Implementation and Evaluation of an Embedded AI System on a Resource-Constrained Platform

Nezar Sanad

University of Twente

Abstract: There is increasing interest in deploying miniature AI systems on the edge, lessening reliance on cloud computing and improving security and latency. These systems are also increasingly being used in applications where high reliability and radiation resistance are a necessity, hence the need for studying the reliability of such systems. The current design flow for edge AI accelerators tends to involve the development of AI systems using the Xilinx design framework. SRAM-based FPGAs such as those from Xilinx are not ideal for such studies, as Flash-based FPGAs are inherently better suited than SRAM-based FPGAs to such environments and can provide radiation resistance at much more suitable prices. This necessitates the creation of a suitable system on a flash-based FPGA to assess its performance when subjected to reliability testing. Such a system is created on a Microchip SmartFusion 2 FPGA, incorporating a model built using the Conifer framework, which is designed to convert trained boosted decision trees into extremely low latency FPGA firmware. An adapter was designed to allow models produced by Conifer to interface with the ARM AMBA standard busses. This allows easy integration into microcontroller-based systems, allowing the ML model to interface with any memory on the bus matrix while being software-controllable. This results in a highly adaptable system to facilitate reliability testing of a flash FPGA-based AI system along with its internal and external memories.

1. Introduction

The increasingly widespread use of AI has also affected embedded systems, some of which operate in applications where reliability is of paramount importance. While most FPGA AI research is conducted on high-power FPGAs designed to be used in data centers, this work will focus on implementing and evaluating an AI accelerator on a low-cost and low-power flash FPGA meant for applications where reliability is a necessity. This work will aim to provide a tool that can be used to easily implement embedded AI systems on similarly sized devices.

2. Design Considerations and Background

The original goal was to merely implement an AI accelerator in a flash-based FPGA and perform tests on the effects of using the RAM embedded in the FPGA fabric (BRAM) versus using the external DRAM. This was to be done with the goal of creating an embedded AI system on a flash-based FPGA, That would be subject to reliability tests by exposing the FPGA to directed radiation. The most common type of FPGA is SRAM-based. SRAM FPGAs, primarily those from Xilinx and Altera FPGAs dominate the market, with Microchip and Lattice having smaller shares. What makes the FPGAs manufactured by Microchip unique is that they are flash-based FPGAs. This means that the configuration data of these FPGAs is stored in flash cells, instead of SRAM. This allows for much faster startup times than in SRAM-based FPGAs. Another advantage of this type of FPGA is that it is more reliable. SRAM FPGAs could encounter configuration errors as the configuration data is loaded onto it from external non-volatile memory. The most important feature in this context however is their inherent resistance to radiation, flash cells are much more resistant to radiation, as they are immune to single-event upsets [1]. These features make flash-based FPGAs particularly attractive in applications such as the defense and aerospace industries where reliability is crucial. The increasingly widespread use of AI, however, brings an interesting

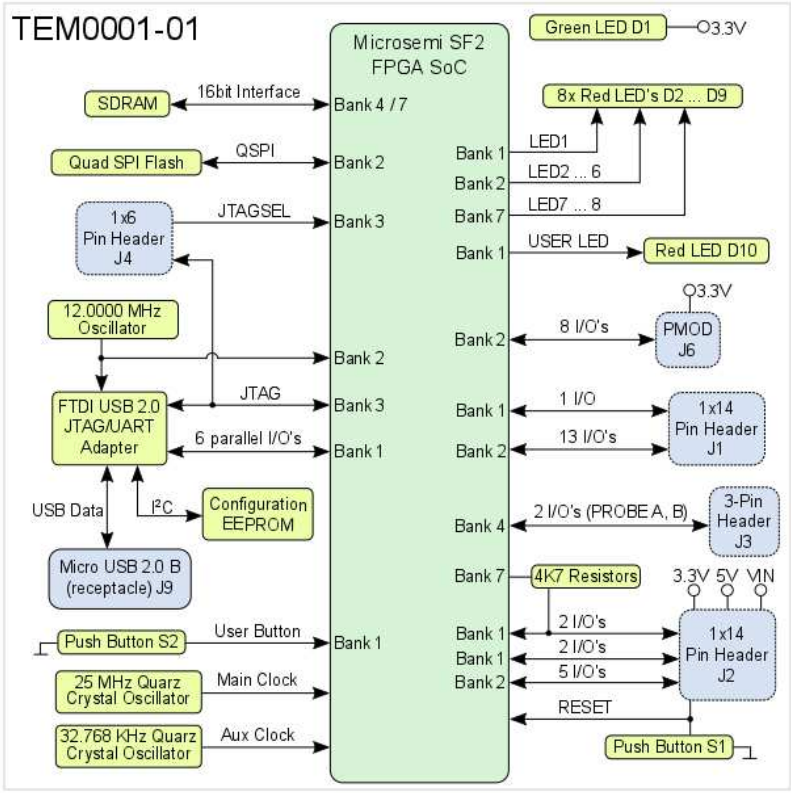

Fig. 1. Block Diagram of the TEM0001 FPGA board [2]

question, what would the effects of radiation and its impact be on an FPGA-based AI system be? What impact would using different memories have? A flash-based FPGA would be ideal for this scenario as it already possesses resistance to radiation and sees use in applications where reliability matters most.

2.1. Target Device

The target device is a Microchip SmartFusion2 M2S010. This FPGA is embedded in the TEM0001 board manufactured by Trenz Electronics. The board includes external memories, oscillators, headers, LEDs, and an FTDI JTAG chip for communication between the FPGA and the computer. External RAM is included, in the form of an 8 Megabyte SDRAM chip (Winbond W9864G6JT). External flash is also included in the form of an 8 Megabyte flash chip (Winbond Serial Flash Memory W74M64FV) with a QSPI interface [2]. The included FPGA is part of Microchip's SmartFusion2 family. These SoC FPGAs are marketed as low-power, highly secure, and exceptionally reliable devices, fit for communications, industrial, defense, and automotive use. These devices include a built-in Arm Cortex-M3 microcontroller equipped with embedded flash and SRAM, a DDR2/3 RAM controller in addition to a plethora of peripherals. This processor is capable of running at clock speeds up to 166MHz. The FPGA fabric on these devices consists of up to 150K logic elements and up to 5Mb of SRAM [3]. The physical layout of a SmartFusion2 FPGA can be seen in Figure 2.

What makes these devices particularly interesting is their high resilience to radiation. They are marketed as being immune to single event upsets (SEUs), which are single state changes inflicted by ionizing radiation. SmartFusion2 SoCs are also marketed as having a failure in time (FIT) rate of zero [3]. This makes the SmartFusion2 SoC an ideal candidate for testing the reliability of an

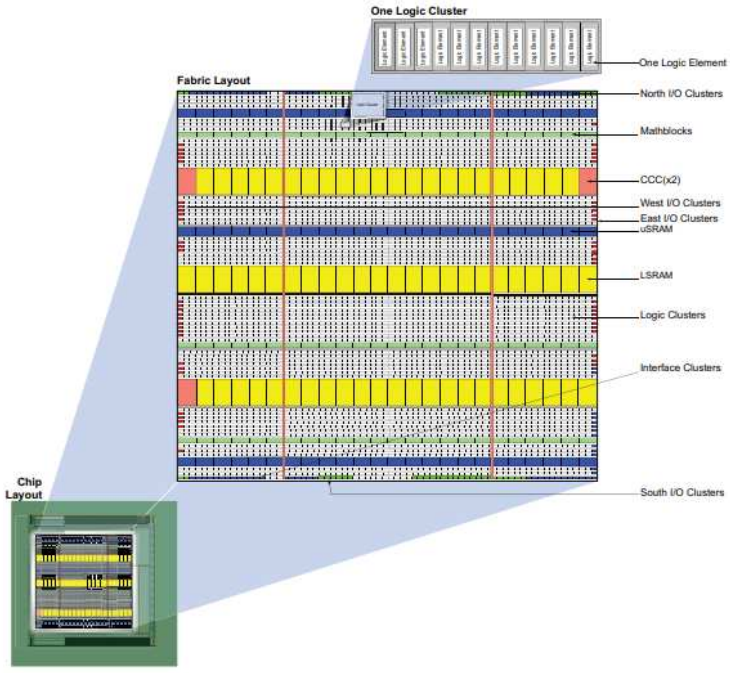

Fig. 2. Fabric architecture of a SmartFusion2 device [4]

embedded AI system on an FPGA. The particular version of this device used for this research is the M2S010 which contains 12084 logic elements, 22 math blocks, 2 PLLs, 21 18Kb SRAM blocks, 22 1Kb SRAM blocks, 64Kb of embedded SRAM, and 256Kb of embedded non-volatile memory (eNVM). The exceedingly low amount of logic elements will pose a particularly serious design constraint as AI accelerators tend to require large amounts of logic. Most of the difficulty will lie in overcoming this issue.

3. Design Tools and Methods

3.1. Libero

The main tool for this undertaking is the Libero design suite. This is the tool provided by Microchip to facilitate the development of hardware on their FPGAs. Libero provides a place to edit HDL files, compose projects, and connect HDL entities together with Microchip IP blocks and other HDL entities. Libero is more a bundle of tools then it is a single tool. The design flow within it contains either ModelSim or QuestaSim to allow for pre-synthesis, post-synthesis, and post-implementation simulation of the design. Synthesis is carried out by a special version of Synopsys Synplify, Synplify Pro Microchip Edition. This tool also does the placing and routing to map the given design to the primitives on the FPGA (like LUTs and DFFs) and determines the routing between them. Once the implementation of the HDL is created, the bitstream can then be flashed onto the FPGA using the tool provided by Microchip (FlashPro) either within Libero itself or, by using the standalone tool. Debugging of the on-chip flash, RAM, and flip flops is done using SmartDebug. Finally, Libero has a useful tool that outputs the hardware abstraction layer and any necessary drivers for used IP blocks, to be used in developing the firmware running on the built-in processor.

3.2. QuestaSim

Libero users have choice between two different software for simulation, ModelSim and QuestaSim. Both are developed by Mentor Graphics and are quite similar. ModelSim is the older and cheaper

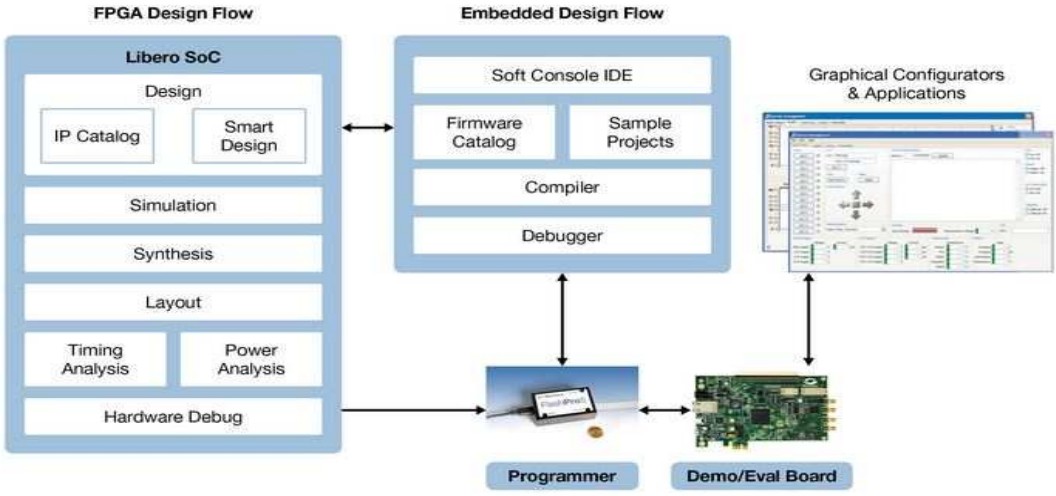

Fig. 3. Libero Design Flow [5]

version and is what students usually use. QuestaSim is Mentor Graphics' flagship product and is merely an improved version of ModelSim. QuestaSim is 64-bit and can take advantage of multi-core processors, unlike the 32-bit ModelSim. QuestaSim also has support for more features than ModelSim, such as full System Verilog support. QuestaSim was used for the development of this hardware, as there is no cost difference due to the included license and it can process simulation faster than ModelSim.

3.3. Synplify

Synplify Pro Microchip Edition is integrated into the design flow of Libero and carries out the synthesis and place-and-routing. It can also be opened interactively to allow the user to examine both the RTL and the post-layout implementation of the design. It is particularly useful as it can provide a clearer timing analysis than what is visible in Libero, to determine what the critical paths are.

3.4. SmartDebug

SmartDebug is an invaluable tool provided by Microchip that allows the user to read the eNVM of the SoC as it is running, in addition to having the ability to both read and write into all the RAM and Flip flops utilized by the design, which retain the same labels they had in the HDL.

3.5. SoftConsole

SoftConsole is the Eclipse-based IDE provided by Microchip to enable easier embedded software development on their devices. SoftConsole allows for a smooth transition from writing HDL to writing software for the target device. It also supports a variety of useful debugging features through the openOCD debugger, such as pausing running programs, inspecting their variables and assembly, and quickly determining crash sources.

4. Design Process

4.1. Initial Plan

The initial plan was to study the effect of memory resource distribution on the performance of embedded AI system implementations on flash-based FPGAs. The efficiency and security benefits of processing AI models on the edge instead of the cloud require suitable devices to run the AI models efficiently, making FPGAs a very suitable option. Implementing an AI accelerator on a flash-based FPGA allows for an ideal test-bed to study the effects of radiation and bit flips on embedded AI systems for use in applications requiring high reliability. The performance effects of running the accelerator from the BRAM vs external DRAM can then be investigated and analyzed. Research was then done to select a suitable AI accelerator to implement on the FPGA.

The initial focus was on neural network accelerators, as neural networks are the most widely used. Neural networks consist of multiple layers of ”neurons”, which take in a set of inputs and map it to a scalar value by calculating a weighted sum and passing it through a neuron activation function. These weights used by the neurons are the parameters of the neural network. Particular interest was placed in quantized neural network accelerators. Large models can have millions of parameters, resulting in megabytes of storage required to store these parameters, which are usually floating point numbers. While floating point numbers are good for model accuracy, they require a lot of memory, especially within the context of embedded AI systems. The use of floating point numbers also requires more computing power and makes adaption to devices like FPGAs difficult. The solution to these problems is to quantize these parameters into fixed point numbers. If done correctly, the size of the model and the complexity of hardware needed to run it can be greatly reduced, without a significant loss in accuracy. Mixed precision quantization is also used, allowing different parameters to be quantized to different precisions based on their sensitivity.

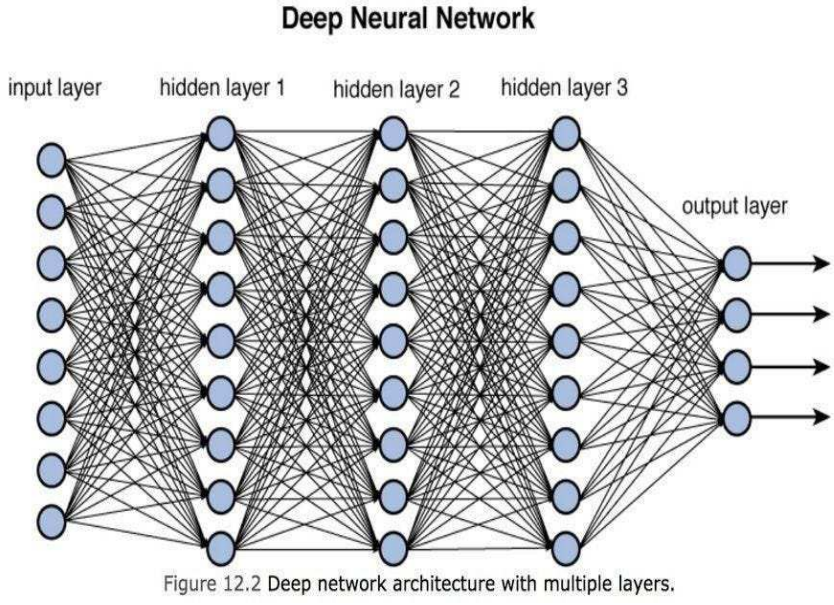

Fig. 4. Deep neural network visual representation [6]

The ability to quantize neural networks allows for easier implementation of embedded AI systems, however dedicated hardware is needed to fully take advantage of this. Several such implementations exist. DNNbuilder [7], is one such implementation. DNNbuilder is a software tool designed to allow fast implementation of deep neural networks on FPGAs. These FPGA-based neural network implementations can deliver higher efficiency than GPU-based implementations (up to 4.35x). DNNbuilder however has drawbacks. The hardware is made for a specific neural network,

requiring re-configurable devices such as FPGAs. This hardware also implements all layers of a neural network on the FPGA at once, which constrains the size of the neural network due to the limited resources on an FPGA. While DNNbuilder supports models quantized down to 4 bits, it does not support mixed precision quantization. Another implementation similar to DNNbuilder is FINN [8]. FINN and its improved version FINN-R [9], are also software tools designed to generate FPGA-based hardware implementations of neural networks. FINN does this in a different way to DNNbuilder, however despite allowing models quantized down to binary, it suffers from the same drawbacks as DNNbuilder, such as an inability to create mixed precision neural networks. FILM-QNN [10] overcomes these constraints by allowing arbitrary precision and mixed precision models, although the bit-packing scheme it uses for the DSP blocks limits it to 4/8 bit weights and 5-bit activations.

The implementation that was chosen to be used is BARVINN [7]. BARVINN is a software-programmable DNN accelerator, capable of processing arbitrary and mixed precision DNNs. The hardware architecture can be seen in Figure 5. BARVINN consists of 8 matrix vector units (MVUs). These MVUs are controlled by a RISC-V processor. Each MVU outputs a 64-element vector each clock cycle, which is calculated using a 64-element input vector, and a 64x64 matrix of weights. Within each MVU is a matrix-vector product unit (MVP), which carries matrix operations, using fixed point numbers with precision ranging from 1 to 16 bits. In addition to the MVP, the MVU contains a multiplier/adder unit, a pooling/ReLU unit, and a quantizer/serializer unit in the pipeline to allow additional operations to be performed on the data [7]. This architecture allows a large amount of flexibility, as different kinds of layers such as convolutional or fully connected layers can be assigned to an MVU. An MVU can also be assigned either multiple layers within the DNN, or a single layer can be divided over multiple MVUs. These MVUs are controlled by a barrel RISC-V processor. Due to the requirements of controlling all 8 MVUs at once, the processor needs to be capable of running 8 threads simultaneously. To prevent the large resource usage of an 8-core microprocessor, a barrel processor was used instead which which can execute 8 hardware threads by running each thread once very 8 cycles. The chosen processor implements the RV32I RISC-V instruction set to allow for easier integration with existing software tools. To allow for easy programming of the controller, a software tool was made to process ONNX format neural networks into RISC-V code.

The initial plan was to focus on the effects of different memory distribution schemes on BARVINN. The performance of on-chip BRAM, off-chip memory, and various combinations of the two will be tested and analyzed. BRAM offers the advantage of having lower latency than off-chip memory. Off-chip memory however has the advantage of not taking up FPGA logic, memory blocks, and chip space, while offering much greater quantities of memory. Most papers focus on either the processing hardware or the memory allocation for software-based models, leaving this particular aspect of designing hardware AI systems overlooked.

4.2. Initial Efforts

Upon the start of the work, the target device was analyzed and it became immediately clear that the initial plan was not possible. As mentioned in Section 2.1, the Microchip SmartFusion2 has 12084 LUTs, 12084 DFFs, and 22 math blocks, while the BARVINN implementation when implemented by its researchers used over 200 thousand LUTs and 512 DSP blocks on their FPGA [7]. The realization that the model requires more than an order of magnitude more resources than my FPGA can provide encouraged me to look at the alternatives that I researched before starting the work. These models required an amount of FPGA logic cells far exceeding what I had available. Extensive research was then conducted to find a suitable neural network accelerator to implement on the target device. This effort was however fruitless, what was learned is that FPGA accelerators are primarily designed for AI applications that require large amounts of computational power. Tools such as Vivado Vitis High-Level Synthesis (HLS), which is a way to design hardware in C++ without using VHDL, were commonly used. These types of projects would be quite problematic to port to the Libero design suite. While the previous issue is one that could be overcome, the largest issue

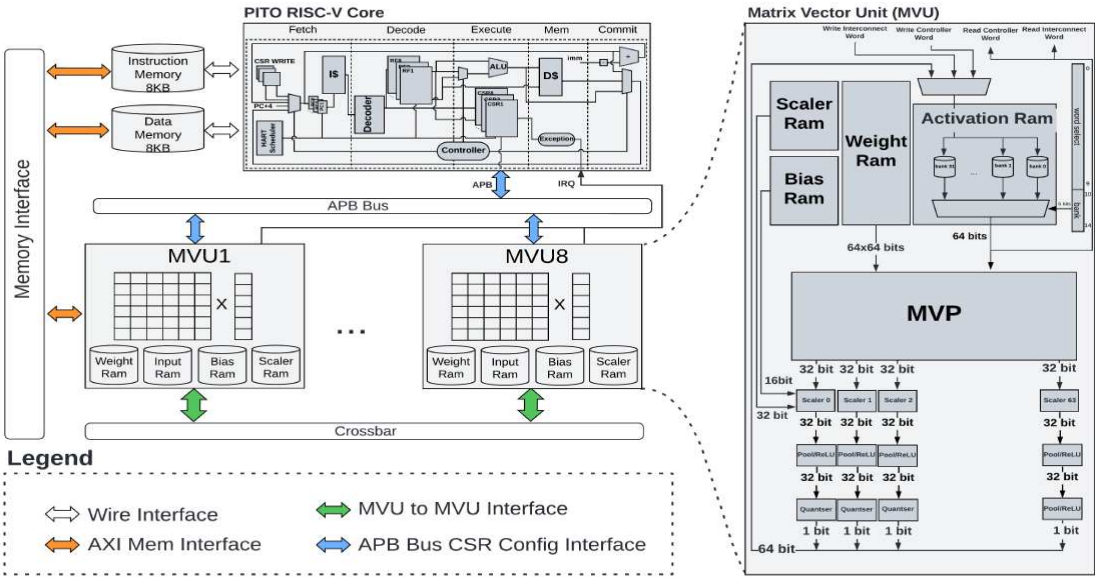

Fig. 5. BARVINN hardware architecture diagram [7]

is that due to the most commonly used applications of neural networks being computationally heavy tasks such as image recognition, most FPGA-based AI accelerators are designed and implemented on very large FPGAs. For example, BARVINN was implemented on a Xilinx Alveo U250 [7], the cost of such an FPGA is in the many thousands of euros, a fair bit more than the 41 Euro (excl. VAT) cost of the target device at the time of writing [11]. Most FPGA accelerator research found was done on large FPGAs such as the aforementioned Alveo U250 which are designed for use in data centers and similar applications, and not in low-power embedded devices. While this research was ongoing, efforts were made to become familiar with the tools mentioned

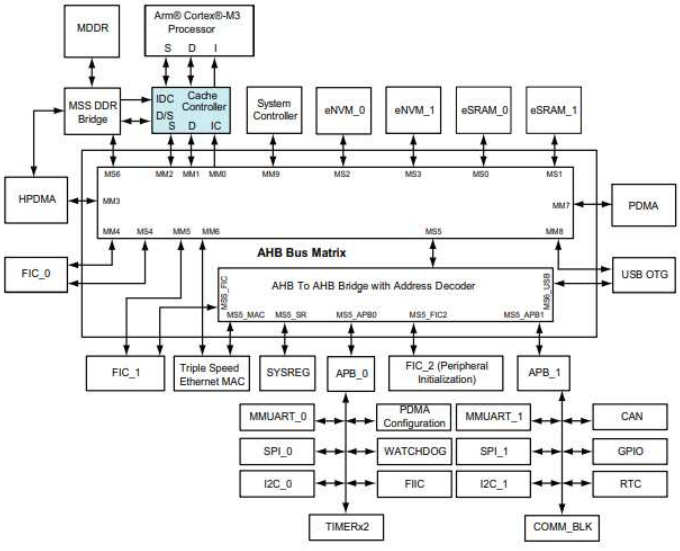

Fig. 6. MSS diagram

in Section 3. The tutorials provided by the board manufacturer, Trenz Electronics, were followed to

become familiar with the Libero design suite, the device fabric, and the microcontroller subsystem of the device, shown in Figure 6. The next step was determined to be implementing the NEORV32 RISC-V soft core processor on the FPGA, as its modifiability and the presence of a custom function subsystem (CFS) allows for the easy integration of accelerators and coprocessors.

The implementation of NEORV32 posed an unexpectedly large amount of difficulties. Un-explainable

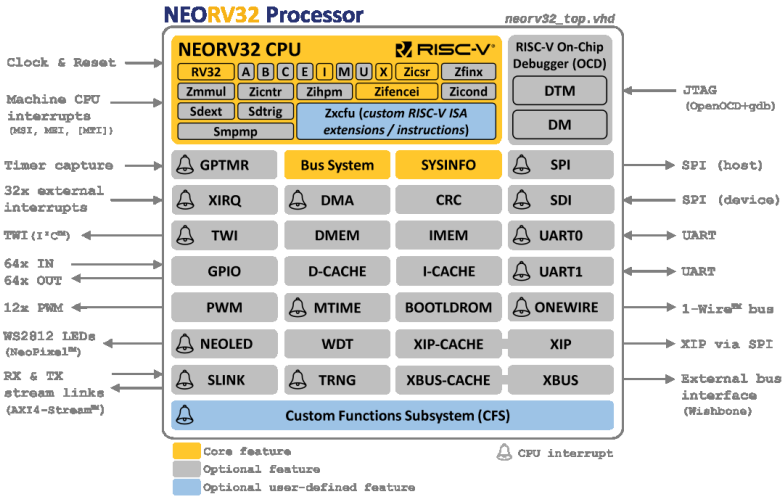

Fig. 7. NEORV32 overview [12]

errors from Libero resulted in the inability to elaborate the VHDL code, let alone synthesize it. Assistance was finally acquired from the co-supervisor in synthesizing a working NEORV32. However, the implementation did not allow for reprogramming the processor except through re-configuring the eNVM with a binary file every time. Attempts to implement other versions of NEORV32 with extra features such as loading new programs through UART failed for unknown reasons. An attempt to emulate the actions in the TCL scripts provided, one by one in the GUI, also failed due to unknown reasons. The conclusion gained from these experiences was that Libero has some peculiarities that make it very difficult to instantiate the NEORV32 design, presumably due to its files being divided between the work library and another library. Another peculiarity that was found was that the project had to be compiled with the Microchip Igloo2 as the target device instead of the Microchip SmartFusion2 being used, to be able to use the eNVM without using the built-in ARM Cortex M3. Significant technical difficulties were also encountered with other tools. The RISC-V compiler toolchain was very difficult to set up, as the Windows operating system was being used, and the compiler toolchain, along with the script files it required to compile for NEORV32 would only function on Linux. The university server that was being used to run Libero was not an option as the toolchain required dependencies not present on the server which could not be installed without sudo access. The compiler toolchain was finally made functional on a Ubuntu virtual machine using Microsoft Hyper-V, however, file sharing could not be arranged with the home operating system. There was still the problem of getting the openOCD debugger operational with NEORV32 and being able to integrate the firmware generated by Libero into the code. This task would be exceedingly difficult to complete with the setup being used at the time. This setup consisted of using Libero via a remote desktop connection (X2Go) to the university server, which had a barely functioning file-sharing system with Windows. The process to reprogram NEORV32 would involve writing and compiling the code on the Ubuntu virtual machine, sending the compiled binary to the Windows OS through Google Drive, and running a provided Python script to turn the compiled binary into a .mem file which could be used to write into the eNVM, and sending that through google drive again to be received on the university server, where I could

synthesize the design and write into eNVM memory. X2Go does not support port forwarding, however, so a .JOB file would have to be exported from Libero, sent through Google Drive back to the Windows system, as file sharing on X2Go would usually not work, and finally ran on FlashPro 5, a Microchip software used for batch programming devices. Proper debugging would then be impossible as openOCD was not yet set up, and would probably require forwarding the USB port to a Linux system somehow. As is readily apparent, this is a completely untenable setup to do any kind of work or research. While this chain could be optimized by perhaps dual booting Linux on the computer being used, and running VS Code connected to a GitHub repository on all operating systems being used, it still begs the question of whether all this is necessary when compared with the extremely smooth development environment of the embedded ARM processor. At this point, it had been determined that no suitable neural network accelerator would be identified for this target device, as even the accelerator available in the IP catalog of Libero was three times too large for the device. A month of extremely precious time had been wasted dealing with tool and Linux-related technical difficulties and determining that the original objective was not going to be possible, at least not in the way originally envisioned.

4.3. Direction Shift

The shortening amount of time remaining and the apparent impossibility of the task at hand required an immediate change in direction. The focus shifted to placing a random forest accelerator, or a matrix multiplier as a co-processor in a worst-case scenario. No suitable matrix multiplier was found, however, some promising papers were found related to implementing decision trees on FPGAs.

4.3.1. Random Forests and GBDTs

Random forests are machine learning algorithms that determine the outcome based on the outcomes of an ensemble of decision trees. A decision tree uses the features (inputs) given to determine the outcome, this is done by splitting at each decision node based on a certain threshold, until a leaf node is reached which denotes an outcome. Decision trees attempt to find the best splits in the data to arrive at a conclusion, and are typically trained through the Classification and Regression Tree (CART) algorithm [13]. The location of each split is decided based on optimizing metrics such as the Gini impurity or information gain, to maximize the value gained by the split. Decision trees can however run into issues such as overfitting, so a better approach is required. Using an ensemble of decision trees can be much more effective when determining the output. The key to using ensembles of decision trees is to train them such that they are not correlated with each other and provide unique information. The most well known methods to do so are bagging and boosting. Training a random forest through bagging is done by training each decision tree on a subset of features randomly chosen with replacement from the entire feature set, this technique was first introduced by Leo Breiman. The hyperparameters of random forests include the number of trees, maximum depth of a tree, and the maximum amount of features used to split a node.

An improvement over the random forest is machine learning algorithm called Gradient Boosted Decision Trees (GBDT). A GBDT model is also composed of an ensemble of decision trees, but unlike the random forest, bagging is not used to train the decision trees. Gradient boosting involves composing a strong machine learning model out of a collection of weak models, typically decision trees. This is done by training each subsequent weak model to compensate for the faults of the previous one. An illustration of a GBDT model can be seen in Figure 8.

4.3.2. Promising Finds

One of the most interesting finds was the paper titled ”Random Decision Tree Body Part Recognition Using FPGAs” [15]. The paper discusses the implementation of the algorithm used on the Microsoft Kinect to identify human body parts and gestures, on an FPGA. While the study was

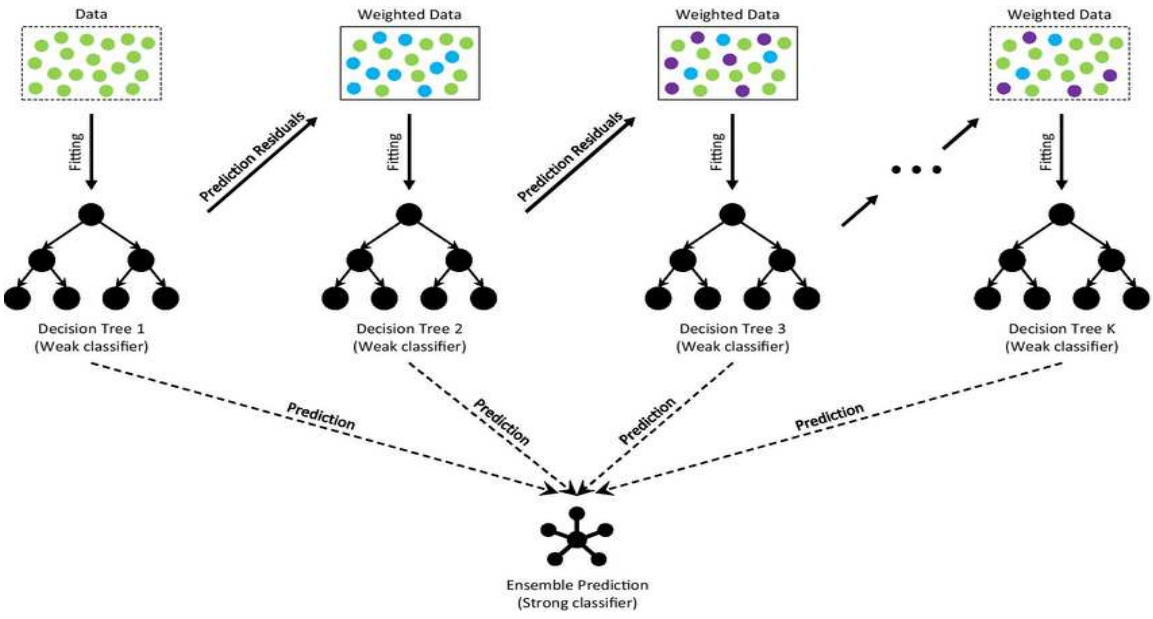

Fig. 8. The Architecture of a GBDT model [14]

conducted on a data center grade FPGA (Xilinx ML605) similar to what is used in similar studies, the intent was to create something that could run on embedded low-power FPGAs. The resultant system can be seen in Figure 9.

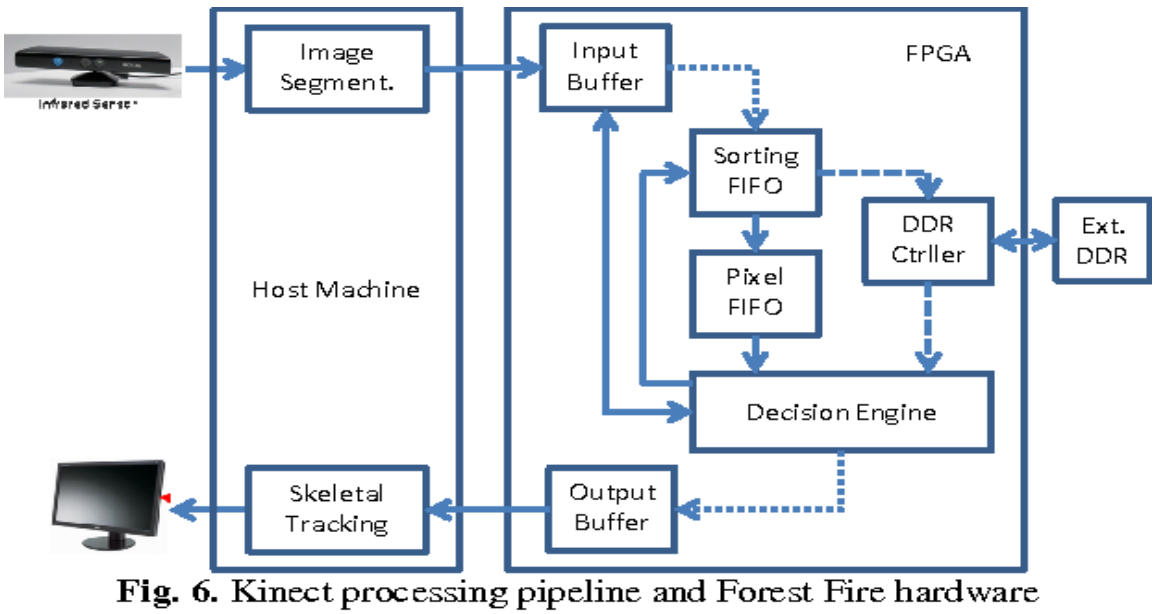

Fig. 6. Kinect processing pipeline and Forest Fire hardware

Fig. 9. Kinect Processing Pipeline [15]

What makes this implementation particularly attractive for the problem at hand is the low resource usage, 2.511 LUTs, and 2.267 FFs for the accelerator itself. The hardware implementation in the paper outperformed the software implementation by a healthy margin, showing the viability

of using FPGA-based random forest accelerators. The implementation provided also suffered from memory issues that prevented it from being easily implemented on small FPGAs as the buffers it uses require large amounts of BRAM. This would pose a great opportunity to do research on analyzing and optimizing its memory usage, which was the original goal. The major issue present however was that there was no publicly available code, and since this research was conducted at Microsoft, the chances of requesting it and getting it in a timely manner, or at all, were slim. Therefore, the search for a suitable system to implement had to continue.

4.3.3. Conifer

The final solution to the problem of implementing an embedded AI system on such a small FPGA was found thanks to a paper titled ”Fast inference of Boosted Decision Trees in FPGAs for particle physics” [16]. The paper discusses the extension of the existing hls4ml library, which allows the creation of hardware representation of neural networks through Xilinx Vitis HLS. The extension, named Conifer [17], allows for the creation of hardware representations of BDT models from commonly used libraries such as scikit-learn and XGboost. The output can be VHDL, HLS or C++ code. The hardware generated is designed to have extremely low latency, as the intended use is for the triggers in the Large Hadron Collider, which receive terabytes of data per second and have to decide which data to keep at extremely high rates. The hardware structure of a decision tree can be seen in Figure 10.

What makes this framework suitable for implementation is: the ease of converting any BDT model trained through one of the common software libraries into VHDL, the existence of a public codebase, and the resource efficiency along with the resource usage being fully dependent on the hyperparameters of the BDT model. The first efforts to create a trivial realization of an example were quite difficult however. Interfacing with Libero IPs necessitates the use of Libero SmartDesign, which acts as a graphical user interface for connecting HDL entities together with Microchip IPs and external ports. Despite what is mentioned in section 6.11.1 of the SmartDesign user guide [18], arrays of arrays are not supported. Finding out how to instantiate the VHDL model from Conifer, whose ports consist of arrays of booleans, and arrays 18 bit signed arrays, none of which are supported by SmartDesign in any form, resulted in days being spent until it was realized that documentation is simply incorrect. Even an array of std_logic_vectors with properly defined ranges could not be used. With just over two weeks now remaining, as a week had been spent finding an AI accelerator capable of fitting on the 12K logic element SmartFusion2, and another week of wasted efforts due to incorrect documentation, it was evident that something needed to be done to solve the issue and deliver a useful output for the thesis. It was decided to create a system capable of interfacing with a standard data bus, to allow any necessary data to flow in between the accelerator and any memory being used.

4.4. System Design

To fully take advantage of the very high speed of the hardware model produced by Conifer, a custom data pipeline. As an example, a ten input model was tested and it was processed in 12 clock cycles, as each feature put in the model has to be 18 bits in size, which would amount to 180 bits per 12 clock cycles, and it can be assumed that the clock used on the SmartFusion2 would be 100MHz. That amounts to 180 bits of input every 120ns, or 1.5 Gbps for a very small model, for image recognition with images as small as MNIST (28x28 pixel, grayscale) that would require an input data rate of 117.6 Gbps (assuming inputs are 18 bit at input and not kept as 8 bit then re-scaled to 18 bit fixed point in intermediate circuitry). Such systems would require custom high speed data pipelines for every application, so such a system would not be ideal for the given circumstances. It was instead decided that the system should be used as a co-processor, which would entail that it has to share an address space and memory data bus with the processor. The goal would then be to choose a widespread bus protocol to interface with. The chosen bus protocol was the Advanced High-performance Bus (AHB), part of ARM Advanced Microcontroller

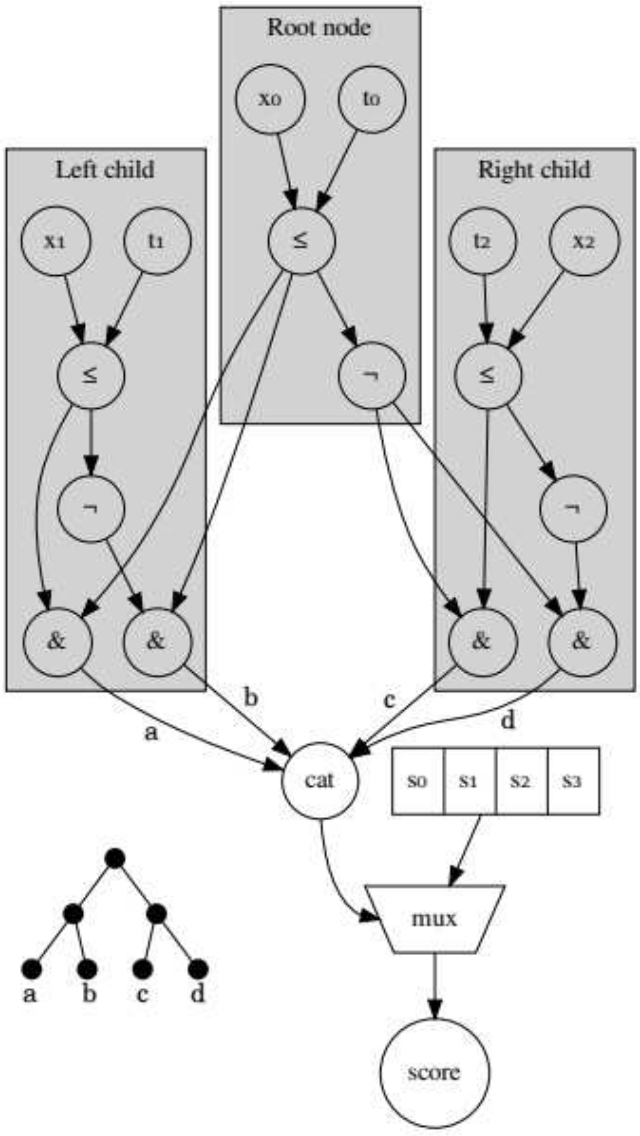

Fig. 10. Schematic of the implementation of decision trees in hls4ml, showing a single tree with a depth of 2. The x are the node features, and "t" are the thresholds. The 'handlebar' is the unary 'not' operator, and '&' the binary 'and'. The Boolean leaf activations are concatenated and used to address a look-up-table of output scores. [16]

Bus Architecture (AMBA). AMBA is a freely available open standard for the connection of blocks in SoCs, such as the extremely widespread ARM Cortex microcontrollers. In fact, the microcontroller section of the SmartFusion2 SoC utilizes an AHB bus matrix to connect the Cortex M3 MCU to all the memories, peripherals, and fabric interface controllers. The AHB protocol is also a very fast protocol, second only to the Advanced eXtensible Interface (AXI), which would be harder to design for, and would get bottlenecked by the AHB interfaces present inside the SmartFusion2 Microcontroller Subsystem (MSS) as seen in Figure 6, and within the Cortex M3 core itself in a SmartFusion2 based system. The widespread use of AHB interfaces on Microchip IP blocks is yet another factor in favor of using AHB as an interface. A possible option could be the Wishbone bus

standard, used as an interface for the NEORV32 microcontroller, although the benefits of using the AHB protocol severely outweigh those of using the Wishbone bus. The second question to be answered would be: how would the accelerator be controlled through software? The initial plan was to use the Custom Function Subsystem (CFS) of the NEORV32 microcontroller to provide the accelerator block with control registers, and control signals. The only remaining issue would be connecting the bus and control interfaces to the BDT model, as the bus operates on serial data (32 bit wide), while the inputs and outputs are loaded and unloaded from the BDT model all at once. The solution to that would be a data loader block, whose purpose would be to take the output of the bus, and buffer it until all the features have arrived, which can then be sent into the BDT model in parallel. The opposite of this process would be done by the data unloader, which would receive the array of output classes, and hand them to the bus interface one at a time. An illustration of the initial design can be seen in Figure 11. Y is an array of 18 bit signed types whose size depends on the amount of classes the model outputs. Y_vld is an array of booleans, indicating a valid output for each class. X is an array of 18 bit signed types whose size is the number of features the model receives. The control interface was not yet precisely defined at the beginning. Two counters would be kept within the AHBL interface, to be used for both choosing the address of memory to read/write and to be sent to the data loader and unloader to inform them which piece of data is being dealt with,

The entire system can be seen in Figure 12. The plan was to connect the accelerator block

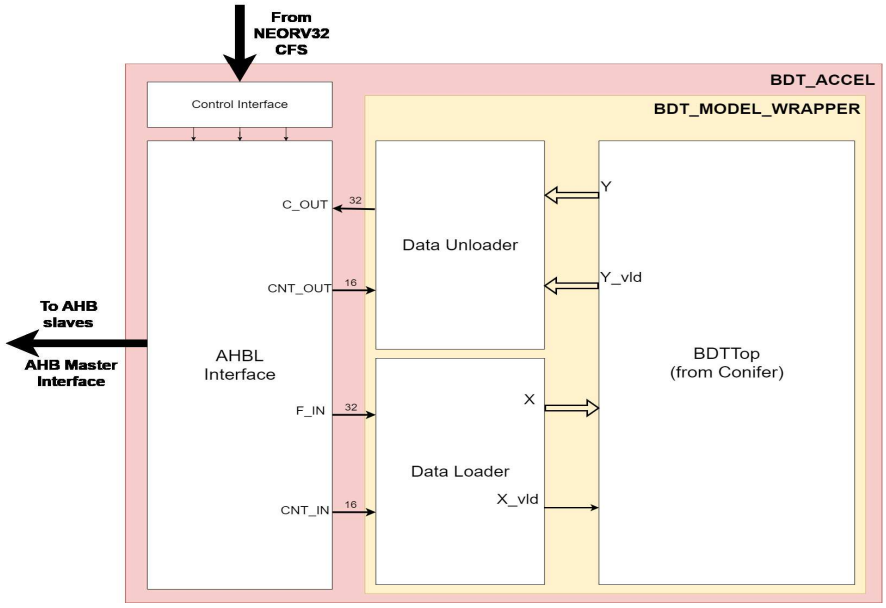

Fig. 11. Diagram of the initially proposed accelerator design

to NEORV32 and the bus matrix while having all the available memories be connected to the AHB bus matrix. This would enable interaction of the accelerator with any available memory, and enable the accelerator to be fully controlled through software.

4.5. Implementation of the System

4.5.1. Implementation of the BDT Model Wrapper

As can be seen in Figure 11, the BDT model wrapper entity consists of three parts, the actual BDT model (provided by Conifer), the data loader, and the data unloader. Works first started on the data loader, the idea was to create a VHDL process that assigns the least significant 18 bits

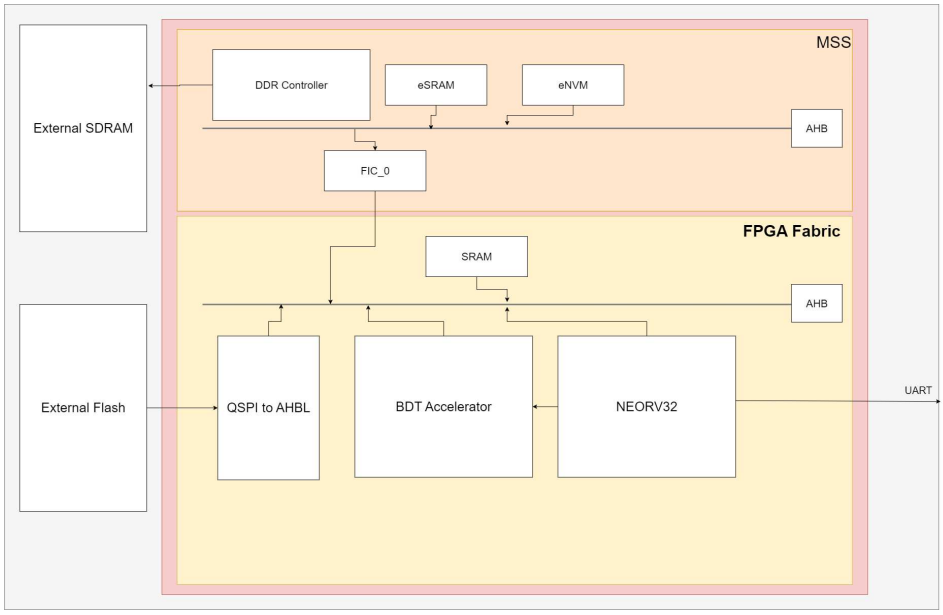

Fig. 12. Diagram of the initially proposed system (Wishbone to AHB bridge for NEORV32 not shown)

of the input 32 bit word (bus width) to a slot in the output array based on the received counter value from the AHBL interface, which would act as an address. The first implementation was made functional in the pre-synthesis simulation, however it showed strange behavior in the post synthesis simulation. After consulting with a person knowledgeable about VHDL, it became clear that the code was not written very well, especially for synthesis. The data loader was subsequently re-written as a shift register, that would be a two dimensional array, shifting in an 18 bit word every time an enable signal is received. This implementation performed well both in pre-synthesis and post-synthesis simulation, and was latter found to be fully functional on the target device. The resource usage for the data loader was also kept to a minimum, as it used just enough flip-flops to buffer all the features, and a small number of LUTs (for example 16 for 100 features). The design change however resulted in the AHBL interface no longer needing to output the feature counter, as it would instead output an enable pulse for the shift register. Work then began on the data unloader. Its operation is simpler as it just has to output the correct 18 bit value (concatenated into 32 bits) based on the input counter. This was easy to make functional, the only struggles were with dealing with the boolean array output by the model and condensing it into a single logic signal. This was accomplished, however, the LUT usage for the data unloader was rather high (1355 for 100 output classes), which could be a point of improvement in the future. Together, these modules allowed the creation of the BDT model wrapper, which finally allowed it to be instantiated in Libero SmartDesign.

4.5.2. Design and Implementation of the AHBL Interface

The AHB protocol will be explained first, and following that the interface design will be discussed. Information and illustrations of the AHB protocol are taken from the AMBA AHB specification [19]. The AHB protocol was designed to be used in high-performance, high-clock frequency systems, and utilizes features such as: burst transfers, single clock edge operations, non-tristate implementation and configurable address and data bus widths. This makes it ideal for use with memory devices and high-bandwidth peripherals (such as an AI accelerator in this scenario). The way the AHB interfaces and bus matrices are designed in Libero, an IP block takes care of address decoding and multiplexing, allowing for the masters and slaves to utilize the AHB Lite

protocol instead of the full AHB specification. AHB lite is a reduced version of the AHB protocol, where there is only one master on the bus. The AHBL interface for the bus would need to be a bus master that reads a chunk of memory starting from the given starting address until it has read a 32-bit word for every feature the model requires. Once the BDT model is done processing the data and gives an output, the bus interface would need to write that array of data to the chosen destination address. It was decided that the first implementation of the bus interface would only do single transfers, as burst transfers would complicate everything. For reference, single transfers with wait states (as those have to be accounted for) are shown in Figure 13.

Figure 3-3 shows a read transfer with two wait states.

Figure 3-3 Read transfer with two wait states

Figure 3-4 shows a write transfer with one wait state.

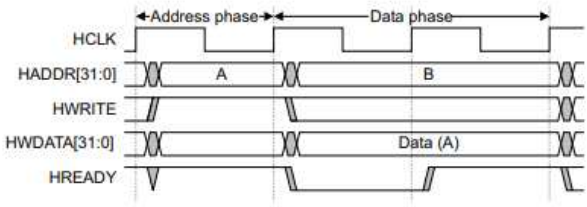

Figure 3-4 Write transfer with one wait state

Fig. 13. AHB single transfers plot [19]

HADDR is driven by the master with the desired address, the master drives HWRITE to signal to the slave whether a write or read operation is occurring, and HREADY is driven by the slave to signal to the master when it is ready to participate in another transfer. HRDATA and HWDATA are the write and read data busses. A read transfer requires the master to drive the HWRITE line low, and place the address on the HADDR bus, and then wait until HREADY is back to 1 to read. A write transfer requires the master to drive HWRITE high, place the address on the HADDR bus, and then keep the data to be written on the HWDATA bus until HREADY is high again. Not shown here is HTRANS, which has to be set to ”10” while a transfer is occurring, and ”00” when the master is idle. There are also other parameters the master has to provide, such as HBURST, which determines whether a burst transfer is occurring (”000” in this case), and HSIZE which determines what size of word is being sent on the bus (full word (32 bit), ”10” in this case). The FSM created for the interface as a result of this is shown in Figure 14.

This version of the interface attempted to rely on a simple FSM with minimal combinational logic. Two control signals were added, GO_AHEAD and DONE. GO_AHEAD determined when the BDT accelerator would start operation, and DONE would signal that it has finished. The interface would read a chunk of memory when the go-ahead signal was given, and then write the results to the chosen place in memory when the BDT model has completed processing. The testing setup for this interface was a simple system connecting the Microchip IP Core_AHBLSRAM (block ram with an AHBL slave interface) to it. A macro file was written to force set a piece of memory to predetermined values, the behavior of the interface and system as a whole would

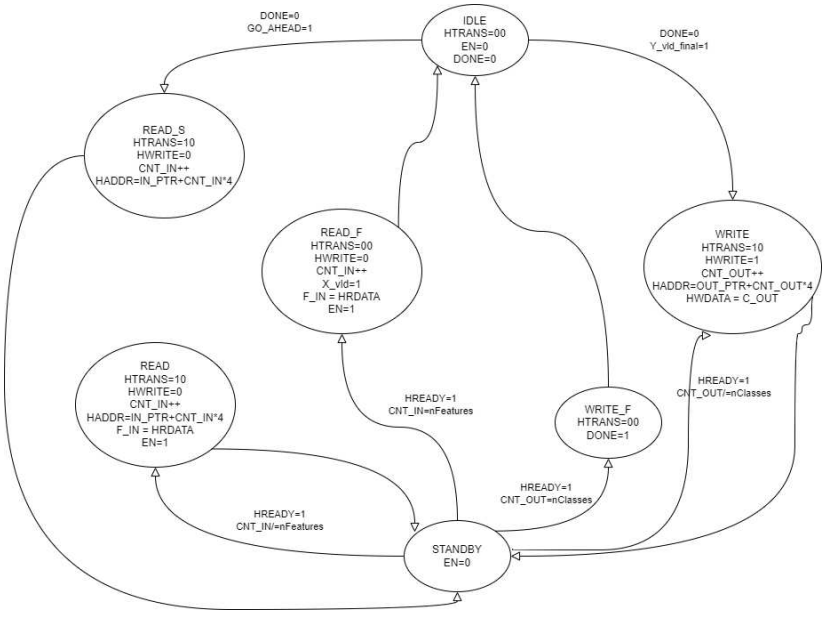

Fig. 14. FSM of first AHBL interface

then be assessed as it read the memory and then placed the output at the chosen destination. After some debugging, the system was functional in pre-synthesis simulation. The VHDL code was written in such a way that synthesis would not be an issue, however, this version of the interface was not continued with. The reason for this is that its design resulted in two extra clock cycles being needed for every transfer. This was an unacceptable delay, as a 3-cycle transfer (the BRAM IP inserts two wait cycles for every transfer, resulting in three-cycle transfers) would take 5 cycles, almost halving the bandwidth of the connection. Such a slow-down due to bad design was unacceptable and the AHBL interface was redesigned. The redesign aimed at including combinational logic along with the FSM, to allow for zero extra delays being inserted, except for those made by the slave. The FSM of the new design is shown in Figure 15.

The new design is no longer a pure FSM, as combinational logic was added in certain states, to be able to respond to changes within the clock cycle. After some debugging, this version of the AHBL interface was capable of interfacing with BRAM at its maximum speed (3 cycles per word), and appeared to be fully functional in pre-synthesis simulations. To make debugging easier, an entity was created that simulated the BDT model but was simply a register file that sent the inputs to the outputs, this would allow for easy testing to determine whether the design is reading and writing to memory correctly, in addition to testing whether it would supply the model with correct inputs. Post-synthesis simulations were attempted, however, the results were nonsensical, as despite all input signals being well-defined, every single signal in the system was undefined. As time was running short, and nothing in the HDL code seemed to be problematic for synthesis, the post-synthesis simulation was ignored and the step was taken to test the model on the target device.

4.5.3. Implementation of Partial System on FPGA

The system was built on top of the reference design provided by the board manufacturer [20]. This would allow for the existence of a fully functional system to test the accelerator design on, without spending valuable time experimenting with how to utilize all the board's features. The test was performed by hard-coding the input and output addresses at synthesis, as the control interface was not ready yet, and then using the embedded Cortex M3 to read and write the

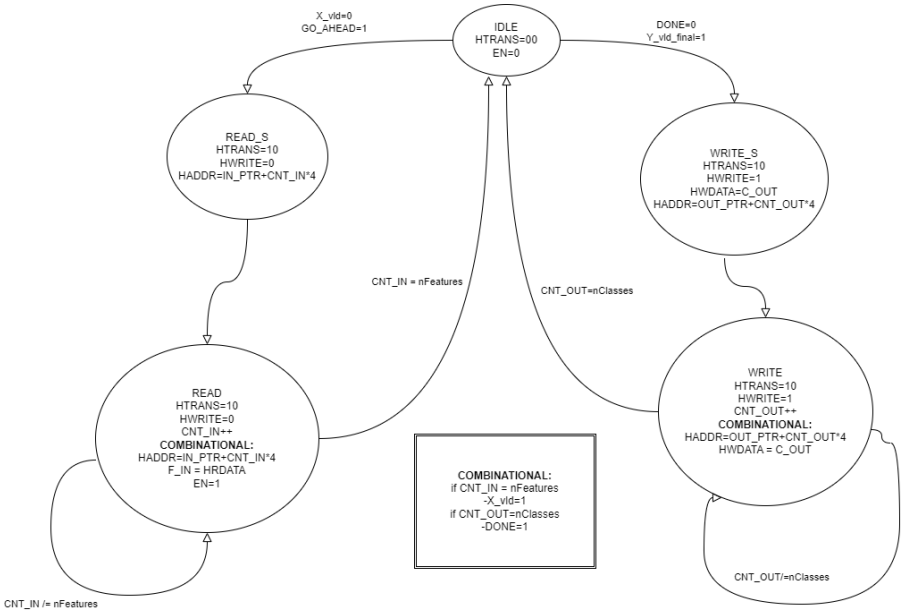

Fig. 15. FSM of second edition AHBL interface

relevant sections of memory while using its GPIO peripheral to control and read the GO_AHEAD and DONE signals. The test was a near-complete success with Accelerator (with the register file replacement for the BDT model) being able to write and read memory properly. A test was also conducted with an example BDT model from Conifer to ensure the output for a given set of inputs was identical to the simulation output, which it was. There was however a bug where the last value written to external RAM would be 0, this did not occur when writing to BRAM. This bug was ignored until the rest of the design could be completed. It was also discovered here that the linker script provided by Microchip to write a program to eNVM was nonfunctional for programs with code sizes above 16Kb. This necessitated the use of a separate linker script, which would load and debug the program from the eSRAM.

4.5.4. Implementation of the Control Interface

The next step would be to implement the control interface. It was decided that the control interface should contain 6 32-bit registers. Two registers would be for the input and output addresses. Three registers would be for three timers, which would keep count of how many clock cycles it took to write, read, and complete the whole operation. The final register would contain control signals, which at the moment only consist of the go-ahead signal. It was also discovered here that even though it is not stated anywhere, Libero ignores synthesis attributes in VHDL code, and synthesis attributes have to be added through SmartDesign. This was also the time at which it was decided to abandon the idea of integrating the accelerator with NEORV32. The difficulties stated in previous sections made NEORV32 particularly difficult to work with, while working with the embedded ARM MCU provided a very smooth design flow with full debugging capabilities. What was realized is that, if the control interface was to also be a standard bus interface, then the control registers would simply be memory addresses that any connected device could write to and control. This would allow easy integration with any microcontroller, be it ARM or RISC-V. It was decided to implement an Advanced Peripheral Bus interface, as the protocol is simpler, and a high bandwidth connection to the control registers is not necessary. The FSM of an APB slave can be seen in Figure 16.

An APB slave interface was implemented for the control registers. The interface functioned in

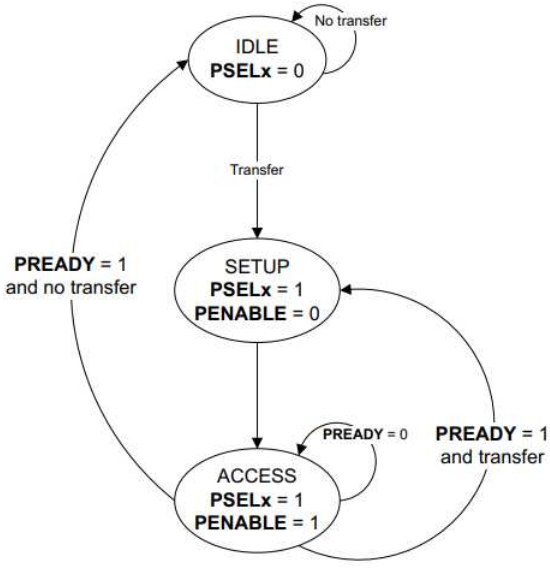

Fig. 16. FSM of APB slave [21]

pre-synthesis simulation, however behavior was sometimes strange, even though the specification was followed properly. An attempt was then made to test the interface on the target device. The test showed that neither writing nor reading the control registers was possible. It was then decided at this point, after some further investigation, that it would be easier to implement an AHB slave interface than it would be to fix the APB interface. The state machine for the AHB slave interface only had two states at first, INIT and READY. It would remain in the READY state until a suitable combination of signals for a read or write transfer would arrive. It would then complete the transfer and return to the ready state within the clock cycle. This was tested on the FPGA, the test showed that read transfers worked properly, however, writes did not function. A WRITE state was then inserted, which would introduce a single delay cycle for every write transfer. This solved the issue and the control interface and counters were now fully functional.

4.5.5. Implementation of Full System

With both the memory interface and control interface functional, the full system could now be tested. The testing method would involve using the BDT model replacement, to output the inputs, and verify the functionality of the whole system. The system was tested by reading 100 inputs and writing 100 outputs to programmed memory locations. This was successful. A couple of fixes were also made solving two important issues, the writing to RAM bug was solved by ensuring HWDATA stays as the last output instead of reverting to 0 within a clock cycle. It was also realized that enabling burst read transfers was as simple as changing HBURST to ’001’ (undefined length burst), and using HTRANS=’11’ (SEQ) when in the READ state. This resulted in a large speed-up when using external RAM and embedded SRAM. No speedup was present when using the BRAM as the IP provided by Microchip does not support burst transfers. After the implementation of these improvements, the system worked flawlessly, being tested with a register file of various sizes, and with two models from the Conifer examples folder, one trained on the Iris dataset, and one trained on the Harpie dataset. However, it was not possible to test whether the BDT models tested gave correct outputs. This is because conifer BDT models use arbitrary precision fixed-point numbers, the library they use is in C++, which could not be compiled alongside the Libero-generated firmware which is in C. Time constraints did not allow for the writing code to

solve this issue. It is however verified that the accelerator does its job properly concerning inputs and outputs to the model, so it can be assumed that everything is fully functional. Time constraints also did not allow for bench-marking the accelerator against a software implementation, however as will be seen in the results, it can be assumed that it is quite a lot faster.

5. Results and Evaluation

5.1. Final System Design

The final accelerator diagram can be seen in Figure 18. The final system diagram can be seen in Figure 17. The accelerator went through some changes, most notably getting an AHB control interface, allowing for integration into nearly any microcontroller system. The AHBL interface was also given burst transfer capabilities, resulting in notable increases in performance.

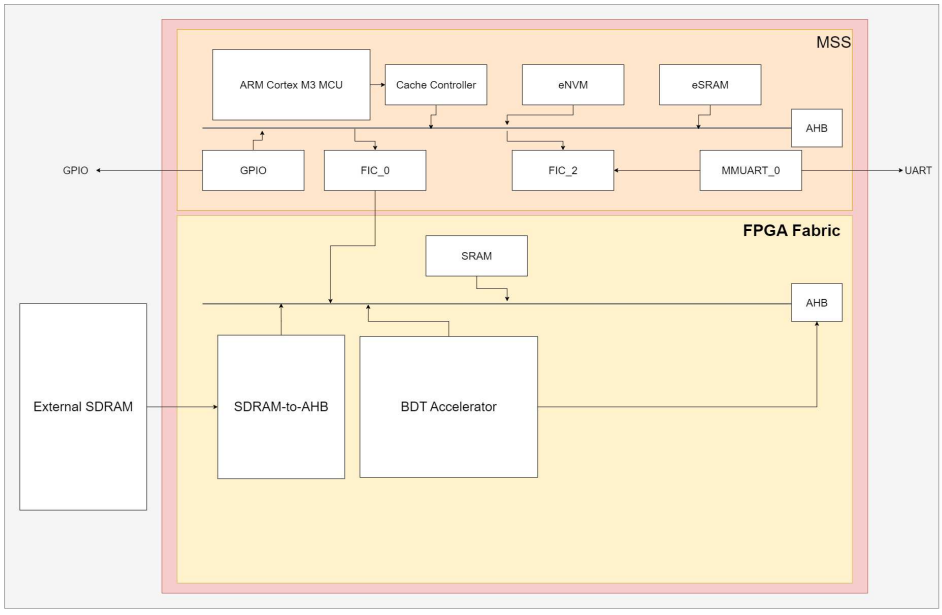

Fig. 17. Diagram of the final system

The system as a whole was built off of the reference design from Trenz Electronics, trimming it down to only the necessary parts, and adding BRAM to the memory bus. The main crystal oscillator was kept as the clock source, with the PPL running the system at a 100MHz clock. This gives the processor access to 3 types of RAM (SDRAM, eSRAM, and BRAM) and two types of flash (eNVM and SPI Flash(not included in figure)). The accelerator is in theory capable of being easily integrated with any BDT model given by Conifer, as long as there are enough FPGA resources available to fit it. The accelerator is also designed to interface with nearly any generic system possessing an AHB bus or some other type of data bus that can be bridged to AHB. It also possesses built-in counters to easily find the read, write and processing speed.

5.2. System Performance

To evaluate system performance, it will be tested with a BDT model and a register file with different amounts of inputs and outputs to test memory read and write speeds, as Conifer BDT models are designed to process data in around 12 clock cycles, making the actual processing time negligible compared to the memory read and write speeds. The results can be seen in Table 5.2. R symbolizes read time in clock cycles, W symbolizes write time, and T symbolizes total time. The system is running at 100MHz, so each clock cycle is 10 nanoseconds long. Rf-X refers to using a register file instead of a BDT model which has a one clock cycle delay and X

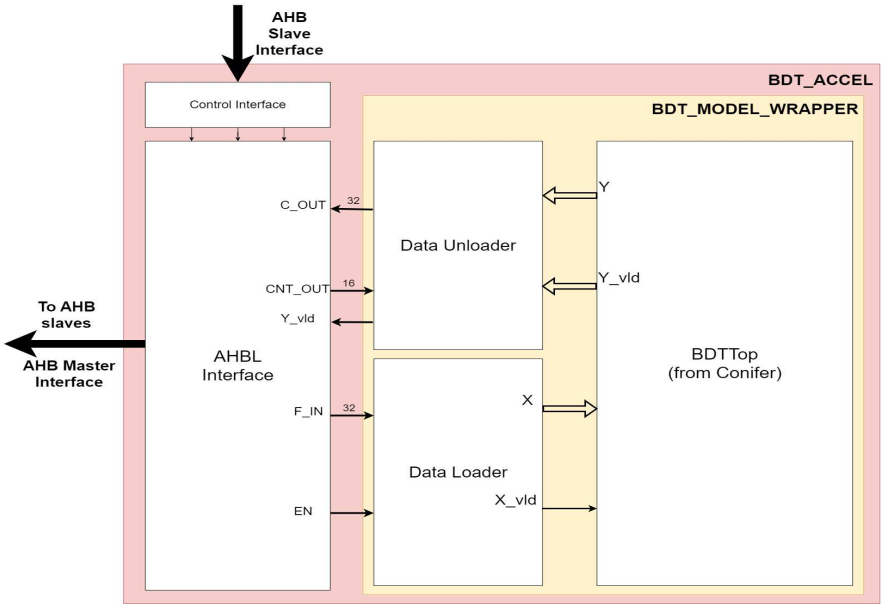

Fig. 18. Diagram of the final accelerator design

	DRAM(r)	DRAM(w)	DRAM(t)	BRAM(r)	BRAM(w)	BRAM(t)	eSRAM(r)	eSRAM(w)	eSRAM(t)
RF-100	2710	1987	4698	305	301	607	204	201	406
RF-50	1355	987	2343	155	151	307	104	101	206
RF-10	275	187	463	35	31	67	24	21	46
Iris (4/3)	113	37	161	17	9	37	12	7	30

inputs and X outputs. The BDT model used is a model from the Conifer examples code which is trained on the Iris dataset and has 4 features and 3 output classes. What can clearly be seen is that DRAM(100MHz) is quite a lot slower than SRAM, usually being around 7 times slower than BRAM, with the difference decreasing as the amount of features and classes decreases. eSRAM can be seen to be consistently faster than BRAM, this is because the embedded SRAM supports AHB burst transfers, while on the SmartFusion2 BRAM does not.

5.3. System Evaluation

The final product is an adapter capable of interfacing any Conifer BDT model, which can be generated from any sci-kit-learn, XGBoost, or ydf BDT model, with standard embedded system architectures involving the AHB or a similar enough bus. This allows even the smallest FPGAs, such as the SmartFusion2 to be converted into embedded AI systems with relative ease. The design performs without errors and is capable of harnessing the full speed of the AHB data bus, however, more rigorous testing is required to ensure truly reliable operation. Due to time constraints, a proper driver could not be written, so no testing was performed to compare the speed and efficiency of this system to a pure software implementation. However, since the actual data processing takes no time, and memory retrieval only needs to retrieve features and not parameters, in addition to being able to take advantage of burst transfers as memory is read in a single chunk, it can be confidently assumed that the execution of an ML model on this platform will be many times, if not orders of magnitude faster than a pure software implementation, no matter which memory is being used. It would also be very beneficial to attempt running an embedded AI system on the SmartFusion2 that would be of actual use, beyond just a proof-of-concept.

6. Conclusions

In conclusion, with the goal of setting up an embedded AI system on a SmartFusion2, an adapter was developed to easily integrate any hardware model from the Conifer framework into any existing microcontroller-based system. Tests were also performed to find the performance impact of using different memories with the AI accelerator. Despite being plagued by time constraints and early misdirected efforts, this work served as a proof-of-concept of running an embedded AI system on a resource-constrained FPGA, in addition to providing the tools to easily implement embedded AI systems on similar embedded devices. Another possible use of the output of this work is the ability to test the effects of radiation exposure on a SmartFusion2 embedded AI system.

References

[1] D. Dsilva, J.-J. Wang, N. Rezzak, and N. Jat, "Neutron SEE Testing of the 65nm SmartFusion2 Flash-Based FPGA," in *2015 IEEE Radiation Effects Data Workshop (REDW)*. Boston, MA, USA: IEEE, Jul. 2015, pp. 1–5. [Online]. Available: http://ieeexplore.ieee.org/document/7336722/

[2] "TEM0001 TRM - Public Docs - Trenz Electronic Wiki." [Online]. Available: https://wiki.trenz-electronic.de/display/PD/TEM0001+TRM

[3] "SmartFusion2 SoC Product Brochure." [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/Brochures/SmartFusion2-FPGA-Product-Brochure.pdf

[4] "UG0445: SmartFusion2 SoC FPGA and IGLOO2 FPGA Fabric User Guide." [Online]. Available: https://ww1.microchip.com/downloads/aemDocuments/documents/FPGA/ProductDocuments/SoC

[5] "Libero SoC v12.0 and later | Microsemi." [Online]. Available: https://www.microsemi.com/product-directory/vectorblox-ai/5598-libero-soc

[6] R. Parmar, "Training Deep Neural Networks," Sep. 2018. [Online]. Available: https://towardsdatascience.com/training-deep-neural-networks-9fdb1964b964

[7] X. Zhang, J. Wang, C. Zhu, Y. Lin, J. Xiong, W.-m. Hwu, and D. Chen, "DNNBuilder: an automated tool for building high-performance DNN hardware accelerators for FPGAs," in *Proceedings of the International Conference on Computer-Aided Design*. San Diego California: ACM, Nov. 2018, pp. 1–8. [Online]. Available: https://dl.acm.org/doi/10.1145/3240765.3240801

[8] Y. Umuroglu, N. J. Fraser, G. Gambardella, M. Blott, P. Leong, M. Jahre, and K. Vissers, "FINN: A Framework for Fast, Scalable Binarized Neural Network Inference," in *Proceedings of the 2017 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Monterey California USA: ACM, Feb. 2017, pp. 65–74. [Online]. Available: https://dl.acm.org/doi/10.1145/3020078.3021744

[9] M. Blott, T. B. Preußer, N. J. Fraser, G. Gambardella, K. O'brien, Y. Umuroglu, M. Leeser, and K. Vissers, "FINN- *R*: An End-to-End Deep-Learning Framework for Fast Exploration of Quantized Neural Networks," *ACM Transactions on Reconfigurable Technology and Systems*, vol. 11, no. 3, pp. 1–23, Sep. 2018. [Online]. Available: https://dl.acm.org/doi/10.1145/3242897

[10] M. Sun, Z. Li, A. Lu, Y. Li, S.-E. Chang, X. Ma, X. Lin, and Z. Fang, "FILM-QNN: Efficient FPGA Acceleration of Deep Neural Networks with Intra-Layer, Mixed-Precision Quantization," in *Proceedings of the 2022 ACM/SIGDA International Symposium on Field-Programmable Gate Arrays*. Virtual Event USA: ACM, Feb. 2022, pp. 134–145. [Online]. Available: https://dl.acm.org/doi/10.1145/3490422.3502364

[11] T. E. GmbH, "SMF2000 FPGA Module with Microchip SmartFusion® 2, 8 MByte SDRAM." [Online]. Available: https://shop.trenz-electronic.de/en/TEM0001-01A-ABC-2-SMF2000-FPGA-Module-with-Microchip-SmartFusion-2-8-MByte-SDRAM

[12] S. Nolting and A. t. A. Contributors, "The NEORV32 RISC-V Processor," Aug. 2023. [Online]. Available: https://github.com/stnolting/neorv32

[13] "What Is Random Forest? | IBM," Oct. 2021. [Online]. Available: https://www.ibm.com/topics/random-forest

[14] H. Deng, Y. Zhou, L. Wang, and C. Zhang, "Ensemble learning for the early prediction of neonatal jaundice with genetic features," *BMC Medical Informatics and Decision Making*, vol. 21, no. 1, p. 338, Dec. 2021. [Online]. Available: https://bmcmedinformdecismak.biomedcentral.com/articles/10.1186/s12911-021-01701-9

[15] J. Oberg, K. Eguro, R. Bittner, and A. Forin, "Random decision tree body part recognition using FPGAs," in *22nd International Conference on Field Programmable Logic and Applications (FPL)*, Aug. 2012, pp. 330–337, iSSN: 1946-1488. [Online]. Available: https://ieeexplore.ieee.org/document/6339226

[16] S. Summers, G. D. Guglielmo, J. Duarte, P. Harris, D. Hoang, S. Jindariani, E. Kreinar, V. Loncar, J. Ngadiuba, M. Pierini, D. Rankin, N. Tran, and Z. Wu, "Fast inference of Boosted Decision Trees in FPGAs for particle physics," *Journal of Instrumentation*, vol. 15, no. 05, p. P05026, May 2020. [Online]. Available: https://dx.doi.org/10.1088/1748-0221/15/05/P05026

[17] S. Summers, "thesps/conifer," Jun. 2024, original-date: 2020-04-20T14:37:01Z. [Online]. Available: https://github.com/thesps/conifer

[18] "SmartDesign User Guide." [Online]. Available: https://onlinedocs.microchip.com/pr/GUID-BB41100F-72D4-4B5E-BF44-5F63DB805A02-en-US-5/index.html?GUID-4A2C253C-5BAD-414E-905E-8EEAC5791C7B

[19] "AMBA AHB Protocol Specification." [Online]. Available: https://developer.arm.com/documentation/ihi0033/latest/

[20] “TEM0001 Test Board - Public Docs - Trenz Electronic Wiki.” [Online]. Available: https://wiki.trenz-electronic.de/display/PD/TEM0001+Test+Board
[21] “AMBA APB Protocol Specification.” [Online]. Available: https://developer.arm.com/documentation/ihi0024/latest/