SiFive E31 User Guide

v19.08p2p0

SiFive E31 User Guide

Proprietary Notice

Release Information

Version	Date	Changes
v19.08p2p0	December 06, 2019	• Fixed erratum in which the TDO pin may remain driven after reset
v19.08p1p0	November 08, 2019	• Fixed erratum in which Debug.SBCS had incorrect reset value for SBACCESS • Fixed typos and other minor documentation errors
v19.08p0	September 17, 2019	• The Debug Module memory region is no longer accessible in M-mode
v19.05p2	August 26, 2019	• Fix for errata on 3-series cores with L1 data caches or L2 caches in which CFLUSH.D.L1 followed by a load that is nack'd could cause core lockup
v19.05p1	July 22, 2019	• SiFive Insight is enabled • Use AHB-Lite for external ports • Enable debugger reads of Debug Module registers when periphery is in reset • Fix errata to get illegal instruction exception executing DRET outside of debug mode
v19.05	June 09, 2019	• Updated deliverables description • Updated Simulation Testbench chapter to describe new Testbench and the Verilator Testbench target
v19.02	February 28, 2019	• Changed the date based release numbering system • Added the Verilog defines and Test Mode signals table
v3p0	June 01, 2018	• Moved Interface and Debug Interface chapters from the Core Complex Manual into this User Guide • Added chapters for Port Bridges • Added chapters Implementation Chapter
v1p0	May 11, 2017	• Initial release of the E31 Core Complex Product User Guide • Describes the deliverables of the E31 Core Complex v1p0 Product Release

Contents

Chapter 1

Introduction

1.1 About this Document

This document describes the E31 v19.08p2p0 production deliverables. To learn more about the functionality of the E31 please read the E31 Manual.

1.2 About this Release

This is the Production release of the E31 v19.08p2p0.

Chapter 2

Deliverables

This chapter describes the contents of the deliverables for the E31 v19.08p2p0.

2.1 Folder Structure

The folder structure of the delivery is as follows:

arty_a7_100t-sifive
Contains the designs FPGA bitstream.

freedom-e-sdk
Software SDK for the design including Freedom Metal BSP and applications.

bsp
Freedom Metal BSP for the RTL testbench and, where applicable, the FPGA bitstream. Note that Freedom Metal BSPs also include a design's Device Tree file (DTS).

freedom-devicetree-tools
Tools used to generate Freedom Metal BSPs from DTS files. Can be used to re-generate BSPs in the case of a hand-edited DTS file.

freedom-metal
Source code for the Freedom Metal library.

scripts
Helper scripts used by the main makefile.

software
One folder for each included Freedom Metal application. Each application includes a pre-built hex file in its *release* directory which can be run directly on the testbench without needed to re-compile from source. Note: In some cases there will not be a pre-built hex file for every application. This will be the case when a particular application is not expected to run correctly on the selected core configuration. Possible reasons for this are:

- Executable image does not fit in configured boot memory
- Application data does not fit in configured data memory
- Multicore application excluded for single-core configuration

Makefile
Top-level SDK makefile which can be used to re-build all included examples from source. Readme.md; Readme file describing how to use the SDK's top-level makefile.

info
Files which describe the design.

mems.conf
Configuration file which describes the memory instances of the design.

modules_to_be_retimed.txt
Contains the list of modules which need to be retimed.

sifive_insight.yml
Contains a .yml description of the SiFive Insight signals included in the design. See Section 10.3 for more details on SiFive Insight.

rtl
The E31 RTL.

memories
A single verilog file containing all memories in the design.

testbench
Includes all the modules in the synthesizable testbench, the test driver, and extracted simulation constructs (assertions) that are bound to locations in the DUT.

design
The E31 itself. Includes the top-level module `E3_CoreIPSubsystem` and all submodules.

sifive_insight
Contains all the System Verilog files defining and binding the SiFive Insight signals to modules in the design.

.F files
Manifest files for the associated folder. A complete list of files to be synthesized as part of the design can be found in `design.F`.

Makefile
Used to execute the test bench described in Chapter 10.

Chapter 3

Memories

This chapter describes the memories used in the E31 design.

3.1 RAM Instances

The Core Complex RAM instances consist of synchronous single-ported SRAMs. These are contained within wrapper modules that expose a standardized generic interface for the E31. For each of the modules specified in Table 1, the implementer is required to provide a module definition that instantiates the SRAM macros and connects the macro-specific pins to the interface described in Table 2.

Behavioral models of the RAMS are provided as part of the deliverable in the file: `verilog/memories/CoreIPSubsystem*`.

The E31 RAM instances are delivered as is and are not configurable. It is, however, possible to construct the memory instances from multiple smaller instances.

Module Name	Depth	Address Width (N_{addr})	Data Width (N_{data})	Write Mask Granularity (N_{part})	Description
data_arrays_0_0_ext	2048	11	64	32	ICache Data Array
data_arrays_0_ext	4096	12	32	8	
tag_array_ext	128	7	40	20	

Table 1: SRAM Modules and Configuration

Name	Direction	Width	Description
RW0_clk	Input	1	Memory clock
RW0_en	Input	1	Active-high signal indicating that the memory is being access. This may be used for clock gating.
RW0_addr	Input	N_{addr}	Address of access.
RW0_rdata	Output	N_{data}	Read data
RW0_wmode	Input	1	Active-high signal indicating that the access is a write operation.
RW0_wdata	Input	N_{data}	Write data.
RW0_wmask	Input	N_{data}/N_{part}	Active-high write mask. Each bit controls whether or not the corresponding N_{part}-bit subword is written. This is present only in memories that require mask write functionality.

Table 2: SRAM Signals

3.2 DTIM Sizing

It is possible to implement less than the maximum specified 16 KiB DTIM. When doing so, boot-time software must program a Locked PMP region spanning the unimplemented address space to guarantee that accesses to unimplemented memory space are trapped accordingly. Please see the Physical Memory Protection Chapter in the E31 Manual for more details on how to configure PMP.

Chapter 4

E31 Interfaces

This chapter describes the primary interfaces to the E31.

4.1 Clock & Reset

The `core_clock_0`, `clock`, `rtc_toggle`, `reset`, and `reset_vector_0` inputs are described in Table 3.

The relationship between the clock input frequencies are as follows:

`clock > 2 * rtc_toggle` and `core_clock_0 >= clock`

Name	Direction	Width	Description
`core_clock_0`	Input	1	The core pipeline and cache clock
`clock`	Input	1	Clock input to the PLIC and the external ports. Has a `1/m` frequency relationship with `core_clock_0` where m ≥ 1.
`rtc_toggle`	Input	1	The Real Time Clock input. Must run at strictly less than half the rate of `clock`.
`reset`	Input	1	Synchronous reset signal. Active high. Must be asserted for 16 cycles of `clock` and synchronously de-asserted.
`reset_vector_0`	Input	32	Reset Vector Address. Implementations MUST set this signal to a valid address.

Table 3: Clock and Reset Interfaces

4.1.1 Real Time Clock (`rtc_toggle`)

As defined in the RISC-V privileged specification, RISC-V implementations must expose a real-time counter via the `mtime` register. In the E31 the `rtc_toggle` input is used as the real-time counter. `rtc_toggle` must run at strictly less than half the frequency of `clock`. Furthermore, for

RISC-V compliance, the frequency of `rtc_toggle` must remain constant, and software must be made aware of this frequency.

4.1.2 Peripheral Clock (`clock`)

The peripheral clock is used to decouple the frequency of the core from that of some of the on core complex peripherals. `clock` has a $\frac{1}{m}$ frequency relationship with `core_clock_0` where m is any positive integer. Additionally, these clocks must be phase-aligned.

The peripherals connected to `clock` are described in Section 9.2.

4.2 Ports

This section will describe all of the Ports in the E31.

Name	Base Address	Top	Protocol	Description
`front_port_ahb_0`			AHB	32-bit data width. Synchronous to `clock`
`periph_port_ahb_0`	`0x2000_0000`	`0x2000_1FFF`	AHB	32-bit data width. Synchronous to `clock`
`sys_port_ahb_0`	`0x4000_0000`	`0x4000_1FFF`	AHB	32-bit data width. Synchronous to `clock`

Table 4: E31 Platform Bus Interfaces

4.2.1 Front Port

The E31 has one master bus interface called the Front Port. This port can be used by external masters to read and write into any memory-mapped device in the system. Note that an external master using a Front Port can trigger an I-Cache to reconfigure itself by using the procedure described in the E31 Manual.

Reads and writes to a Front Port interface can also pass through to the Peripheral and System bus interfaces if a transaction falls within their address space. Note that transactions through a Front Port do not pass through the Physical Memory Protection (PMP) unit.

The E31 Front Port interface is fully compliant with the *AMBA 3 AHB-Lite Protocol, Version 1.0* specification, and Front Port transactions pass through a(n) AHB to TileLink Bridge bridge (AHB2TL).

The Front Port is described in Table 4. The AHB2TL bridge is described in Chapter 7.

4.2.2 Peripheral Port

The E31 has one Peripheral Port, which is typically used to access peripheral devices in off-core-complex address space. Errors that propagate to the processor via the Peripheral Port have the effects described in Chapter 5. The Peripheral Port is described in Table 4.

The E31 Peripheral Port pass through a TileLink to AHB Bridge (TL2AHB) which is described in Chapter 6.

4.2.3 System Port

The E31 has one System Port, which is typically used to access higher-bandwidth peripheral devices in off-core-complex address space. Errors that propagate to the processor via the Peripheral Port have the effects described in Chapter 5. The System Port is described in Table 4.

The E31 System Port pass through a TileLink to AHB Bridge (TL2AHB) which is described in Chapter 6.

4.3 E31 Interrupt Interfaces

This chapter describes all of the interrupt signals in the E31.

4.3.1 External Global Interrupts

Global interrupts are interrupts which are connected to the PLIC from external interrupt sources. Please see the E31 Manual for a detailed description of global interrupts.

Name	Direction	Width	Description
`global_interrupts`	Input	127	External interrupts from off-chip or peripheral sources. These are level-based interrupt signals connected to the PLIC and must be synchronous with `clock`.

Table 5: External Global Interrupt Interface

4.3.2 Local External Interrupts

Local interrupts are interrupts which can be connected to peripheral sources and signaled directly to an individual hart. Please see the E31 Manual for a detailed description of local interrupts.

Name	Direction	Width	Description
local_interrupts_0	Input	16	Interrupts from peripheral sources destined to core 0. These are level-based interrupt signals connected directly to the core and must be synchronous with clock

Table 6: Local External Interrupt Interface

4.4 Debug Output Signals

Signals which are outputs from the debug module are shown in Table 7.

Name	Direction	Width	Description
debug_ndreset	Output	1	This signal is a reset signal driven by the debug logic of the chip. It can be used to reset parts of the SoC or the entire chip. It should NOT be wired into logic which feeds back into the debug_systemjtag_reset signal for this block. This signal may be left unconnected.
debug_dmactive	Output	1	This signal, 0 at reset, indicates that debug logic is active. This may be used to prevent power gating of debug logic, etc. It may be left unconnected.

Table 7: External Debug Logic Control Pins

4.5 JTAG Debug Interface Pinout

SiFive uses the industry-standard JTAG interface which includes the four standard signals, TCK, TMS, TDI, and TDO. A test logic reset signal must also be driven on the debug_systemjtag_reset input. This reset is synchronized internally to the design. The test logic reset must be pulsed before the core reset is deasserted.

Name	Direction	Width	Description
debug_systemjtag_TCK	Input	1	JTAG Test Clock
debug_systemjtag_TMS	Input	1	JTAG Test Mode Select
debug_systemjtag_TDI	Input	1	JTAG Test Data Input
debug_systemjtag_TDO_data	Output	1	JTAG Test Data Output
debug_systemjtag_TDO_driven	Output	1	JTAG Test Data Output Enable
debug_systemjtag_reset	Input	1	Active-high Reset
debug_systemjtag_mfr_id	Input	11	The SoC Manufacturer ID which will be reported by the JTAG IDCODE instruction.

Table 8: SiFive standard JTAG interface for off-chip external TAPC

Chapter 5

E31 Error Handling

This chapter describes how the E31 handles errors from its memories and interfaces.

Errors can be introduced to the core via ECC errors or error responses returned on the various port interfaces. For port interfaces that are not natively TileLink, their error responses are translated into TileLink errors as described in the respective bridge chapters. The core's behavior is purely determined by the type of TileLink or ECC error that it receives. The behavior is also dependent on the type of core.

5.1 3/5/7 Series Error Handling

For 3-, 5-, or 7-series cores, on the various interfaces to the E31:

- TileLink denied/corrupt on I$ refills result in precise exceptions
- TL denied on D$ refill prevents refill from occurring
- TL corrupt on D$ refill performs the refill
- TL denied/corrupt on MMIO is ignored by the core

Chapter 6

TileLink to AHB Bridge (TL2AHB)

6.1 Introduction

SiFive's TileLink to AHB Bridge (TL2AHB) can be used to connect SiFive Core Complex IP to AMBA 3 AHB Protocol v1.0 based systems. SiFive Core Complex IP natively uses TileLink for all system communication external to the Core Complex. The TL2AHB bridge translates TileLink transactions to AMBA 3 AHB Protocol v1.0.

6.2 Compliance

- The SiFive TL2AHB is fully compliant with AMBA 3 AHB Protocol v1.0 and this document should be read in conjunction with the AMBA 3 AHB Protocol v1.0 Protocol Specification.
- The SiFive TL2AHB is fully compatible with SiFive Core Complex IP. Some properties of the TL2AHB are specific to a given Core Complex implementation. This document should be read in conjunction with the Core Complex IP Manual.

6.3 Block Diagram

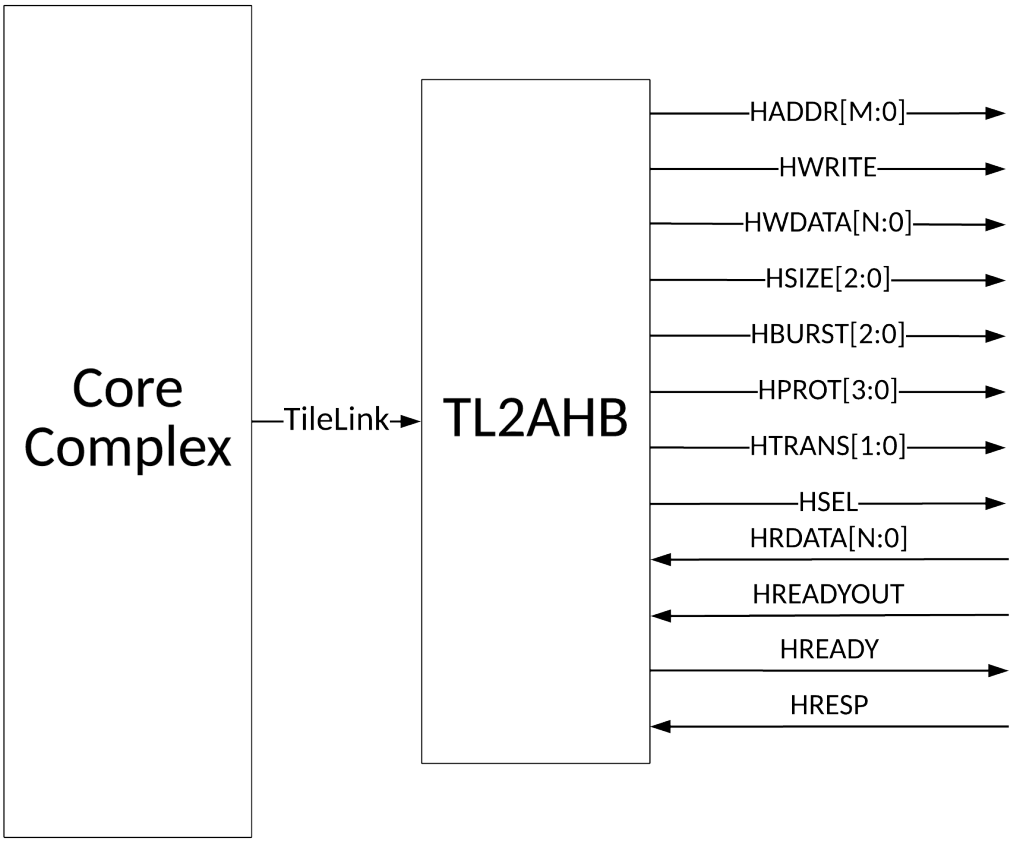

Figure 1: TL2AHB Block Diagram

6.4 TL2AHB Interface

Name	Direction	Width	Description
HADDR	Out	[M:0]	Transfer address where M is the minimum width necessary for the address range assigned to TL2AHB in a given Core Complex implementation.
HWRITE	Out	1	When HIGH this signal indicates a write transfer. When LOW this signal indicates a read transfer.
HWDATA	Out	[N:0]	Write transfer data where N is dependent on a given Core Complex implementation.
HSIZE	Out	[2:0]	The size of the transfer.
HBURST	Out	[2:0]	Burst type indicator.
HPROT	Out	[3:0]	Protection control signals.
HTRANS	Out	[1:0]	Transfer type of current transfer.
HRDATA	In	[N:0]	Read transfer data where N is dependent on a given Core Complex implementation.
HREADYOUT	In	1	Used by a slave to indicate when its transfer has finished. See Section 6.5.3 for more details.
HREADY	Out	1	Used to indicate to the master and all slaves that the previous transfer has finished. See Section 6.5.3 for more details.
HRESP	In	1	Response signal used to indicate transfer status: OKAY or ERROR.
HSEL	Out	1	Indicates that the current transfer is intended for the selected slave. See Section 6.5.3 for more details.

Table 9: AHB Interface

6.5 Functional Description

This section will describe the functional behavior of the TL2AHB in more detail.

6.5.1 Atomic Memory Operations (AMO)

- TileLink AMOs are translated to Read-Modify-Write operations and therefore no longer atomic.
- Because AMOs issued through the TL2AHB can not guarantee atomicity, they should not be issued through the TL2AHB in a multi-master system.

6.5.2 Bursts

- The TL2AHB will fragment a TileLink burst transaction to the largest supported AHB burst size
- Request are always aligned to the burst size.

- Only fixed size incrementing or single beat burst are issued, specifically:
  - SINGLE
  - INCR4
  - INCR8
  - INCR16
- Multi-beat narrow burst are never issued.

HADDR

- For a given Core Complex implementation, the width of HADDR is the minimum width necessary for the the address range of the TileLink bus it is connected to.

HMASTLOCK

- HMASTLOCK is always tied to 0.

HPROT

- HPROT is tied to 0x3: Non-cacheable, Non-bufferable, privileged, data access.

HSEL

- A single HSEL bit is implemented and used to indicate when the port is active.
- An external arbiter and decoder is required to select more than one slave device. See Section 6.5.3.

HRESP

- When HRESP indicates a transfer error, the signal is translated into the TileLink response **d_error**. OKAY = LOW.

6.5.3 TL2AHB System Integration

The TL2AHB is implemented in a way which allows for some flexibility when integrating into a larger system. It is possible to:

- Directly connect a single slave to the TL2AHB described in Section 6.5.3.1.
- Connect to a Decoder/Multiplexor which allows for multiple slave connections described in Section 6.5.3.2.

TL2AHB Direct Slave Connection

- In this use case, the TL2AHB signals map directly onto the Slave's interface. This is depicted in Figure 2.

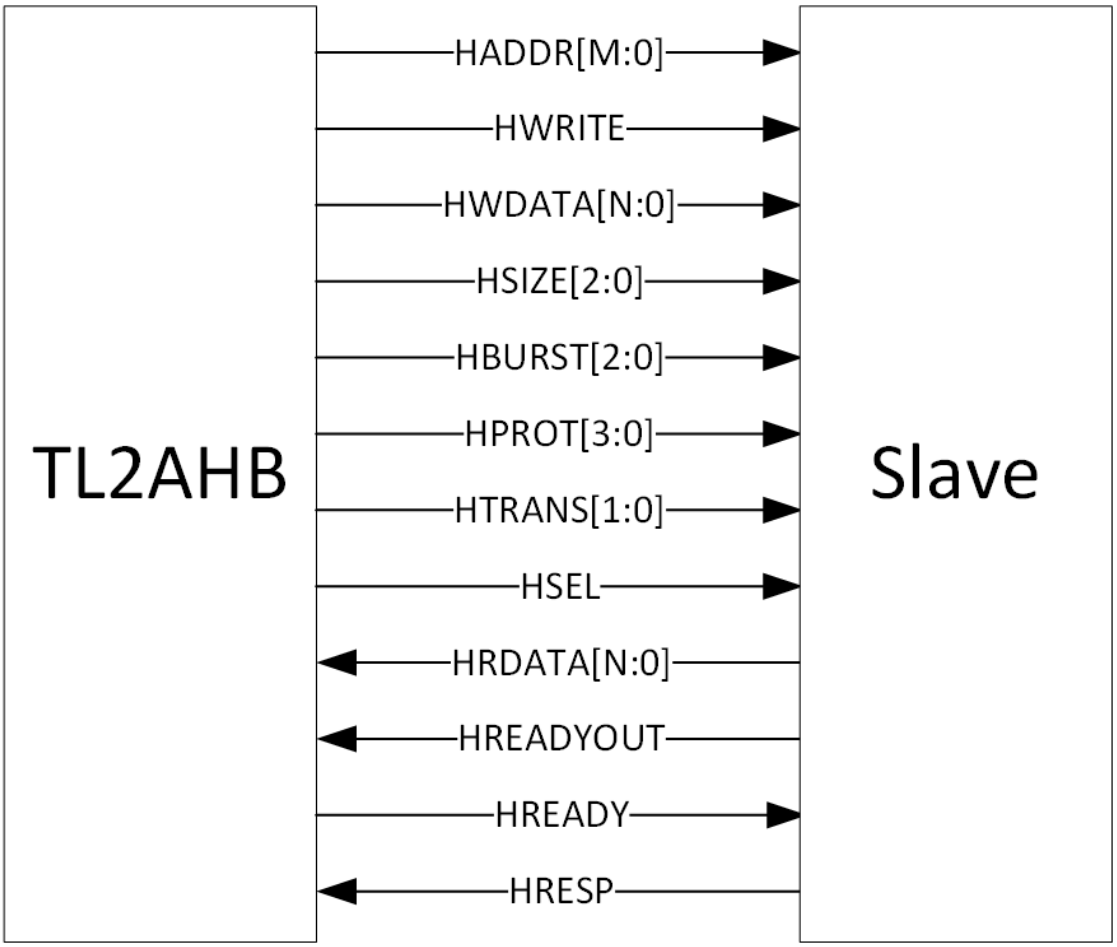

Figure 2: TL2AHB connected directly to a slave

TL2AHB Decoder/Multiplexor Connection

- The HREADY signal from the decoder should be connected to the TL2AHB's HREADYOUT signal.
- HSEL and HREADY on the TL2AHB should be left floating.

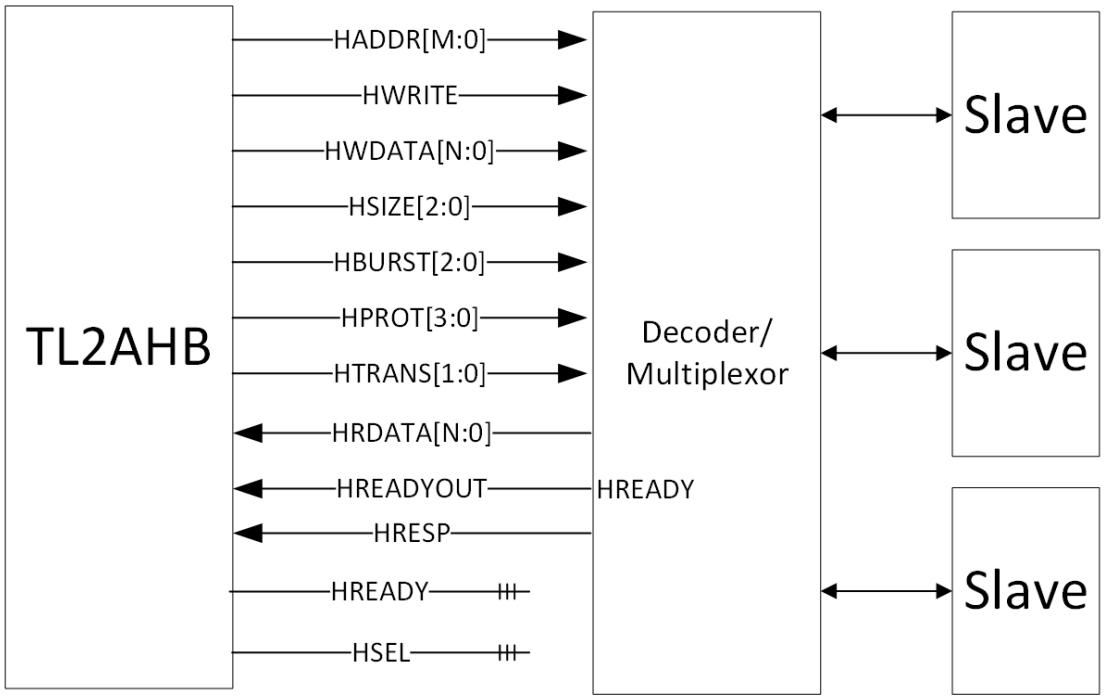

Figure 3: TL2AHB connected directly to a slave

Chapter 7

AHB to TileLink Bridge (AHB2TL)

7.1 Introduction

SiFive's AHB to TileLink Bridge (AHB2TL) can be used to connect SiFive Core Complex IP to AMBA 3 AHB Protocol v1.0 based systems. SiFive Core Complex IP natively uses TileLink for all system communication external to the Core Complex. The AHB2TL bridge translates AMBA 3 AHB Protocol v1.0 transactions to TileLink.

7.2 Compliance

- The SiFive AHB2TL is fully compliant with AMBA 3 AHB Protocol v1.0 and this document should be read in conjunction with the AMBA 3 AHB Protocol v1.0 Protocol Specification.
- The SiFive AHB2TL is fully compatible with SiFive Core Complex IP. Some properties of the AHB2TL are specific to a given Core Complex implementation. This document should be read in conjunction with the Core Complex IP Manual.

7.3 Block Diagram

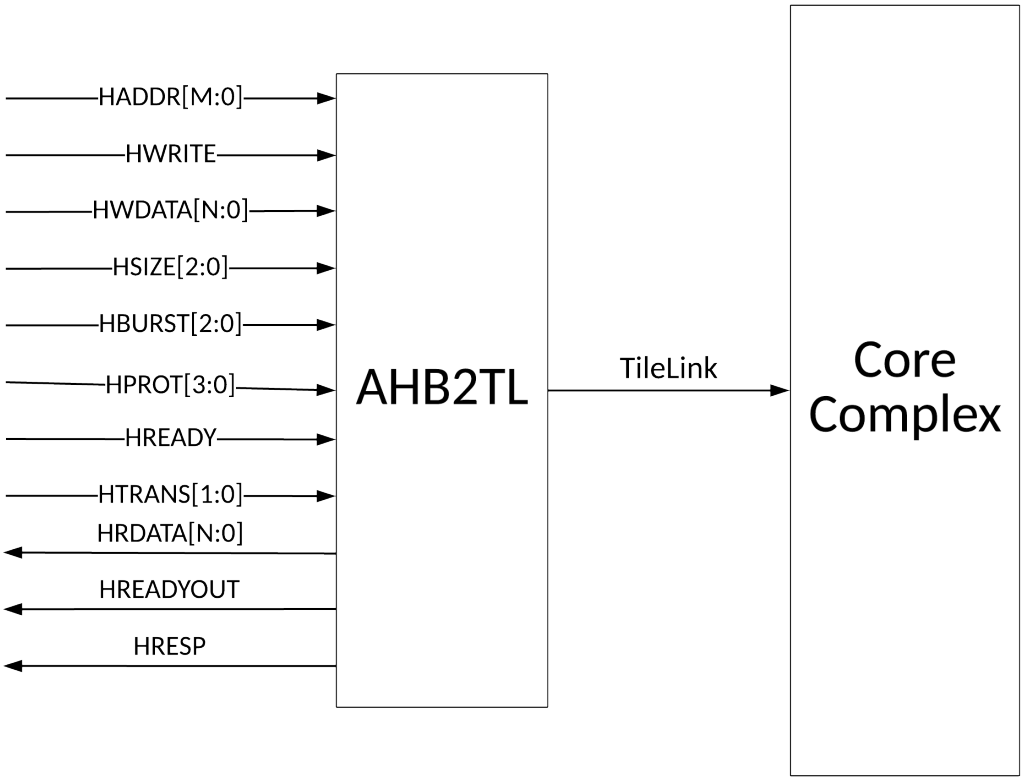

Figure 4: AHB2TL Block Diagram

7.4 AHB2TL Interface

Name	Direction	Width	Description
HADDR	In	[M:0]	Transfer address where M is the minimum width necessary for the address range assigned to AHB2TL in a given Core Complex implementation.
HWRITE	In	1	When HIGH this signal indicates a write transfer. When LOW this signal indicates a read transfer.
HWDATA	In	[N:0]	Write transfer data where N is dependent on a given Core Complex implementation.
HSIZE	In	[2:0]	The size of the transfer.
HBURST	In	[2:0]	Burst type indicator.
HPROT	In	[3:0]	Protection control signals.
HTRANS	In	[1:0]	Transfer type of current transfer.
HRDATA	Out	[N:0]	Read transfer data where N is dependent on a given Core Complex implementation.
HREADYOUT	Out	1	Used by a slave to indicate when its transfer has finished.
HREADY	In	1	Used to indicate to the master and all slaves that the previous transfer has finished.
HRESP	Out	1	Response signal used to indicate transfer status: OKAY or ERROR.
HSEL	In	1	Indicates that the current transfer is intended for the selected slave.

Table 10: AHB Interface

7.5 Functional Description

This section will describe the functional behavior of the AHB2TL in more detail.

7.5.1 Bursts

- All legal AHB bursts are supported.
- Narrow bursts, and bursts that are not aligned to the transfer size, do not result in improved throughput.

7.5.2 HADDR

- For a given Core Complex implementation, the width of HADDR in the AHB2TL bridge is the entire addressable address space including ITIM and DTIM. Transactions to off core complex addresses will pass through to the Core Complex's bus master interfaces.
- Illegal addresses result in HRESP=1.

7.5.3 HMASTLOCK

- HMASTLOCK is ignored.

7.5.4 HPROT

- HPROT is ignored.

Chapter 8

Debug Interface

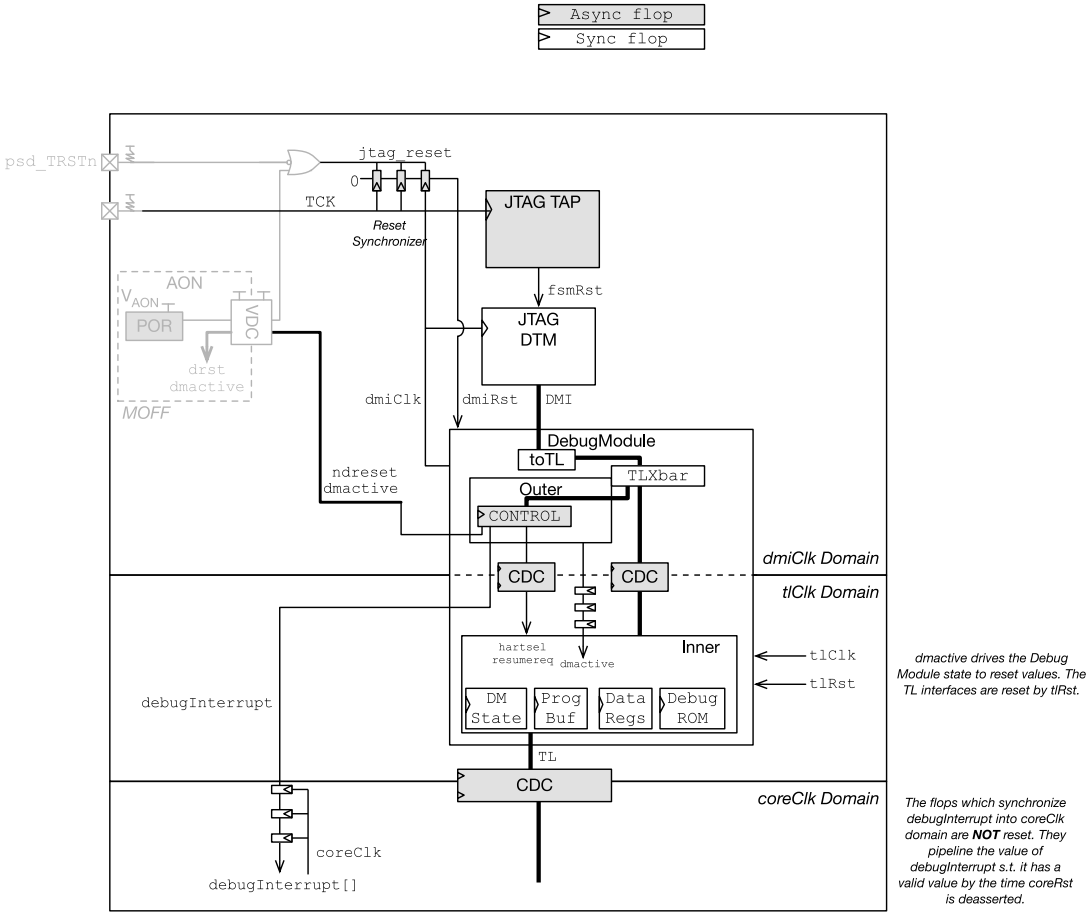

Figure 5: Debug Transport Module and Debug Module for HW Debug.

The SiFive E31 includes the JTAG debug transport module (DTM) described in *The RISC-V Debug Specification, Version 0.13*. This enables a single external industry-standard 1149.1 JTAG interface to test and debug the system. The JTAG interface can be directly connected off-

chip in a single-chip microcontroller or can be an embedded JTAG controller for a RISC-V Core IP designed to be included in a larger SoC.

The DTM and debug module are depicted in Figure 5.

On-chip JTAG connections must be driven (no pullups), with a normal 2-state driver for TDO under the expectation that on-chip mux logic will be used to select between alternate on-chip JTAG controllers' TDO outputs. TDO logic changes on the falling edge of TCK.

8.1 JTAG TAPC State Machine

The JTAG controller includes the standard TAPC state machine shown in Figure 6. The state machine is clocked with TCK. All transitions are labelled with the value on TMS, except for the arc showing asynchronous reset when TRST=0.

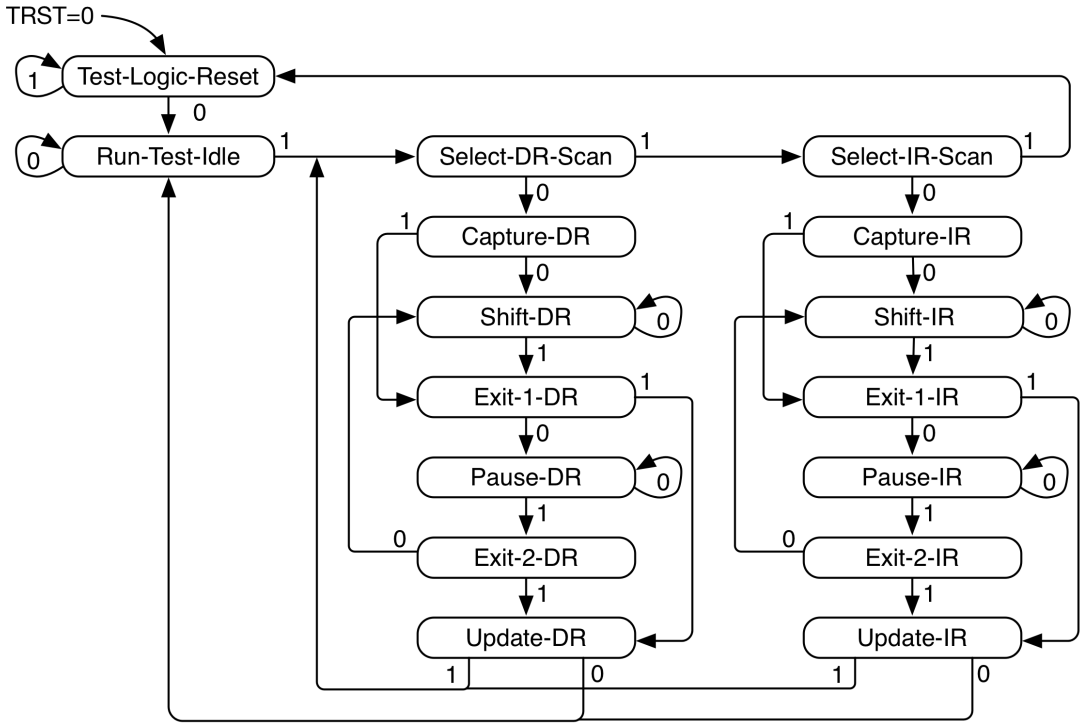

Figure 6: JTAG TAPC state machine.

8.2 Resetting JTAG Logic

The JTAG logic must be asynchronously reset by asserting `jtag_reset` before `coreReset` is deasserted.

Asserting `jtag_reset` resets both the JTAG DTM and debug module test logic. Because parts of the debug logic require synchronous reset, the `jtag_reset` signal is synchronized inside the E31.

During operation, the JTAG DTM logic can also be reset without `jtag_reset` by issuing 5 `jtag_TCK` clock ticks with `jtag_TMS` asserted. This action resets only the JTAG DTM, not the debug module.

8.3 JTAG Clocking

The JTAG logic always operates in its own clock domain clocked by `jtag_TCK`. The JTAG logic is fully static and has no minimum clock frequency. The maximum `jtag_TCK` frequency is part-specific.

8.4 JTAG Standard Instructions

The JTAG DTM implements the BYPASS and IDCODE instructions.

The Manufacturer ID field of IDCODE is provided by the RISC-V Core IP integrator, on the `jtag_mfr_id` input.

8.5 JTAG Debug Commands

The JTAG DEBUG instruction gives access to the SiFive debug module by connecting the debug scan register between `jtag_TDI` and `jtag_TDO`.

The debug scan register includes a 2-bit opcode field, a 7-bit debug module address field, and a 32-bit data field.

to allow various memory-mapped read/write operations to be specified with a single scan of the debug scan register.

These are described in *The RISC-V Debug Specification, Version 0.13*.

8.6 Using Debug Outputs

The debug module logic in SiFive Systems drives two output signals: `ndreset` and `dmactive`. These signals can be used in integration. It is suggested that the `ndreset` signal contribute to the system reset. It must be synchronized before it contributes back to the RISC-V Core IP's overall `reset` signal. This signal must not contribute to the `jtag_reset` signal. The `dmactive` signal can be used, for example, to prevent clock or power gating of the debug module logic while debugging is in progress.

Chapter 9

Implementation

This chapter will describe the steps necessary to synthesize E31.

9.1 Top Level

The top level verilog module is defined in the file:
`verilog/design/E3_CoreIPSubsystem.v`

All top level interfaces are described in the the E31 Chapter 4.

9.2 Clocking

The E31 has the following main clock inputs: `core_clock_X`, `clock`, `rtc_toggle`. `core_clock` is used to clock the processor and Level 1 memory system. `clock` is used to clock the bus, PLIC and debug interfaces. `rtc_toggle` is exposed via the architectually defined real time counter exposed in the `mtime` CSR.

The relationship between the clock input frequencies are as follows:

$$core\_clock \geq clock > (2 \times rtc\_toggle)$$

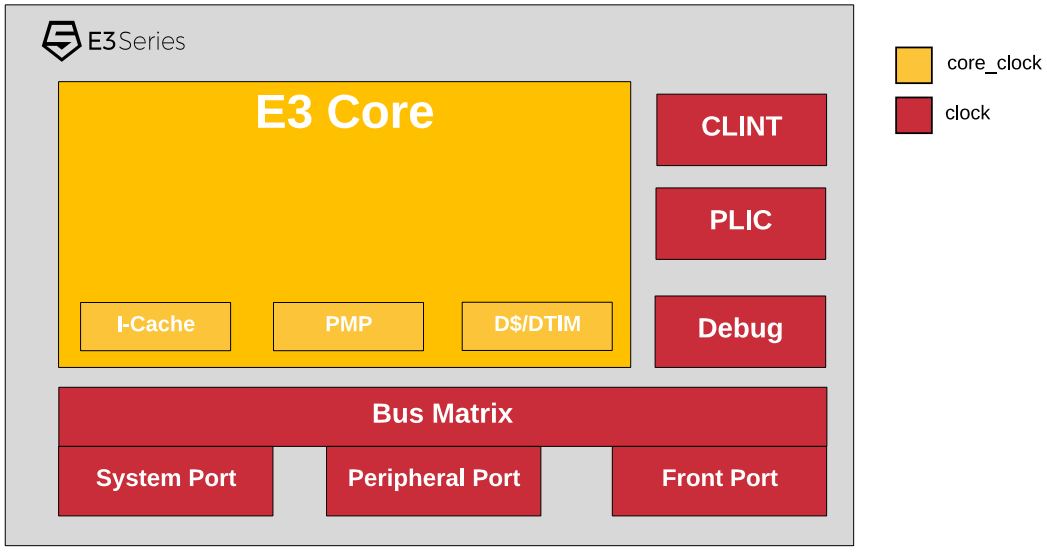

Figure 7: E31 Clock Diagram.

9.2.1 Clocking Guidelines

Depending on the maximum frequency of a given implementation, it may be advantageous to have `clock` run a lower frequency to `core_clock`. As depicted in Figure 7, `clock` is used by the bus interfaces, PLIC, and debug interfaces. It is common at higher frequencies for the busses, or a PLIC with a large number of interrupts and/or interrupt privilege levels, to become the critical path. In order to achieve higher frequencies, it is therefore recommended that the frequency of `clock` be half of `core_clock`. As a result of this, the PLIC will have slightly increased latency generating interrupts, and the maximum bandwidth available in the bus interfaces will be reduced. In most circumstances, these are acceptable tradeoffs. If targeting lower frequencies, it is OK to have `clock` and `core_clock` be of equal frequencies.

`rtc_toggle` is used by software for time keeping and therefore not necessary to have a high frequency. Generally, `rtc_toggle` is connected to either a Real Time Clock (e.g. 32.768 kHz) or to the base clock input frequency of the platform. The only restrictions on `rtc_toggle` are that it must be strictly less than half the frequency of `clock` and that it must remain constant. Furthermore, for RISC-V compliance, software must be made aware of this frequency via a header file or similar.

9.3 Retiming

A list of all modules (if any) which require retiming is included in the design deliverables. The list can be found in the *info/retiming_modules.txt* file.

9.4 Gate Level Simulation

To avoid X propagation, all state such as flip-flops and SRAMs must be initialized at the beginning of simulation, as well as after each power-up event if conducting power-aware simulation. Additionally, ensure that the SRAM models never produce X values by modifying them as necessary.

For Synopsys Design Compiler and IC Compiler, the command `all_registers -output_pins` can be used to enumerate all state elements in the design. The UCLI command `force -deposit <inst> <value>` can be used in VCS simulation to force the output pins to 0 or 1. VCS also requires a PLI table file to enable wn capability (debug access) on these instances.

Note that if the standard cell logic library contains flip-flops with inverted outputs (i.e., Q and QN), those pins must be initialized to opposite values.

Chapter 10

Simulation Testbench

10.1 Included Test Bench

The E31 includes an example test simulation environment which is designed to work with Synopsys VCS version K-2015.09-SP2 or higher, Cadence Xcelium version 17.05 or higher, and Verilator version 4.014 or higher. Several tests are included as part of the testbench to verify functionality and can be run using the included Makefile. The testbench tests and their sources are located in the included Freedom E SDK environment, specifically the `freedom-e-sdk/software` folder. The Freedom E SDK environment contains all of the necessary source code, Makefiles, and utilities necessary to recompile the included tests as well as the ability to easily extend the testbench with additional tests. Freedom E SDK requires a suitable toolchain which can be downlaoded from: https://www.sifive.com/boards

For more information on the Freedom E SDK environment, please read the readme file located in the *freedom-e-sdk* directory.

10.1.1 Executing the Testbench

GNU Make is used to build the RTL into a simulator and run the included binary test files. The Make targets are described below:

- clean - Cleans the build
- all - Runs all tests on all simulators
- all-verilator - Runs all tests using verilator
- all-vcs - Runs all tests using vcs
- all-xrun - Runs all tests using xcelium
- all-waves - Runs all tests while dumping waveforms in VPD format from all simulators
- all-verilator-waves - Runs all tests using verilator while dumping wafeforms in VPD format
- all-vcs-waves - Runs all tests using vcs while dumping wafeforms in VPD format
- all-xrun-waves - Runs all tests using xcelium while dumping wafeforms in VPD format

- test.out - Runs the test called `test`
- test.vpd - Runs the test called `test` and produces a waveform in VPD format

Executing the command:

```
make all-waves
```

will run all of the tests in the `tests` folder and produce the resulting `<test_name>.out` and `<test_name>.vpd` files which can then be analyzed for detailed execution information.

10.2 Testbench Output

The testbench is capable of producing two types of output files: `.out` and `.vpd`. `.out` files contain a trace of all instructions executed by the processor. `.vpd` files contain VPD waveforms of the design which can be viewed with a waveform viewer such as Synopsys DVE.

10.2.1 Testbench Output - Trace

The test bench will produce output files with the filename `<test_name>.out`. The output files contain a cycle-by-cycle dump of a core's write-back stage. An example output is provided below:

```
Format:
core id: cycle [valid] pc=[address] Written[register=value][valid] Read[regis-
ter=value] Read[register=value]

Example:
C0: 483 [1] pc=[00000002138] W[r 3=000000007fff7fff][1] R[r 1=000000007fffffff] R[r
2=fffffffffff8000]
C0: 484 [1] pc=[0000000213c] W[r29=000000007fff8000][1] R[r31=ffffffff80007ffe]
R[r31=0000000000000005]
C0: 485 [0] pc=[00000002140] W[r 0=0000000000000000][0] R[r 0=0000000000000000] R[r
0=0000000000000000]
```

The first `[1]` at cycle 483, core 0, shows that there's a valid instruction at PC `0x2138` in the writeback stage. The second `[1]` tells us that the register file is writing r3 with the corresponding value `0x7fff7fff`. When the add instruction was in the decode stage, the pipeline had read r1 and r2 with the corresponding values next to it. Similarly at cycle 484, there's a valid instruction at PC `0x213c` in the writeback stage. At cycle 485, there isn't a valid instruction in the writeback stage, perhaps, because of a instruction cache miss at PC `0x2140`.

10.2.2 Testbench Output - Waves

When running the included testbench with the `all-waves` or `<test_name>.vpd` targets, the testbench will create VPD formatted waveforms which can be viewed with a waveform viewer such as Synopsys DVE. The waveform along with the trace log can be helpful when debugging tests run on the testbench.

10.2.3 Adding Tests To The Included Testbench

The simplest way to add new tests to the testbench is to start from one of the included tests. The return-pass test contains an empty main function which returns 0 (pass value) and is suitable for starting a new test. The example below demonstrates how to build a new test starting from return-pass.

- In the *freedom-e-sdk/software* directory, make a copy of the *return-pass* folder and name the copied folder to the name of your test. For this example we will use $TEST.
- In the $TEST directory, edit the makefile variable *PROGRAM* to match the name $TEST.
- In the $TEST directory, change the filename of return-pass.c to $TEST.c
- Use the Freedom E SDK makefile to build the test targeting the *RTL* BSP and the *release* Configuration. The name of your BSP can be found in the *freedom-e-sdk/bsp* directory.

```
make TARGET=deliverable-name-rtl PROGRAM=$TEST CONFIGURATION=release software
```

- In the base directory of the deliverable, it is now possible to run the new test using the testbench makefile.

```
make $TEST.out
```

For more information on using Freedom E SDK and its environment, please read the readme file located in the *freedom-e-sdk* directory.

10.3 SiFive Insight

E31 is enabled with SiFive Insight technology which provides deep visibility into the design while at the same time being easily accessible. SiFive Insight is a verilog module that contains a curated list of signals chosen by the designers and presented in an intuitive hierarchy with descriptive names. SiFive Insight is primarily meant to be used during simulation waveform debugging and allows for a deep understanding of what is happening inside the SiFive deliverable without knowing details of the design.

Note that some signals in SiFive Insight are pseudo-signals which represent several signals with logic applied to them in order to present a more useful higher-level function. For example, the `Instruction Commit` signal in a design may be the logical combination of several signals. SiFive Insight also manages the grouping of signals to improve readability. An example of this would be how SiFive Insight presents the `mstatus` CSR. SiFive Insight presents `mstatus` such that each field in the CSR is grouped together, improving readability directly from the waveform viewer.

A complete list of SiFive Insight signals, along with descriptions, can be found in the `info/sifive_insight.yml` file located in the deliverables.

10.3.1 Viewing SiFive Insight Signals

Follow the instructions in Section 10.1.1 to execute the testbench and generate the resulting VPD wave files. This can be done with either the `make all-waves` or `make <test_name>.vpd` Make targets.

Once the test completes, open the VPD wave file with a waveform viewer such as Synopsys DVE. The SiFive Insight module can be found under the Verilog module hierarchy `TestDriver/testHarness/system/SiFive_Insight` or by simply searching for `SiFive_Insight`.

From here it is possible to add the SiFive Insight signals to the waveform viewer.

10.3.2 Enabling SiFive Insight Outside of the SiFive Testbench

To enable SiFive Insight in testbenches other than the one provided with the SiFive deliverables, simply include the SiFive Insight Verilog files with the testbench and design under test during the compilation of the simulator. The SiFive Insight Verilog files are located in the `verilog/sifive_insight/` directory. Waveform dumps will automatically include the SiFive Insight signals.