

**THE IMAGINATION UNIVERSITY PROGRAMME**

**RVfpga-SoC Lab 5**

**Tensorflow LITE**

# Introduction :

In this Lab, we show how to build a Tensorflow LITE project (hello\_world) for Zephyr that is a real-time operating system (RTOS), and then run on SweRVolf.

# Requirements :

To complete this lab, you will need to install the following:

* Verilator (v4.106) (Refer to Installation Guide (Page No.13))
* FuseSoC (Refer to Installation Guide (Page No.14))
* OpenOCD (RISC-V-specific version) (Refer to Installation Guide (Page No.14))
* Zephyr Prerequisites (Refer to Installation Guide (Page No.15))
* Zephyr SDK (v0.12.4) (Refer to Installation Guide (Page No.15))
* PuTTY

**IMPORTANT:** Before starting RVfpga-SoC Labs, we highly recommend completing the RVfpga-SoC Installation Guide.

For example, if you have not already, install Xilinx’s Vivado and Verilator following the instructions in the RVfpga-SoC Installation Guide. Make sure that you have copied the RVfpga-SoC folder that you downloaded from Imagination’s University Programme to your machine.

# Tensorflow-Lite Overview:

TensorFlow Lite is a set of tools that enables on-device machine learning by helping developers run their models on mobile, embedded, and IoT devices.

Its key features are:

* Optimized for on-device machine learning, by addressing 5 key constraints: latency (there's no round-trip to a server), privacy (no personal data leaves the device), connectivity (internet connectivity is not required), size (reduced model and binary size), and power consumption (efficient inference and a lack of network connections).
* Multiple platform support, covering Android and iOS devices, embedded Linux, and microcontrollers.
* Diverse language support, which includes Java, Swift, Objective-C, C++, and Python.
* High performance, with hardware acceleration and model optimization.
* End-to-end examples, for common machine learning tasks such as image classification, object detection, pose estimation, question answering, text classification, etc. on multiple platforms.

# Tensorflow’s Hello World Example :

The Hello World example is designed to demonstrate the absolute basics of using TensorFlow Lite for Microcontrollers. We train and run a model that replicates a sine function, i.e, it takes a single number as its input and outputs the number's sine value. When deployed to the microcontroller, its predictions are used to either blink LEDs or control an animation.

The end-to-end workflow involves the following steps:

Train a model (in Python): A jupyter notebook to train, convert and optimize a model for on-device use.

Run inference (in C++ 11): An end-to-end unit test that runs inference on the model using the C++ library.

The following sections walk through the example's [hello\_world\_test.cc](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/examples/hello_world/hello_world_test.cc), a unit test that demonstrates how to run inference using TensorFlow Lite for Microcontrollers. It loads the model and runs inference several times.

### **1. Include the library headers**

To use the TensorFlow Lite for Microcontrollers library, we must include the following header files:

|  |
| --- |
| #include "tensorflow/lite/micro/all\_ops\_resolver.h"  #include "tensorflow/lite/micro/micro\_error\_reporter.h"  #include "tensorflow/lite/micro/micro\_interpreter.h"  #include "tensorflow/lite/schema/schema\_generated.h"  #include "tensorflow/lite/version.h" |

* [all\_ops\_resolver.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/all_ops_resolver.h) provides the operations used by the interpreter to run the model.
* [micro\_error\_reporter.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/micro_error_reporter.h) outputs debug information.
* [micro\_interpreter.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/micro_interpreter.h) contains code to load and run models.
* [schema\_generated.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/schema/schema_generated.h) contains the schema for the TensorFlow Lite [FlatBuffer](https://google.github.io/flatbuffers/) model file format.
* [version.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/version.h) provides versioning information for the TensorFlow Lite schema.

### **2. Include the model header**

The TensorFlow Lite for Microcontrollers interpreter expects the model to be provided as a C++ array. The model is defined in model.h and model.cc files. The header is included with the following line:

|  |
| --- |
| #include "tensorflow/lite/micro/examples/hello\_world/model.h" |

### **3. Include the unit test framework header**

In order to create a unit test, we include the TensorFlow Lite for Microcontrollers unit test framework by including the following line:

|  |
| --- |
| #include "tensorflow/lite/micro/testing/micro\_test.h" |

The test is defined using the following macros:

|  |
| --- |
| TF\_LITE\_MICRO\_TESTS\_BEGIN  TF\_LITE\_MICRO\_TEST(LoadModelAndPerformInference) {  . // add code here  .  }  TF\_LITE\_MICRO\_TESTS\_END |

We now discuss the code included in the macro above.

### **4. Set up logging**

To set up logging, a tflite::ErrorReporter pointer is created using a pointer to a tflite::MicroErrorReporter instance:

|  |
| --- |
| tflite::MicroErrorReporter micro\_error\_reporter;  tflite::ErrorReporter\* error\_reporter = &micro\_error\_reporter; |

This variable will be passed into the interpreter, which allows it to write logs. Since microcontrollers often have a variety of mechanisms for logging, the implementation of tflite::MicroErrorReporter is designed to be customized for your particular device.

### **5. Load a model**

In the following code, the model is instantiated using data from a char array, g\_model, which is declared in model.h. We then check the model to ensure its schema version is compatible with the version we are using:

|  |
| --- |
| const tflite::Model\* model = ::tflite::GetModel(g\_model);  if (model->version() != TFLITE\_SCHEMA\_VERSION) {  TF\_LITE\_REPORT\_ERROR(error\_reporter,  "Model provided is schema version %d not equal "  "to supported version %d.\n",  model->version(), TFLITE\_SCHEMA\_VERSION);  } |

### **6. Instantiate operations resolver**

An [AllOpsResolver](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/all_ops_resolver.h) instance is declared. This will be used by the interpreter to access the operations that are used by the model:

|  |
| --- |
| tflite::AllOpsResolver resolver; |

The AllOpsResolver loads all of the operations available in TensorFlow Lite for Microcontrollers, which uses a lot of memory. Since a given model will only use a subset of these operations, it's recommended that real-world applications load only the operations that are needed.

This is done using a different class, MicroMutableOpResolver. You can see how to use it in the *Micro speech* example's [micro\_speech\_test.cc](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/examples/micro_speech/micro_speech_test.cc).

### **7. Allocate memory**

We need to preallocate a certain amount of memory for input, output, and intermediate arrays. This is provided as a uint8\_t array of size tensor\_arena\_size:

|  |
| --- |
| const int tensor\_arena\_size = 2 \* 1024;  uint8\_t tensor\_arena[tensor\_arena\_size]; |

The size required will depend on the model you are using and may need to be determined by experimentation.

### **8. Instantiate interpreter**

We create a tflite::MicroInterpreter instance, passing in the variables created earlier:

|  |
| --- |
| tflite::MicroInterpreter interpreter(model, resolver, tensor\_arena,  tensor\_arena\_size, error\_reporter); |

### **9. Allocate tensors**

We tell the interpreter to allocate memory from the tensor\_arena for the model's tensors:

|  |
| --- |
| interpreter.AllocateTensors(); |

### **10. Validate input shape**

The MicroInterpreter instance can provide us with a pointer to the model's input tensor by calling .input(0), where 0 represents the first (and only) input tensor:

|  |
| --- |
| // Obtain a pointer to the model's input tensor  TfLiteTensor\* input = interpreter.input(0); |

We then inspect this tensor to confirm that its shape and type are what we are expecting:

|  |
| --- |
| // Make sure the input has the properties we expect  TF\_LITE\_MICRO\_EXPECT\_NE(nullptr, input);  // The property "dims" tells us the tensor's shape. It has one element for  // each dimension. Our input is a 2D tensor containing 1 element, so "dims"  // should have size 2.  TF\_LITE\_MICRO\_EXPECT\_EQ(2, input->dims->size);  // The value of each element gives the length of the corresponding tensor.  // We should expect two single element tensors (one is contained within the  // other).  TF\_LITE\_MICRO\_EXPECT\_EQ(1, input->dims->data[0]);  TF\_LITE\_MICRO\_EXPECT\_EQ(1, input->dims->data[1]);  // The input is a 32 bit floating point value  TF\_LITE\_MICRO\_EXPECT\_EQ(kTfLiteFloat32, input->type); |

The enum value kTfLiteFloat32 is a reference to one of the TensorFlow Lite data types, and is defined in [common.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/c/common.h).

### **11. Provide an input value**

To provide input to the model, we set the contents of the input tensor, as follows:

|  |
| --- |
| input->data.f[0] = 0.; |

In this case, we input a floating-point value representing 0.

### **12. Run the model**

To run the model, we can call Invoke() on our tflite::MicroInterpreter instance:

|  |
| --- |
| TfLiteStatus invoke\_status = interpreter.Invoke();  if (invoke\_status != kTfLiteOk) {  TF\_LITE\_REPORT\_ERROR(error\_reporter, "Invoke failed\n");  } |

We can check the return value, a TfLiteStatus, to determine if the run was successful. The possible values of TfLiteStatus, defined in [common.h](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/c/common.h), are kTfLiteOk and kTfLiteError.

The following code asserts that the value is kTfLiteOk, meaning inference was successfully run.

|  |
| --- |
| TF\_LITE\_MICRO\_EXPECT\_EQ(kTfLiteOk, invoke\_status); |

### 

### **13. Obtain the output**

The model's output tensor can be obtained by calling output(0) on the tflite::MicroInterpreter, where 0 represents the first (and only) output tensor.

In the example, the model's output is a single floating point value contained within a 2D tensor:

|  |
| --- |
| TfLiteTensor\* output = interpreter.output(0);  TF\_LITE\_MICRO\_EXPECT\_EQ(2, output->dims->size);  TF\_LITE\_MICRO\_EXPECT\_EQ(1, input->dims->data[0]);  TF\_LITE\_MICRO\_EXPECT\_EQ(1, input->dims->data[1]);  TF\_LITE\_MICRO\_EXPECT\_EQ(kTfLiteFloat32, output->type); |

We can read the value directly from the output tensor and assert that it is what we expect:

|  |
| --- |
| // Obtain the output value from the tensor  float value = output->data.f[0];  // Check that the output value is within 0.05 of the expected value  TF\_LITE\_MICRO\_EXPECT\_NEAR(0., value, 0.05); |

### **14. Run inference again**

The remainder of the code runs inference several more times. In each instance, we assign a value to the input tensor, invoke the interpreter, and read the result from the output tensor:

|  |
| --- |
| input->data.f[0] = 1.;  interpreter.Invoke();  value = output->data.f[0];  TF\_LITE\_MICRO\_EXPECT\_NEAR(0.841, value, 0.05);  input->data.f[0] = 3.;  interpreter.Invoke();  value = output->data.f[0];  TF\_LITE\_MICRO\_EXPECT\_NEAR(0.141, value, 0.05);  input->data.f[0] = 5.;  interpreter.Invoke();  value = output->data.f[0];  TF\_LITE\_MICRO\_EXPECT\_NEAR(-0.959, value, 0.05); |

### **15. Read the application code**

Once you have walked through this unit test, you should be able to understand the example's application code, located in [main\_functions.cc](https://github.com/tensorflow/tensorflow/blob/master/tensorflow/lite/micro/examples/hello_world/main_functions.cc). It follows a similar process, but generates an input value based on how many inferences have been run, and calls a device-specific function that displays the model's output to the user.

# Setting up The Environment For Tensorflow:

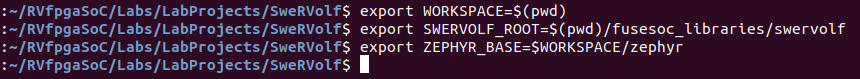
Open your Ubuntu terminal and complete the following steps :

**Step 1.** Navigate to the directory “**SweRVolf**”. We have to set the following shell variables. To do that, we run the following:

**$** export WORKSPACE=$(pwd)

**$** export SWERVOLF\_ROOT=$WORKSPACE/fusesoc\_libraries/swervolf

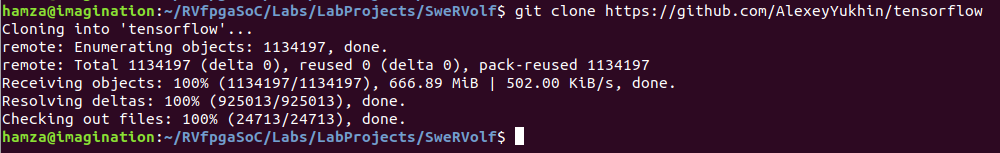
**$** export ZEPHYR\_BASE=$WORKSPACE/zephyr



**Figure 1. Set the shell variables**

**Step 2.** Clone the Tensorflow GitHub repository.

**$** git clone https://github.com/AlexeyYukhin/tensorflow

****

**Figure 2. Tensorflow**

Now navigate to the “tensorflow” directory.

**$** cd tensorflow



**Figure 3. Navigate to the “tensorflow” directory**

Checkout the specific branch of the repository by the following command :

**$** git checkout -b adc570a50410be7aba1c33522854f45fa0f349be



**Figure 4. git checkout**

**Step 3.** Navigate to the “zephyr” directory to install the required packages.

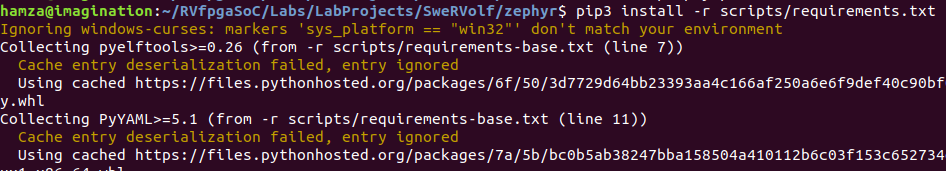
**$** cd ../zephyr/



**Figure 5. Navigate to the “zephyr” directory**

Install the required packages listed in the “scripts“ or requirements.txt” file using the following command.

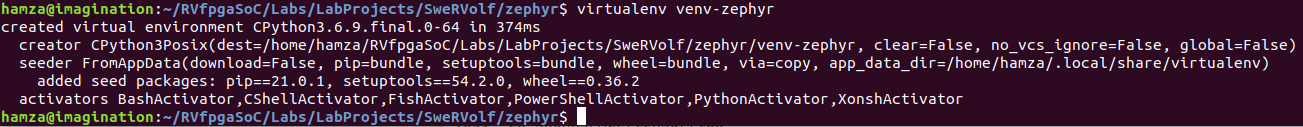
**$** pip3 install -r scripts/requirements.txt



**Figure 6. Installing required packages**

**Step 4.** Create a virtual environment using the following command.

**$** virtualenv venv-zephyr



**Figure 7. Creating venv-zephyr**

**Step 5.** Open a new terminal tab using “Ctrl + Shift + T”. Enter the following command in the new terminal tab to activate the virtual environment that we have created in the last step.

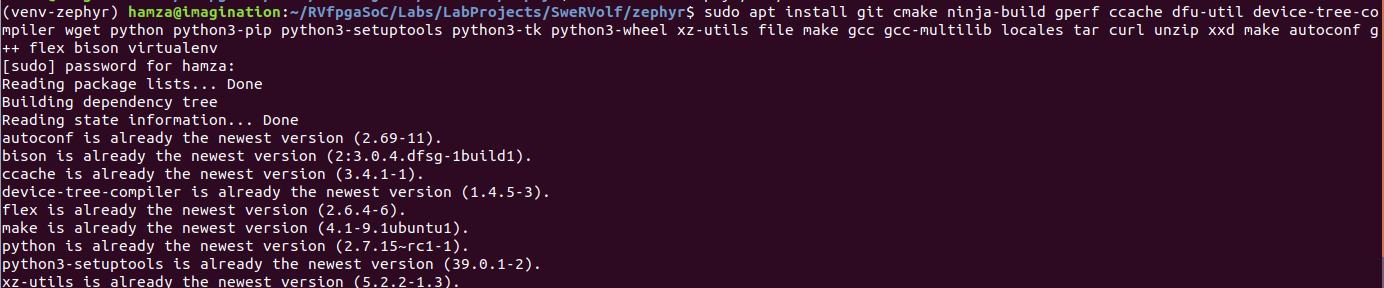
**$** source venv-zephyr/bin/activate



**Figure 8. Activating venv-zephyr**

**Step 6.** Install the packages in the virtual environment.

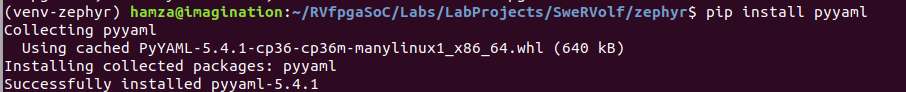
**$** sudo apt install git cmake ninja-build gperf ccache dfu-util device-tree-compiler wget python python3-pip python3-setuptools python3-tk python3-wheel xz-utils file make gcc gcc-multilib locales tar curl unzip xxd make autoconf g++ flex bison virtualenv



**Figure 9. Installing packages in venv-zephyr**

Install the “pyyaml” package using the following command:

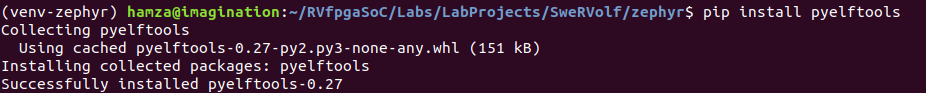
**$** pip install pyyaml



**Figure 10. Installing pyyaml in venv-zephyr**

also, install the “pyelftools” package using the following command:

**$** pip install pyelftools



**Figure 11. Installing pyelftools in venv-zephyr**

Now we can close this terminal tab and return to our main terminal tab where we will be building the “hello\_world” example.

# Building Hello World Example for Swervolf:

In this section, we will be building the “hello\_world” example for Swervolf. We will be generating the “**zephyr.bin**” and “**zephyr.elf**” files for the “hello\_world” example.

First, we will navigate to the tensorflow directory

**$** cd ../tensorflow/



**Figure 12. Navigating to the “tensorflow” directory**

To build a runnable binary for a given project (such as an example application), we can use the following command, replacing <project\_name> with the project we wish to build:

**$** make -f tensorflow/lite/micro/tools/make/Makefile <project\_name>\_bin

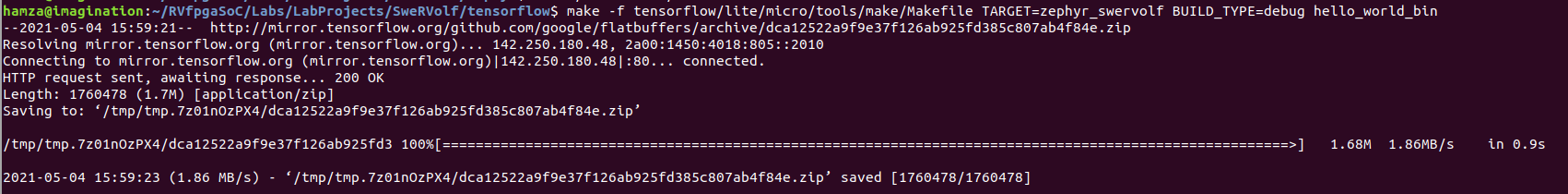
As we are building binaries for the “hello\_world” project so we can replace the <project\_name> with “hello\_world”.

**$** make -f tensorflow/lite/micro/tools/make/Makefile hello\_world\_bin

We will target it for Swervolf using the parameter “TARGET”.

Now run the following command :

**$** make -f tensorflow/lite/micro/tools/make/Makefile TARGET=zephyr\_swervolf BUILD\_TYPE=debug hello\_world\_bin

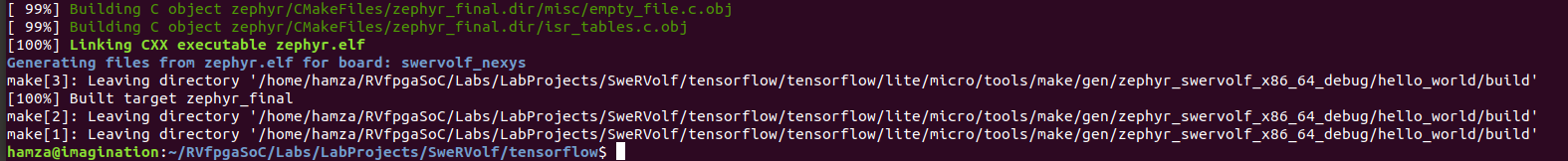


**Figure 13. Building hello\_world example**

This will take a few minutes since it has to download some toolchains for the dependencies. Once it has finished, you should see some folders created inside a path like

*tensorflow/lite/micro/tools/make/gen/zephyr\_swervolf\_x86\_64\_debug/hello\_world/*

These folders contain the generated project and source files.



**Figure 14. Building hello\_world example completed**

The resulting binaries (zephyr.bin and zephyr.elf) will be generated in the following path:

*tensorflow/lite/micro/tools/make/gen/zephyr\_swervolf\_x86\_64\_debug/hello\_world*

*/build/zephyr*

# Running Hello World Example on Verilator:

In this section, we will be converting the “zephyr.bin” file into a “.hex” file and then load it in as the initial ram file while running the simulator for swervolf.

**Step 1.** Navigate to the “hello\_world” project directory. Enter the following command to enter that directory:

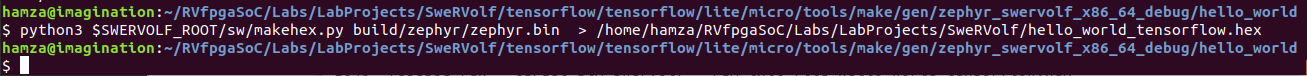
**$** cd tensorflow/lite/micro/tools/make/gen/zephyr\_swervolf\_x86\_64\_debug/hello\_world/



**Figure 15. “hello\_world” project path**

**Step 2.** Convert the “**.bin**” file to the “**.hex**” file. To create the “**.hex**” file, run the following command from the hello\_world directory :

**$** python3 $SWERVOLF\_ROOT/sw/makehex.py build/zephyr/zephyr.bin > /home/{YourUsername}/RVfpgaSoC/Labs/LabProjects/SweRVolf/hello\_world\_tensorflow.hex



**Figure 16. convert “.bin” to “.hex”**

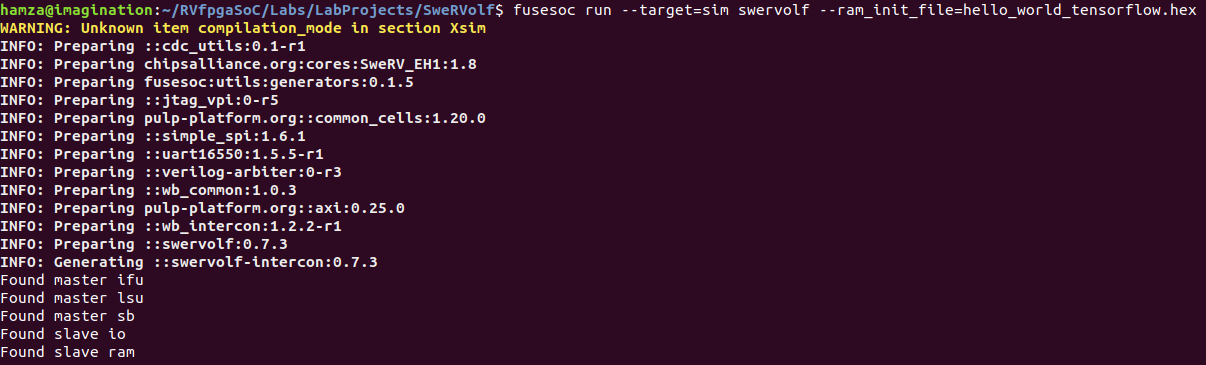
**Step 3.** Navigate back to the “WORKSPACE” directory



**Figure 9. Installing packages in venv-zephyr**

**Step 4.** Load the “**.hex**” file in the simulator :

**$** fusesoc run --target=sim swervolf --ram\_init\_file=hello\_world\_tensorflow.hex



**Figure 17. Loading “.hex” file in the simulator**

We can see the output of the hello\_world example (See Figure 18).

The program prints the “X” and “Y” coordinates of the sine function that the TensorFlow model is plotting.



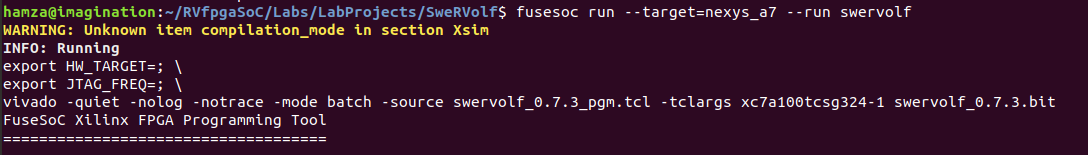
**Figure 18. “hello\_world” output**

# Running Hello World Example on the Nexys A7 Board:

In this section, we will be running the “hello\_world” project on the board using OpenOCD.

**Step 1**. Connect the Nexys A7 board to your computer and then run the FPGA build command in the Workspace directory.

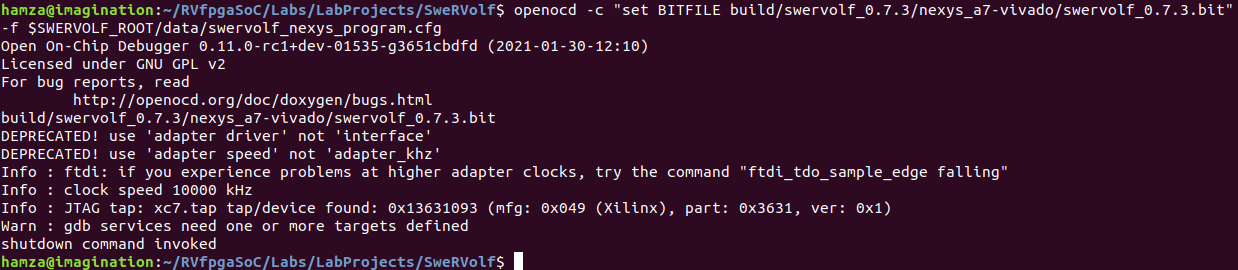
**$** fusesoc run --target=nexys\_a7 --run swervolf



**Figure 19. Run the FPGA build**

**Step 2.** Program the board with OpenOCD.

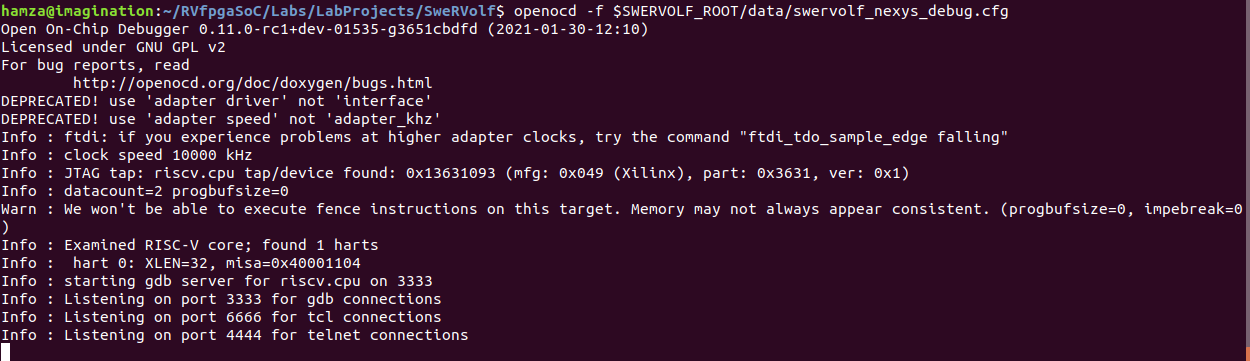
**$** openocd -c "set BITFILE build/swervolf\_0.7.3/nexys\_a7-vivado/swervolf\_0.7.3.bit" -f $SWERVOLF\_ROOT/data/swervolf\_nexys\_program.cfg



**Figure 20. Run OpenOCD**

**Step 3.** Connect OpenOCD with SweRVolf.

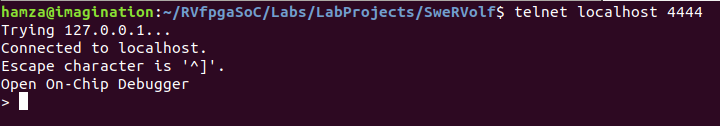
**$** openocd -f $SWERVOLF\_ROOT/data/swervolf\_nexys\_debug.cfg



**Figure 21. OpenOCD connected**

**Step 3**. Open a new terminal using “Ctrl + Shift + t” & connect to the debug session through OpenOCD using the following command:

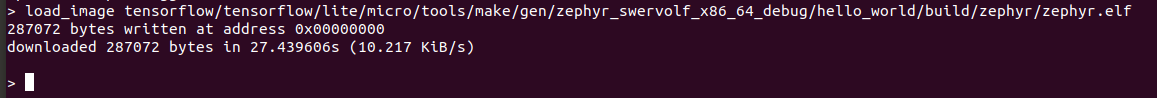
**$** telnet localhost 4444



**Figure 22. telnet localhost 4444**

OpenOCD supports loading ELF program files by running *load\_image /path/to/file.elf*. Remember that the path is relative to the directory from where OpenOCD was launched.

> load\_image tensorflow/tensorflow/lite/micro/tools/make/gen/zephyr\_swervolf\_x86\_64\_debug/hello\_world/build/zephyr/zephyr.elf



**Figure 23. loading the “.elf” file**

After the program has been loaded, set the program counter to address zero using the following command:

**>** reg pc 0



**Figure 24. Set program counter to zero**

Now start the program using this command:

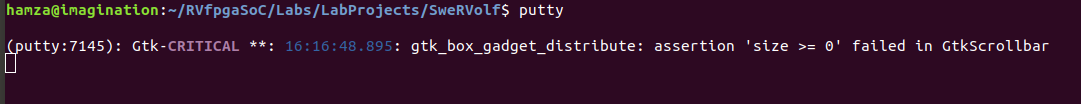
**>** resume



**Figure 25. Start the program**

**Step 4**. Open a new terminal using “Ctrl + Shift + t”. Open “PuTTY” using the command

**$** putty



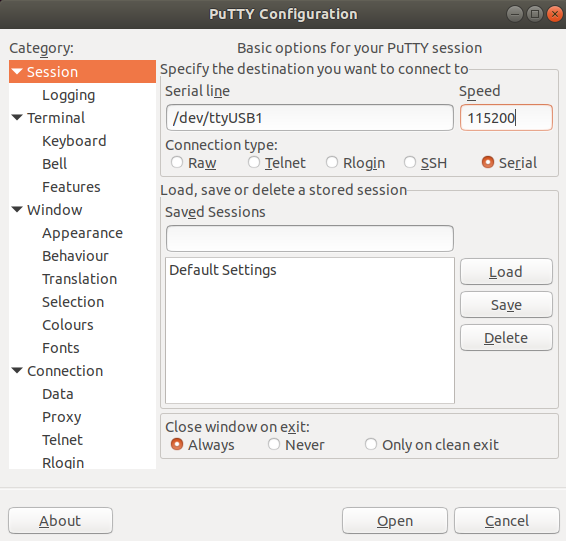
**Figure 26. Open PuTTY**

PuTTY is a free and open-source terminal emulator, serial console, and network file transfer application. We will be using it here as a serial console for our Nexys A7 board.

**Step 5**. Set the following configuration:

Select the connection type as “**Serial**”, then enter “**/dev/ttyUSB1**” as the serial line, and set the speed equal to “**115200**”.

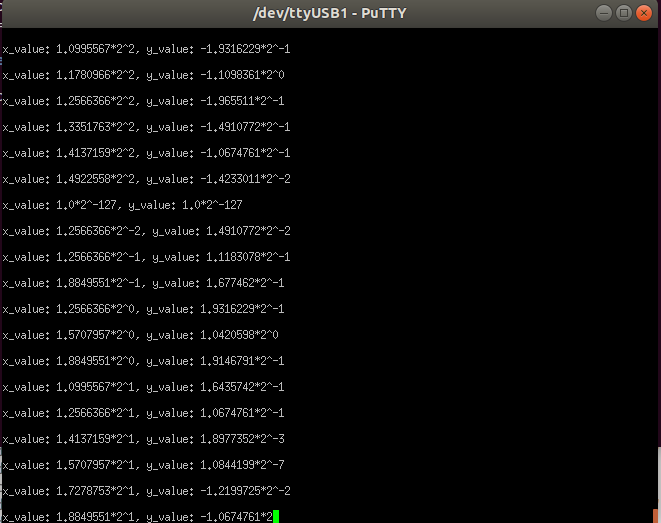
Now click “Open” to start the serial console.



**Figure 27. PuTTY configuration**

In the serial console, we can see the output of the hello\_world example (See Figure 28).

Again as we saw in the simulation section, the program prints the “X” and “Y” coordinates of the sine function that the TensorFlow model is plotting.



**Figure 28. serial console**