Speedster7t Machine Learning Processing User Guide (UG088)

Speedster FPGAs

Achronix®
Data Acceleration

Copyrights, Trademarks and Disclaimers

Achronix Semiconductor Corporation

2903 Bunker Hill Lane
Santa Clara, CA 95054
USA

Website: www.achronix.com
E-mail : info@achronix.com

Table of Contents

Chapter - 1: Introduction

The machine learning processing block (MLP) is an array of up to 32 multipliers, followed by an adder tree, an accumulator, and a rounding/saturation/normalize block.The MLP also includes two memory blocks, a BRAM72k and LRAM2k, that can be used individually or in conjunction with the array of multipliers. The number of multipliers available varies with the bit width of each operand and the total width of input data. When the MLP is used in conjunction with a BRAM72k, the amount of data inputs to the MLP block increases along with the number of multipliers available.

The MLP offers a range of features including integer multiply with optional accumulate, bfloat16 operations, floating point 16, block floating point, and floating point 24. Below is a list of features available with the MLP block:

- Configurable multiply precision and multiplier count
    - Up to 32 multiplies for 4-bit integers or block floating point 12 in a single MLP
    - Up to 16 multiplies for 8-bit integers or block floating point 16 in a single MLP
    - Support for 16-bit integer, Bfloat16, (Brain Float), floating point 16, and floating point 24 in a single MLP
- Multiple number formats (fixed and floating point)
- Deep adder tree and accumulator block
- Multiple rounding and saturation features
- Tightly coupled circular register file (LRAM) for easily caching and feeding back results
- Tightly coupled BRAM for reusable input data such as kernels or weights
- Cascade paths up a column of MLPs
    - Allows for broadcast of operands up a column of MLPs without using up critical routing resources
    - Allows for adder trees to extend across multiple MLPs
    - Broadcast read/write to tightly coupled BRAMs up a column of MLPs to efficiently create large memories

Along with the numerous multiply configurations, the MLP block includes optional input and pipelining registers at various locations to support high-frequency designs. There is a deep adder tree after the multipliers, with the option to bypass the adders and output the multiplier products directly. There is a normalization block for floating-point and block-floating-point operations. In addition, a feedback path allows for accumulation within the MLP block.

Below are block diagrams showing the MLP using the fixed or floating-point formats.

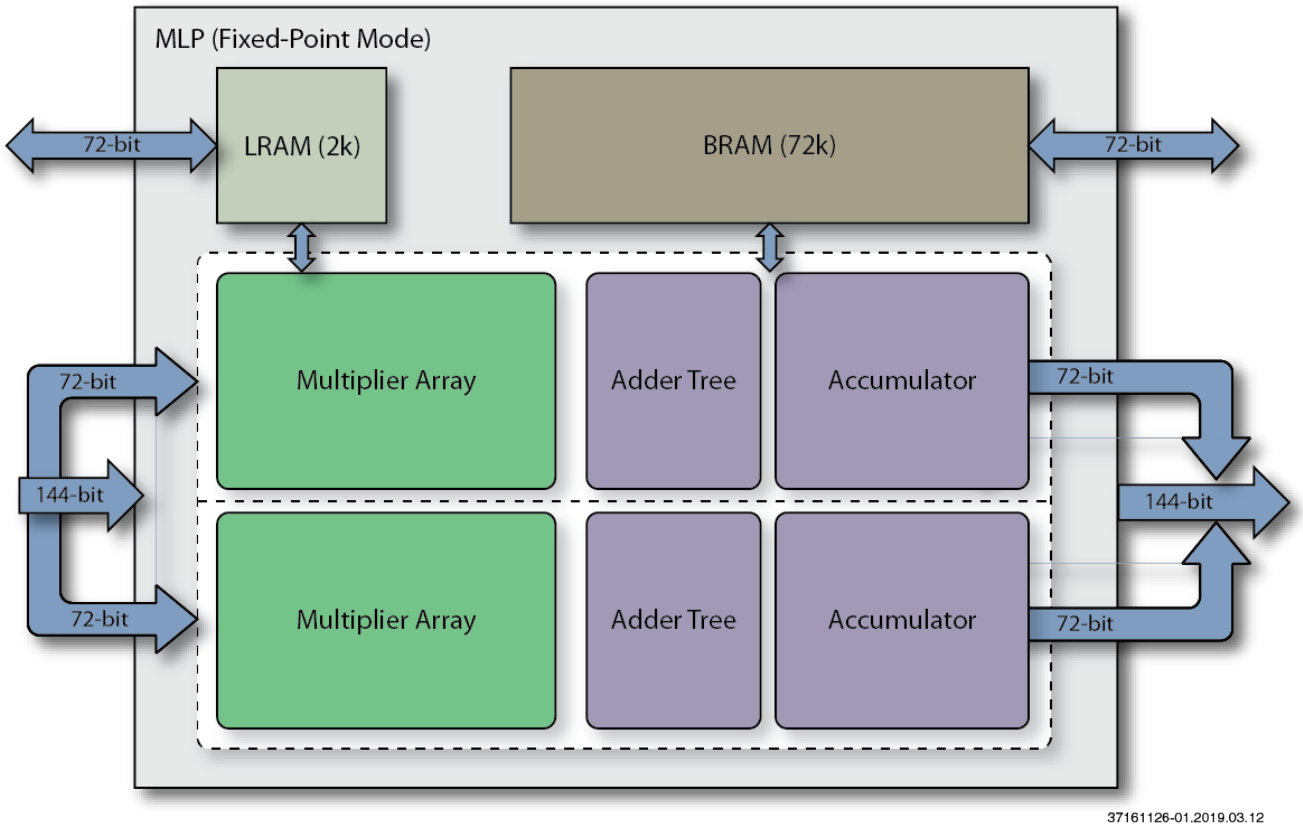

Figure 1: ***MLP Using Fixed-Point Mode***

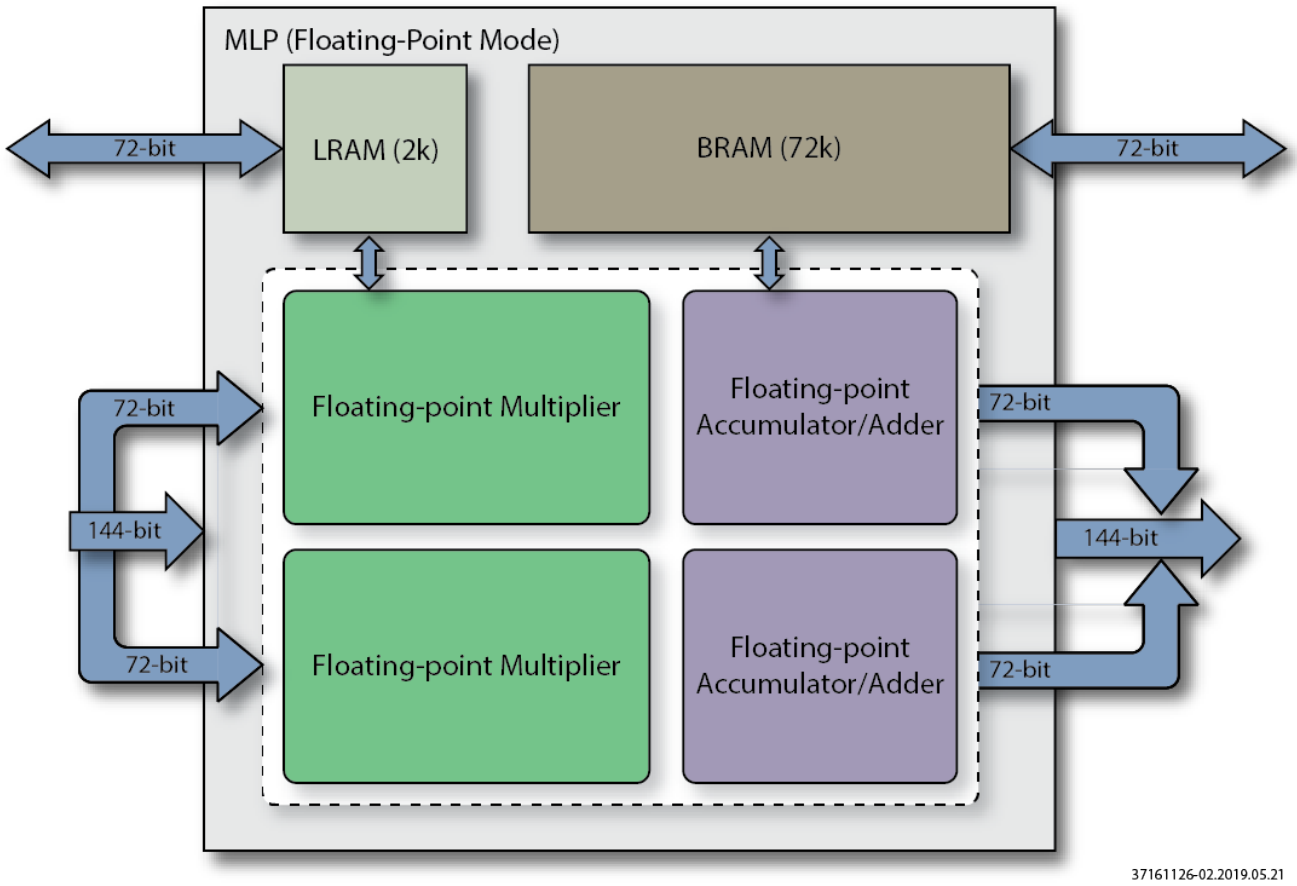

Figure 2: ***MLP Using Floating-Point Mode***

A cascade path allows for the adder tree to extend across multiple MLP blocks in a column without using extra fabric resources, and a data cascade/broadcast path is available to send operands across multiple MLP blocks. Below is a diagram showing the cascade paths across MLPs.

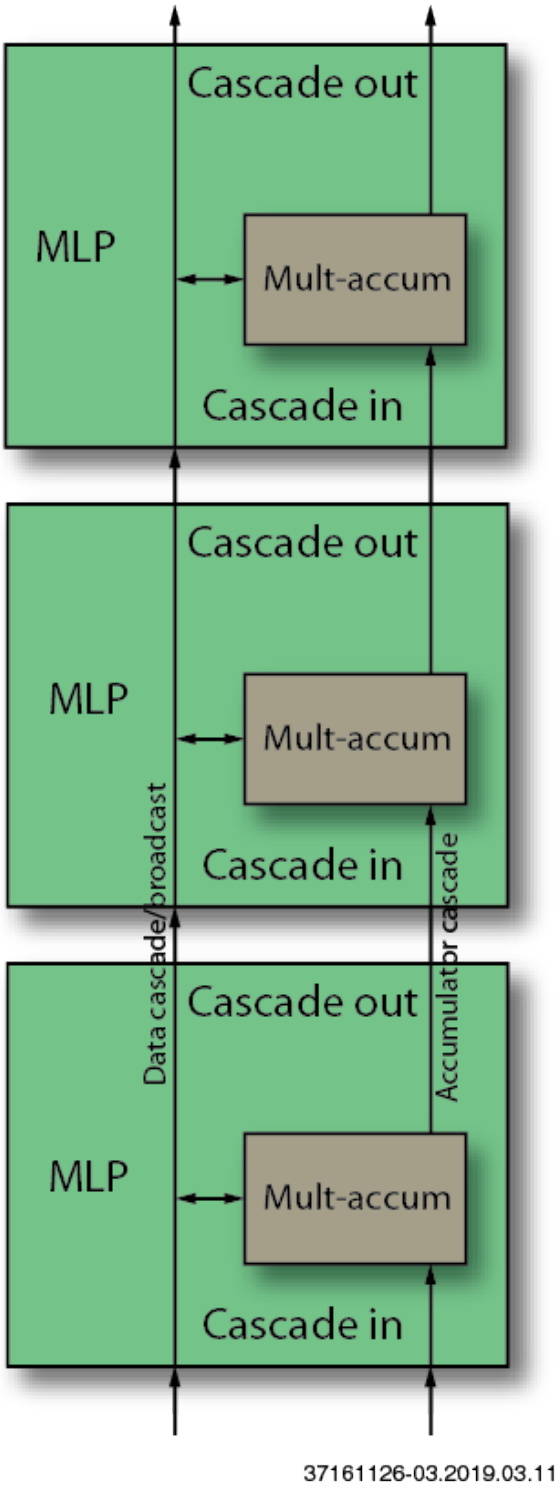

Figure 3: ***MLP Cascade Path***

Scope

This user guide is intended to be used in conjunction with Speedster7t IP Component Library User Guide which gives details of the MLP72 primitive, listing each port and parameter, and containing detailed internal block diagrams. This document discusses mathematical applications and how the MLP72 may best be configured to achieve them. In addition this document lists advanced modes of the MLP72 that allow it to perform complex mathematical manipulations in a very efficient form

Sections

This document contains sections on the following;

- Number Formats (see page 11) : This details the variety of numerical formats that can be used by the MLP72. Understanding the formats greatly aids understanding how various parameters relating to the formats should be set, and equally how issues such as precision and rounding are defined and controlled
- Integer Multiplication Primitives (see page 18) : A number of pre-defined integer multiplication primitives, derived from the MLP72, are available. These are easy to instantiate and in certain cases may be inferred. If the application only requires integer multiplication, then these are recommended over instantiating the full MLP72
- Floating Point Primitives (see page 19) : Two pre-defined floating point primitives,derived from the MLP72, are available. These are easy to instantiate and if the application simply requires floating point multiplication, then these are recommended over instantiating the full MLP72
- Dot Product Reference Designs (see page 20) : A suite of reference designs are available which show how to perform matrix dot product multiplication.
- Cascade Paths (see page 21) : The MLP72, (and associated BRAM72K and LRAM2K), have dedicated high speed cascade paths which allow connection of several primitives in series. Using these cascade paths allows for complex, multi-element, structures to be performed. Multiple MLP72 can be operated in parallel, cascading input data or results along the paths, in parallel with their associated BRAM passing pre-stored kernels or weights; or alternatively the BRAM address, allowing for different kernel and weights to be applied to each MLP72 whilst they share the same input data.
- Closely Coupled LRAM (see page 24) : The MLP72 has a closely coupled LRAM2K, which can be used as a register file, storing output results, and inserting them back into the inputs to be part of subsequent calculations. Alternatively the LRAM can be used to cache results allowing for subsequent efficient burst delivery.

Chapter - 2: Number Formats

Within the machine learning processor (MLP72), a variety of different number formats are used. It is important to understand these formats, how they are represented and what their resolution is, so that the user can make the correct choices with regard to math they wish to perform, and subsequently how to configure correctly for the chosen number formats.

Integer Formats

Integer values can be represented in three possible formats

- Signed – The highest order bit (MSB) represents the sign. The other bits equate to the distance from either 0 (for a positive number, sign bit = 1'b0) or the most negative possible value (sign bit = 1'b1). This format is commonly known as signed two's compliment.
- Unsigned – Only positive values are represented, with all bits representing the binary value, i.e., the distance from zero.
- Signed magnitude – The MSB is the sign bit. The remaining bits represent the value in the same form as unsigned, i.e., the distance from zero. Signed magnitude has a negative range one less than that of signed due to the fact that zero can be represented by 1000_0000 or 0000_0000 (using 8-bit values as the example).

The following examples shown how the same values can be represented by each method

Table 1: ***Integer Format Examples***

Decimal Value	Format	Bit 7	Bits [6:0]
24	Unsigned	0	001_1000
24	Signed	0	001_1000
24	Signed magnitude	0	001_1000
–24	Unsigned	Cannot be represented	
–24	Signed	1	110_1000(†)
–24	Signed magnitude	1	001_1000

Note

† For an 8-bit signed integer the most negative value is −128. The distance from the most negative value is 128 – 24 = 104. The binary representation of 104 is 110_1000.

Integer Groups

The formats are listed in their groups, which represent their size in bits. These group names are used in the MLP parameters to represent the size of the integers that will be directed to each respective multiplier. Within a group, each named format is followed by it's token name; these token names are used by the MLP parameters to configure the modes of each multiplier.

INT16

Table 2: ***Signed 16 Bit (Signed16)***

Bit Position	15	[14:0]	Range
Function	Sign bit	Value	−32768 to +32767

Table 3: ***Unsigned 16 Bit (Unsigned16)***

Bit Position	[15:0]	Range
Function	Value	0 to +65535

Table 4: ***Signed Magnitude 16 Bit (SMAG16)***

Bit Position	15	[14:0]	Range
Function	Sign bit	Value	−32767 to +32767

INT8

Table 5: ***Signed 8 Bit (Signed8)***

Bit Position	7	[6:0]	Range
Function	Sign bit	Value	−128 to +127

Table 6: ***Unsigned 8 Bit (Unsigned8)***

Bit Position	[7:0]	Range
Function	Value	0 to +255

Table 7: ***Signed Magnitude 8 Bit (SMAG8)***

Bit Position	7	[6:0]	Range
Function	Sign bit	Value	−127 to +127

INT7

Table 8: ***Signed 7 Bit (Signed7)***

Bit Position	6	[5:0]	Range
Function	Sign bit	Value	−64 to +63

Table 9: ***Unsigned 7 Bit (Unsigned7)***

Bit Position	[6:0]	Range
Function	Value	0 to +127

Table 10: ***Signed Magnitude 7 Bit (SMAG7)***

Bit Position	6	[5:0]	Range
Function	Sign bit	Value	−63 to +63

INT6

Table 11: ***Signed 6 Bit (Signed6)***

Bit Position	5	[4:0]	Range
Function	Sign bit	Value	–32 to +31

Table 12: ***Unsigned 6 Bit (Unsigned6)***

Bit Position	[5:0]	Range
Function	Value	0 to +63

Table 13: ***Signed Magnitude 6 Bit (SMAG6)***

Bit Position	5	[4:0]	Range
Function	Sign bit	Value	−31 to +31

INT4

Table 14: ***Signed 4 Bit (Signed4)***

Bit Position	3	[2:0]	Range
Function	Sign bit	Value	−8 to +7

Table 15: *Unsigned 4 Bit (Unsigned4)*

Bit Position	[3:0]	Range
Function	Value	0 to +15

Table 16: *Signed Magnitude 4 Bit (SMAG4)*

Bit Position	3	[2:0]	Range
Function	Sign bit	Value	−7 to +7

INT3

Table 17: *Signed 3 Bit (Signed3)*

Bit Position	2	[1:0]	Range
Function	Sign bit	Value	−4 to +3

Table 18: *Unsigned 3 Bit (Unsigned3)*

Bit Position	[2:0]	Range
Function	Value	0 to +7

Table 19: *Signed Magnitude 3 Bit (SMAG3)*

Bit Position	2	[1:0]	Range
Function	Sign bit	Value	−3 to +3

Floating Point Formats

A key feature of the MLP72 is the ability to process and manipulate floating-point (FP) numbers, which have a wider range of methods for representing values. Floating-point representations divide the bits into a mantissa and an exponent. The mantissa represents the value, and the exponent represents the equivalent of a bit shift left or right of this value. This shift is equivalent to multiplying the value by 2^n, where *n* represents the exponent value.

Formats

MLP72 supports three floating-point formats, characterized by their total size (`fp_size`) and by the number of exponent bits (`fp_exp_size`). The difference, `fp_size` − `fp_exp_size`, is the size of the mantissa bits and represents the precision. The three supported formats are listed below:

Table 20: *Supported Floating-Point Formats*

Format	FP Size	FP Exponent Size	Precision	MLP Name	Alternative Names
fp24	24	8	16	FP24	
fp16	16	5	11	FP16	binary16, half precision
bf16	16	8	8	BFLOAT16	bfloat16 (brain float). Not to be confused with block floating point.

Table Notes

The fp16 format is defined in the IEEE-754 standard as binary16. The other formats follow the IEEE-754 standard's rules for representation and rounding, but are not specifically defined in the standard. The bfloat16 format is supported by TensorFlow.

The MLP72 supports all three formats for both input and output, but internally all operations are performed with fp24. For example, when the internal accumulator is used, the accumulation is done with the extra precision of fp24, even if the output is one of the 16-bit formats.

Representation

The binary representation of a floating-point number has the form:

Table 21: *Floating-Point Number Representation*

	Sign	Exponent	Mantissa
Bits	1	fp_exp_size	(precision-1)

Positive numbers have sign = 0; negative numbers have sign = 1. The special cases of 0.0 and infinity can both have a sign.

The mantissa is normalized to have MSB = 1. Since the MSB is always 1, it is not stored. This "hidden 1" is why the precision is one higher than the number of bits in the mantissa. An exponent *e* is stored as *e* + bias, where the bias is defined in the table below. This table also lists the limits for the (absolute) value that can be represented.

There are two special exponents:

- If the exponent field is 0, the value of the floating-point number is +0.0 or −0.0.
- If the exponent field is all 1s (255 or 31, depending on fp_exp_size), the value is ±infinity.

If a number is not 0.0 and not infinity, its value is 1.mantissa × $2^{(exp-bias)}$ with the appropriate sign. Here, 1.mantissa starts with the hidden 1 and has the mantissa as binary fraction.

Table 22: *Floating-Point Number Range*

Format	Bias	Exp for inf	Minimum Positive	Maximum Positive
fp24	127	255	$2^{(-126)}$	$2^{128} - 2^{112}$
fp16	15	31	$2^{(-14)}$	$2^{16} - 2^{5} = 65504$
bf16	127	255	$2^{(-126)}$	$2^{128} - 2^{120}$

Subnormal numbers (numbers too close to 0 to be normalized) are not supported; they are changed to 0.0. Likewise, NaN ("not a number", for invalid operations), is not supported; it is changed to infinity.

Rounding

When the result of an operation does not fit exactly within the precision, it is rounded to the nearest value that can be represented. If the true result is equally close to two values, it is rounded to the even one (the one that has LSB = 0).

If the absolute value of a result is too large to be represented, the result is changed to infinity. Similarly, if the absolute value of a result is too small to be represented, it is changed to 0. The latter case is called underflow, describing the situation where the floating-point result is 0.0 but mathematically the result should not be 0 (for example, subtracting two not quite equal numbers may result in underflow if the numbers are small).

Internally, all operations are performed with fp24, including the rounding of multiplications and additions. If a 16-bit output format is selected, the fp24 result is then rounded a second time to 16 bits. In rare cases, this double rounding may give a slightly different result than if the operation had been done with 16 bits only.

Block Floating Point

Block floating point (block fp) is not a number format *per se*, rather it is a method of processing a collection of floating point numbers efficiently. The principle is that each floating point number consists of a mantissa and an exponent. If the exponents are made the same (normalized), then the mantissas can be arithmetically combined in the same way as integer numbers. The result of the mantissa calculation can be recombined with the normalized exponent, resulting in a full floating point result. During this process there are two semi-independent input formats involved:

- The original floating point format
- The integer format used for the multiplication

There are a number of considerations when normalizing the exponent among a collection of floating point numbers. The value of a floating point number is

`mantissa × 2exponent` where `mantissa` is of the form 1.`fraction`.

However, in order to increase accuracy for the same number of bits, the '1' is not stored, it is implicit. A floating point number is stored as {`sign_bit, exponent, fraction`}.

When converting from a floating point value to a block floating point value, firstly the '1' needs to be added to the fraction in order to give the full mantissa. The mantissa is then right-shifted, while incrementing the exponent, until the exponent has the desired value equivalent to the maximum exponent in the block. It is this shift process that requires the implicit '1' to firstly be re-inserted with the `fraction`; after shifting the implicit '1' is no longer in the MSB position.

The shifted mantissa plus sign bit must fit within the integer format that is used. If it is necessary to right-shift a significant amount, then precision is lost. It is possible to end up with 0 if the original exponent was small enough. The choice of integer size, therefore, depends on the required precision and the range of exponent values that are to be processed together.

Chapter - 3: Integer Multiplication Primitives

If the user application only requires integer multiplication, with no complex data re-use or results caching; then there are three integer multiplication primitives available. These primitives are detailed in Speedster7t IP Component Library User Guide. The primitives offer the following capabilities

MLP72_INT

The MLP72_INT supports up to 12 integer multiply operations, followed by an adder tree and an optional accumulate. The number of arithmetic operations that can be supported depends on the operand width, where more arithmetic operations can be supported per clock cycle with narrower operands. Inputs can be encoded as unsigned integers, signed two's-complement integers, or signed-magnitude integers. Operands width can be in the range 3 to 16 bits. Outputs are always 48-bit signed integers.

MLP72_INT8_MULT_4X

The MLP72_INT8_MULT_4X primitive is a simple multiplier block with support for up to four parallel multipliers using 8-bit two's-complement signed, signed magnitude, or unsigned integers. For higher performance operation, additional input and/or output registers can be enabled. Enabling each register causes an additional cycle of latency.

MLP72_INT16_MULT_2X

The MLP72_INT16_MULT_2X primitive is a simple multiplier block with support for up to two parallel 16-bit multipliers using 16-bit two's-complement signed, signed magnitude, or unsigned integers. For higher performance operation, additional input and/or output registers can be enabled. Enabling each register causes an additional cycle of latency.

In the circumstance where one of the above primitives would fulfil the application needs, the user is recommended to use the macro in preference to instantiating the full MLP72 primitive. The primitives offer a simpler, easier to understand interface, and use the optimal configuration of the MLP72, to achieve the highest possible results, with the minimum of debug effort from the user. None of the macros can currently be inferred, however instantiation templates for each macro are provided in Speedster7t IP Component Library User Guide

Chapter - 4: Floating Point Primitives

The user is recommended to read the introduction to the floating point library in Speedster7t IP Component Library User Guide as this explains the principles behind the floating point primitives, and also details files that need to be included in the users build environment in order to be able to use these floating point primitives. The primitives are detailed belowIf the user application requires simple floating point multiplication with one or two floating point multiplications per MLP72, then there are two floating point primitives that are available. These primitives greatly simplify using the MLP72 in floating point mode, and if the application does not require advanced data re-use, or summation of multiple results, then they are recommended over directly instantiating the MLP72.

FP_MULT

FP_MULT computes a × b, with optional accumulation. The floating point range is up to fp24, with an 8 bit exponent and 16 bit precision. For further details on floating point number formats, please refer to Number Formats (see page 11). Currently this module has to be directly instantiated; it cannot be inferred.

FP_MULT_2X

This macro is similar to FP_MULT (see page 19) but uses a single MLP72 to compute two products in parallel, with optional accumulation. The two operations are:

- `dout_ab = a × b`
- `dout_cd = c × d`

The floating point number range is the same as for FP_MULT (see page 19). Similarly this module has to be directly instantiated; it cannot be inferred.

Chapter - 5: Dot Product Reference Designs

The flexibility of the MLP72 results in there being a number of different methods for structuring the data flow to perform dot-product multiplication. To aid the user, there are a number of reference designs that perform dot product calculations on differing numerical formats, and differing vector sizes. These dot product reference designs are detailed in the Speedster7t Reference Designs User Guide. The code for these reference designs can be downloaded from the Achronix secure FTP site at secure.achronix.comOne of the most common uses of the MLP72, certainly within the Machine Learning, or Artificial Intelligence, (AI/ML), solution space, is the ability to multiply matrices together. This process whereby each element of a row is multiplied by the value of its equivalent column location, and the sum of all the multiplications forms the result in the new matrix, is known as dot-product. The MLP72 is ideally suited to this dot-product manipulation, offering summation and accumulation from each of the 16 possible multiplications.

If the user needs to perform matrix multiplication, and hence dot-product calculation, they are recommended to refer to these designs to help determine the correct architecture for their application.

There are three dot product reference designs

dot_product_N_8x8

This design demonstrates the dot product of a sequence of int8 values. On each cycle *N* int8 inputs are multiplied and summed. The result is then accumulated and available two cycles after the input. The accumulation can be controlled by first and last indicators inputs.

dot_product_2bf16_doutcasc4

This design demonstrates the dot product of a number of bfloat16 (brain-float) inputs. The design consists of four MLP72, connected using their cascade paths. Each MLP72 sums the result of two parallel multiplications, with each multiplication being the result of an a input multiplied by a b input (both being bfloat16 numbers). The sum from each MLP72 is cascaded up the column of MLP72 to the next block above. In the last MLP72, on each cycle, the sum of the eight parallel bf16 multiplications is calculated.

The final result is the accumulated sum across a number of input cycles.

dot_product_10bfp_fp10_6x6

This design computes the accumulated sum of a block of 10 floating-point number pairs. Each pair of numbers is multiplied and the result added to the sum. The inputs are in fp10 format (5 bits mantissa + 5 bits exponent). This design uses a single MLP72, and its tightly coupled BRAM72K, to demonstrate how the combination of blocks can process a total input bit width that is larger than any of the individual inputs bit widths. This capability is achieved by the design operating in two phases; in phase one the b inputs are written to the BRAM. In phase two the a inputs are applied to the MLP72 as the b values are read from the BRAM72K.

An advantage of this structure is that if the b values need to be used more than once, such as for a kernel or weights, then they only need be written the once and can be re-read multiple times as different sets of a inputs are applied.

During phase two, the final result is the accumulated sum across the number of input cycles. The number of cycles of a input must match the number of stored b values written.

Chapter - 6: Cascade Paths

The MLP72, (and associated BRAM72K and LRAM2K), have dedicated high speed cascade paths which allow connection of several primitives in series. Using these cascade paths allows for complex, multi-element, structures to be performed. Multiple MLP72 can be operated in parallel, cascading input data or results along the paths, in parallel with their associated BRAM passing pre-stored kernels or weights; or alternatively the BRAM address, allowing for different kernel and weights to be applied to each MLP72 whilst they share the same input data. Introduction

The MLP72 cascade paths are listed below

Table 23: *MLP72 Cascade Paths*

Port	Direction	Description
fwdi_multa_h[71:0]	Input	Forward cascade path inputs for multiplier A inputs, higher multiplier block
fwdi_multb_h[71:0]	Input	Forward cascade path inputs for multiplier B inputs, higher multiplier block
fwdi_multa_l[71:0]	Input	Forward cascade path inputs for multiplier A inputs, lower multiplier block
fwdi_multb_l[71:0]	Input	Forward cascade path inputs for multiplier B inputs, lower multiplier block
fwdo_multa_h[71:0]	Output	Forward cascade path output for multiplier A inputs, higher multiplier block
fwdo_multb_h[71:0]	Output	Forward cascade path output for multiplier B inputs, higher multiplier block.
fwdo_multa_l[71:0]	Output	Forward cascade path output for multiplier A inputs, lower multiplier block
fwdo_multb_l[71:0]	Output	Forward cascade path output for multiplier B inputs, lower multiplier block.
fwdi_dout[47:0]	Input	MLP72 internally calculated result, cascaded from MLP72 below
fwdo_dout[47:0]	Output	MLP72 internally calculated results, cascaded up to MLP72 above

Usage Conditions

A number of conditions are required to use cascade paths;

- Cascade paths run up, (and also down), a column of identical elements. In the case of the MLP72, the cascade paths only run up a column, and as such are named forward, (fw), paths.
- All forward cascade paths inputs will come from the MLP72 directly below
- All forward cascade path outputs will go to the MLP72 directly above
- It is not possible to bypass an MLP72 in a column that is using cascade paths
- It is not possible to drive the cascade path inputs from the fabric. For the start of the path, the user may use the ACX_FLOAT macro to create a floating signal which may be connected to the input of the lowest MLP72, (see below)

- It is not possible to use the cascade outputs in the fabric. The top of the cascade chain must be left open circuit.

The following code is suggested for implementing the floating input to the bottom cascade chain

```
code

// ------------------------------------------
// Cascade paths
// ------------------------------------------
// Create a float wire for cascade signals
wire float;
ACX_FLOAT X_ACX_FLOAT(.y(float));


// ------------------------------------------
// Example connectivity
// ------------------------------------------
wire [71:0]  fwd_multa_h[NUMBER_OF_MLP:0];
assign       fwd_multa_h[0] = {72{float}};
```

Solutions

There are a number of application requirements that can be solved by the use of the cascade paths

Input re-use

The four buses that feed the multiplier array, `multa_l`, `multa_h`, `multb_l` and `multb_h` are all output as similarly named forward cascade paths, (`fwdo_multa_l` etc.). Equally each is available as an cascade path input, (`fwdi_multa_l` etc.) to its respective input selection mux.

Therefore data values can be input to the bottom of a column of MLP72, and cascaded up to all MLP72 in the column. This greatly simplifies routing, and also improves timing as the data source only has to fanout to a single input on the bottom MLP72. In this scenario one of the multiplier inputs, for example all the "A" inputs, would be cascaded up the column, and then the corresponding "B" inputs would be input directly to each MLP72. This allows for a column of MLP72 to process the same source data, (the "A" inputs), with different coefficients or weights, (the "B" inputs), in a parallel structure.

In order to achieve high performance, it is recommended that the delay registers, `del_multa_l` etc. are enabled. With these enabled the data is able to traverse the cascade chain at approximately 750MHz, (fastest speed grade). Having the registers enabled results in each MLP72 in the column processing the data one cycle after the MLP72 below it. If the second data source, (the "B" inputs") are equally delayed by one cycle per MLP72, then the resultant output from each MLP72 will also be delayed by a cycle for each MLP72 in the column. This can have advantages in terms of collecting the results, and streaming them to either a following processing stage, or storing the results into memory.

Example

An example of the input cascade paths is to be found in the Speedster7t Reference Designs User Guide with the mlp_conv2d reference design. This design passes the image data that is to be processed up the cascade path, and this data is then convolved with weights read from each MLP72's local BRAM72K. This allows a single stream of image data to be processed by 16 MLP72s in parallel, and the whole design applies 60 MLP72s in parallel, with the source data only needing to fanout to the 4 MLP72 at the bottom of each column.

To further illustrate the power of the cascade paths, in the above mlp_conv2d reference design, the BRAM72K themselves are programmed using the cascade path, so allowing the weight data source to again only fanout to the four columns of MLP72/BRAM72K. In addition, as the weights are read from BRAM, the read address is also cascaded up the column, with a delay stage enabled between each BRAM. This structure ensures that each MLP72 has the image data and weights correctly phased with each MLP72 operating one cycle behind the one below it.

Output addition

The internally calculated 48 bit result of the MLP72 calculation is output as fwdo_dout. This cascade path connects to the MLP72 above, into the sum / accumulator for each half of the multiplier tree. This allows for the sum of the MLP72 below to be added, or subtracted, from the current MLP72 sum. Using cascaded outputs it is possible to build extended MLP72 which support accumulation and summation across a wide number of inputs

Chapter - 7: Closely Coupled LRAM

Introduction

The MLP72 has an integrated Logic 2k bit RAM, (LRAM), tightly bonded to both its external inputs, and internal signals. This LRAM enables local storage and reuse of both input values, and output results. The LRAM is often referred to as a register file, particularly when it is configured to store and replay MLP72 results. The LRAM can be configured as 36 bits × 64, 72 bits × 32, or 144 bits × 16, dependant upon application.

Modes

When the LRAM is used as an integrated part of the MLP72, it can be operated in three modes; (the modes values correspond to the values set for the `lram_input_control_mode` and `lram_output_control_mode` parameters.

- Mode 0, (default). LRAM is slaved to co-sited BRAM72K. The LRAM operates as an extension to the BRAM72K, supporting additional address space. The data, read and write signals are connected from the BRAM72K to the LRAM using the dedicated signal paths. This mode is intended for initialising the LRAM, via the NOC, during power up.
- Mode 1. LRAM operates as either RAM or FIFO. Re-purposing several dual use MLP72 inputs the LRAM can store the results of the MLP72 calculation, and its output can be routed back into the MLP72 Input Selection (see page) stage.
- Mode 2. With the LRAM set to operate as a FIFO in Mode 1, Mode 2 adds additional signals that allow reset of the FIFO address generators. This additional flexibility allows for when the LRAM may be storing groups of results or coefficients that do not necessarily match the length of the FIFO, i.e. their length is not a power of 2^n.

Solutions

The LRAM can be operated as a register file, storing output results, and adding them back into the multiplier outputs

Revision History

Version	Date	Description
0.9	06 Aug 2019	• Initial preliminary release.