all components will be connected to a single system bus.
4. CSR - Control and Status Register.
5. DM - Debug Module.
6. DMI - Debug Module Interface.
7. DR - JTAG Data Register.
8. DTM - Debug Transport Module.

9. DXLEN - Debug XLEN, which is the widest XLEN a hart supports, ignoring the current value of MXL in `misa`.
10. GPR - General Purpose Register.
11. hardware platform - A single system consisting of one or more components.
12. hart - A hardware thread in a RISC-V core.
13. IDCODE - 32-bit Identification CODE, and a JTAG instruction that returns the IDCODE value.
14. IR - JTAG Instruction Register.
15. JTAG - Refers to work done by IEEE's Joint Test Action Group, described in IEEE 1149.1.
16. NAPOT - Naturally Aligned Powers-Of-Two.
17. NMI - Non-Maskable Interrupt.
18. physical address - An address that is directly usable on the system bus.
19. SBA - System Bus Access.
20. TAP - Test Access Port, defined in IEEE 1149.1.
21. TM - Trigger Module.
22. virtual address- An address as a hart sees it. If the hart is using address translation this may be different from the physical address. If there is no translation then it will be the same.

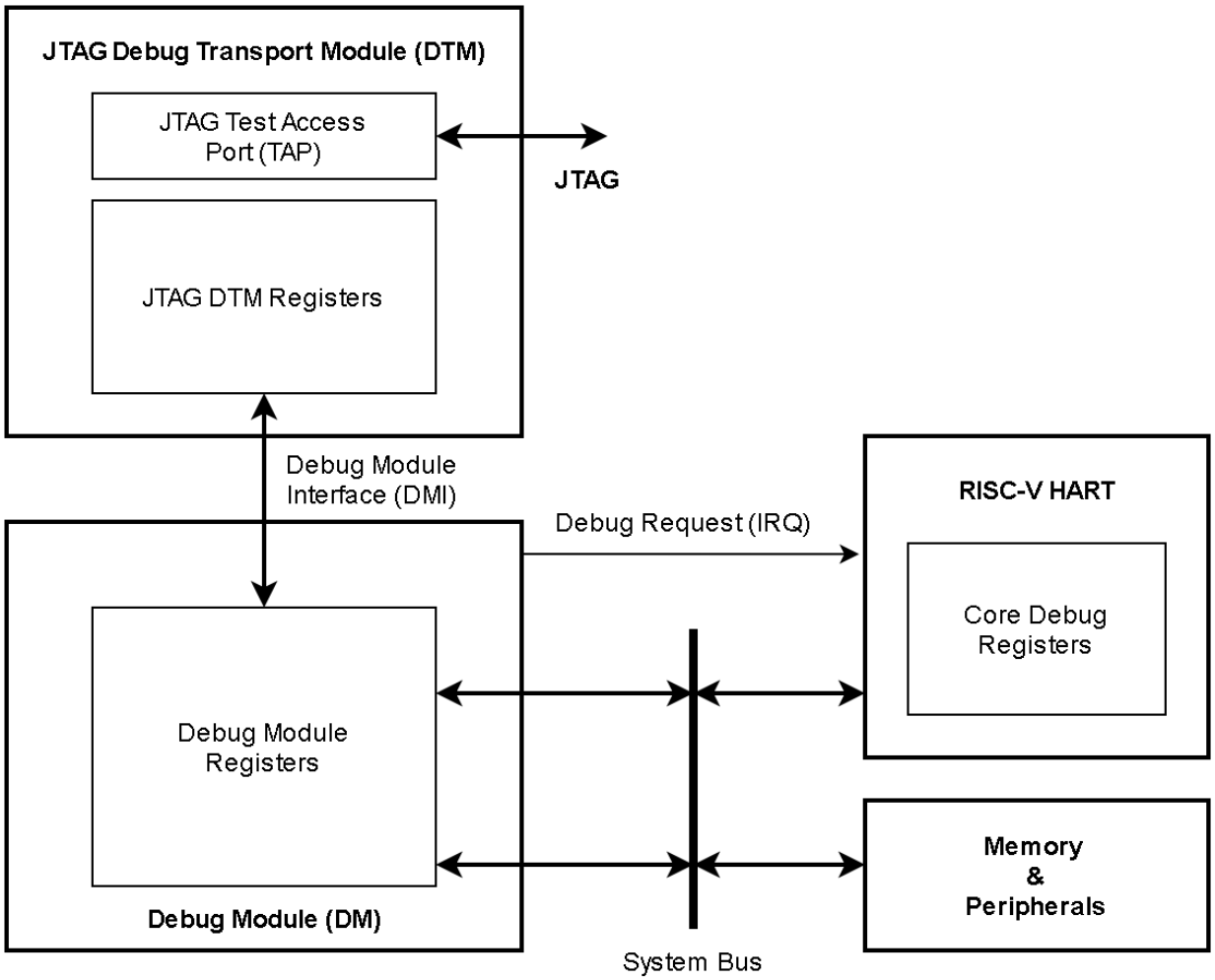
Architectural Overview

This chapter describes the architectural overview of the Debug support interfaces and components and introduces the functionality of the major Debug components. It contains the following sections:

- System Overview

System Overview

The user interacts with the Debug Host (e.g. laptop), which is running a debugger (e.g. gdb). The debugger communicates with a Debug Translator (e.g. OpenOCD, which may include a hardware driver) to communicate with Debug Transport Hardware (e.g. Olimex USB-JTAG adapter). The Debug Transport Hardware connects the Debug Host to the hardware platform's Debug Transport Module (DTM). The DTM provides access to one or more Debug Modules (DMs) using the Debug Module Interface (DMI).



The hart in the hardware platform is controlled by exactly one DM.

DMs provide run control of the hart in the hardware 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 also be used to access memory.

The current version of the specification considers only a single hart. All dependencies are concerning the single hart until mentioned explicitly.

Debug Module (DM)

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

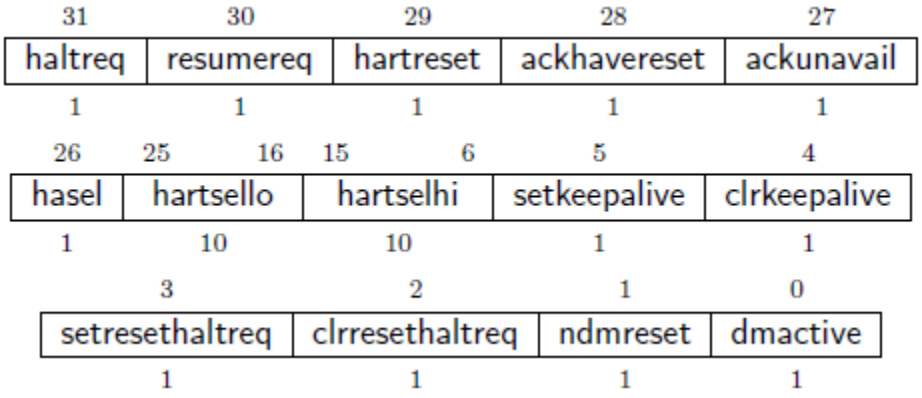
1. Give the debugger necessary information about the implementation.
2. Allow hart to be halted and resumed
3. Provides abstract read and write access to a halted hart's GPRs.
4. Provides access to a reset signal that allows debugging from the very first instruction after reset.
5. Provides a Program Buffer to force the hart to execute arbitrary instructions.
6. Single-Step execution support.

DM Registers

Debug functionality lies within the DM Registers. The debug registers implemented as a part of this specification is as follows.

Debug Module Control (`dmcontrol`)

This register controls the overall Debug Module as well as the currently selected hart.



- **haltreq -** Bit used to control the halt request for the selected hart.
- **resumereq -** Bit used to control the resume request for the selected hart.
- **hartreset** - Bit used to control the reset for the selected hart.
- **ackhavereset** - Acknowledgement bit used to clear the havereset bits.
- **ackunavail** - Acknowledgement bit used to clear the unavail bits.
- **hasel** - Selects the definition of currently selected harts.
- **hasello** - The low 10 bits of hartsel: the DM-specific index of the hart to select.
- **haselhi** - The high 10 bits of hartsel: the DM-specific index of the hart to select.
- **setresethaltreq & clrresethaltreq** - bits used to control the halt-on-reset feature.
- **ndmreset** - Bit used to control the reset signal from the DM to the rest of the hardware platform.

- **dmactive** - Bit used to control the reset signal for the Debug Module itself

On any given write, a debugger should only write 1 to at most one of the following bits: resumereq, hartreset, ackhavereset, setresethaltreq, and clrresethaltreq. The others must be written 0.

Hart DM States

A hart is exactly in any one of the following four DM states:

1. Non-existent
2. Unavailable
3. Running
4. Halted

Non-existent

Harts are nonexistent if they will never be part of this hardware platform, no matter how long a user waits. In a simple single-hart hardware platform, only one hart exists, and all others are nonexistent.

Unavailable

Harts are unavailable if they might exist/become available at a later time, or if there are other harts with higher indexes than this one. In a single-hart hardware platform, the hart may be unavailable for one of the following reasons

- Reset
- Temporarily powered down
- Not plugged into the hardware platform.

There are no guarantees about the state of the hart when it becomes available.

Running

Harts are running when they are executing normally as if no debugger was attached. This includes being in a low power mode or waiting for an interrupt, as long as a halt request will result in the hart being halted.

Halted

Harts are halted when they are in Debug Mode, only performing tasks on behalf of the debugger.

Harts will be unavailable while reset is asserted, and sometime after reset is de-asserted. They might transition to running for some time after the reset is de-asserted. Finally, the hart will end up either running or halted, depending on `haltreq` and `resethaltreq`.

Reset Control

Two methods allow a debugger to reset harts.

Non-Debug Module Reset

`ndmreset` resets all the harts in the hardware platform and all other parts of the hardware platform except for the Debug Modules, Debug Transport Modules, and Debug Module Interface.

Exactly what is affected by this reset is implementation-dependent, but it is possible to debug programs from the first instruction executed.

The sequence is as follows:

1. Set (Assert) the Non-Debug Module Reset bit
   `ndmreset = 1`
2. Clear (De-assert) the Non-Debug Module Reset bit
   `ndmreset = 0`

Hart Reset

`hartreset` resets the hart.

The sequence is as follows

1. Set (Assert) the Hart Reset bit
   `hartreset = 1`
2. Clear (De-assert) the Hart Reset bit
   `hartreset = 0`

The actual reset may start as soon as the bit is asserted, but may start an arbitrarily long time after the bit is de-asserted. The reset itself may also take an arbitrarily long time.

While the reset is ongoing, harts are either running, indicating it's possible to perform some abstract commands during this time, or in the unavailable state, indicating it's not possible to perform any abstract commands during this time. Once a hart's reset is complete, `havereset` is set.

When a hart comes out of reset and if either `haltreq` or `resethaltreq` is set, the hart will immediately enter Debug Mode (halted state). Otherwise, if the hart was initially running it will execute normally (running state) and if the hart was initially halted it will now be running but may be halted.

Debug Module Reset

The Debug Module's state and registers will be reset at power-up and while `dmactive` is 0. All the harts accessible to the DM will be reset.

Due to clock and power domain crossing issues, it might not be possible to perform arbitrary DMI accesses across hardware platform reset. While `ndmreset` or any external reset is asserted, the only supported DM operations are reading and writing `dmcontrol`.

When harts have been reset, they are set a sticky `havereset` state bit. The conceptual `havereset` state bits are reflected in `anyhavereset` and `allhavereset`.

These bits are set regardless of the cause of the reset. The `havereset` bit for the harts is cleared by writing 1 to `ackhavereset`. The `havereset` bits are cleared when `dmactive` is low.

Selecting Hart

A single DM supports up to 2^{20} harts. The debugger will select the hart and all the subsequent halt, resume, reset and abstract commands are specific to that hart.

In the current implementation, the Zilla core has a single hart bearing a hart index of 0. So, the `hartsello` is set to 0, and `hartselhi` is hardwired to 0.

Run Control

Debug Module tracks 4 conceptual bits of state:

1. halt request
2. resume ack
3. halt-on-reset request (optional)
4. hart reset (optional)

The DM receives halted, running, and havereset signals from the hart.

Halting

When a debugger writes 1 to `haltreq`, the hart's halt request bit is set. When a running hart, or a hart just coming out of reset, sees its halt request bit high, it responds by halting, de-asserting its running signal, and asserting its halted signal.

The hart will ignore the halt request bit if it is already halted.

Resuming

When a debugger writes 1 to `resumereq`, the hart's resume ack bit is cleared and the hart is sent a resume request. Harts respond by resuming, clearing their halted signal, and asserting their running signal. At the end of this process, the resume ack bit is set.

The hart will ignore the resume request if it is already running.

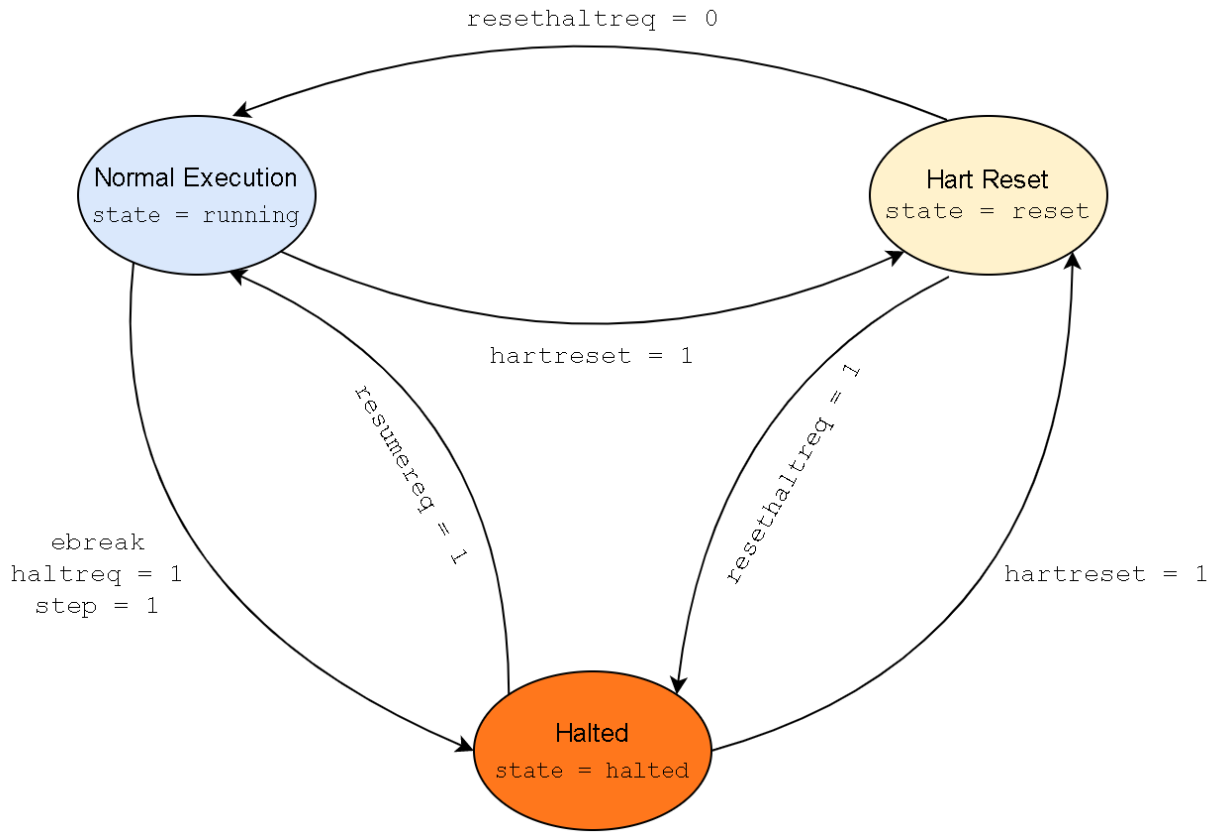
Halt-on-Reset

The implementation of the Halt-on-Reset feature is indicated by `hasresethaltreq` being set to 1. The halt-on-reset functionality is controlled by `setresethaltreq` and `clrresethaltreq`.

The halt-on-reset request bit for the hart is set by writing 1 to `setresethaltreq`. When a hart's halt-on-reset request bit is set, the hart will immediately enter debug mode on the next de-assertion of its reset.

This is true regardless of the reset's cause. The hart's halt-on-reset request bit remains set until cleared by the debugger writing 1 to `clrresethaltreq`, or by DM reset.

State-Machine



Abstract Commands

The DM supports Access Register Abstract Command to facilitate this functionality. This `command` gives the debugger access to CPU registers and allows it to execute the Program Buffer.

Abstract Data

These are DM registers that are read and written by the Abstract Commands.

The number of data registers implemented is reflected in `datacount`.

Accessing these registers while an abstract command is executing causes `cmderr` to be set to 1 (busy).

The contents in these registers are preserved after the execution of abstract commands and only cleared when DM is reset.

Register Memory Map

The Register Numbers for Abstract Access are as shown in the table.

Numbers	Group Description
0x0000 - 0x0fff	CSRs. The “PC” can be accessed here through `dpc`.
0x1000 - 0x101f	GPRs
0x1020 - 0x103f	Floating Point Registers
0xc000 - 0xffff	Reserved for non-standard extensions and internal use.

Abstract Command Execution

The sequence of operation is as follows:

1. The `command` is executed based on the following table.

`transfer`	`write`	Copy Data	
		From	To
0	0	-	-
0	1	-	-
1	0	regno.	arg0
1	1	arg0	regno.

2. If `postexec` is set, the Program Buffer is executed.

If any of these operations fail, `cmderr` is set and none of the remaining steps are executed.
If the failure is that the requested register does not exist in the hart, `cmderr` is set to 3 (exception).

Debuggers execute abstract commands when the `command` is written. The control and status of the execution of `command` are reflected in the `abstractcs`.

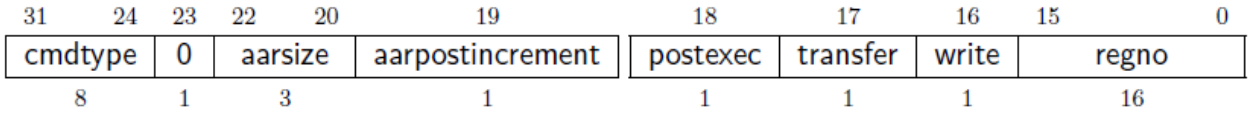


Fig: Abstract command

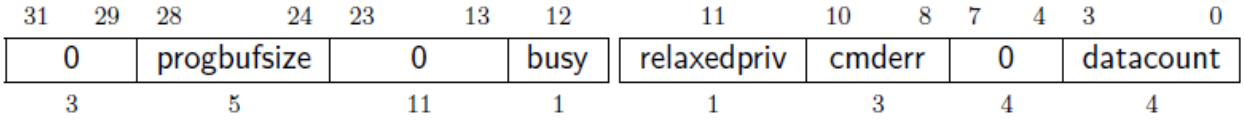


Fig: Abstract Control and Status

Before Execution

Before the execution of `command`, the following conditions have to be met.

- `haltreq`, `resumereq`, and `ackhavereset` should be 0.
- `cmderr` is 0.

The `busy` bit is set to 1 as soon as the `command` is written.

During Execution

- The `haltreq`, `resumereq`, `ackhavereset`, `setresethaltreq`, or `clrresethaltreq` bits should not be written.

Additional Cases

- If an abstract command does not complete in the expected time and appears to be hung, the debugger has to reset the hart (using `hartreset` or `ndmreset`).
- If that doesn't clear `busy`, then Debug Module has to be reset (using `dmactive`).
- If an abstract command is started while the selected hart is unavailable or if a hart becomes unavailable while executing an abstract command, then the Debug Module will terminate the abstract command,
    - setting `busy` low, and
    - `cmderr` to 4 (halt/resume).

Program Buffer

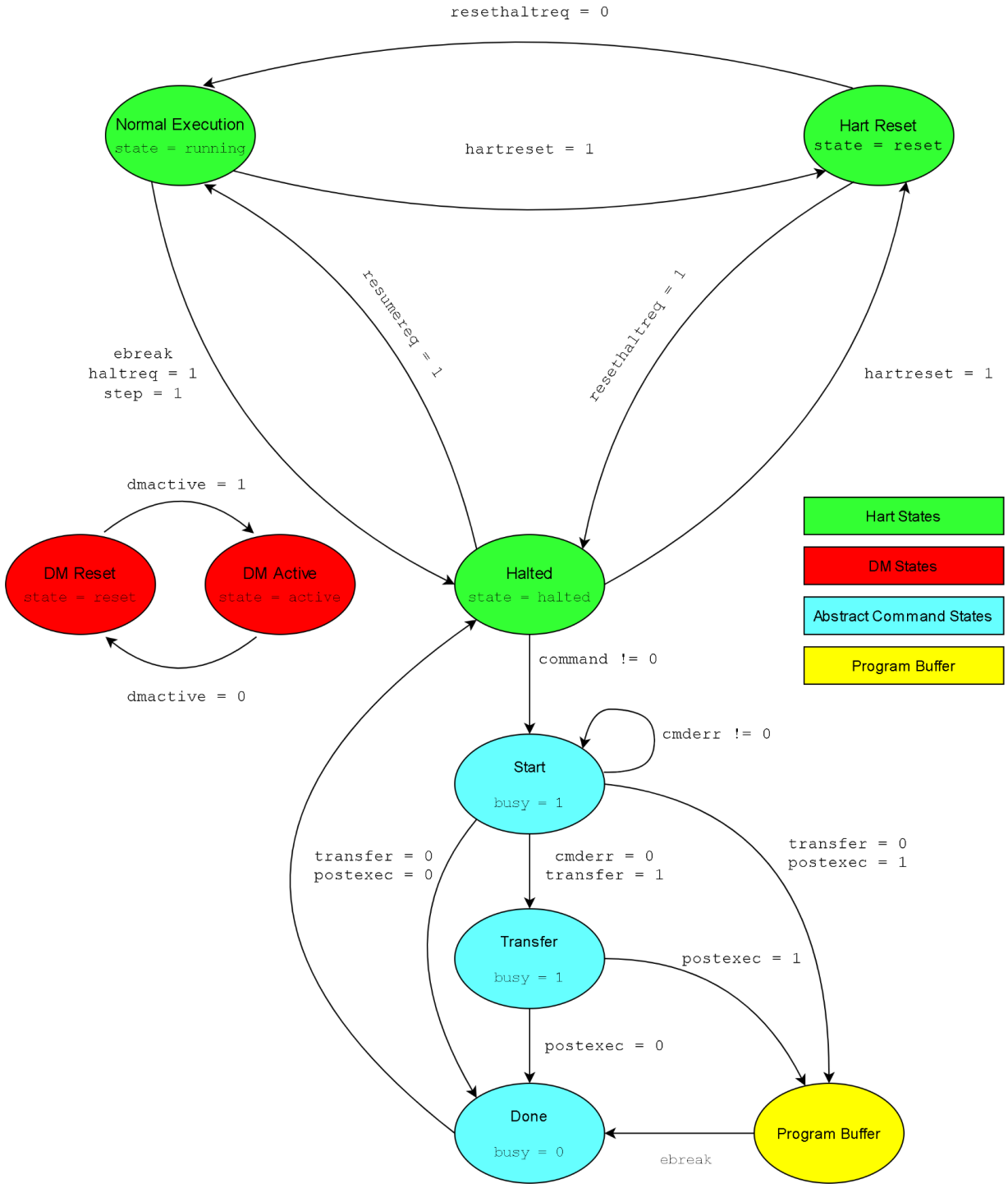
Arbitrary instructions can be executed on the halted hart, using the Program Buffer feature. The instructions written to the Program Buffer are executed exactly once with the Access Register Abstract Command, by setting the `postexec` bit.

The size of the Program Buffer implemented is reflected in `progbufsize`. Valid sizes are 0-16.

The instructions are pushed to the halted pipeline, but the hart will still be in debug mode (halted). The Program Buffer will end with an `ebreak` statement. At the end of the program

buffer, the processor enters Debug Mode again due to the execution of `ebreak` instruction. Consequently, the `cause` is updated to 1 (`ebreak`).

An implicit `ebreak` is supported, indicated by `impebreak`. With this feature, the complete Program Buffer size (16-words) can be utilized for efficient debugging.



Programmer’s Model

This chapter describes the DM and DTM registers. It contains the following sections:
About the Programmer’s model
DM Register Description
DTM Register Description

About the Programmer’s model

DM Register Map

All the registers are 32-bit wide unless specified explicitly. The registers described in this section are accessed over the DMI bus. Each DM has a base address and since there is only one DM in this implementation, all the addresses mentioned are absolute addresses.

Address	**Register Name**	**Description**
0x04 - 0x0f	`data0 - data11`	Abstract Data 0 - Abstract Data 11
0x10	`dmcontrol`	Debug Module Control
0x11	`dmstatus`	Debug Module Status
0x12	`hartinfo`	Hart Information
0x16	`abstractcs`	Abstract Control and Status
0x17	`command`	Abstract Command
0x20 - 0x2f	`progbuf0 - progbuf15`	Program Buffer 0 - Program Buffer 15

DTM Register Map

Address	**Register Name**	**Description**
0x00	`BYPASS`	JTAG recommends this encoding
0x01	`IDCODE`	To identify a specific silicon version
0x10	`dtmcs`	DTM Control and Status
0x11	`dmi`	Debug Module Interface Access
0x12 - 0x17	`Reserved (BYPASS)`	Reserved for future RISC-V debugging

0x17	`command`	Abstract Command
0x1f	`BYPASS`	JTAG requires this encoding

DM Register Description

Abstract Data (`data0-data11`, 0x04-0x0f)

Debug Module Control (`dmcontrol`, at 0x10)

Debug Module Status (`dmstatus`, at 0x11)

Hart Information (`hartinfo`, at 0x12)

Abstract Control and Status (`abstractcs`, at 0x16)

Abstract Command (`command`, at 0x17)

Program Buffer (`progbuf0 - progbuf15`, 0x20-0x2f)

DTM Register Description

`IDCODE` (at 0x01)0000_0001

DTM Control and Status (`dtmcs`, at 0x10)0001_0000

Debug Module Interface Access (`dmi`, at 0x11)0001_0001

`BYPASS` (at 0x1f)0001_1111