ANN IP Core Specification

Author: David Aledo, Félix Moreno
david.aledo@upm.es

Rev. [0.1]
June 2, 2016

This page has been intentionally left blank.

Revision History

Rev.	Date	Author	Description
0.1	23/05/16	David Aledo	First Draft

Contents

1 Introduction

This IP core is a configurable feedforward Artificial Neural Network (ANN). ANNs are Artificial Intelligence (AI) algorithms biologically inspired on the brain. An ANN can be trained to learn a specific task. Typical tasks are pattern recognition and classification, but not exclusively. ANN is based on a simple model of a neuron, described on Equation 1.

$$a_n = \mathrm{f}\left(\sum_{i=0}^{M}(w_{n,i} \cdot p_i) + b_n\right)$$

a_n : is the output of the neuron *n*.

f : is the activation function.

M : is the number of inputs.

$w_{n,i}$: is the weight for the input *i* of the neuron *n*.

p_i : is the input *i*.

b_n : is the bias of the neuron *n*.

Neuron's models are grouped by layers are connected in a network. This IP performs full feedforward connections between consecutive layers. All neurons' outputs of a layer become the inputs for the next layer. This ANN architecture is also known as Multi-Layer Perceptron (MLP) when is trained with a supervised learning algorithm. There is extensive literature about ANN, please consult it for more information.

Different kinds of activation functions can be added easily coding them in the provided VHDL template and following the steps described in Appendix A: Adding user activation functions.

This IP core is provided in two parts: kernel plus wrapper. The kernel is the optimized ANN with basic logic interfaces. The kernel should be instantiated inside a wrapper to connect it with the user's system buses. Currently, an example wrapper is provided for instantiate it on Xilinx Vivado, which uses AXI4 interfaces for AMBA buses [1,2]. Developers hope to add more wrappers in the future with the help of the OpenCores community.

The kernel of the ANN IP core is configurable through parameters. Bit widths, number of layers, inputs, number of neurons in each layer, activation function of each layer, and layer type of each one can be configured. There are three layer types regarding on if inputs and outputs are parallel or serial: “SP”, “PS”, and “PP” (in the current version “PP” layer type is not available). See Section Kernel Architecture for more information.

Computations are performed in fixed point format in order to achieve good performance on a reasonable area. Read Section Operation to know how to adjust data into the ANN input and output formats.

2 Kernel architecture

The architecture of the ANN can be configured through parameters. Table 1 summarizes the parameter list and its function. Input and outputs are always serial stream data. Data flow in a pipeline way inside the ANN. Data between layers can be serial or parallel depending on the configuration.

Parameter	Type	Description
Nlayer	integer	Number of layers
NbitW	natural[4]	Weight-and-bias bit-width
NumIn	natural	Number of inputs to the network
NbitIn	natural	Input bit-width
NumN	int_vector[1]	Number of neurons in each layer
l_type	string[2]	Layer type of each layer
f_type	string[3]	Activation function type of each layer
LSbit	int_vector[1]	LSB of the output of each layer
NbitO	int_vector[1]	Output bit-width of each layer
NbitOut	natural	Network output bit width

[1] int_vector is an array of integers defined on the provided package layers_pkg.vhd.

[2] l_type string must contain as layer types as number of layers, separated by spaces. Each layer type is a two character string “SP”, “PS”, or “PP” (see Architecture section for more information).

[3] f_type string must contain as activation function types as number of layers, separated by spaces. Each layer type is a six character string.

[4] At current version, NbitW must be a multiple of 8 due to memory alignment.

Table 1: List of parameters (generics)

The main structural blocks of the ANN are the layers. Three types of layer can be selected (currently only two of them) depending if the input and output data are parallel or serial.

- The Serial-input Parallel-output (“SP”) layer type is an array of Multiplier-and-Accumulators (MACs). Its resource utilization depends on the number of neurons, and its latency depends on the number of inputs. This is the most common implementation and is perfect for first processing layers which receive inputs serially.

- The Parallel-input Serial-output (“PS”) layer type implements one neuron that is reused to calculate all neurons of the layer. Its resource utilization depends on the number of inputs, and its latency depends on the number of neurons plus the logarithm to the base 2 of the number of inputs (because the adder tree). A drawback of this layer type is that there is not perfect mapping of the multipliers and the adder tree into embedded DSP48Es of Xilinx. They have one multiplier and one adder, configurable to perform different operations like MACs, but to combine all multipliers with the adder tree adders is not possible, causing more DSPs utilization. Nevertheless, this layer type is good for output layers which need serial outputs.
- The Parrallel-input Parallel-output (“PP”) is not available in the current version. This layer type is the full parallel implementation of a layer, achieving the maximum performance at expenses of the largest resource utilization.

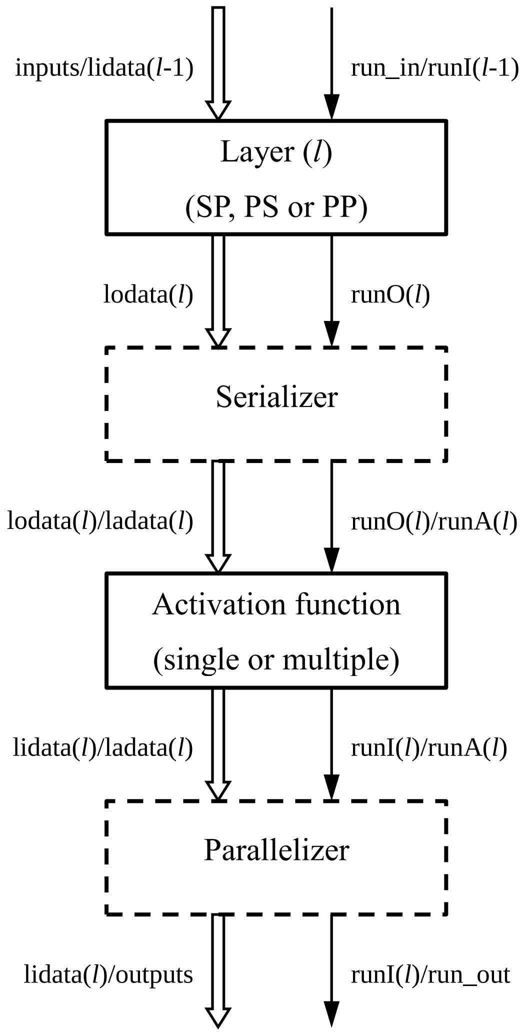
When serial-input and serial-output is needed, it is automatically accomplished by inserting a serializer after a “SP” layer or a parallelizer before a “PS” layer. Serializer is a shift register with parallel load, and parallelizer is a shift register with parallel unload.

The selected activation function is inserted wisely between layers. If a parallel output precedes a serial input, a serializer is inserted before an only one activation function block. If a serial output precedes a parallel input, one activation function block is inserted before a parallelizer. Only when parallel output precedes parallel input an array of parallel activation function blocks is inserted.

Each block has its own control based on the propagation of a ready signal.

Figure 1 shows the block diagram around a generic layer *l*. The ANN IP core kernel generates as many of these blocks as *Nlayers*.

The kernel of the ANN IP core does not implement any control register. Registers may be implemented on wrappers.



If $l = 0$, inputs and run_in are arriving to layer (*l*) instead of lidata(*l*-1) and rinI(*l*-1). If $l = Nlayer - 1$, outputs and run_out are leaving parallelizer or a single activation function instead of lidata(*l*) and runI(*l*). If the serializer is present, its outputs are ladata(*l*) and runA(*l*). If the parallelizer is present, its inputs are ladata(*l*) and runA(*l*).

Figure 1: Block diagram around a generic layer *l*.

3

Weight and bias memories architecture

Understanding the weight-and-bias memories architecture is key point to properly use the ANN IP core. Due to the high configurability of the ANN IP core, the weight-and-bias memories architecture is automatically generated based on the ANN parameters.

Although the address space of the weight-and-bias memories is seen as a unique BRAM space address through the BRAM interface, it is actually composed by several BRAMs and registers, with some addresses without physical implementation. The user must take care to avoid write or read to these unimplemented addresses. Figure 2 shows the general scheme of the address space.

The *total address* length is calculated by the formula on Equation 2.

$$addr\_l = \log_2(Nlayer) + 1 + \max\left(\bigcup_{l=0}^{Nlayer} \left(\log_2(NumIn_l) + \log_2(NumN_l)\right)\right)$$

addr_l : is the *total address* length seen by the host.

Nlayer : is the ANN IP core parameter which describe the number of layers in the ANN.

$NumIn_l$: is the number of inputs of the layer l. $NumIn_0 = NumIn$ (the ANN IP core parameter which describe the number of inputs of the ANN), and $NumIn_l = NumN_{l-1}$ when $l \neq 0$.

$NumN_l$: is the number of neurons in the layer l. Every one of them is an element of the integer array parameter of the ANN IP core *NumN*.

The combination of all $NumN_l$ forms the integer array parameter of the ANN IP core *NumN*.

They are each one of the elements of the integer array parameter of the ANN IP core *NumN*.

The first $\log_2(Nlayer)$ bits address the layer to be accessed. Starting from the input layer labeled 0, to the output layer labeled $Nlayer - 1$. The layer number is coded with little-endian binary numbers (i.e. unsigned(addr_l-1 downto addr_l-$\log_2$(Nlayer)).

The next bit of the weight-and-bias address selects between the weights ('0') or the biases ('1') of the layer. This bit is called *bias select*.

The remaining bits are used to address the weights or biases of the layer with a memory structure that depends on the layer type. The *layer RAM address* length used for each layer depends on the number of inputs and neuron on the layer. The *layer RAM address* length is defined as $lra\_l = \log_2(NumN_l) + \log_2(NumN_l)$. When the *bias select* is asserted, a bias is addressed with unsigned(bra_l-1 downto 0). $bra\_l = \log_2(NumN_l)$ is the *bias RAM address* length. The corresponding neuron number of the bias is coded with little-endian binary numbers. When the *bias select* is deasserted, a weight $w_{n,i}$ is addressed using the top part of the *layer RAM address* unsigned(lra_l-1 downto wra_l) to select input *i*, and the bottom part unsigned(wra_l-1 downto 0) to select neuron *n*. $wra\_l = \log_2(NumIn_l)$ is the *weight RAM address* length. Both *n* and *i* are coded with little-endian binary numbers.

"SP" layers: biases are stored on registers because they have to be accessed simultaneously from the MACs. Weights are stored on BRAMs. There is one BRAM per neuron. All the *n* weights ($w_{n,i}$) for an input *i* are accessed simultaneously from the MACs.

"PS" layers: biases are stored in a BRAM because they do not have to be accessed simultaneously. Weights are stored on BRAMs. There is one BRAM per input. All the *i* weights ($w_{n,i}$) for a neuron *n* are accessed simultaneously from the neuron.

"PP" layers: as all the weights and biases have to be accessed simultaneously, all of them are stored in registers.

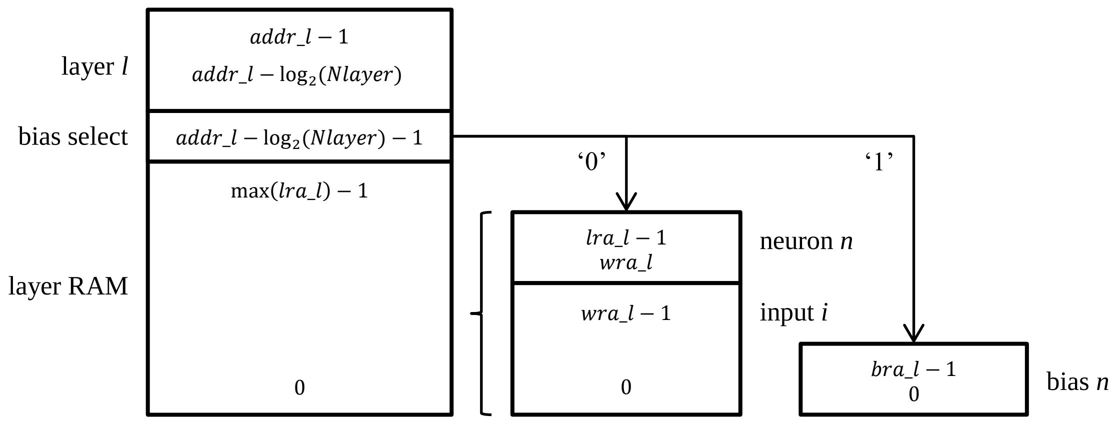


Figure 2: General scheme of the weight and bias memories space address.

In order to correctly access to weight-and-bias memories from a host, when host code is written in C/C++, the best way is through 2-D arrays for weights and 1-D arrays for bias, which base addresses must be separated $\max(lra\_l)$ spaces. Size of the first dimension of weight arrays must be defined as the $NumIn_l$ next power of two. Although first dimension of the 2-D weight arrays have to be defined as the $NumIn_l$ next power of two, unless $NumIn_l$ is a power of two, indexes between $NumIn_l$ and its next power of two must not be accessed. The reason for this way of defining weight arrays is because then compiler will automatically jump to the correct address.

The header file ann.h is provided to help C/C++ programmers to access the weight-and-bias memories. This header file should be edited to set the parameters related to user's ANN.

4 Operation

Before sending input data to de ANN, weight-and-bias memories should be initialized through the BRAM port. Weight-and-bias memories can be written and read by a host though the BRAM interface.

Once the weights and biases are configured, data to the ANN can be sent serially with a validation bit (run_in). The ANN kernel does not check if the weights and biases have been initialized. If input data are sent and the run_in validation bit is asserted without weight and bias initialization, the IP core will use the uninitialized data contained in the weight-and-bias memory blocks.

Output data is produced with a throughput of 1 clock cycle. The output validation bit run_out is asserted every clock cycle that a valid output data is produced.

Computations are performed in fixed point format in order to achieve good performance on a reasonable area. Format of the fixed point signals can be configured through their correspondent bit-width parameters and the LSbit array parameter. Internal layer registers and signals are sized to prevent overflow. However, to assure reasonable resource utilization, data between layers is re-sized. Independently of the layer type, results of the multiplications have a length $mul\_l = NbitW + NbitIn_l$, and final result of all the additions a length $res\_l = mul\_l + \log_2(NumIn_l)$. $NbitIn_0 = NbitIn$, and $NbitInl = NbitO_{l-1}$ when $l \neq 0$. After all layer computations, only $NbitO_l$ bits are sent to the next layer, starting the output Less Significan Bit (LSB) on the position defined by the $LSbit_l$ parameter. Overflow issues are avoided by saturating the result to de most positive or negative values when needed. However, underflow or excessive saturation issues may appear if the *NbitO* and *LSbit* parameters are not chosen carefully.

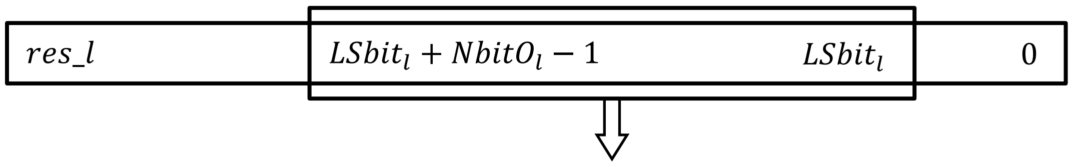


Figure 3: Fixed point format of layer outputs.

5 Clocks

The current version of the ANN kernel has only one clock which is named clk. It is used to synchronize every register and memory of the design. Future versions may include alternative clocks for the weight and bias memories to separate them form the ANN processing pipeline.

6 Kernel IO ports

Table 2 lists the kernel IO ports. In order to connect this basic logic interfaces in a system, a wrapper should be added to interface with system buses.

Port	Width	Direction	Description
reset	1	Input	Synchronous active high reset input
clk	1	Input	Clock input
run_in	1	Input	Start and input data validation
m_en	1	Input	Weight and bias memory enable (BRAM interface)
m_we	1	Input	Weight and bias memory write enable (BRAM interface)
inputs	NbitIn	Input	Input data (serial stream)
wdata	NbitW	Input	Weight and bias memory write data (BRAM interface)
addr	addr_l[1]	Input	Weight and bias memory address (BRAM interface)
run_out	1	Output	Output data validation
rdata	NbitW	Output	Weight and bias memory read data (BRAM interface)
outputs	NbitOut	Output	Output data (serial stream)

[1] the address length is calculated with the formula on Equation 2. See Section Weight and bias memories architecture for more information.

Table 2: List of kernel IO ports

7 Validation

The ANN IP core has been validated through simulation. The VHDL testbench performs a supervised training of an example application. The example application is an autoencoder used to compress an image (which is read from a file).

The VHDL testbench files are NOT provided. The simulation needs excessive RAM and takes long time to finish. To validate small changes in the ANN IP kernel modules is recommended to write Ad-Hoc testebenches for the modified modules. Validation also can be done through the provided example application (see Appendix C: Example application).

Appendix A

Adding user activation functions

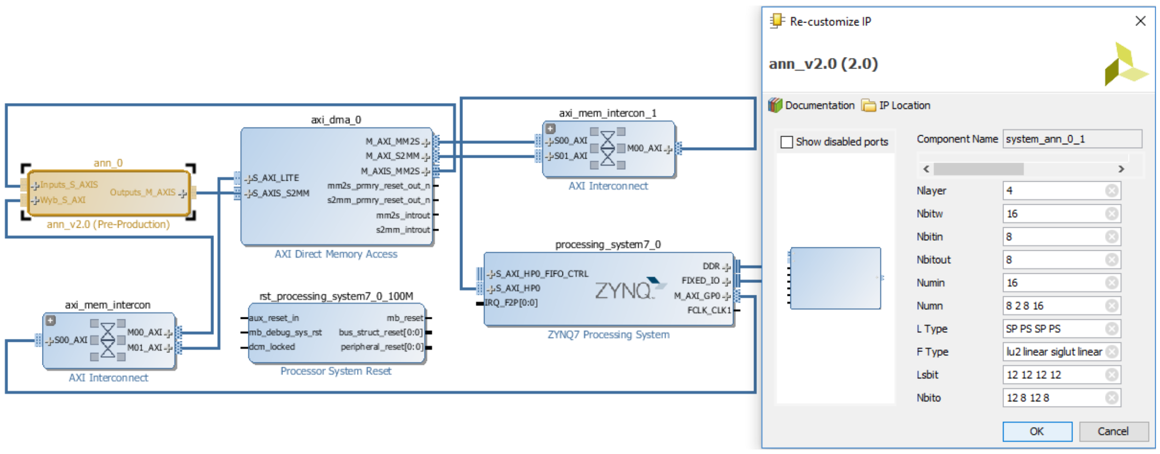
User activation functions can be easily added following these steps:

1. Write the user activation function description into the template af_template.vhd. Change file and entity name accordingly, but do not modify the rest of the entity template. Add user description into the module architecture. af_sigmoid.vhd file can be taken as an example.
2. Edit Structural architecture of activation_function.vhd file to instantiate the new user activation function inside an if-generate, like in its commented template. Choose a six characters tag for the new user activation function. This step allows the ANN IP core kernel to select the user activation function from the f_type parameter if the chosen tag is present on it.
3. Re-synthetize all the ANN IP core.
4. When configuring the ANN IP core, the new activation function tag can be used for the f_type parameter.

Appendix B

Wrapper for Vivado

The ann_v2_0.vhd, and ann_v2_0_*.vhd are the files automatically generated by the Vivado's Create and Package New IP wizard, and then edited to instantiate and connect the ANN IP core kernel. They have been tested for non-burst writes and reads on the weight and bias memories, and AXI stream interfaces connected through a DMA without SG engine. Test has been performed on a Zybo [3] using Vivado 2015.4. Figure 4 shows the test block design on Vivado GUI.



ann_v2.0 (2.0)

Component Name	system_ann_0_1
Nlayer	4
Nbitw	16
Nbitin	8
Nbitout	8
Numin	16
Numn	8 2 8 16
L Type	SP PS SP PS
F Type	lu2 linear siglut linear
Lsbit	12 12 12 12
Nbito	12 8 12 8

Figure 4: Block design connection of the ANN IP core on Vivado.

Note: ANN IP kernel needs a synchronous active high reset, but AXI4 interfaces use active low resets. A synchronous active high reset must be connected on the input port ann_areset.

Appendix C
Example application

The example application is a C program for a host controlling the ANN IP core. It targets the system of the Figure 4.

The example application is based on the popular MNIST database [4]. Although in order to make it suitable for embedded computing, the images have been reduced to half in both directions by averaging every 4x4 block. So, the inputs to this network are 14x14=196 pixel images.

The outputs of this example are ten values which each one should stand at its maximum to select its assigned digit, and zero in other case. This implies that the output layer of this ANN has again a fixed size of 10 neurons.

Files, and more information will be available soon. Authors expect it before the end of June 2016.

References

1. ARM AMBA AXI Protocol v2.0 Specification

2. AMBA4 AXI4-Stream Protocol v1.0

3. http://store.digilentinc.com/zybo-zynq-7000-arm-fpga-soc-trainer-board/

4. Yann LeCun, Corinna Cortes, Christopher J.C. Burges. The MNIST database of handwritten digits. http://yann.lecun.com/exdb/mnist/index.html