logic fruit Technologies

WHITEPAPER

Enabling RISC-V Based System Development

This whitepaper focuses on providing a jump start on RISC-V development. It shows how to build a verification environment quickly involving a RISC-V core and required peripherals based on selected applications

- By

Sandeep Nasa &
Sagar Thakran

Enabling RISC-V Based System Development

VERIFYING A RISC-V CORE-BASED DESIGN: A PRIMER

This article focuses on providing a jump start on RISC-V development. It shows how to build a verification environment quickly involving a RISC-V core and required peripherals based on selected applications.

RISC-V is an open-source instruction set architecture (ISA) specification. It is a general-purpose ISA developed at U.C. Berkeley, which is designed for supporting a wide variety of applications, from micropower embedded devices to high-performance cloud server multi-processors, and is freely available for anyone to build a processor core compliant to its ISA. Because it is open-source, it is possible to customize the processor's core and still be compliant to the RISC-V ISA, which has led to a rapidly-growing ecosystem in the market based on application requirements. All these custom-based processor ecosystems are used for various applications by integrating required peripherals. Many companies/organizations have developed RISC-V cores for targeted applications and made them available for further enhancement via open source.

To make a working application, we'll need a verification environment to verify the intended functionality. A good verification environment flow is required to verify the targeted application and showcase performance for commercial needs. To fulfill the goal of creating a system with the core and peripherals based on an application, we need to enable the verification environment along with selected test cases that suits our needs. To achieve this, we need to select one of the standard cores along with its test suite. We have selected a 32-bit RISC-V core from Western Digital (named SweRV_EH1).

The rest of this article will explain the steps needed to enable a SystemVerilog-based verification environment on QuestaSim for the SweRV_EH1 RISC-V core. One test case is selected which is modified as per traffic application we are targeting here. At the end of this article, you will find a link to the source for this project so you may walk through the code on your own.

Areas of Focus:

1. Providing an example with all required steps to jump start RISC-V core-based application development.
2. Enabling System-Level Verification.
3. Providing one working test case for ready reference for further enhancement based on need.
4. Providing a Simplified Approach to Enable the Simulation Environment in QuestaSim.

INTRODUCTION

The emergence of the RISC-V ISA as an open-source platform is rapidly attracting interest in the industry, which has resulted in many companies building their own custom-based processors to meet the needs of their targeted applications. An ISA is the basic vocabulary that allows hardware and software to communicate. Since the same ISA is the target. It also helps to have a common benchmark to compare cores from different sources on various criteria, such as performance, and energy efficiency for a variety of applications.

RISC-V is designed with a small, fixed-base ISA. It includes modular fixed-standard extensions that can be used with most of the code. This architecture enables the development of application-specific extensions without needing to modify the standard ISA core. In addition to enabling customization, this approach is expected to prevent, or at least minimize, fragmentation of the RISC-V software ecosystem.

Designing an application using a RISC-V core will require some time to build a development environment. Details mentioned in this abstract will speed up the process to build a RISC-V core based system using the Western Digital core named SweRV_EH1.

SweRV EH1 CORE BRIEF DESCRIPTION

SweRV EH1 core by Western Digital is based on the RISC-V ISA architecture and targeted for high performance embedded applications. The micro architecture of this core is implemented on 28nm technology by TSMC and has amazing performance benchmarks.

SweRV EH1 core is open-sourced and available on GitHub as well (please see the link below). The SweRV EH1 has a core complex which consists of a microarchitecture core along with other components. The Core Complex Block Diagram is shown below:

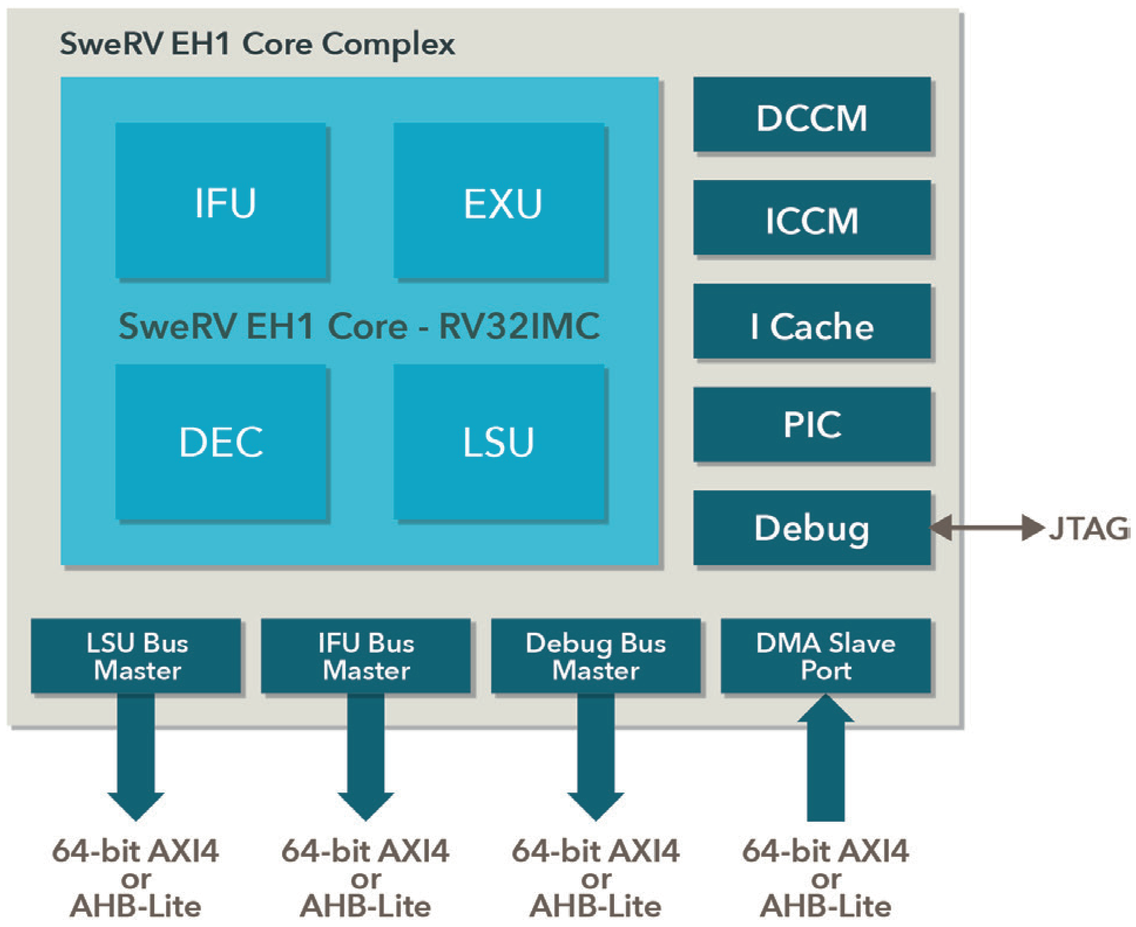

Figure 1: SwerRV EH1 Core Complex

The features of the SweRV EH1 Core complex are as follows:

a. The micro architecture core is an RV32IMCcompliant RISC-V core with branch predictor.

b. Optional instruction and data closely coupled memories with ECC protection.

c. Optional 4-way set-associative instruction cache with parity or ECC protection.

d. Optional programmable interrupt controller supporting up to 255 external interrupts.

e. Four system bus interfaces for instruction fetch, data accesses, debug accesses, and external DMA accesses to closely coupled memories (configurable as 64-bit AXI4 or AHB-Lite).

f. A core debug unit compliant with the RISC-V debug specification.

g. 1GHz target frequency (for 28nm technology node).

Architectural details of the SweRV EH1 micro architecture core is explained below:

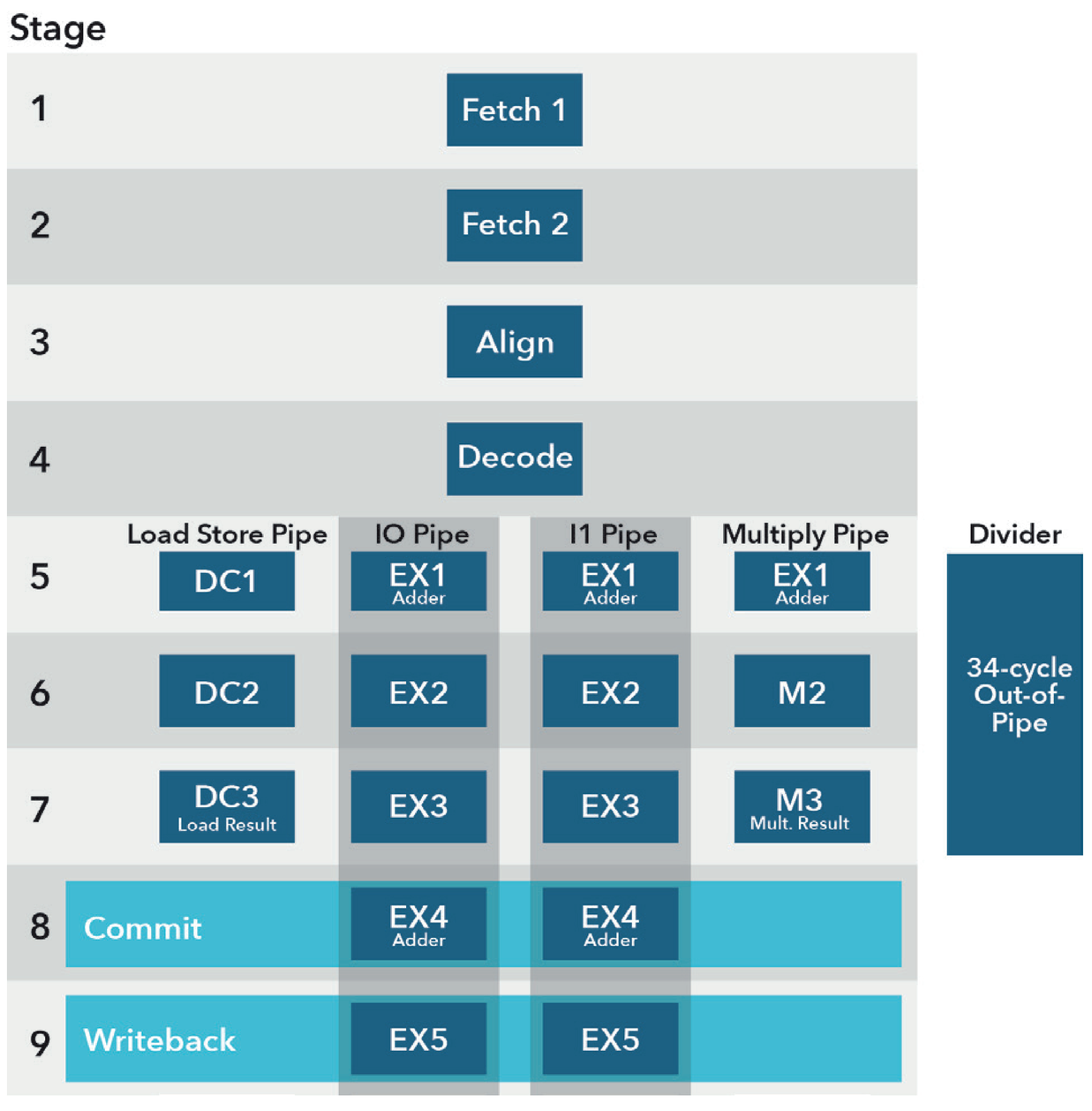

Figure 2: SwerRV EH1 Core Pipeline

- The core has superscalar architecture, with dual issue 9-stage pipeline supporting 4 arithmetic logic units (ALU) labeled as EX1 to EX4, each in two pipelines I0 and I1.
- There are 9 stages present in the pipeline including writeback, and 4 stall points: Fetch1, Align, Decode, and Commit.
- The Fetch unit has 2 stages. Align will form instructions from the 3 fetch buffers. Decode will decode up to 2 instructions from 4 fetch buffers. Commit will commit up to 2 instructions per cycle, based on the workload.
- It includes one load/store pipeline, one multiplier pipeline and one 34-cycle out-of pipeline divider unit.
- Compared to previous open-source RISC-V cores such as Rocket or Pulpino, SweRV uses a superscalar dual-issue micro architecture which results in improving the various performance benchmarks by 20-30%, at a relatively small expense of core gate count or implementation area.

STEPS/SEQUENCE TO BUILD ENVIRONMENT WITH RISC-V CORE BASED APPLICATION

In this section we review the steps to execute one test case for the traffic application, which can be enhanced for any other application as well. If you encounter any issues in recreating these steps in your environment, please refer to the "Issues and Resolution" section of this article.

The steps are listed below:

Replicate SweRV Core Based Application Environment and execute default test case [using QuestaSim]:

To start working on the RISC-V core based application development, the first step is to enable the database in the local machine by running the default test case. Follow the below steps to do the same:

a. Download the RISC-V core repository from Github. (see reference 1)

b. Set the following environment variables:

1. RV_ROOT = <Repository_Path> / Cores-SweRV-master
2. BUILD_PATH = <Sim_Output_Path>

c. Check the default configuration of the SweRV-core, and if necessary change it based on the application requirements.

d. Check which parameters are configured by default:

$RV_ROOT/configs/swerv.config -h

e. Once done with configuration, we need to modify the Makefile slightly to run the example in QuestaSim:

Change line "all: clean verilator" to "all: clean vlog"

f. Execute the default test case to print "Hello World" in the console window:

make -f $RV_ROOT/tools/Makefile

Salient Features of the Environment

There are a few features of the SweRV Environment that will need to be modified to implement our traffic controller application.

a. There are AXI and AHB peripherals present in the environment. AXI is selected by default and AHB can be made default by using switch *-target=default_ahb* in make command.

b. We can select environment user options and parameters based on application requirements. Some important ones are as follows.

User options:

-target = { default, default_ahb, default_pd, high_perf}

Parameters settings:

-ahb_lite

build with AHB-lite bus interface.

default is AXI4

-pic_total_int = { 1, 2, 3, ..., 255 }

number of interrupt sources in PIC

c. The address used to write into AHB peripheral is 0xD0580000.

Traffic Light Application details

Below you will find the block diagram of the Traffic Light Application with description:

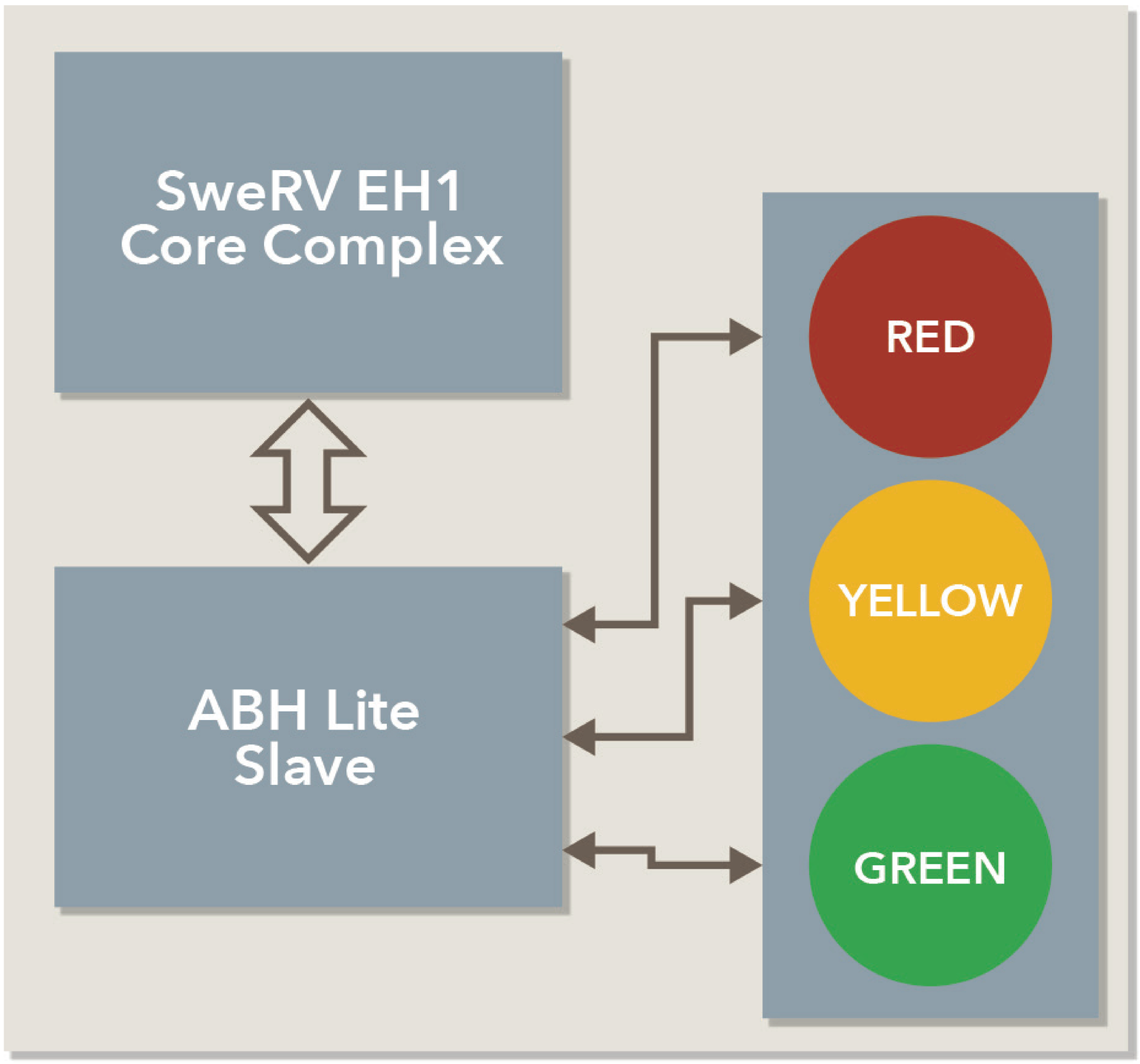

Figure 3: Basic Block Diagram of a Traffic Light Application

a. This is a basic block diagram of an application that controls a traffic light for smooth traffic management. Here we are configuring the time duration for each light to turn on along with the sequence in which the different colored lights are turned on.

b. We are using 3 colors here so 3 byte registers will be used for time duration and two byte registers for sequence control.

Execution of Test Case for Traffic Application

The execution of the test cases is explained in following steps:

a. First, we need to install the RISC-V based GCC supported toolchain. We need to set an environment variable to give a path for this tool so that the Makefile can get these compiled libraries. We have used ***SysGCC*** tool chain for this [see number 2 in the references].

b. The Environment variable to be set fortool chain is:

RISCV=C:\SysGCC\risc-v\bin

c. Once done with the toolchain , we need to select the AHB interface as the default connection. To enable this, we need to use the option *-target=default_ahb* in the makefile command.

d. Modifications done in *hello_word.s* file There are some modifications required in the default assembly language test as shown below. Also the file name is renamed to traffic_app.s

Original File:

```
    // Load string from hw_data
    // and write to stdout address

    li x3, STDOUT
    la x4, hw_data

loop:

    lb x5, 0(x4)

    sb x5, 0(x3)
    addi x4, x4, 1
    bnez x5, loop
```

Updated File:

Lines shown in original snapshot are commented and some new lines are added as shown in the snapshot below.

```
    // Load data into t1
    // and write to stdout address

    li x3, STDOUT # Load register t0 with an address
    li t1, 0x3c #RED light time
    sb t1, 0(x3)
        li t1, 0x1e #YELLOW light time
    sb t1, 0(x3)
        li t1, 0x12 #GREEN light time
    sb t1, 0(x3)
        li t1, 0x200 #CTRL SEQUENCE  1byte
    sb t1, 0(x3)

//   loop:
// lb x5, 0(x4)
//  sb x5, 0(x3)
//  addi x4, x4, 1
//  bnez x5, loop
```

e. Modifications done in tb_top.sv file. In this file address and data are captured and printed to showcase that the data written in the testcase reaches the AHB slave. The portion of code used to print to the console is used for this. We get output on the terminal as well as in console.log file.

Original File:

```
cansol Monitor
 mailbox_data_val & mailbox_write) begin
 $fwrite(fd,"%c", WriteData[7:0]);
 $write("%c", WriteData[7:0]);
```

Updated File:

The lines shown in original file are commented and 2 extra printing lines are added below to represent that the info sent by the testcase reaches the ahb-lite slave.

```
// cansol Monitor
if( mailbox_data_val & mailbox_write) begin
   // $fwrite(fd,"%c", WriteData[7:0]);
   // $write("%c", WriteData[7:0]);
   $fwrite(fd,"\tAddress->%0h\n\t Data\t->%0h\n",WriteAddr,WriteData[31:0]);
   $write("\tAddress->%0h\n\tData\t->%0h\n",WriteAddr,WriteData[31:0]);
```

f. Once done with the configuration part, run the makefile command:

make -f $RV_ROOT/tools/Makefile -target=default_ahb

g. The steps followed inside the Makefile are shown in the flowchart for easy understanding:

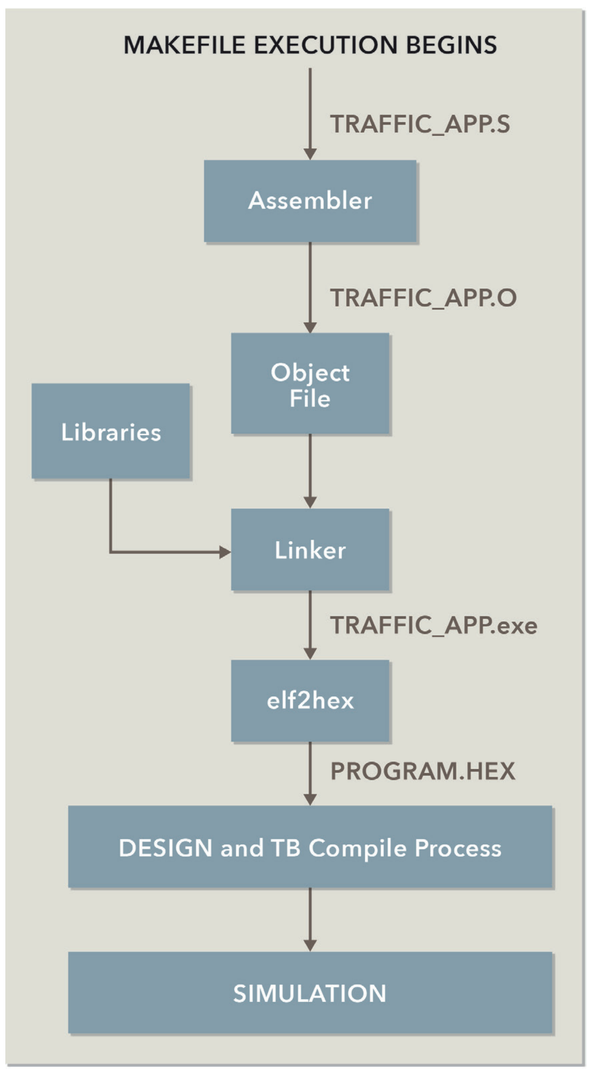

Figure 4: Makefile Steps

In the flow chart first the traffic_app.s file is taken as input and the assembler will convert it to the traffic_app.o object file. This object file is linked to the standard riscv libraries by the linker and the executable file named traffic_app.exe is generated.

After this compilation is done, simulation finally proceeds.

The output captured in AHB-LITE Slave is shown below which shows that data sent to ahb slave:

Address	d0580000
Data	3f
Address	d0580000
Data	22
Address	d0580000
Data	40
Address	d0580000
Data	41

Figure 5: Data Sent to the ABH Slave

ISSUES AND RESOLUTIONS

We have highlighted below a few issues we faced during the bring up of the RISC-V based application development, including an issue faced in PATH set to RV_ROOT in cygwin terminal.

Issue: we were setting a path with “/” in cygwin with a path starting from /cygdrive/d/RISC-V/Cores-SweRV-master which is not taken properly by Makefile.

Root Cause: Internally Makefile is using path of os prompt so it expects path in dos style.

Resolution: Give PATH in cygwin style using "/" and the path is starting with D:/RISC-V/Cores-SweRV-master so that DOS path is available to Makefile. After this change execution started working.

CONCLUSION AND FUTURE ENHANCEMENT

The RISC-V based application development flow has numerous possibilities for future enhancements such as:

a. The current example can be used as a starting point for any other similar application. We showed how to add one address, but more addresses can be added for multiple peripherals using a similar scheme.

b. The algorithm used can also be enhanced for any complex application.

c. Peripheral code can be attached after the AHB slave depending on the application requirement.

d. Enhancements can be done by changing the configuration. Some examples are provided by Western Digital.

e. Users can move on to the advanced version of the Existing core , as Western Digital also has EH2 Core also which is a better version of the previous one.

f. Here we used assembly language but the same infrastructure can be used to run c-based code if desired.

ENVIRONMENT STRUCTURE

The Environment Structure for understanding the hierarchy is shown below:

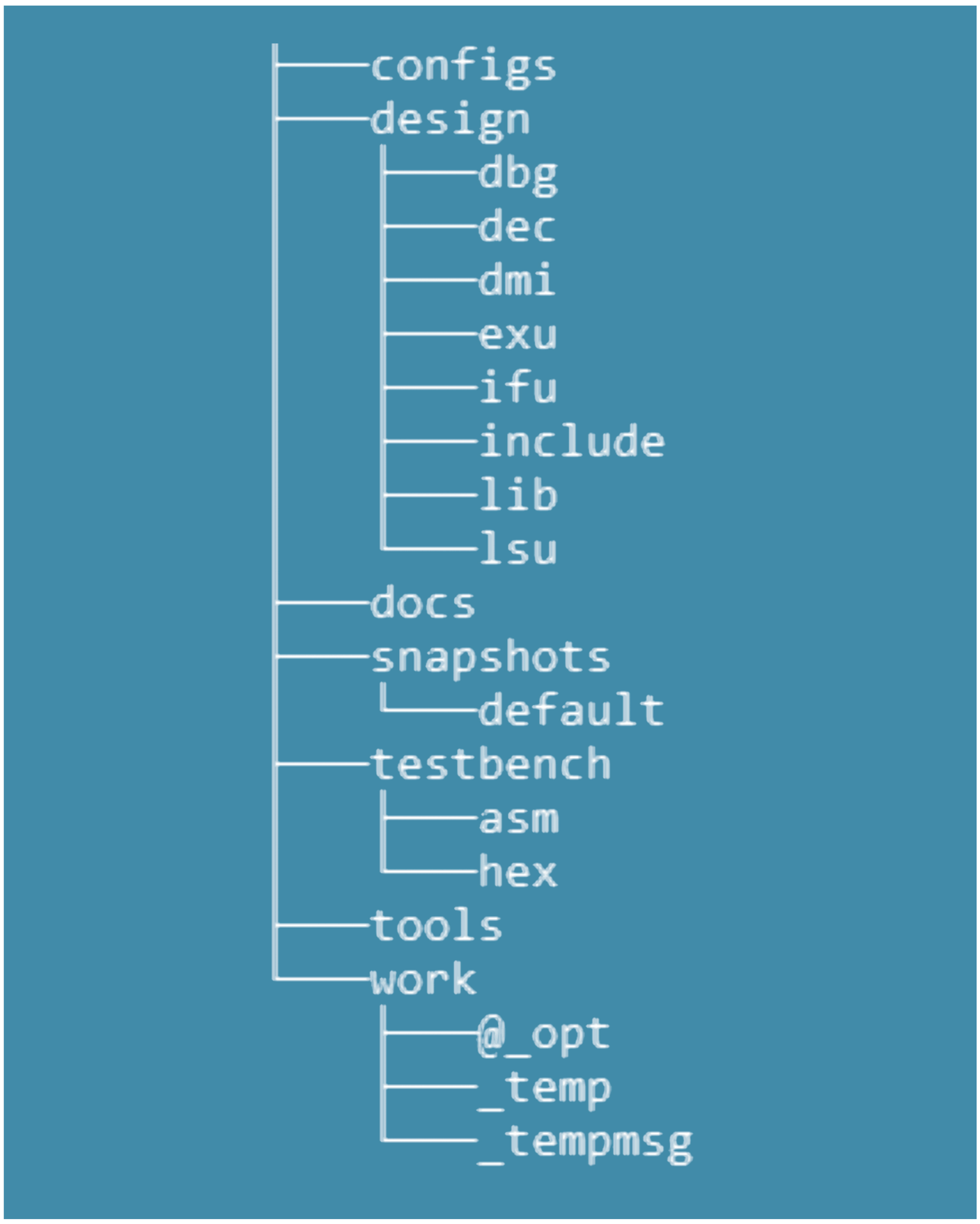

Figure 6: Environment Structure

1) Configs: Configuration Directory consists of a configuration script for SweRV.
This script will generate a consistent set of `defines needed for the design and testbench.

2) Design: Design Root directory, which contains all the design description files of the SweRV EH1 core.

This directory has the following subdirectories which represent different blocks of the core having design description files written in SystemVerilog:

IFU: Instruction fetch Unit

EXU: Execution Unit

DEC: Decoders and Registers

LSU: Load and Store Unit

DBG: Debugger Unit

3) Docs: Docs Directory consists of Reference Documents for understanding the technical specifications of the SweRV_EH1_Core.
This Directory also contains a Programmer's Reference Manual which enables the user to understand the register description and commands to start writing the assembly language program.

4) Snapshots: Output Directory formed where generated configuration files are created using a configuration script.

5) Testbench: Testbench directory consists of testbench files such as tb_top and interface files along with two sub directories named as asm and hex.

asm: A subdirectory under Testbench, where users create application specific assembly files for test case execution.

hex: A subdirectory under Testbench, where ready-made hex files are present for default test case execution if no RISC-V software is installed.

6) Tools: Tools Directory consists of tool specific scripts [for QuestaSim, Verilator and other tools] and the Makefile utility for test cases execution.

7) Work: Work directory consists of a compiled snapshot of the design and testbench.

Users can load the work directory to see the design hierarchy and perform the simulations using the QuestaSim tool.

REFERENCE

1. SweRV Repository: *https://github.com/chipsalliance/Cores-SweRV*
2. GCC 10.1.0 for RISC-V development: *https://gnutoolchains.com/risc-v/*

logic fruit Technologies

Thank You!

Does anyone have any questions?

Contact Us

806, 8th Floor BPTP Park Centra
Sector–30, NH–8 Gurgaon – 122001
Haryana (India)

+91-0124 4643950

info@logic-fruit.com

- By

Sandeep Nasa & Sagar Thakran