TinyML - Machine Learning on Edge-Devices

Dual Degree Project Phase - I

Submitted in partial fulfillment of the requirements
for the degree of

Bachelor and Master of Technology

by

Aryan Lall
Roll No. - 17D070053

under the guidance of
Prof. Siddharth Tallur

Department of Electrical Engineering

INDIAN INSTITUTE OF TECHNOLOGY BOMBAY
Powai, Mumbai - 400076
October 2021

Abstract

This report presents the preliminary work done towards the development and demonstration of the toolchain for deploying the machine learning frameworks on edge-devices including the resource-constrained FPGAs. This includes the comparative analysis of the performance obtained on different platforms using benchmark applications. Once the toolchain has been developed and tested, the idea is to implement an application on a suitable hardware by utilizing the tested methodologies.

First section of this report would provide an overview of TinyML, its applications and about the TensorFlow Lite framework. Further, the report addresses the implementation details and results obtained on different hardware platforms including ST microcontrollers and FPGAs. Future tasks include selecting an appropriate application and further implementing it on the hardware. These applications might include problems from domains such as genomics, machine and equipment health monitoring, computer vision and maneuvering, etc. All the required steps have been properly documented to enable quicker reproduction of the aforementioned work.

Contents

Chapter 1

Introduction

1.1 TinyML and its applications

Machine learning has transformed the way we visualize the data and offers promising solutions to a vast variety of problems. These data-driven approaches demand large computational power and storage for their handling and analysis. This poses a problem for their implementation on low-power edge devices which are powered by Microcontroller Units (MCUs).

The TinyML paradigm proposes to integrate Machine Learning(ML)-based mechanisms within small objects powered by MCUs [1]. This provides the door for the creation of new apps and services that do not require cloud processing assistance which on the other hand consumes higher energy, contributes to latency, and poses data security and privacy issues. The incorporation of intelligence into these tiny devices has a number of benefits which include energy-efficient devices, cheaper and faster product which offers data security. However, considering the enormous heterogeneous range of existing devices with limited computational power and memory, it still remains a challenge to implement machine learning frameworks on these tiny devices. With great power, comes great responsibilities!

The applications of TinyML can range from smart devices to the recent booming industry of AgroTech. Many IoT applications are already integrated in our daily personal and professional activities. Health sector offers large opportunities where applications such as health monitoring, visual assistance, hearing aids, personal sensing, activity recognition, etc. requires reliable smart devices. Apart from these, few other applications from the never-ending list includes augmented reality, real-time voice recognition, language translation, context-aware support systems,

machine health monitoring, Hearables [2], etc. Such low-power requirement even allows the operation of these devices in remote areas. Recently, machine learning is also paving its way in bioinformatics and a lot of opportunities exist in the field of genomics. TinyML would be the next wave of digital revolution.

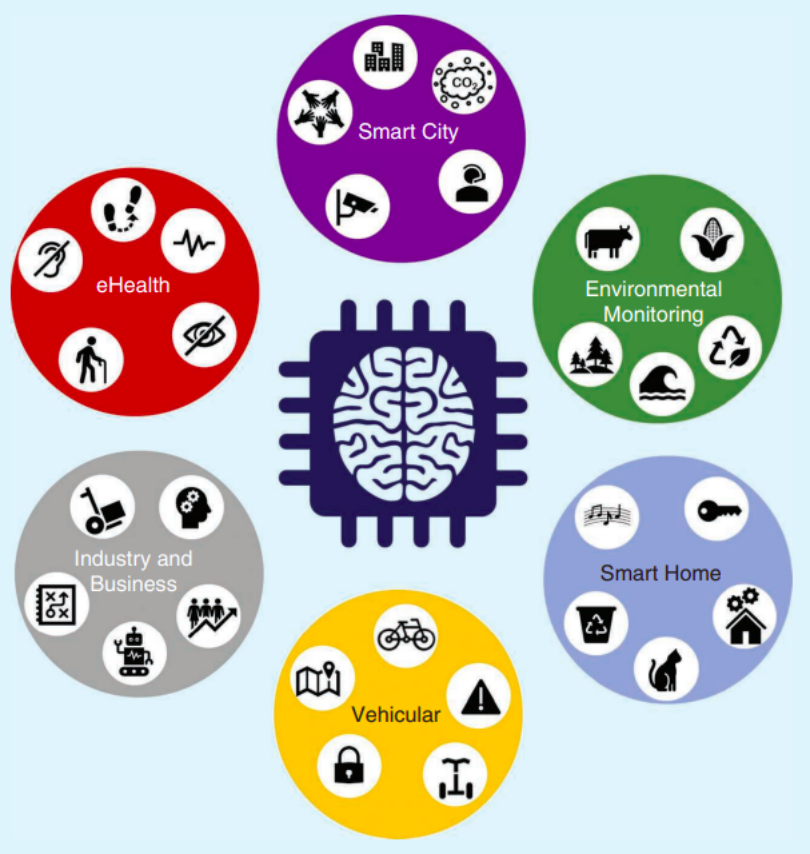


Figure 1.1: Applications of TinyML [Source [1]]

1.2 TensorFlow Lite for Microcontrollers

1.2.1 Overview

TensorFlow Lite Micro (TFLM) is an open-source ML inference framework for running deep-learning models on embedded systems [3]. This framework employs an unified flexible interpreter which addresses the resource-constraints and heterogeneity of the embedded platforms. Beyond deploying a model to an embedded target, the framework must also have a means of training a model on a higher compute plat-

form. The training can thus be performed using the tensorflow APIs with maximum efficiency, and TFLite micro can further be used as the inference engine in MCUs. It doesn't require operating system support, any standard C or C++ libraries, or dynamic memory allocation and also supports floating and quantized (int8, uint8) inference.

TFLM makes it easy to get TinyML applications running across architectures, and it allows hardware vendors to incrementally optimize kernels for their devices. It gives vendors a neutral platform to prove their performance and offers countless benefits which include portability, flexibility, minimization of external dependencies, custom accelerators, etc.

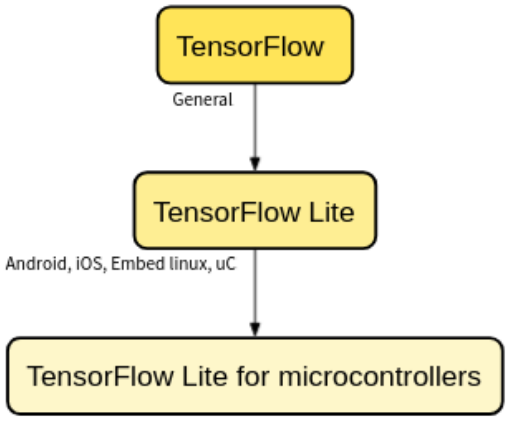


Figure 1.2: TensorFlow Lite framework for microcontrollers can be thought as a subset which utilizes the features offered by the higher TensorFlow APIs in a limited fashion. TensorFlow Lite supports a limited subset of TensorFlow operations, with limited set of devices.

TensorFlow Lite for Microcontrollers is written in **C++11** and requires a **32-bit** platform. It has undergone testing and it has been deployed extensively with many processors based on the Arm Cortex-M architecture. It has been ported to other architectures including ESP32 and many digital signal processors (DSPs). The framework is also available as an Arduino library. It can generate projects for environments such as Mbed as well.

1.2.2 Implementation Workflow

Once the model training has been completed using the tensorflow framework or the high-level APIs such as Keras, a tensorflow model is obtained. To convert this

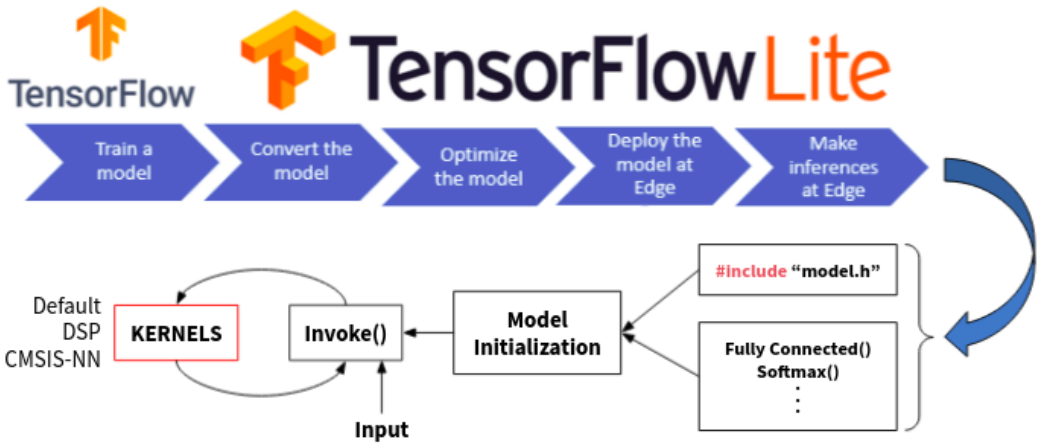


Figure 1.3: Implementation workflow for TensorFlow Lite for Microcontrollers

trained model to run on microcontrollers, it needs to be converted into TensorFlow Lite model. On Python, this conversion can be carried out using the TensorFlow Lite converter Python API [4]. This will convert the model into a FlatBuffer, reducing the model size, and modify it to use TensorFlow Lite operations.

Further, the users can carry out Post-training quantization which can reduce the model size while also improving the CPU and hardware accelerator latency. All the model parameters can be expressed using 8-bit integral or half-precision floating representations. However, this can have trade-off and might reduce the model accuracy, although this downgrading effect is negligible and can be tolerated. This technique can drastically reduce the latency while implementing applications on microcontrollers or FPGAs as integer arithmetic operations are much faster as compared to the corresponding floating operations. This quantization can be carried out in python itself while converting the tensorflow model to TFLite model [5]. Further, this TFLite model is converted to a C byte array (flatbuffer) using standard tools to store it in a read-only program memory on device. Figure 2.1 shows the workflow for the TFLite model deployment. *model.h* file contains the final C byte array which represents the entire model. Further, the individual model layers or operations can be registered and the model can then be initialized using the registered operations and the flatbuffer. Finally, we can invoke the inference engine using the user-inputs.

Each layer operation (fully connected, etc.) requires a Kernel for execution. These individual kernels are by-default written in C++11, and thus they can be accelerated on hardware using DSP instructions or libraries such as CMSIS-NN (for Cortex-M processors), etc. More details would be discussed in further sections.

1.2.3 Obtaining the TFLite model file

Once we have trained our tensorflow model on a host PC, we need to convert this model into a special, space-efficient format for use on memory-constrained devices. We'll use the TensorFlow Lite Converter for this purpose. For a simple demonstration, the following python code coverts a trained keras model to a TFLite model.

```
converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.OPTIMIZE_FOR_SIZE]
tflite_model = converter.convert()
open("my_model.tflite", 'wb').write(tflite_model)
```

As we have discussed in the previous section, we can obtain an 8-bit integer(int8) quantized model by making the following changes in the converter configuration.

```
# Provide a representative dataset to ensure correct quantization of input values
def representative_data():
  for input_value in tf.data.Dataset.from_tensor_slices(training_data).batch(1).take(100):
    yield [input_value]

converter = tf.lite.TFLiteConverter.from_keras_model(model)
converter.optimizations = [tf.lite.Optimize.DEFAULT]
converter.representative_dataset = representative_data
# Ensure that if any ops can't be quantized, the converter throws an error
converter.target_spec.supported_ops = [tf.lite.OpsSet.TFLITE_BUILTINS_INT8]
# Set the input and output tensors to int8
converter.inference_input_type = tf.int8
converter.inference_output_type = tf.int8

tflite_model_quant = converter.convert()
open("my_model_quant.tflite", 'wb').write(tflite_model_quant)
```

Finally, we need to convert the TensorFlow Lite model (.tflite) into a .h header file that can be loaded by TensorFlow Lite for Microcontrollers. The *xxd* utility in linux can very conveniently perform this operation as follows-

```
# Install xxd if it is not available
$ apt-get update && apt-get -qq install xxd
# Convert to a C source file (TFLite micro model)
$ xxd -i 'my_model.tflite' > 'model.h'
```

This *model.h* file contains a C byte array of our neural network model in the FlatBuffer format. This would be further used while configuring the microcontroller application project.

Chapter 2

Implementation on STM32 boards

The STM32 eval boards have been designed as a complete demonstration and development platform for the STM32 MCUs and MPUs. STM32 is a family of 32-bit microcontroller integrated circuits by STMicroelectronics. These chips are grouped into related series that are based around the same 32-bit ARM processor core, such as the Cortex-M33F, Cortex-M7F, Cortex-M4F, Cortex-M3, Cortex-M0+, or Cortex-M0. Internally, each microcontroller consists of the processor core, static RAM, flash memory, debugging interface, and various peripherals [6]. This offers very high performance, real-time capabilities, digital signal processing, low-power operation, and connectivity, while maintaining full integration and ease of development.

We have started out by implementing the TFLite framework on the STM32 boards. Following are the boards on which we tested our implementation -

MCU	**Core**	**RAM(kB)**	**Flash(kB)**	**DSP & FPU**	f_{MAX}**(MHz)**
STM32F401RE	Cortex-M4	96	512	Yes	84
STM32F746NG	Cortex-M7	320	1024	Yes	216

Table 2.1: Specifications for the tested STM32 boards

The above boards were readily available at the time of project commencement and they also support the DSP instructions, thus allowing custom acceleration of the execution kernels. Both of these boards can be easily programmed/debugged using the mini-USB cable and the applications were developed using the STM32 Cube IDE. This eclipse based environment simplifies the project creation and development. The first step towards the implementation of TFLite framework involves the generation of source code files which can be used as a library. Further details are provided in the following sections.

2.1 Generating source files for TFLite

TensorFlow Lite for Microcontrollers is able to generate standalone projects that contain all of the necessary source files, using a Makefile. The current supported environments are Keil, Make, and Mbed. By default, the project will be compiled for the host operating system, although we can specify a different target architecture. The original guide for the TensorFlow lite micro uses the Make build tool to generate a number of example projects that we can use as templates for our microcontroller projects. The idea is to use the TensorFlow Lite as a library instead of a starter project.

For a successful execution of Make, we require few additional tools which can be installed via the following set of commands.

```
$ sudo apt update
$ sudo apt install make git python3.7 python3-pip zip
```

Further we can clone the newest version of the TensorFlow repository.

```
$ git clone --depth 1 https://github.com/tensorflow/tflite-micro.git
```

The last step would be to navigate to the cloned directory and run the Makefile in the TensorFlow Lite for Microcontrollers directory.

```
$ cd tflite-micro
$ make -f tensorflow/lite/micro/tools/make/Makefile
    OPTIMIZED_KERNEL_DIR=cmsis_nn generate_projects
```

Once this is completed, it generates a number of example projects at *tensorflow/lite/micro/tools/make/gen/linux_x86_64/prj*. The exact path would depend on the host operating system. Since, we were using Linux to build the applications, we have *linux_x86_64/* in our directory path. Inside the desired project e.g *hello_world*, we have projects created for the current supported environments such as Keil, Make, and Mbed. Since Make projects are target neutral, we would navigate to this directory. This contains the two most important folders (*tensorflow* [1] and *third_party* [2]) which consists of all the required TFLite source and header files, and these would be further imported in the application project as libraries.

An important flag to notice while running the Make command is the OPTIMIZED_KERNEL_DIR option. Using this option generates the required CMSIS_NN kernel files and they are located in the *tensorflow/lite/micro/kernels* directory of the above generated tensorflow[1] folder. The value of this flag (*cmsis_nn*) is the folder name where the CMSIS_NN kernels are stored. TFLite source files are now generated :)

2.2 Configuring Application project

We can create an application project in the STM32 Cube IDE with C++ as the target language. Make sure to enable required peripherals/pins and configure them properly in the software application code. For example, we require a Timer module for profiling the run-time execution. We can enable any of the available timer modules and instantiate it with proper configuration values such as prescalar, period, etc.

Once the project has been created, we can configure it for accommodating and compiling our tensorflow model using the following steps -

Copying the TensorFlow Lite model & source code files to our project:
We need to copy our TFLite model file (*model.h*) into the include directory of the project. For STM32 Cube IDE project, this is located at *<project_directory>/Core/Inc*.

Further, we create a new folder *tensorflow_lite* in the *<project_directory>* and copy the two important folders that we created earlier (*tensorflow* [1] and *third_party* [2]) from the *hello_world* project at this location. We must also delete the examples folder located in the *<project_directory>/tensorflow_lite/tensorflow/lite/micro* as it contains unnecessary C files which must be avoided while compiling. The project directory structure should finally appear as follows -

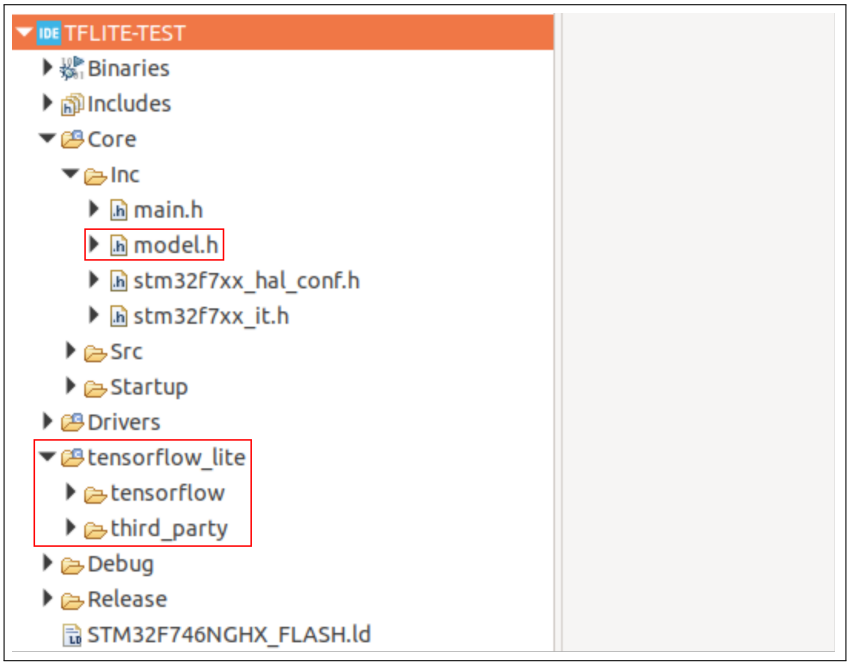


Figure 2.1: Project directory structure

Now, we would only make required changes in the created *tensorflow_lite* folder.

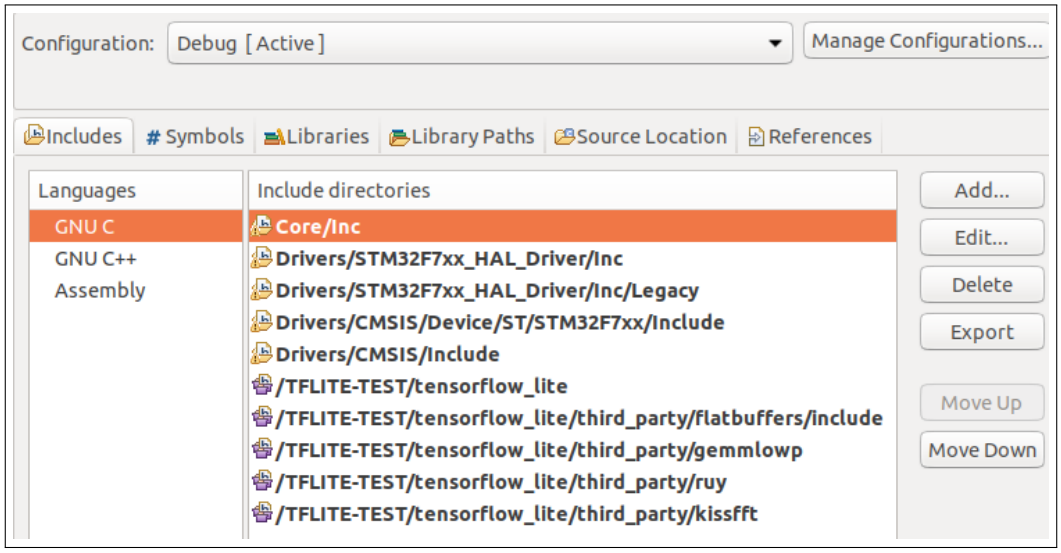
Include Headers and Source files in build process:

Even though the source files are located in our project, we still need to tell our IDE to include them in the build process. Go to **Project >Properties**. In that window, go to **C/C++ General >Paths and Symbols >Includes tab >GNU C**. Click **Add** and in the pop-up window, click **Workspace**. Select the ***tensorflow_lite*** directory in the project. Check **Add to all configurations** and **Add to all languages**.

Similarly, add the following directories also -

- *tensorflow_lite/third_party/flatbuffers/include*
- *tensorflow_lite/third_party/gemmlowp*
- *tensorflow_lite/third_party/kissfft*
- *tensorflow_lite/third_party/ruy*

Make sure that these include directories are also reflected in the **GNU C++** and **Assembly** languages, for each of the **Debug** and **Release** configuration. Finally, we would include the source directory of TFLite. Go to the **Source Location** tab and add *<project_directory>/tensorflow_lite* to both the Debug and Release configurations. Click Apply and Close.



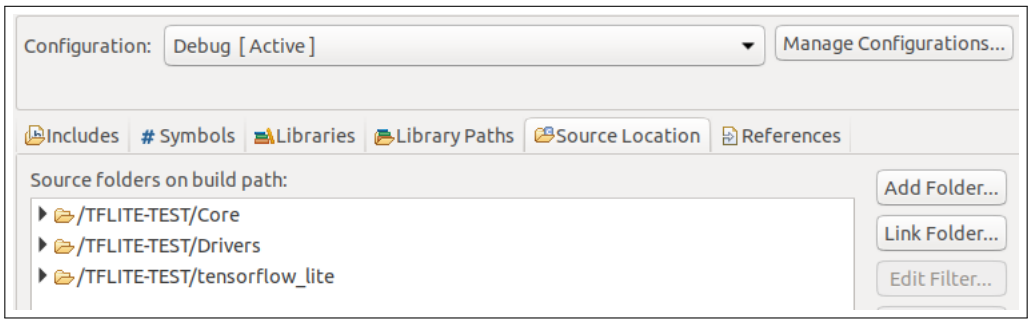


Figure 2.2: Includes and Source Location tabs after above changes

The required header and source files are now included in our project. Thus, we can easily use the TFLite framework as a target neutral library for various kind of projects. Lastly, we need to delete few unnecessary files before we compile the code. Following folders are required to be deleted (*these files will be used later) -

1. *<project_directory>/tensorflow_lite/tensorflow/lite/micro/tools/make/downloads/cmsis/CMSIS/NN*

2. *<project_directory>/tensorflow_lite/tensorflow/lite/micro/kernels/<cmsis_nn folder >*

Writing the Application code:

We can now edit the *main.cpp* file and add our application code. For using the tensorflow variables and functions, we need to 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 "model.h"  // Include the model file
```

Depending on the hardware platform, we need to initialize the peripherals with proper configuration values. The STM Cube IDE automatically generates these initialization functions and thus, they can be easily called inside the code. Rest of the discussion focuses on explaining the different sections inside the TFLite application code. For an example code, please refer to main.cpp.

First, we initialize the TFLite micro model with the parameters of our model file.

```
model = tflite::GetModel(my_model);  //my_model is the C byte array
```

Next, we must register the layer operations that our model uses. This will be used by the interpreter to access the operations that are used by the model. We can do this in two different ways:
1. Include all the operations available in tensorFlow lite for microcontrollers. Although this is an easier way to load the operations, it uses a lot of memory as various unnecessary operations are also included. This method is not recommended for microcontroller applications. Following code shows how to declare the operation resolver using this method -

```
tflite::AllOpsResolver resolver  // All operations registered;
```

2. We can manually register only those operations which are required by the model. 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* as follows -

```
static tflite::MicroMutableOpResolver<1> micro_op_resolver;
tflite_status = micro_op_resolver.AddFullyConnected();
if (tflite_status != kTfLiteOk)
{
  error_reporter->Report("Could not add FULLY CONNECTED op");
}
```

Next, we would build the interpreter to run the model and allocate memory for the required tensors as follows -

```
static tflite::MicroInterpreter static_interpreter(
    model, micro_op_resolver, tensor_arena, kTensorArenaSize,
       error_reporter);
interpreter = &static_interpreter;

tflite_status = interpreter->AllocateTensors();
```

Finally, we can assign the user values to the input tensor and invoke the TFLite model to obtain the output results.

```
model_input->data.f[0] = 2.0f;  // User value
tflite_status = interpreter->Invoke();   // Invoke TFLite model
float my_output = model_output->data.f[0];  // Get output
```

Depending on the application, the above code can be modified for implementing complex algorithms. For example. we have also implemented the MNIST classification problem on the STM Disco board.

Miscellaneous:

We can also modify the Debug code of tensorflow lite to allow error messages to be sent over the UART peripheral, thus making the debugging easier. Open the file *<project_directory>/tensorflow_lite/tensorflow/lite/micro/debug_log.cc* and update it's code as follows -

```
#include "tensorflow/lite/micro/debug_log.h"
extern "C" void __attribute__ ((weak)) DebugLog(const char* s){
  // To be defined by the user further
}
```

Go to the *main.cpp* file and redefine this debug function as follows -

```
extern "C" void DebugLog(const char* s)
{
 // Enable error messages to be sent via UART
 HAL_UART_Transmit(&huart1, (uint8_t *)s, strlen(s), 100);
}
```

In STM32 Cube IDE, *printf* (and variants) does not support floating point values. To add that, we need to go to *Project >Properties >C/C++ Build >Settings >Tool Settings tab >MCU G++ Linker >Miscellaneous*. In the *Other flags* pane, add the line **-u_printf_float** for both Debug and Release configurations. Finally, we can build our project and generate the .elf file.

Running the application:

Once the application has been built successfully, we can run it on the STM32 board. We can use utilities such as PuTTY to send or receive the UART messages over the serial terminal. In python, pySerial can be used for communicating with the board over the serial port.

```
arm-none-eabi-objdump -h -S  TFLITE-TEST.elf  > "TFLITE-TEST.list"
arm-none-eabi-objcopy  -O binary  TFLITE-TEST.elf  "TFLITE-TEST.bin"
   text    data     bss     dec     hex filename
  95964  205748   53464  355176   56b68 TFLITE-TEST.elf
```

Figure 2.3: Size (bytes) of different sections in the .elf file (dec is total size)

2.3 Acceleration using DSP instructions

Since the TFLite kernels are written in C++, it gives us an opportunity to modify and accelerate them using various hardware optimizations. This enables us to completely utilize the power of the processor core and leverage the performance of our application. One such optimization could be to use the available DSP instructions which many of the ARM cores offer.
The **CMSIS DSP** software library is a suite of common signal processing functions for use on Cortex-M and Cortex-A processor based devices. This library has generally separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values. The performance gain would vary across different processor architecture. Rest of the section discusses about the usage of DSP instructions and how can we modify the TFLite kernels. This discussion **assumes** the following -

1. Cortex-M or Cortex-A processor based device which supports DSP instructions
2. User has configured the application project according to section 2.2

2.3.1 Usage in Fully Connected layer

A fully connected layer essentially involves the following 3 stages of computation -

1. A matrix multiplication (weight x input)
2. Addition of 2 vectors (bias and output of previous stage)
3. Apply layer activation function (ReLU, Softmax, etc.)

The first 2 stages of the fully connected layer can be effectively implemented and accelerated using the common CMSIS DSP instructions. The implementation of the last stage using DSP is subject to complexity and availability of the DSP instructions.

Considering that we are working with floating-precision, the following CMSIS DSP functions can help us to accelerate the computation of the fully connected layer -

1. **arm_mat_mult_f32** : This function performs a floating-point matrix multiplication using DSP instructions.
2. **arm_add_f32** : This function performs a floating-point addition between 2 input vectors using DSP instructions.

The matrix input is provided in a special format of matrix data structures, called as an ARM matrix instance. Thus, this matrix instance is first initialized using a function (*arm_mat_init_f32)* which sets the values of the internal structure fields.

2.3.2 Configuring project settings

Before we modify and compile the code, we are required to make few changes in the project settings such as including DSP functions header files, setting preprocessor symbols, etc. These are listed as follows -

Including the DSP Header files:
There are two ways to obtain the CMSIS DSP header files. First, the TFLite make process itself generates the DSP header files and they are located at *<project_directory>/tensorflow_lite/tensorflow/lite/micro/tools/make/downloads/cmsis/CMSIS/DSP/Include*.

Second, we can clone the original github repository of CMSIS. Further, we need to copy the folder *CMSIS_5/CMSIS/DSP/Include* and paste it under *<project_directory>/Drivers/CMSIS/DSP* (new folder).

Go to **Project >Properties**. In that window, go to **C/C++ General >Paths and Symbols >Includes tab >GNU C**. Click **Add** and in the pop-up window, click **Workspace** and select **only** one of the above folders in the project. Check **Add to all configurations** and **Add to all languages**.

Including the DSP Libraries:
We need to include and link the appropriate library in the application. Copy the folder *<STM32Cube_Repository >STM32Cube_FW_F(2/4/7)_V.X.XX.X/Drivers/CMSIS/Lib* and paste it under *<project_directory>/Drivers/CMSIS*. The exact path (2/4/7) would depend on whether Cortex-M(3/4/7) is used, respectively.
In the properties window, go to **C/C++ Build >Settings >MCU G++ Linker > Libraries**. In the Library search path section, click Add and select the GCC library present in the workspace path *<project_directory>/Drivers/CMSIS/Lib/GCC*. Now, to add the specific library to work with, click Add in the Libraries section and insert **arm_cortexM7lfsp_math** (for Cortex-M7) or **arm_cortexM4lf_math** (for Cortex-M4). For more details on which library to include, refer to comments in *arm_math.h* file.

Insert Preprocessor symbols:
Finally, we need to inform the compiler about the presence of DSP math instructions by inserting appropriate preprocessor symbols. Go to **C/C++ Build >Settings >MCU GCC Compiler >Preprocessor** and add **ARM_MATH_CM(3/4/7)** to the Define symbols section. The exact symbol (3/4/7) would depend on whether Cortex-M(3/4/7) is used, respectively. Click Apply and Close.

2.3.3 Modifying the TFLite Kernel

The kernel (*FullyConnected* function) which performs the floating-precision fully connected layer operation is located at -
<project_directory>/tensorflow_lite/tensorflow/lite/kernels/internal/reference/fully_connected.h

By default, the kernel code uses normal *for* loops for computation as follows -

```
for (int b = 0; b < batches; ++b) {
  for (int out_c = 0; out_c < output_depth; ++out_c) {
    float total = 0.f;
    for (int d = 0; d < accum_depth; ++d) {
      total += input_data[b * accum_depth + d] *
               weights_data[out_c * accum_depth + d];
    }
    float bias_value = 0.0f;
    if (bias_data) {
      bias_value = bias_data[out_c];
    }
    output_data[out_c + output_depth * b] =
       ActivationFunctionWithMinMax(
        total + bias_value, output_activation_min,
           output_activation_max);
  }
}
```

Figure 2.4: Original kernel code

The value of variable *batches* is generally 1. Thus, we can ignore this variable for simplifying the loop. The *input_data* vector has *accum_depth* number of elements. Similarly, both the *output_data* and *bias_data* vector has *output_depth* number of ele-

ments. This example uses the ReLU activation. Instead of iterating over each element and performing the computation, we can replace the above code section with few DSP instructions as follows -

```
arm_matrix_instance_f32 Weight, Input, Output;

// Initailize matrix instance
arm_mat_init_f32(&Weight, output_depth, accum_depth, (float32_t*)
    weights_data);
arm_mat_init_f32(&Input, accum_depth, 1, (float32_t*)input_data);
arm_mat_init_f32(&Output,output_depth,1, (float32_t*)output_data)

// Matrix multiplication
arm_mat_mult_f32(&Weight, &Input, &Output);

// Addition of bias
if (bias_data) {
  arm_add_f32(output_data, const_cast<float*>(bias_data),
      output_data, output_depth);
}

// ReLU Activation
for (int out_c = 0; out_c < output_depth; ++out_c) {
  output_data[out_c] = ActivationFunctionWithMinMax(
                output_data[out_c], output_activation_min,
                    output_activation_max);
}
```

Figure 2.5: Updated kernel code using DSP instructions

For more information regarding the DSP function arguments and details, please refer to the CMSIS DSP manual. The above example considers a floating operation, however, we can also work with fixed-precision. The DSP library has separate functions for operating on 8-bit integers, 16-bit integers, 32-bit integer and 32-bit floating-point values. Thus, we can modify the kernel accordingly and accelerate the layer operations.

2.4 Acceleration using CMSIS-NN Library

Another optimization technique for accelerating the TFLIte kernels is to use the CMSIS NN library for Cortex-M processor cores, which is a collection of efficient neural network kernels developed to maximize the performance and minimize the memory footprint of neural networks. This library has separate functions for operating on different weight and activation data types including 8-bit integers (q7_t) and 16-bit integers (q15_t). The description of the kernels are included in the function description. For more implementation details, refer to the paper [7].

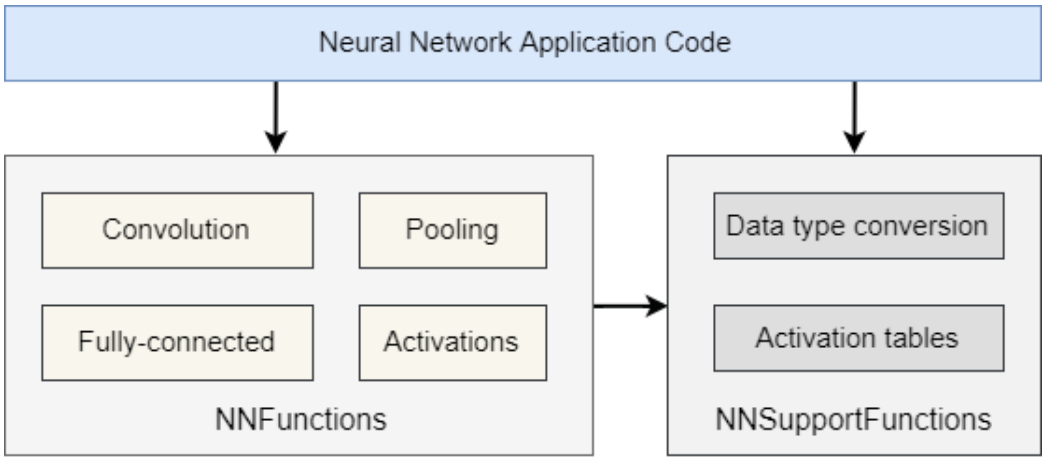


Figure 2.6: CMSIS NN Block diagram. Invoking the model executes the pre-build NN functions.

Various experimental results [7] have shown that neural network inference based on CMSIS-NN kernels achieves **4.6X** improvement in runtime/throughput and **4.9X** improvement in energy efficiency. We can directly use this kernels in our application code to implement neural network algorithms on Arm Cortex-M CPUs. However, the current library doesn't offers support for architectures such as RNNs, Transformers, etc. The performance gain would again vary across different processor architecture. Rest of the section discusses about the usage of these kernels. This discussion **assumes** the following -

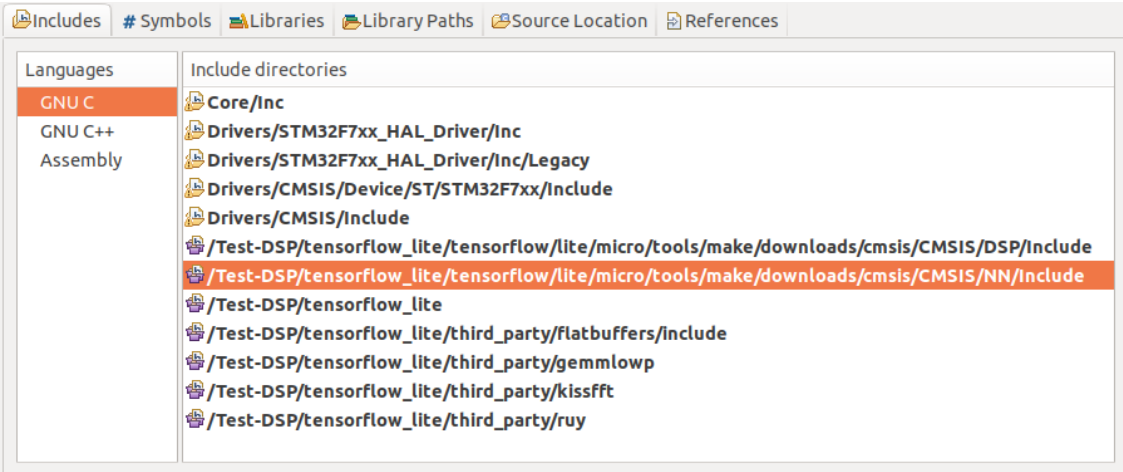
1. Cortex-M processor based device which supports DSP instructions
2. User has configured the application project according to section 2.3.2

2.4.1 Configuring project settings

Restore the folders that were deleted while configuring the project in this section. Both of these folders contain the CMSIS NN kernel source files and thus are required to be added in the project. Configure the project as follows -

Including the Header files:

All the header files are present in the above ***NN/Include*** folder which are auto-generated during the make process of TFLite. However, we can also obtain the header and source files from the CMSIS github repository, as discussed in the DSP section. Include this folder in the project. Check **Add to all configurations** and **Add to all languages**.



Insert Preprocessor symbols:

We need to inform the compiler about the presence of CMSIS_NN kernels by inserting appropriate preprocessor symbols. Insert the following symbols in the project, similar to the way discussed in the DSP section. Click Apply and Close.

1. **CMSIS_NN**
2. **ARM_MATH_DSP** // DSP instructions are supported
3. **__FPU_PRESENT=1** // FPU is present in the device core

For more information on which symbols to include, check out the comments in the file *arm_nnfunctions.h*.

2.4.2 Modifying the TFLite Kernel

The CMSIS NN kenels for TFLite are auto-generated during the make process and they are located at *<project_directory>/tensorflow_lite/tensorflow/lite/micro/kernels/<cmsis_nn folder >*. Consider that we are modifying the kernel for the fully connected layer. The default kernel is located at *<project_directory>/tensorflow_lite/tensorflow/lite/micro/kernels/fully_connected.cc* and this needs to be **replaced** with the one located in the *cmsis_nn* folder. Similarly, we can replace other default kernels with the CMSIS NN kernels. Note that few of the header includes might require some modifications. For example, originally the following include line is present in the file *fully_connected.cc* -

```
#include "CMSIS/NN/Include/arm_nnfunctions.h"
```

This needs to be changed as follows -

```
#include "arm_nnfunctions.h"
```

Once all the required kernels are replaced, we can compile our application project and generate the .elf file. The performance comparison between the different optimization techniques and the resultant performance gain is discussed in the later chapters. This chapter explains the way users can configure their project settings to accommodate the TFLite framework and introduces various techniques for run-time acceleration on the STM32 boards. Users are encouraged to explore new techniques and ways to implement their own kernels and integrate them with the existing setup.

Chapter 3

Implementation on ARM Soft CPU

The last chapter was centered around the implementation of TFLite framework on hard processor cores such as ARM Cortex-M based devices, etc. Thus, we were only required to develop the software and deploy it on the MCU. However in this case, we are highly restricted by the features offered by the MCU vendor and it becomes very difficult to perform any changes on the available hardware.

FPGAs, on the other hand, offer enough flexibility to define and reconfigure the hardware. The parallel processing ability offered by FPGAs allows the development of custom hardware accelerators and is ideal for applications including image processing or artificial intelligence. Complex tasks are often solved by software implementations with fast processors. FPGAs offer a cost-effective alternative, which, via parallelization and adaption to the application, provide a significant speed advantage compared to processor-based solutions. At this point, we may ask a question, Can the machine learning frameworks be implemented on FPGAs?

Few SoC and MPSoC families from Xilinx such as Zynq, UltraScale, etc. combines the power of FPGA with the ARM core processor along with various other peripherals. This offers more design choices to the engineers and this single device forms an even more powerful embedded computing platform. For these powerful boards, we already have development environments such as **Vitis AI** which is a Xilinx's development platform for AI inference on Xilinx hardware platforms, including both edge devices and Alveo™ cards. It consists of optimized IP, tools, libraries, models and is designed with high efficiency and ease-of-use. However, not every FPGA would come with an ARM hardcore processor, especially if it is resource-constrained like CMOD A7-35T board, etc. Thus, the challenge in such devices is the absence of a hardcore PS which is required to perform complex control tasks and prepare the ground for the TFLite runtime.

3.1 ARM Cortex-M on FPGA

The ARM DesignStart FPGA program offers instant and free-of-cost access to Arm Cortex-M soft CPU IP for FPGA designs. There is no license fee and zero royalty associated with DesignStart FPGA so the developers can get started with prototyping and designing commercial products instantly [8]. **Cortex-M3** and **Cortex-M1** soft CPU IPs are available for integration in the FPGA design. Rest of the discussion is based on Cortex-M3 IP, although the project can be modified similarly for using the Cortex-M1 variant.

The Cortex-M3 is a low-power processor that features low gate count, low interrupt latency, and low-cost debug. It is intended for deeply embedded applications that require optimal interrupt response features. The processor implements the ARMv7-M Thumb instruction set, and is binary compatible with the instruction sets and features implemented in other Cortex-M profile processors. On the expense of flexibility offered by these IPs, they also have few **limitations**. The maximum supported clock frequency for Cortex-M3 IP is **50 MHz** and for Cortex-M1, it is 100 MHz. Thus, the slower clock would result in higher latency as compared to the hard-core solutions. Also, these soft CPUs don't have a floating point unit (FPU) and thus, usage of libraries such as CMSIS NN, DSP functions are not allowed. Currently the flow to update a bitstream with new Instruction Tightly Coupled Memory (ITCM) data only supports memory sizes in the range **16KB** to **128KB**. For more information regarding the soft IP, visit the ARM designstart FPGA website.

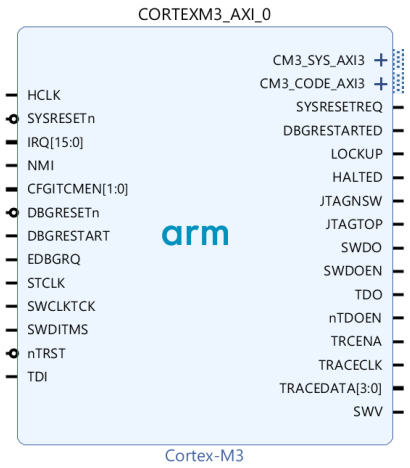


Figure 3.1: Cortex-M3 soft IP

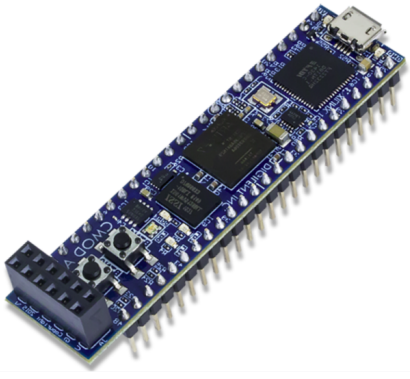


Figure 3.2: CMOD A7-35T Board

The idea is to implement the Cortex-M3 soft CPU IP on a resource-constrained

FPGA board such as Digilent's **CMOD A7-35T**, and further implement the TFLite framework for running various ML applications. The advantage of doing such a thing is that even these smaller boards would also be introduced to the machine learning domain, which wasn't possible earlier. Few specifications of the CMOD A7-35T board are mentioned below -

LUT	Flip-Flop	Block RAM	SRAM	Quad-SPI Flash
20800	41800	225 KB	512 KB	4 MB

Table 3.1: Key specifications of CMOD A7-35T board

3.1.1 Implementation Workflow

The first step towards the implementation of Cortex-M soft IP on FPGA involves the configuration of hardware and the generation of bitstream. We would be using Xilinx **Vivado 2019.1** for this purpose. The Cortex target's program memory is local to the FPGA, and the program memory contents is included in the FPGA bitfile. Xilinx uses an **MMI** file to describe the location of internal FPGA memories, so that their contents may be updated. Thus, we would also generate a .mmi file which would be later used to directly update the bitstream without the need of regenerating it.

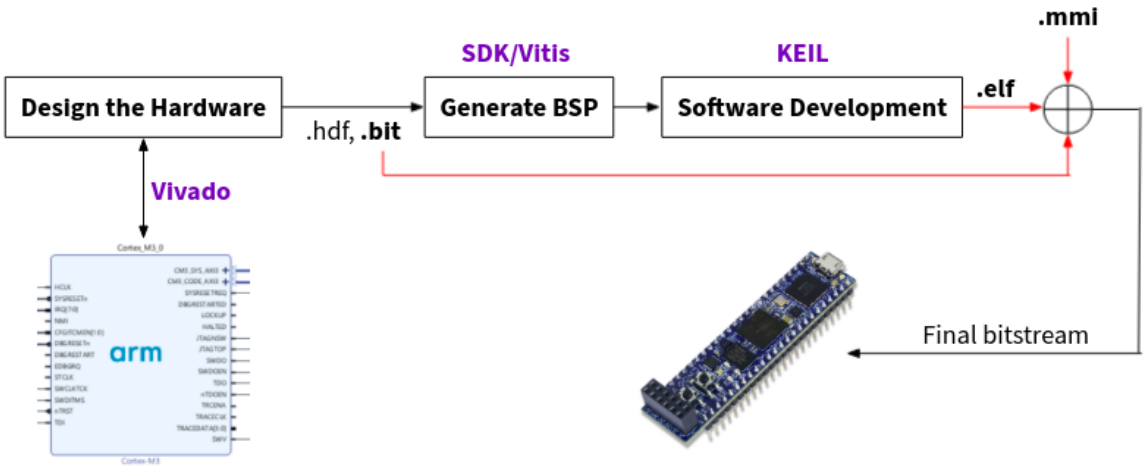


Figure 3.3: Implementation workflow for ARM Cortex-M IP

Once the bitsream is obtained, the next step is to generate the board support package (**BSP**) for the hardware platform. BSP is a set of files that provide low-level drivers for all the hardware. This allows for a consistent interface between

high-level software projects, and the low-level hardware drivers. This package will incorporate the memory map, for example, extracting the locations as consistently named variables. We would be using Xilinx **SDK 2019.1** for this purpose. For the software development, we would use the ARM **Keil-MDK 5** environment which is a software development solution for Arm-based microcontrollers. Although the software development route through Keil is a bit painful, we don't have any other options as these soft IPs are currently not supported by Vitis/SDK.

Once the application project is built successfully, we would obtain an **.elf** file. This, with the help of .mmi file, is used to update the program memory contents inside the bitstream. This method takes less than a minute, which is orders of magnitude faster than regenerating a bitfile. Finally, this updated bitstream is downloaded on the FPGA board. More details regarding the hardware and software configuration would be discussed in the further sections.

3.2 Hardware Configuration & Bitstream Generation

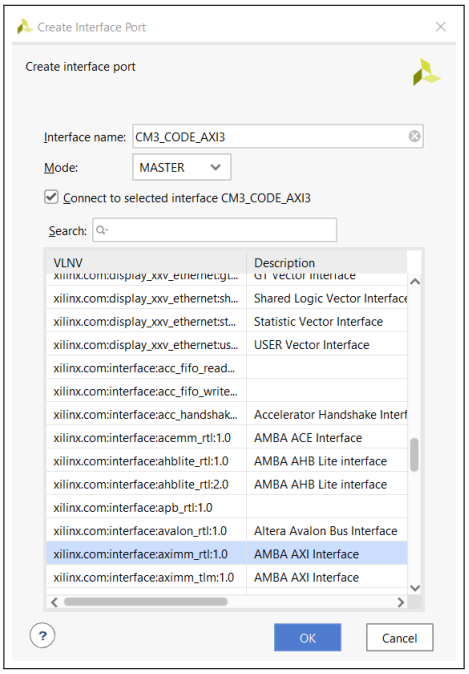
Vivado Design Suite is a software suite produced by Xilinx for synthesis and analysis of HDL designs. It allows us to synthesize and implement the hardware from a block level design using numerous pre-compiled Xilinx IPs and define interconnections among them, which drastically reduces the development time and time-to-market. FPGA is required to be configured with an appropriate hardware design which consists of Cortex-M3 IP as the CPU. The hardware and software package for this IP can be downloaded from the ARM designstart website.

The package comes up with an example vivado design project targeting Arty A7 and S7 boards. Users can modify this example project directly for their own target FPGA board. However, this process can be a bit longer and is prone to errors. Thus, we would create a new project in Vivado and define our design from scratch. Before we create the block deisgn, the Arm IP Integrator (IPI) repository for Cortex-M3 must be added to the list of Vivado IP repositories. This makes the processor available in any new designs. More details regarding the procedure and project initialization are mentioned in the *<package >/docs/arm_cortex_m3_designstart_fpga.../.pdf* doc. Users are requested to configure the project by following instructions till section 2.4.

Next, we can create a new block design in vivado and add the Cortex-M3 IP. Double clicking on this IP opens up the configuration window, where we can set various parameters such as the number of interrupts, debug level, size of ITCM (instruction memory) and DTCM (data memory), etc. Since the TFLite application would be big-

ger, we can set the ICTM size as 128KB. The DTCM size is proportional to the model size, however 64KB is a good starting value. Three of the most important external ports in the Cortex-M3 IP are listed below along with their usage and configuration.

1. **CM3_SYS_AXI3**: A memory-mapped master port which is used to access the external peripherals using the AXI Interconnect. Vivado would automatically connect this port while running the *connection automation*. Thus, we are not required to make any manual changes.
2. **CFGITCMEN[1:0]**: This input signal is the Instruction Tightly Coupled Memory (ITCM) alias enable. Bit [1] sets the upper alias enable line and bit [0] sets the lower alias enable line.
   If CFGITCMEN[1] is set, then the internal RAM ITCM is mapped to the upper address alias in the memory map.
   If CFGITCMEN[0] is set, then the internal RAM ITCM is mapped to the lower address alias in the memory map. Right-click the CFGITCMEN[1:0] input of the Cortex-M3 IP and select *Create Port*; accept pre-filled settings and click OK. Later, this would be assigned the constant value **1** in the top-level HDL code.
3. **CM3_CODE_AXI3**: The processor instruction fetches that are to memory addresses that are not aliased to the ITCM is output on the CM3_CODE_AXI3 port. Select this port in the Cortex IP, then right-click in the design window and select *Create Interface Port*; keep the pre-filled values and click OK.



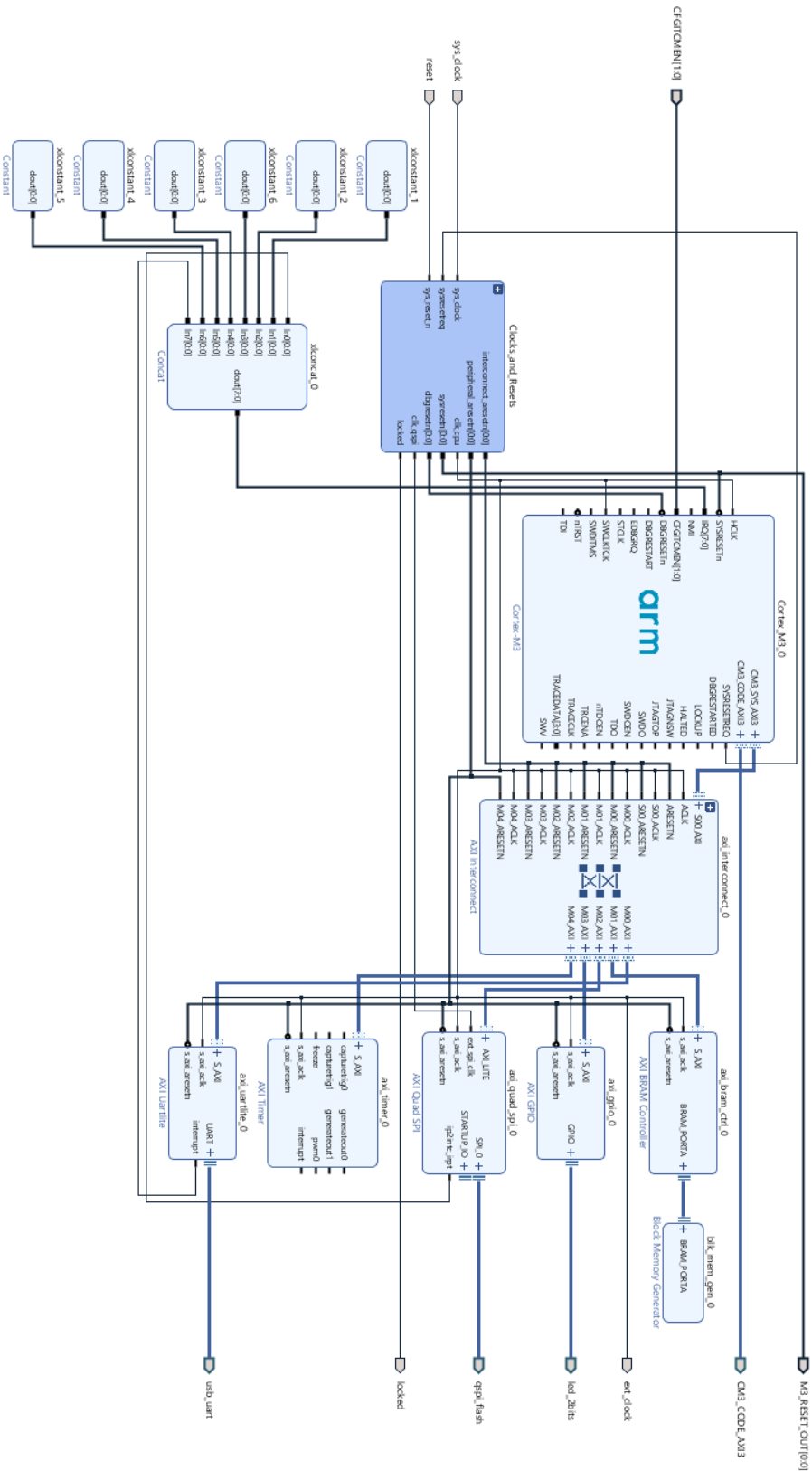


Figure 3.4: Block Design for using Cortex-M3 IP on CMOD A7-35T board

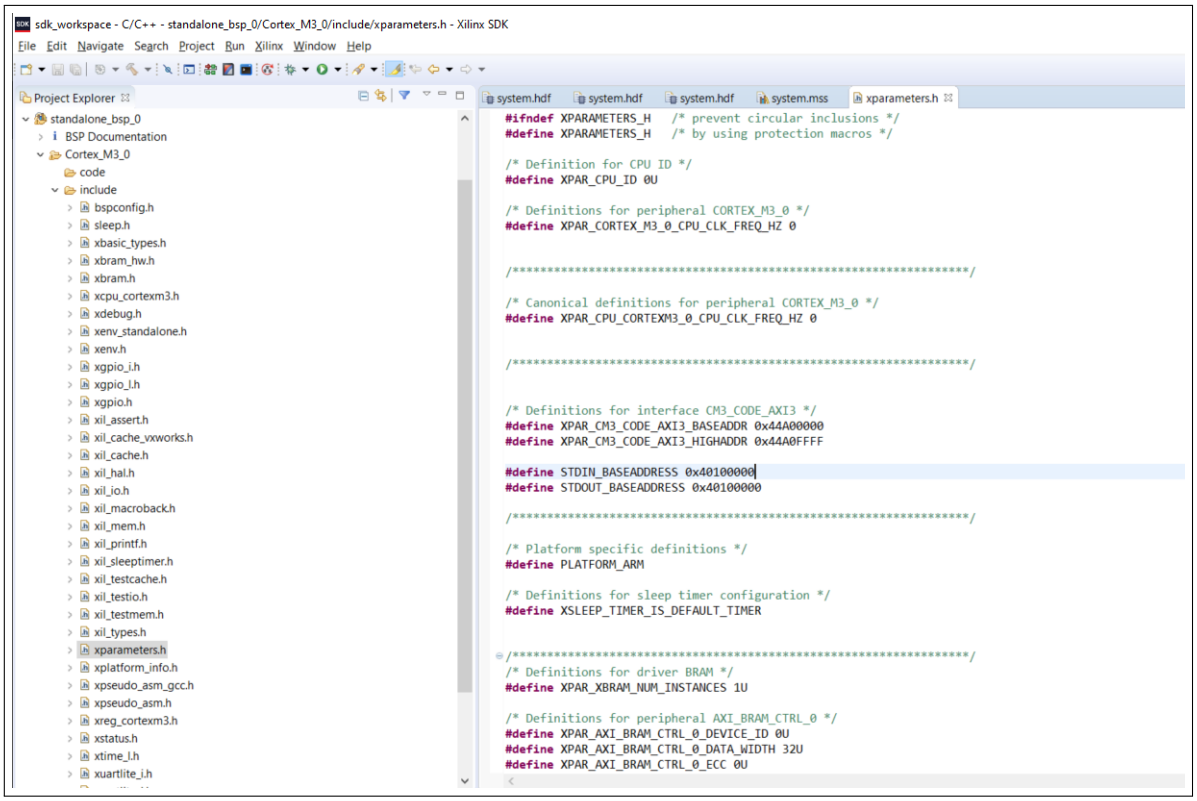
The interrupt port (**IRQ**) of Cortex IP can also be assigned a value of 0 using the constant blocks. Now, we can add and configure the external peripherals in our block design. For example, we can add the AXI Uartlite IP for enabling the UART communication, AXI GPIO for using the external LEDs, AXI Timer for using the timer module, etc. We also require clocks and reset to drive different components in the design. The *Block Automation* feature in vivado automatically connects these memory-mapped peripherals with the processor IP using an AXI interconnect. The final block design is shown in Figure 3.4.

The final step is to specify the top-level HDL file which binds our entire design. Go to the *Sources* tab in the block design window, right-click on the block design file (.bd) under the Design Sources, and select *Create HDL Wrapper*. This would generate a top-level verilog/vhdl file. If you are working with the CMOD A7-35T board, then replace the contents of this top-level file with the contents in cmod_a735t_arm.v. Otherwise, this top-level file can be modified similarly for other boards. Few other external ports (M3_RESET_OUT, locked, etc.) are also created in this design for debugging purpose. However, they can be removed and corresponding changes must be reflected in the top-level HDL file. Also, add the appropriate hardware constraint file (.xdc) for the specific board in the project. Finally, we can generate the **bitstream** (.bit) for our hardware design. Make sure that there are no errors encountered during this process.

3.3 Generation of BSP (Board Support Package)

Once the bitstream is generated successfully, the hardware can be exported to generate the BSP files. Xilinx Software Development Kit (SDK) is an eclipse-based Integrated Development Environment (IDE) for development of embedded software applications targeted towards Xilinx embedded processors. In the current development flow, this is used to generate the required BSP files. Follow the steps mentioned in the *<package>/docs/arm_cortex_m3_designstart_fpga.../.pdf* doc for the BSP generation. The Standalone OS version used in the current design is 6.7. However, the package files can be modified to support even higher versions!
If you are using *stdin* and *stout* or any print statement in your C/C++ application project, then make sure that these are set as *axi_uartlite_0* in the BSP settings. Failure to do so would possibly result in a non-functional UART peripheral. Also, check whether the *STDIN_BASEADDRESS* and *STDOUT_BASEADDRESS* in the *xparame-*

ters.h file are correctly set to the UART peripheral base address.
Lastly, we need to manually copy 2 files into the generated BSP location because of the differences between the SDK and Arm Keil-MDK. Copy the files *xpseudo_asm_rcvt.h* and *xpseudo_asm_rcvt.c* from - *<package>/vivado/Arm_sw_respository/CortexM/bsp/standalone_v6_7/src/arm/cortexm3/armcc* to *<sdk_workspace>/standalone_bsp_0/CORTEX_M3_0/include*. The BSP is complete and is now ready for compilation.



3.4 TFLite Software Development

The example *hello_world* project created during the TFLite build process in section 2.1 also generates the project files for the Keil environment. It contains the *µVision* project (.uvprojx) along with the other TFLite source files. The idea is to modify this example project and make it compatible with the FPGA platform.

Modify Target Configuration:
Before we proceed, we need to copy few files/folders to our current working direc-

tory of Keil project. First, copy the ***cmsis*** folder located under the *software* section of the Cortex-M3 package. Second, copy the bsp folder ***standalone_bsp_0*** which was generated in the previous section. Lastly, copy the *<Cortex-M3 package>/software/m3_for_arty_a7/main/m3_for_arty.h* file into the current working directory.
Open the example Keil project *(keil_project.uvprojx)*. In the Project window, right click on the *hello_world* and select *Options for Target 'hello_world'*. This would open up the target configuration window. Following are the changes that we are required to perform in the target configuration -

1. **Device**: In the Device tab, select the device as ARM Cortex M3 (ARMCM3).

2. **Target**: In the Target tab, the size of IROM1 and IRAM1 needs to be changed. Since our ITCM memory is 128KB, we can set the size of IROM1 as 0x20000. Similarly, size of IRAM1 is set as 0x10000.

3. **User**: Keil generates a .axf file after the application project is built successfully. This .axf file needs to be converted to hex format for BRAM initialization, and to .elf format for bitstream generation. A windows batch file *make_hex_a7.bat* serves this purpose and should be automatically run post-build. This file is available in the downloaded Cortex-M3 package under the *software/Build_Keil* section. However, few changes are required to be made in this file and an example make_hex_a7.bat is provided.
   In the user tab, check the *Run #1* box under the *After Build/Rebuild* option in Command Items. Also, select the corresponding *User Command* and specify the path to the *make_hex_a7.bat* file.

4. **C/C++ (AC6)**: In this tab, we can define the preprocessor symbols and the include file directories. In the *Define* section of *Preprocessor Symbols*, enter the line **ARTY_CM3 CORTEX_M3**. Next, select the *Include Paths* and open the browse window. Here, we need to add few include directories which correspond to the FPGA hardware and ARM platform as shown in Figure 3.5.

5. **Linker**: In this tab, check the *Use Memory Layout from Target Dialog* box. The *Scatter File* section should have automatically filled up. In case it is empty, try deselecting and selecting this box again. Figure 3.6 shows the tab after the required changes. Click OK.

The target configuration is completed. Now, we need to add the source files and manage the run-time environment settings.

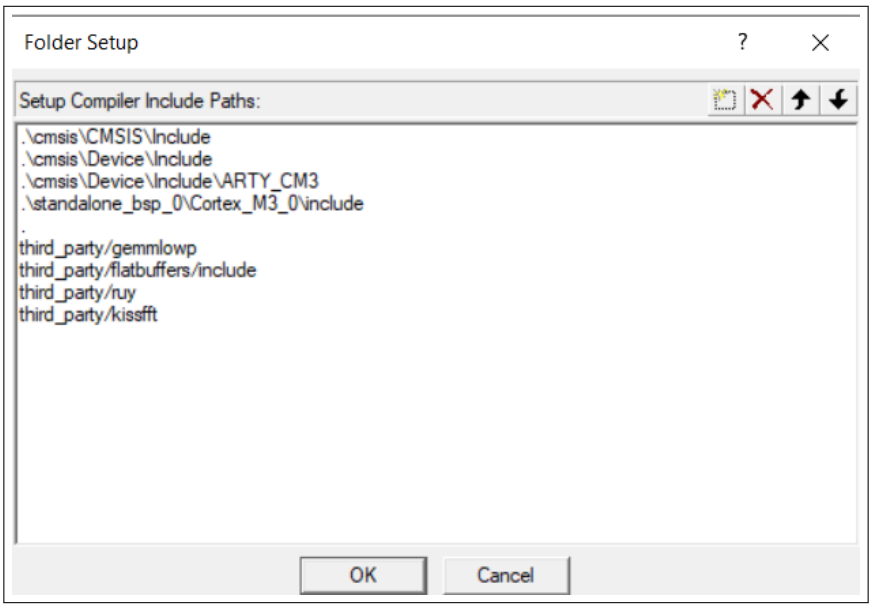


Figure 3.5: Include directories for the Keil project

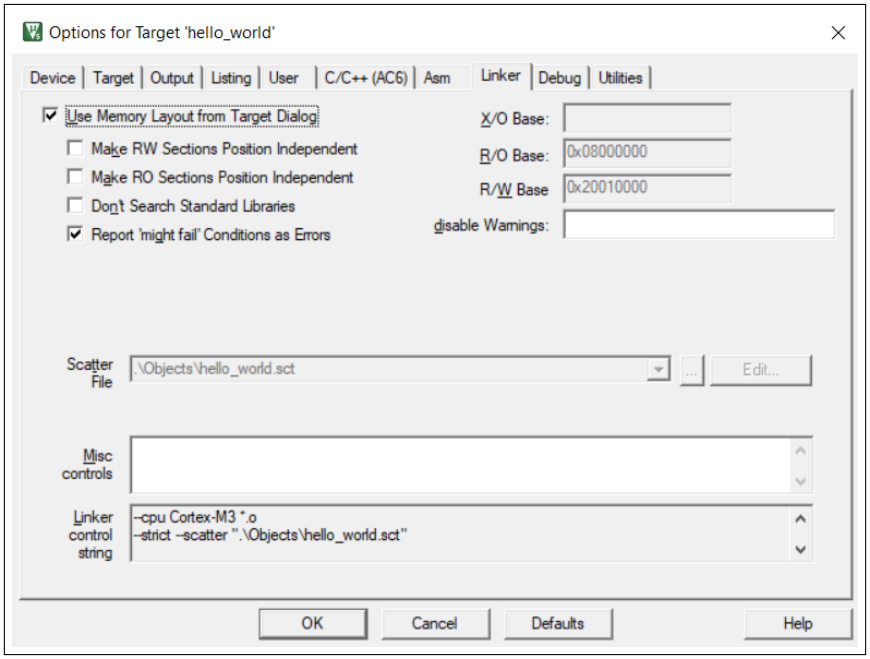


Figure 3.6: Linker tab after required changes

Add Source Files:

In the Project window, right click on the *hello_world* and select *Manage Project Items*. Here, we need to select various source files under the *Project Items* section. The

TFLite source files are already added under the group named ***Source***. Create a new group named ***CMSIS*** and add the following 2 files under it -

1. *cmsis/Device/Source/ARTY_CM3/system_ARTY_CM3.c*
2. *cmsis/Device/Source/ARTY_CM3/ARM/startup_ARTY_CM3.s*

Next, we need to add the BSP specific files. Create a new group named ***Standalone_v6_7*** and add the following files under it -

1. *standalone_bsp_0/Cortex_M3_0/libsrc/standalone_v6_7/src/print.c*
2. *standalone_bsp_0/Cortex_M3_0/libsrc/standalone_v6_7/src/outbyte.c*
3. *standalone_bsp_0/Cortex_M3_0/libsrc/standalone_v6_7/src/xil_assert.c*

Next, we need to add source files for the peripherals. For example, create a new group named ***Xilinx_UART*** and add all the files under the folder *standalone_bsp_0/Cortex_M3_0/libsrc/uartlite_v3_2/src*. The exact path would depend on the version of the peripherals. Repeat this procedure for all the required peripherals in the application project.

Manage Run-Time Environment:

Go to **Project >Manage >Run-Time Environment**. Under the *CMSIS* section, select the checkbox for the **CORE** option. Click OK. The final project items are shown in Figure 3.7. Now, we are ready to write the application code for our TFLite project.

Writing the Application code:

We need to modify 2 files - *main.cc* (application code) and *model.cc* (contains the model parameters/flatbuffer) under the group *Source*. We can simply replace the contents of the flatbuffer array in *model.cc* with the corresponding contents of our trained model file (*model.h*). In the file *main.cc*, we need to include the headers as follows -

```
#include "m3_for_arty.h"   // Platform specific
#include "xparameters.h"   // Identifying peripherals
#include "xuartlite.h"     // UART IP header file
#include "xil_printf.h"    // Using print statements
#include "xtmrctr.h"       // Timer IP header file
```

```
#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 Model file
#include "tensorflow/lite/micro/examples/hello_world/model.h"
```

Rest of the application code is same as explained in the section 2.2, with minor changes in the definition and initialization of the peripherals. For example, to initialize the UART peripheral, we write the following code -

```
// Declare UART IP instance
static XUartLite UART_instance;

// Initialize the UART peripheral
XUartLite_Initialize(&UART_instance, XPAR_AXI_UARTLITE_0_DEVICE_ID);
```

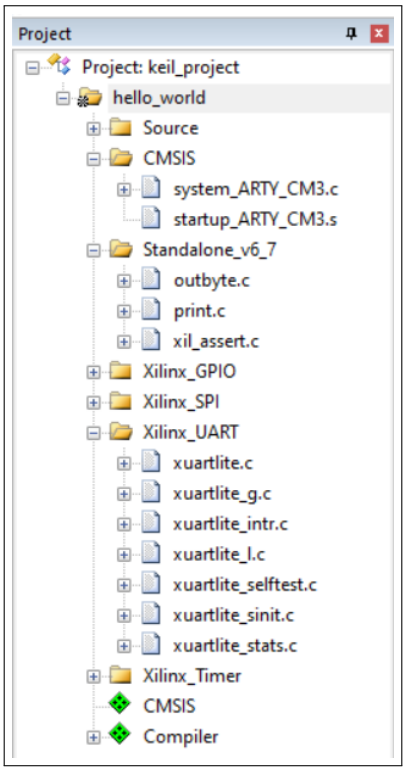


Figure 3.7: Keil Project Items

An example application code is provided at main.cc. Before we build the target, we are required to do some changes in few of the source files as mentioned below -

1. **flatbuffers.h**: This file is present under the *Source* group. We are required to find and comment the following section of code in this file -

```
//extern volatile __attribute__((weak)) const char *
    flatbuffer_version_string;
//volatile __attribute__((weak)) const char *
    flatbuffer_version_string =
//  "FlatBuffers "
//  FLATBUFFERS_STRING(FLATBUFFERS_VERSION_MAJOR) "."
//  FLATBUFFERS_STRING(FLATBUFFERS_VERSION_MINOR) "."
//  FLATBUFFERS_STRING(FLATBUFFERS_VERSION_REVISION);
```

   Failure to do so would result in an error like -
   ERROR: flatbuffers::flatbuffer_version_string Multiply defined Global Symbol

2. **xpseudo_asm_gcc.h**: This file is located in *standalone_bsp_0/Cortex_M3_0/include*. The *str(adr, val)* function defined in this file overlaps with one of the function defined in the TFLite source files. Thus, this function must be commented out in the file *xpseudo_asm_gcc.h*.

```
    /*#define str(adr, val)      __asm__ __volatile__(\
        "str     %0,[%1]\n"\
        : : "r" (val), "r" (adr)\
      )*/
```

3. **debug_log.cc**: This file is located under the *Source* group. We must modify the function *DebugLog* as follows. This function is later redefined in *main.cc* as shown in the example file.

```
extern "C" void __attribute__ ((weak)) DebugLog(const char* s){
   // To be implemented by user
}
```

Finally, we can build the target and obtain the **.elf** file. Note that *make_hex_a7.bat* is automatically run post-build and the generated files (.hex and .elf) are located in the current working directory of the Keil project.

3.5 Running the Application

As discussed earlier, we require a .mmi file to identify the program memory locations inside the bitstream. We can either manually write this file which is prone to human error, or we can use a helper script ***make_mmi_file.tcl*** to automatically generate this file. Also, to automate the process of merging the .elf file with the bitstream, we have a batch file ***make_prog_files.bat***. Both of these scripts come along with the Cortex-M3 package and are located under the hardware section. We need to copy 4 files - *make_mmi_file.tcl*, *make_prog_files.bat*, *make_prog_files.tcl* and the generated .elf file in the previous section (***bram_a7.elf***) into the vivado project directory, and make few changes inside them as discussed further.

Generating .mmi file:
Open the *make_mmi_file.tcl* file and edit the *set part* line to the correct part for your board. For CMOD A7-35T, the part is *xc7a35tcpg236-1*. In Vivado, click "Open Implemented Design", then in the TCL console, navigate to the project directory and run: *$ source make_mmi_file.tcl*. This would generate a MMI file named **m3.mmi**.

Update the Bitstream:
We need to edit the file *make_prog_files.tcl*. Open it and make the following changes -

1. Rename *source_bit_file* to *"./<runs folder >/impl_1/<Bit file name >"*
2. Rename *output_bit_file* to *"final.bit"*

We can also modify the names for MMI and ELF file, however this should be consistent with the previous sections. Also, add the path of bin folder for both Keil and Vivado in the system and user path environment variables, respectively. Finally, since all the required files are now available (*m3.mmi*, *bram_a7.elf* and bitstream), we can run the batch file *make_prog_files.bat*. This would generate the final updated bitstream which can later be downloaded on the FPGA board. Done!

3.6 Challenges

Since the current software development uses the Keil environment, this poses various challenges in the development process as follows -

1. **License Issue**: The free version of Keil-MDK has various limitations. For example, the programs that generate more than **32 KB** of code and data will not compile, assemble, or link. TFLite applications are generally much bigger (>100KB) and thus, they can't be compiled using the free version. For this report, we are using a 30-day free license of Keil.

2. **Complex process**: The development through Keil is a bit difficult as the IDE environment is non-intuitive. Also, it becomes very difficult to import and modify various source or header files inside the project. Navigation across the project is also non-trivial.

Working with **Eclipse**-based IDEs such as Vitis, SDK, etc. is much easier and faster. Also, there is no limitation in terms of the code size. We have tried our best to develop the present software using the Vitis/SDK environment, however, the board doesn't responds to any of our changes. I have even worked with low-level linker, startup files and have tried to modify the project for a GCC environment. The ARM-MDK uses the ARM compiler, whereas the SDK/Vitis environment uses the GCC toolchain. Although, I have also tried to use the ARM CC toolchain to compile the code in Vitis/SDK, yet there was no success. I am able to compile the code and generate the .elf file, however the final bitstream is not working. I think that the issue lies with the compiler and the linker flags.

This chapter was centered around the implementation of TFLite applications on the ARM Soft-IPs, which also allows the resource-constrained FPGA boards such as CMOD A7-35T to execute these applications. We can also use this process to develop and deploy an ARM-based software and thus, this can be used for the software prototyping and testing without the need of an actual hardcore ARM processor. Although the current IP has few limitations, it gives us a lot of opportunities to develop various interesting applications.

Chapter 4

Implementation on MicroBlaze

The last chapter introduced the advantages of using an FPGA for developing machine-learning applications and the implementation on a ARM soft CPU. Since the TFLite library that we created during the build process in section 2.1 is target-neutral, it can be easily imported and used with the other target architectures. One such platform is the Xilinx's soft-core processor, MicroBlaze which also overcomes many of the existing limitations of the ARM soft-IPs and the development process. Rest of the chapter focuses on the implementation of TFLite applications on MicroBlaze.

4.1 Introduction to MicroBlaze

The MicroBlaze CPU is a family of drop-in, modifiable preset 32-bit/64-bit Harvard RISC microprocessor architecture optimized for implementation in Xilinx FPGAs [9]. It has 32-bit instruction set and general purpose registers with 32-bit address bus, extensible to 64 bits. MicroBlaze is a highly configurable processor. The basic design can be user configured with more advanced features such as barrel shifter, divider, multiplier, single precision floating-point unit (FPU), instruction and data caches, exception handling, debug logic, Fast Simplex Link (FSL) interfaces and others. With few exceptions, the MicroBlaze can issue a new instruction every cycle, maintaining single-cycle throughput under most circumstances.
This flexibility allows the user to balance the required performance of the target application against the logic area cost of the soft processor. Although, the overall throughput of MicroBlaze is substantially less than a comparable hard CPU core (such as the ARM Cortex-A9 in the Zynq), the developer can make the appropriate design trade-offs for a specific set of host hardware and application software requirements. Since the MicroBlaze is a soft-core microprocessor, any optional fea-

tures which are not used will not be implemented and thus, will not take up any of the FPGA resources. Figure 4.1 shows a functional block diagram of the MicroBlaze core. MicroBlaze can be used as a stand-alone processor in all Xilinx FPGAs or as a co-processor in a various SoC systems such as Zynq. It is commonly used in one of three preset configurations as shown in the table below: a simple micro-controller running bare-metal applications; a real-time processor featuring cache and a memory protection unit interfacing to tightly coupled on-chip memory running FreeRTOS; and finally, an application processor with a memory management unit running Linux. The table below shows the performance and utilization estimates for these configurations -

	Microcontroller	**Real-Time**	**Application**
Max Clock freq. (MHz)	204	172	146
Logic Cells	1900	4000	7000

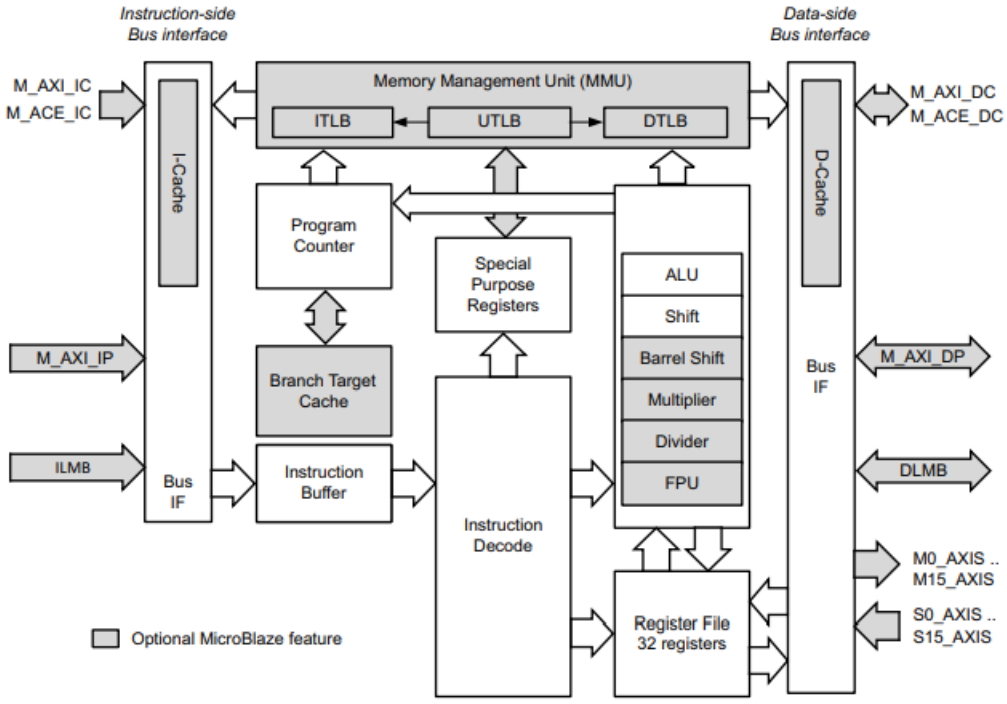


Figure 4.1: MicroBlaze Core Block Diagram (Source [10])

Since we require to run a bare-metal TFLite application, we would be using the Microcontroller variant. More details regarding the hardware and software configuration would be discussed in the further sections.

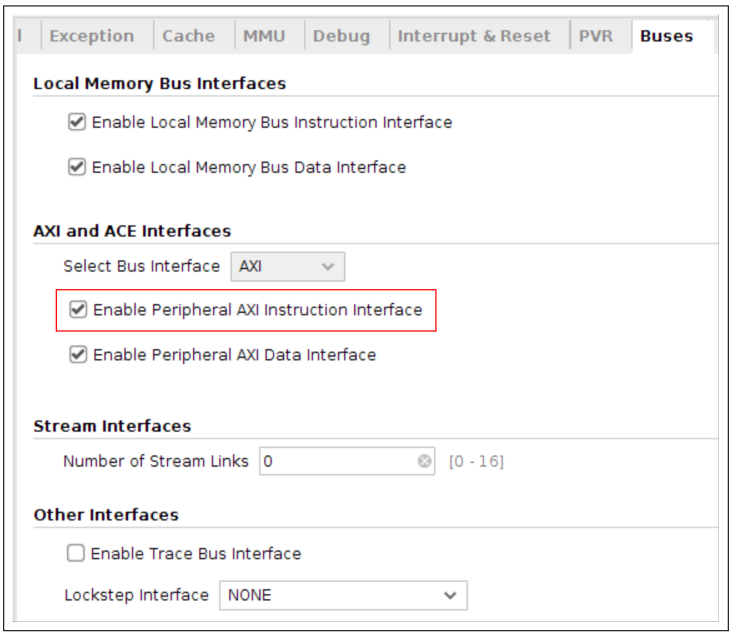
4.2 Hardware Configuration & Bitstream Generation

The hardware configuration would be similar to section 3.2. However, in this case, we would be using the MicroBlaze soft IP as the CPU. We are not required to download any IP package as microblaze is fully supported by Vivado and SDK/Vitis.

Open the Vivado Design suite and create a new project for the CMOD A7-35T board. Further, we need to create a new block design and add the MicroBlaze IP. Double clicking on this IP opens up the configuration window. Since, we want the most performance-optimized version of the MicroBlaze, we need to enable various options in the *General* tab including the Barrel shifter, FPU (Extended), Integer multiplier, etc. as shown in the Figure 4.2. A few more changes in the microblaze configuration are listed as follows. Note that the following changes are specific to the CMOD A7-35T board. For example, we may not require an instruction cache if enough amount of BRAM is available for code memory.

Enable AXI Instruction Interface:

The TFLite applications would not fit in the limited amount of BRAM memory (225KB) available on the CMOD A7-35T board. Thus, we require some external memory device to hold the .text section of our code memory. Fortunately, we have an external SRAM (512KB) available on the board. To access the instructions from this external memory, MicroBlaze requires a peripheral AXI instruction interface and thus, this needs to be enabled as shown in the following figure.



Enable Instruction & Data Cache:

External memory devices are slower and would result in higher latency while fetching instructions. Thus, we should enable the **Instruction cache** of microblaze. It uses the internal BRAM as the cache, thus the access latency is close to 1 cycle. The base and high address in the cache must be carefully set to that of the external SRAM.

Data cache plays a vital role while accelerating the kernels using the Custom IPs. The ILMB and DLMB memory are private to the microblaze and hence, their data can't be accessed by any other external peripheral or IP. However, we can add a shared memory location (BRAM) which can be accessed by both microblaze and the custom IPs. This step should be applicable for all the FPGA boards which require custom IPs for the acceleration.
First, we add the AXI BRAM Controller IP and a Block Memory Generator(BRAM) in the block design. Running connection automation should automatically connect this BRAM with the AXI BRAM controller. Further, this AXI BRAM controller IP is manually connected with the *M_AXI_DC* port of the microblaze. Note that the base and high address in the data cache of microblaze must be carefully set to that of this BRAM address. The microblaze cache configuration is shown in Figure 4.3.

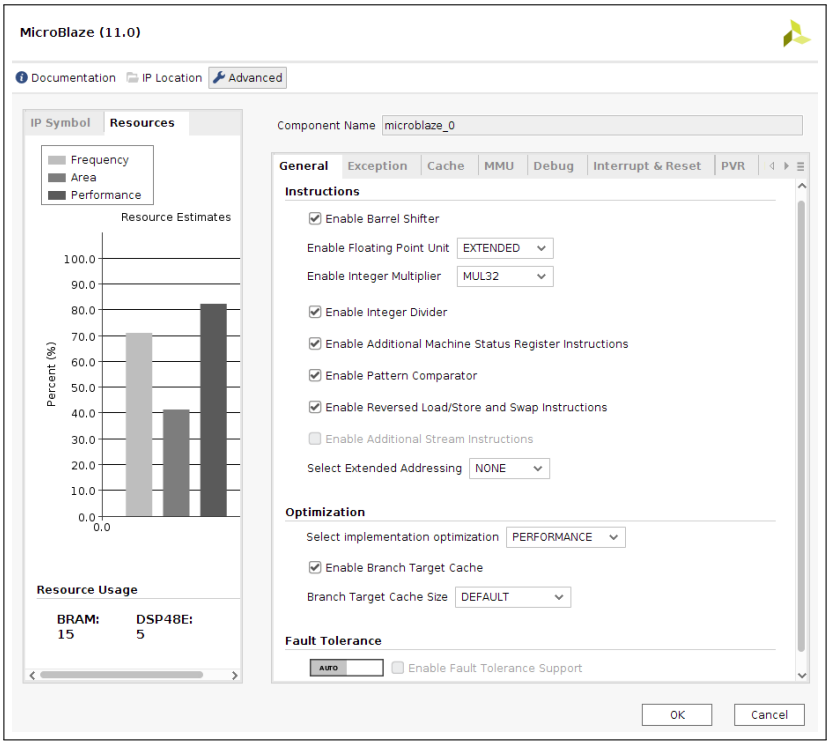


Figure 4.2: MicroBlaze configuration

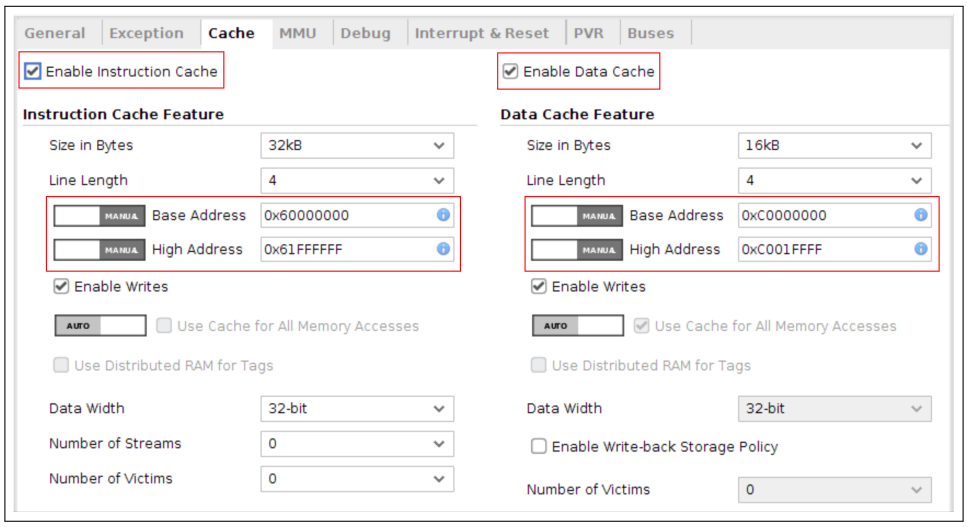


Figure 4.3: Cache configuration in MicroBlaze

Till here, the microblaze connections should look similar to the Figure 4.4. The above changes would increase both the **performance** and the **resource-utilization** of the processor. Finally, we can add the required peripheral IPs in our block design. For this discussion, I have created a basic design which consists of the external SRAM and UART peripheral. Running the *block* and *connection automation* in vivado would automatically connect these to the microblaze using the AXI Interconnect bridge. The *M_AXI_DP* (peripheral data port), *M_AXI_IP* (instruction port) and *M_AXI_IC* (instruction cache) are also connected to the AXI interconnect. The final block diagram is shown in Figure 4.5.

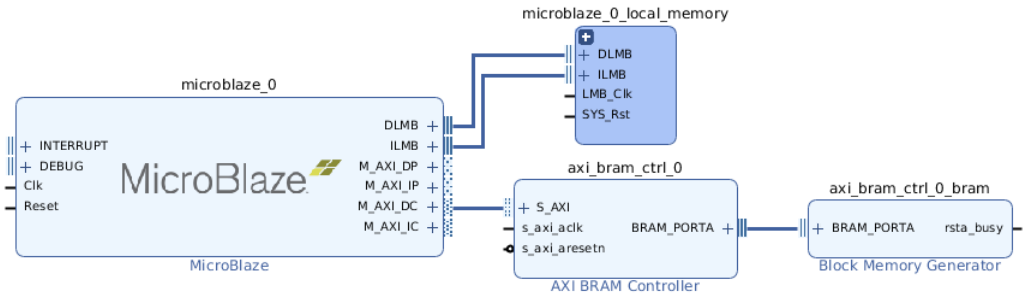


Figure 4.4: Data cache connection in MicroBlaze

Further, we can create the top-level HDL wrapper for our design and generate the bitstream. Next, we would develop the software using Xilinx SDK for running the TFLite applications on microblaze.

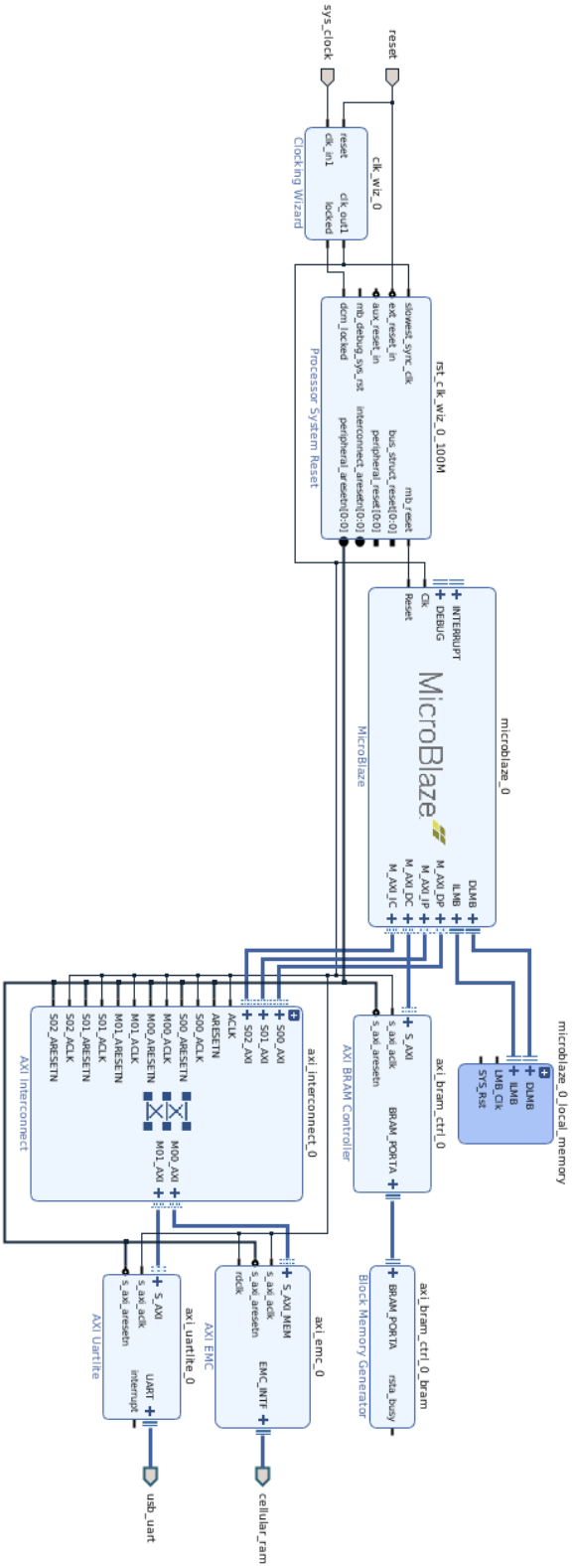


Figure 4.5: Block Design for MicroBlaze

4.3 TFLite Software Development

Since MicroBlaze is fully supported by the SDK/Vitis, we can directly create an application project for our hardware platform. This would auto-generate the required board support package (BSP) files. The implementation workflow is as follows -

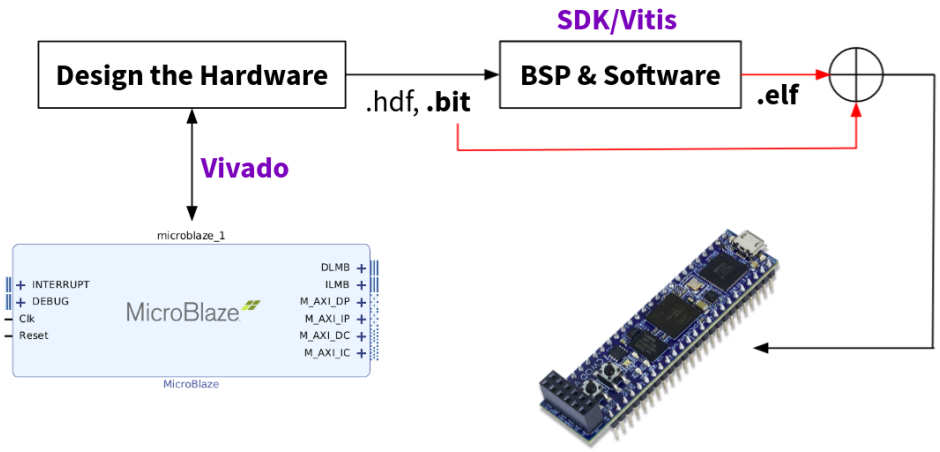


Figure 4.6: Implementation workflow for MicroBlaze application development

Create a C++ application project in SDK/Vitis using the exported hardware definition file. Since SDK/Vitis is an eclipse-based IDE, this project would be configured similar to section 2.2. Copy the flatbuffer file *model.h* into *<sdk_project>/src*. Also, create a new folder *tensorflow_lite* under *<sdk_project>* and copy the TFLite source folders (*tensorflow* & *third_party*) into this. A sample file structure is shown in Figure 4.7. Next, we must specify the include directories in our SDK project as shown in Figure 4.8. Source location for TFLite is already specified due to the file structure in SDK project (no changes required). Don't forget to delete the DSP and CMSIS-NN related files as these aren't supported by microblaze. Finally, we can write our TFLite application code similar to section 2.2, with minor changes in the definition and initialization of the peripherals (discussed in previous chapter). An example application code is provided at main.cc.

Before we compile our application, we need to generate the **Linker** file for mapping different code sections at appropriate memory locations. Select the application project and go to **Xilinx >Generate linker script**. Specify the location of Code sections (SRAM), Data and Heap sections(BRAM). Click Generate and compile the application to obtain the .elf file. To reduce the code memory, the Optimization setting in C++ build can be selected as *Optimize for size (**-Os**)*.

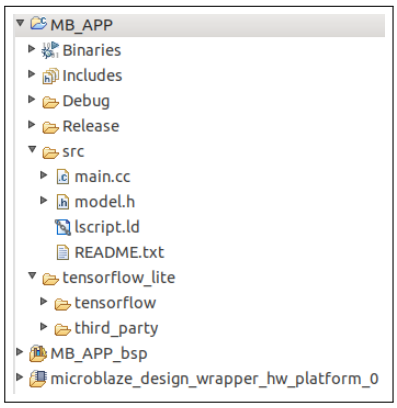


Figure 4.7: SDK Project file structure

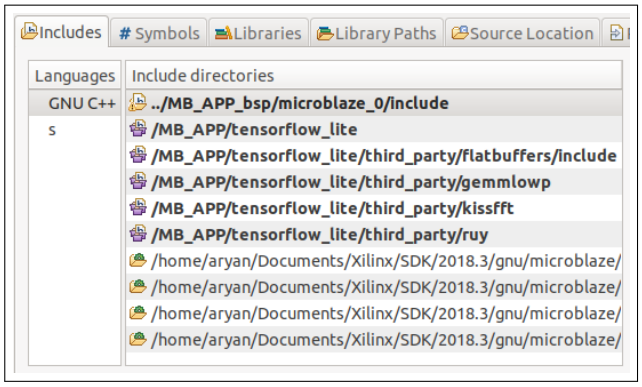


Figure 4.8: SDK Include directories

4.4 Acceleration using Custom IPs

The FPGA fabric allows us to develop custom hardware accelerators for accelerating our application and improving the run-time. We can offload few data-intensive operations from the microblaze and execute them via the implemented custom IPs. One such operation could be the Fully connected layer operation as discussed in section 2.3.1. In this example, we intend to accelerate the matrix multiplication stage in the fully connected layer using a custom IP. The first step would be to synthesize and implement this IP, and later integrate it with the current hardware design.

4.4.1 Creation of Custom IP using Vivado HLS

The Xilinx Vivado High-Level Synthesis (HLS) tool transforms a C specification into a register transfer level (RTL) implementation which we can synthesize into a Xilinx FPGA. We can control the C-synthesis process using various optimization pragmas to create high-performance implementations. This includes pipelining loops, unrolling, etc. Thus, the hardware designers can work at a higher level of abstraction while creating high-performance hardware.

I have designed a custom IP which evaluates the dot product of 2 arbitrary-sized vectors, after adding a constant offset. This IP would be later used in the integer TFLite kernel. For accelerating the process and improving parallelism, the multiply and accumulate stage is implemented in the form of an ***Adder tree*** as shown in the Figure 4.9. The outermost loop is unrolled 8-times. This would highly increase the resource-utilization while improving the operation latency.

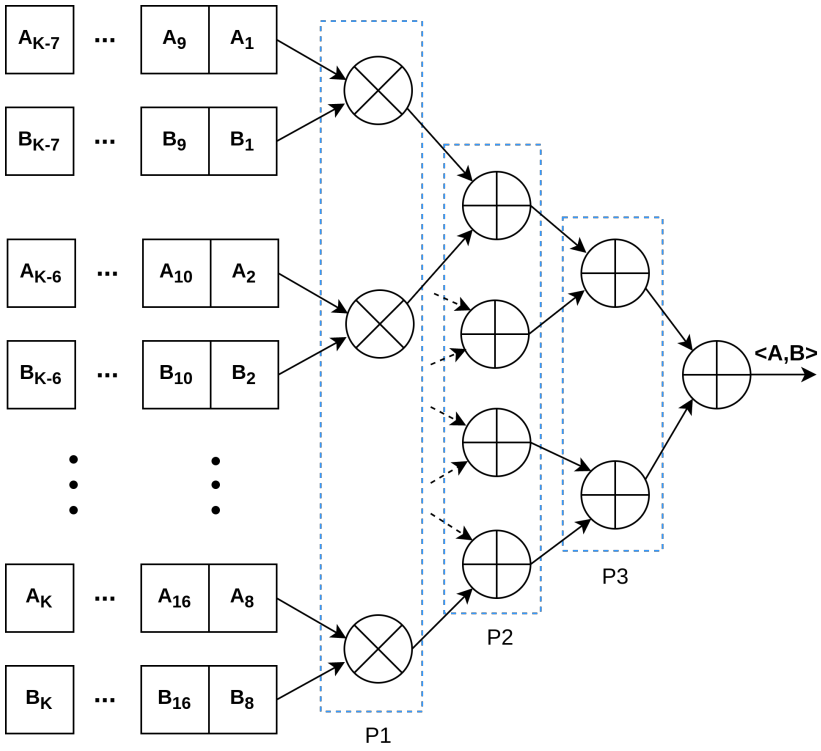


Figure 4.9: Multiply & Accumulate stage in the Custom IP

In HLS, the above functionality can be implemented as follows -

```
// inp -> Input vector
// wt -> Weight vector

for(int j=0; j<8; j++){
  #pragma HLS UNROLL factor=8
  mult[j] = inp[j] * wt[j];
}
for(int j=0; j<4; j++){
  add1_arr[j] = mult[2*j] + mult[2*j + 1];
}
for(int j=0; j<2; j++){
  add2_arr[j] = add1_arr[2*j] + add1_arr[2*j + 1];
}
acc += add2_arr[0] + add2_arr[1];
```

These individual arrays are partitioned in memory, so that multiple elements can be accessed simultaneously. Using HLS, we can easily define the type of interface for

the individual operands using the pragma *HLS INTERFACE*. The 2 input vectors are read from a memory-mapped AXI interface (***m_axi***). Thus, we can treat these inputs as array pointers. Rest of the inputs (input offset, weight offset and size of vector) are declared as AXI4-Lite interface (***s_axilite***) as shown below -

```
int FullyConnected(signed char* input,
                   signed char* weight,
                   int input_offset,
                   int weight_offset,
                   int accum_depth)
{
#pragma HLS INTERFACE m_axi depth=200 port=input offset=slave
    bundle=MASTER_BUS
#pragma HLS INTERFACE m_axi depth=200 port=weight offset=slave
    bundle=MASTER_BUS
#pragma HLS INTERFACE s_axilite port=input_offset bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=weight_offset bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=accum_depth bundle=CRTL_BUS
#pragma HLS INTERFACE s_axilite port=return bundle=CRTL_BUS
 ....
}
```

The **bottleneck** here is the memory-access! We can't read multiple elements (input and weight vector) from the data memory of microblaze simultaneously as it has only a single read port. Nevertheless, rest of the operations are now much faster. The complete HLS code is provided at fullyconnected.cpp.
Once the design is finalized, we can synthesize and implement this Custom IP. Timing violations must be checked, if any. Table 4.1 shows the resource-utilization for this IP. Finally, we can export the RTL to integrate it in our Vivado project.

BRAM_36K	DSP	Flip-Flop	LUT
1	24	2661	3988

Table 4.1: Resource utilization for the Custom IP (Fully Connected)

4.4.2 Integration in Vivado

Open the microblaze vivado project and add the IP repository of the fully connected module in the project settings. This allows vivado to identify and import our custom

IP in the block design. Now, add this Custom IP to our block design.

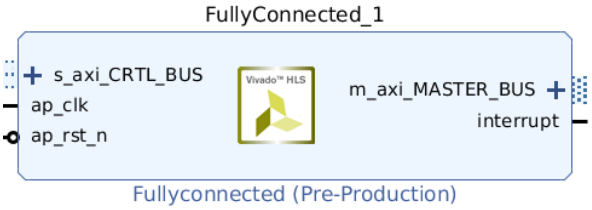
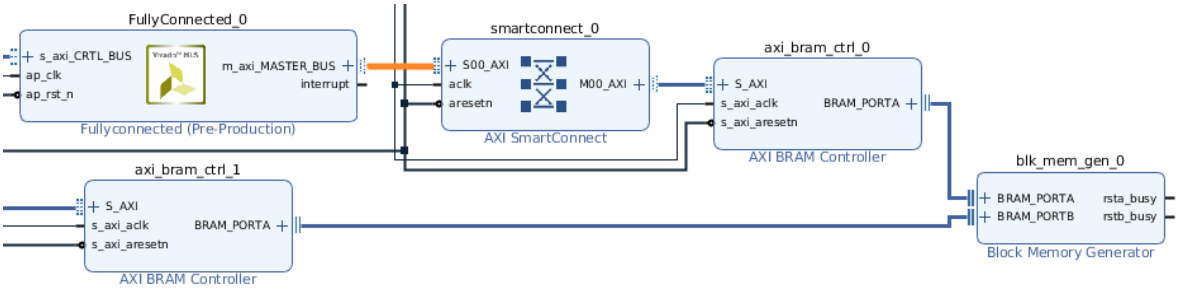


Figure 4.10: Fully Connected IP

Since we want to share and access the data memory of microblaze, we need to increase the BRAM ports of the microblaze data memory. Double click on the corresponding BRAM and select the *Memory Type* as *True Dual Port RAM*. Thus, one of the port will be accessed by the microblaze and the other would be accessed by various custom IPs in the design.
To connect the fully connected IP with the data BRAM, we need to include 2 IPs in our block design: AXI SmartConnect and AXI BRAM Controller. The memory-mapped interface of the custom IP would be connected to the slave interface of AXI SmartConnect, whose master port is further connected with the AXI BRAM controller. This is again connected with the other BRAM port. The final connection should appear as follows -



If the design consists of more than one Custom IP, then we can simply increase the number of slave ports of the AXI SmartConnect, and connect them with the individual custom IPs. More number of custom IPs would reduce the available data bandwidth and might slow down the data access. Once the design is validated, we can generate the bitstream and export the hardware.

4.4.3 Modifying the TFLite Kernel

The SDK project must be upgraded with the newly generated hardware description file in the previous section. Before using the custom IP in our application, we are first required to initialize it as follows -

```
#include "xparameters.h"

// Instance of the Custom IP
XFullyconnected xfc;
XFullyconnected_Config *xfc_cfg;

xfc_cfg = XFullyconnected_LookupConfig(XPAR_FULLYCONNECTED_0_DEVICE_ID);
if(xfc_cfg){
   int status = XFullyconnected_CfgInitialize(&xfc, xfc_cfg);
   if(status != XST_SUCCESS)
      return XST_FAILURE;
}
```

Now, we will make changes in the integer TFLite kernel which is located at - *<sdk_project>/tensorflow_lite/tensorflow/lite/kernels/internal/reference/integer_ops/* ***fully_connected.h***.

By default, the kernel code uses normal *for* loops for computation as follows -

```
for (int b = 0; b < batches; ++b) {
    for (int out_c = 0; out_c < output_depth; ++out_c) {
      int32_t acc = 0;
      for (int d = 0; d < accum_depth; ++d) {
        int32_t input_val = input_data[b * accum_depth + d];
        int32_t filter_val = filter_data[out_c * accum_depth + d];
        acc += (filter_val + filter_offset) * (input_val + input_offset);
      }
      if (bias_data) {
        acc += bias_data[out_c];}
      acc = MultiplyByQuantizedMultiplier(acc, output_multiplier,
          output_shift);
      acc += output_offset;
      acc = std::max(acc, output_activation_min);
      acc = std::min(acc, output_activation_max);
      output_data[out_c + output_depth * b] = static_cast<int8_t>(acc);
    }
}
```

The value of variable *batches* is generally 1. Thus, we can ignore this variable for simplifying the loop. Instead of iterating over each element, we can perform the above operation using our custom IP as follows -

```
// Set Inputs
XFullyconnected_Set_input_r(&xfc, (u32)input_data);
XFullyconnected_Set_input_offset(&xfc, input_offset);
XFullyconnected_Set_weight_offset(&xfc, filter_offset);
XFullyconnected_Set_accum_depth(&xfc, accum_depth);

for (int b = 0; b < batches; ++b) {
    for (int out_c = 0; out_c < output_depth; ++out_c) {

      XFullyconnected_Set_weight(&xfc, (u32)(filter_data + (out_c *
          accum_depth)));

      XFullyconnected_Start(&xfc);            // Start the computation
      while(!XFullyconnected_IsDone(&xfc)); // Wait till over

      int32_t acc = XFullyconnected_Get_return(&xfc);  // Get Output

      if (bias_data) {
        acc += bias_data[out_c];
      }
      acc = MultiplyByQuantizedMultiplier(acc, output_multiplier,
          output_shift);
      acc += output_offset;
      acc = std::max(acc, output_activation_min);
      acc = std::min(acc, output_activation_max);
      output_data[out_c + output_depth * b] = static_cast<int8_t>(acc);
    }
}
```

Thus, we can create our own custom IPs and integrate them further in the hardware design to accelerate the TFLite applications.

This chapter was centered around the implementation of TFLite applications on MicroBlaze, which is a highly configurable soft processor. It also sheds light on the usage and implementation of custom hardware accelerators which can drastically reduce the application run-time. An easier development process and high amount of flexibility makes the MicroBlaze favourable for developing TFLite applications on resource-constrained FPGAs.

Chapter 5

Performance Analysis

The previous chapters discussed about the implementation of TFLite framework on various platforms such as STM32 boards, ARM soft-IP and MicroBlaze. We can now compare the performance obtained on these platforms and the corresponding trade-offs. For this comparison, we require few benchmark models or applications via which we can compare the run-time latency. I have trained and tested 2 NN models: ***Sine*** (computes the sine value of a number), and the well known ***MNIST*** classification. The later model is also quantized to experiment with various optimization techniques. The resource-utilization in FPGAs are also reported.

5.1 Floating-precision Models

Sine Model:

A simple model which computes the sine value of an input number. This model has **321** parameters and 3 fully connected layer with the following architecture -

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
dense (Dense)                (None, 16)                32
_________________________________________________________________
dense_1 (Dense)              (None, 16)                272
_________________________________________________________________
dense_2 (Dense)              (None, 1)                 17
=================================================================
Total params: 321
Trainable params: 321
Non-trainable params: 0
```

The hidden layers have ReLU activation and the model is trained with a simple loss function based on least-square error. The corresponding flatbuffer size (*model.h*) is ≈

3KB. This model was implemented on the STM32 boards (Disco Cortex-M7), ARM Cortex-M3 soft IP and the MicroBlaze, with the corresponding run-time obtained as follows -

	Disco (Cortex-M7)	**ARM IP (Cortex-M3)**	**MicroBlaze**
Clock freq. (MHz)	100	50	100
Inference Time (μs)	**140**	**512**	**113**

Table 5.1: Inference timings for Sine model (Default kernels)

Note that here the inference engine uses the default floating-precision kernels.

DSP Acceleration in MNIST Model:

This model performs the classification of numbers in the MNIST dataset (28x28 images) with a test accuracy of **97.13%**. This model has **50890** parameters and 2 fully connected layer with the following architecture -

```
_________________________________________________________________
Layer (type)                 Output Shape              Param #
=================================================================
flatten (Flatten)            (None, 784)               0
_________________________________________________________________
dense (Dense)                (None, 64)                50240
_________________________________________________________________
dense_1 (Dense)              (None, 10)                650
=================================================================
Total params: 50,890
Trainable params: 50,890
Non-trainable params: 0
```

The hidden layer has ReLU activation and the output layer has Softmax activation. This model is implemented on the STM32 boards (Disco Cortex-M7) and the fully connected kernel is also accelerated with the DSP instructions. Following are the results obtained -

	w/o DSP	**w DSP**
Inference Time (μs)	**11238**	**9332**

Table 5.2: Inference timings for MNIST model on Disco (Cortex-M7) board

The corresponding flatbuffer (*model.h*) size is $\approx$ **200KB**. This model is not implemented on the ARM soft-IP or MicroBlaze due to the limited memory available on the CMOD A7-35T board.

5.2 Integer Models

The above MNIST model is quantized so that all the model parameters can be expressed using 8-bit signed integers. As a result, the corresponding flatbuffer (*model.h*) size reduces to $\approx$ **52KB**. The model accuracy on the test data is **97.16%**.
This model is implemented on the STM32 boards (Disco Cortex-M7) and MicroBlaze, with the corresponding run-time using integer default kernels obtained as follows -

	Disco (Cortex-M7)	**MicroBlaze**
Inference Time (μs)	**15777**	**8674**

Table 5.3: Inference timings for Quantized MNIST model (Default kernels)

Again, this model is not implemented on ARM soft-IP due to shortage of memory. Further, the fully connected kernels are optimized using CMSIS-NN library in Disco Cortex-M7 board, and by using Custom IPs in MicroBlaze. The corresponding run-time is reported as follows -

	Disco (Cortex-M7)	**MicroBlaze**
Optimization	CMSIS-NN	Custom IP
Inference Time (μs)	**2390**	**5007**

Table 5.4: Inference timings for Quantized MNIST model (Optimized kernels)

As evident, the CMSIS-NN kernel available for Cortex-M cores is faster than the other alternatives, which on the other hand, are much more flexible and cheaper.

5.3 Resource Utilization

Both the ARM Cortex-M3 soft IP and MicroBlaze are implemented on the Artix-7 FPGA. The corresponding resource utilization for these processors is reported as follows -

	LUT	**Flip-Flop**	**BRAM_36K**	**DSP**
Cortex-M3 IP	13261	4492	48	3
MicroBlaze	3082	2446	47	5

Table 5.5: FPGA Resource utilization by soft processors

The Memory footprint of the final generated .elf file is also an important indicator of the memory consumption. These are reported as follows -

	text	**data**	**bss**	**dec**
Disco (Cortex-M7)	43768	3484	4248	51500
MicroBlaze	171784	4480	6752	183016

Table 5.6: Memory footprint for Sine model implementation

	text	**data**	**bss**	**dec**
Disco (Cortex-M7)	55896	53568	53504	162968
MicroBlaze	195232	54576	15024	264832

Table 5.7: Memory footprint for MNIST model implementation (integer)

As evident, MicroBlaze code consumes a lot more memory as compared to the hardcore Cortex-M7 counterpart. This is also one of the cons for using microblaze. This difference can be attributed to the fact that these processors follow different ISA. The compiler also plays a definite role in reducing the code memory (MicroBlaze compiler vs GCC ARM toolchain). These design constraints and performance trade-offs must be considered while selecting an appropriate hardware.

Chapter 6

Applications

The final goal is to demonstrate an application on suitable hardware by utilizing the methodologies discussed in the previous chapters. Before this, we need to benchmark a problem statement, solving which might have real-life implications.

These applications can range from the development of energy-efficient smart devices to real-time machine health monitoring, visual assistance, cognitive controls, maneuvering, etc. These small low-power devices can be installed at the critical junctions of big machinery and can be helpful in detecting early failures or catastrophes. FPGA is a very suitable candidate for the development of Speech processing-based technologies. This opens up a new door for the development of offline-processing based devices, which also perform better in terms of latency and complete privacy. Recently, there is also a huge surge in the domain of Augmented virtual reality, where the usage of these devices can make the overall product cheaper and more energy-efficient. A lot of machine-learning applications are arising in the field of Agro-tech. Many pest control companies are using them to identify the various bacteria, bugs, and vermins. It can also identify which conditions will produce the best yield, thus boosting the yield of crops. Since they consume very little power, they can be operated using solar energy, and thus even the farmers can directly use it. Machine learning is also paving its way in bioinformatics and a lot of opportunities exist in the field of genomics. With such small hand-held devices, we can even perform genome sequencing for RT-PCR tests! Nanopore sequencing can also be targeted using these devices. This is still a growing field and we hope that a lot of interesting work can be done in this area.

The immediate future task of this project is to discuss and select a suitable application. We are currently talking with various experts in different fields to get an idea of what might be done using the developed methodologies.

Bibliography

[1] R. Sanchez-Iborra and A. F. Skarmeta, "Tinyml-enabled frugal smart objects: Challenges and opportunities," *IEEE Circuits and Systems Magazine*, vol. 20, no. 3, pp. 4–18, 2020.

[2] P. Crum, "Hearables: Here come the: Technology tucked inside your ears will augment your daily life," *IEEE Spectrum*, vol. 56, no. 5, pp. 38–43, 2019.

[3] R. David, J. Duke, A. Jain, V. J. Reddi, N. Jeffries, J. Li, N. Kreeger, I. Nappier, M. Natraj, S. Regev, *et al.*, "Tensorflow lite micro: Embedded machine learning on tinyml systems," *arXiv preprint arXiv:2010.08678*, 2020.

[4] *https://www.tensorflow.org/lite/microcontrollers/build_convert#model_conversion*.

[5] *https://www.tensorflow.org/lite/performance/post_training_quantization*.

[6] *https://www.st.com/en/microcontrollers-microprocessors/stm32-32-bit-arm-cortex-mcus.html*.

[7] L. Lai, N. Suda, and V. Chandra, "Cmsis-nn: Efficient neural network kernels for arm cortex-m cpus," *arXiv preprint arXiv:1801.06601*, 2018.

[8] *https://www.arm.com/resources/free-arm-cortex-m-on-fpga*.

[9] *https://www.xilinx.com/products/design-tools/microblaze.html*.

[10] *https://www.xilinx.com/support/documentation/sw_manuals/xilinx2021_1/ug984-vivado-microblaze-ref.pdf*.