Speedster7t Machine Learning Processor User Guide (UG088)

Speedster FPGAs

Copyrights, Trademarks and Disclaimers

Copyright © 2026 Achronix Semiconductor Corporation. All rights reserved. Achronix, Speedcore, Speedster, and ACE are trademarks of Achronix Semiconductor Corporation in the U.S. and/or other countries All other trademarks are the property of their respective owners. All specifications subject to change without notice.

Notice of Disclaimer

The information given in this document is believed to be accurate and reliable. However, Achronix Semiconductor Corporation does not give any representations or warranties as to the completeness or accuracy of such information and shall have no liability for the use of the information contained herein. Achronix Semiconductor Corporation reserves the right to make changes to this document and the information contained herein at any time and without notice. All Achronix trademarks, registered trademarks, disclaimers and patents are listed at http://www.achronix.com/legal.

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 1
Scope 4
Sections 4
Chapter 2 : Number Formats 5
Integer Formats 5
Integer Sizes 5
Floating-Point Formats 6
Formats 6
Representation 7
Rounding 8
Block Floating Point 8
Chapter 3 : Integer Multiplication Primitives 10
MLP72_INT 10
MLP72_INT8_MULT_4X 10
MLP72_INT16_MULT_2X 10
Chapter 4 : Floating Point Primitives 11
Speedster7t FP_ADD 11
Speedster7t FP_MULT 11

Speedster7t FP_MULT_PLUS 11
Speedster7t FP_MULT_2X 11
Speedster7t FP_MULT_ADD 11
Chapter 5 : MLP Reference Designs 12
Reference Designs Available 12
Split MLP Shared BRAM Stack 12
Dot Product 12
dot_product_N_8x8 .... *13*
dot_product_bfloat16_4mlp *13*
dot_product_fp16_4mlp *13*
Matrix Vector Multiply 13
Chapter 6 : MLP Cascade Paths 14
Usage Conditions 15
Solutions 15
Input Re-use 15
Output Addition 16
Example *17*
Chapter 7 : Closely Coupled LRAM 18
Modes 18
Solutions 18
Chapter 8 : Revision History 19

Chapter 1 : Introduction

The machine learning processor block (MLP) is an array of up to 32 multipliers, followed by an adder tree, and an accumulator. The MLP is also tightly coupled with two memory blocks, a BRAM72k and LRAM2k. These memories 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 number of data inputs to the MLP block increases, enabling the use of additional multipliers.

The MLP offers a range of features:

- Configurable multiply precision and multiplier count. Any of the following modes are available:
  - Up to 32 multiplies for 4-bit integers or 4-bit block floating-point values in a single MLP
  - Up to 16 multiplies for 8-bit integers or 8-bit block floating-point values in a single MLP
  - Up to 4 multiplies for 16-bit integers or 16-bit block floating-point values in a single MLP
  - One multiplier for 32-bit integers in a single MLP
  - Up to 2 multiplies for 16-bit floating point with both 5-bit and 8-bit exponents in a single MLP
  - Up to 2 multiplies for 24-bit floating point in a single MLP
- Multiple number formats:
  - Integer
  - Floating point 16 (IEEE half-precision and TensorFlow bfloat16 formats)
  - Floating point 24
  - Block floating point, a method that combines the efficiency of the integer multiplier-adder tree with the range of the floating point accumulators
- Adder tree and accumulator block
- Tightly-coupled register file (LRAM) with an optional sequence controller 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. In addition, a feedback path allows for accumulation within the MLP block.

The following block diagrams show the MLP using the fixed or floating-point formats:

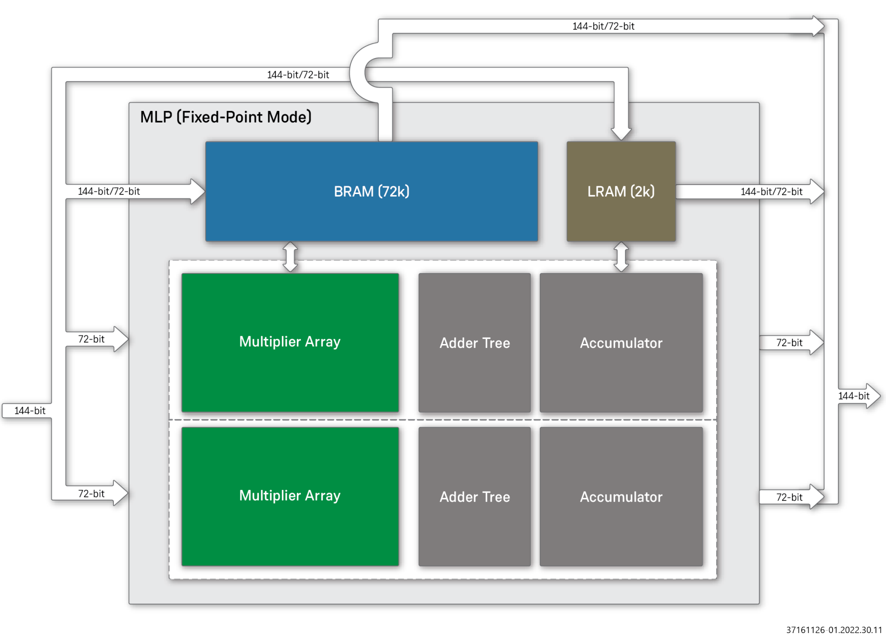

Figure 1 • MLP Using Fixed-Point Mode

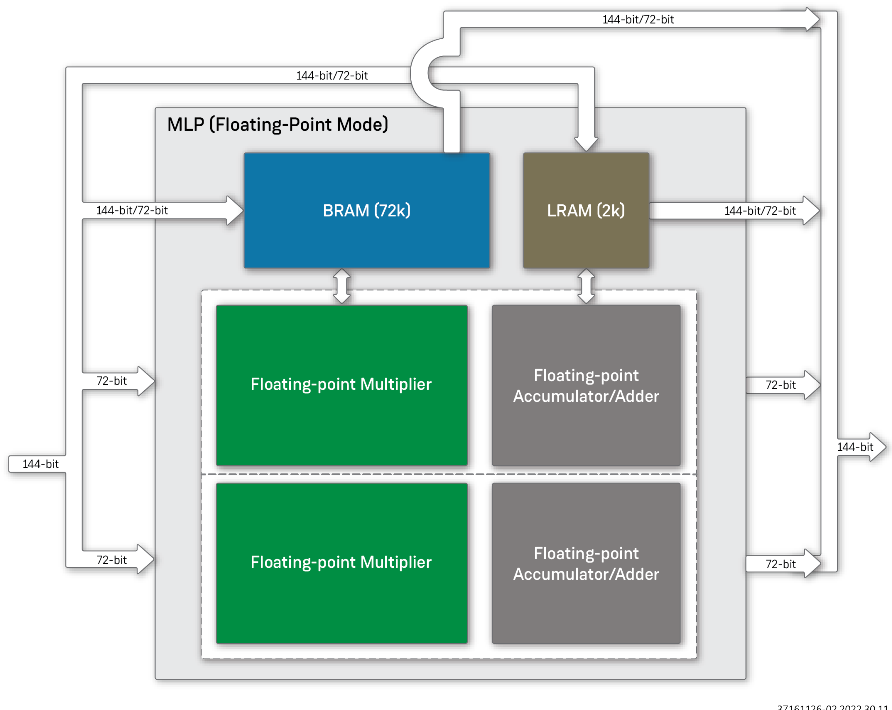

Figure 2 • MLP Using Floating-Point Mode

A powerful feature available in the Achronix MLP is the ability to connect several MLPs with dedicated high-speed cascade paths. The cascade paths allow for the adder tree to extend across multiple MLP blocks in a column without using extra fabric routing resources, and a data cascade/broadcast path is available to send operands across multiple MLP blocks. Cascading input or result data to multiple MLPs in parallel allows for complex, multi-element operations to be performed efficiently without the need for extra routing. The following diagram shows the cascade paths across MLPs:

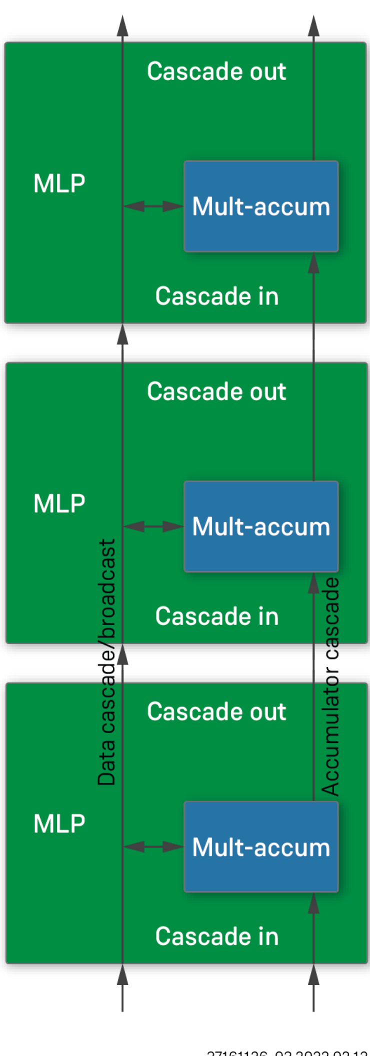

Figure 3 • MLP Cascade Path

Scope

This user guide is intended to be used in conjunction with ***Speedster7t Component Library User Guide* (UG086)**[1] 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** (page 5) – This section 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** (page 10) – A number of predefined integer multiplication primitives, derived from the MLP72, are available. These primitives 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** (page 11) – Two predefined floating-point primitives, derived from the MLP72, are available. These primitives 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** (page 12) – A suite of reference designs is available which show how to perform matrix dot-product multiplication.
- **Cascade Paths** (page 14) – 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 prestored kernels or weights; or alternatively the BRAM address, allowing for different kernel and weights to be applied to each MLP72 while they share the same input data.
- **Closely Coupled LRAM** (page 18) – 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.

---

1 https://www.achronix.com/documentation/speedster7t-component-library-user-guide-ug086

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 the resolutions they provide — so that appropriate decisions can be made about the mathematical operations to perform and how to correctly configure the block for the selected number format.

Integer Formats

Integer values can be represented in three possible formats:

- Signed – Signed binary value in two's complement format. The same format used for Verilog signed integers.
- Unsigned – Standard unsigned binary number, the same format used for unsigned Verilog integers.
- Signed magnitude – The MSB is the sign bit, and the remaining bits represent the unsigned absolute value.

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**
>
> The MLP72 supports any of the three integer formats as the input format, selected with parameters. However, the integer output format is either unsigned (if all inputs are unsigned), or signed (two's complement); signed magnitude is not supported as the output format.

Integer Sizes

The MLP72 supports the following integer input sizes.

Size in Bits	Signed Range	Unsigned Range	Signed Magnitude Range
32	−2,147,483,648 to +2,147,483,647	0 to +4,294,967,295	Unsupported
16	−32,768 to +32,767	0 to +65,535	
8	−128 to +127	0 to +255	−127 to +127
7	−64 to +63	Unsupported	−63 to +63
6	−32 to +31		−31 to +31
4	−8 to +7		−7 to +7
3	−4 to +3		−3 to +3

Floating-Point Formats

A key feature of the MLP72 is the ability to process and manipulate floating-point (FP) numbers. 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 2 • Supported Floating-Point Formats

Format	FP Size	FP Exponent Size	Precision	Alternative Names
fp24	24	8	16	
fp16	16	5	11	binary16, half precision
fp16e8	16	8	8	bfloat16 (brain float). Not to be confused with block floating point.

Format	FP Size	FP Exponent Size	Precision	Alternative Names

Table Note

- 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 performed 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 3 • 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 a binary fraction.

Table 4 • 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	$2^{(-14)}$	$2^{16} - 2^{5} = 65504$

Format	Bias	Exp for inf	Minimum Positive	Maximum Positive
fp16e8	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 might 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 can give a slightly different result than if the operation had been performed with only 16 bits.

Block Floating Point

Block floating point (called MXINT by the Open Compute Project) 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 (by shifting the mantissas), 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 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

The value of a floating-point number is

$$\text{mantissa} \times 2^{\text{exponent}}$$

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, 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 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 group of floating-point numbers that is converted to a common exponent is called a block. Typically, a smaller block size is preferred because it leads to less loss of precision. The MLP72 takes in one or two blocks every clock cycle, supporting a block size as small as 8 values for int8.

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 Component Library User Guide*** (UG086)[2].

In the circumstance where one of the following primitives would fulfill the application needs, the user is recommended to use these primitives in preference to instantiating the full MLP72 primitive. These 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. The instantiation templates for each are provided in ***Speedster7t Component Library User Guide*** (UG086)[3]. In addition to these primitives, the Speedster7t Integer Library, also documented in the same guide, has macros that map standard integer operations to an MLP. These macros can also be instantiated with the IP Configuration tool within ACE.

These 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.

2 https://www.achronix.com/documentation/speedster7t-component-library-user-guide-ug086
3 https://www.achronix.com/documentation/speedster7t-component-library-user-guide-ug086

Chapter 4 : Floating Point Primitives

The user is recommended to read the introduction to the floating-point library in the **Speedster7t Component Library User Guide**[4] (UG086) 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 this floating-point library. If the user application requires simple floating-point multiplication or addition with one or two floating-point multiplications per MLP72, then the library components listed should support these requirements. 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.

All the library components listed below support a floating-point range of up to fp24, with an 8-bit exponent and 16-bit mantissa precision. For further details on floating-point number formats, refer to chapter, "**Number Formats** (page 5)".

The library modules have to be directly instantiated; they *cannot* be inferred. Suitable instantiation templates are included in the **Speedster7t Component Library User Guide**[5] (UG086). These modules can also be instantiated with the IP Configuration tool within ACE.

Speedster7t FP_ADD

FP_ADD computes a + b, with optional accumulation.

Speedster7t FP_MULT

FP_MULT computes a × b, with optional accumulation.

Speedster7t FP_MULT_PLUS

FP_MULT_PLUS computes a × b + c, with optional accumulation.

Speedster7t FP_MULT_2X

This macro is similar to **FP_MULT** (page 0) 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`

Speedster7t FP_MULT_ADD

FP_MULT_ADD computes (a x b) + (c x d), with optional accumulation.

4 https://www.achronix.com/documentation/speedster7t-component-library-user-guide-ug086
5 https://www.achronix.com/documentation/speedster7t-component-library-user-guide-ug086

Chapter 5 : MLP 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 or vector/matrix multiplication. To aid the user, Achronix has created a number of reference designs that perform these calculations on differing numerical formats, and differing vector and matrix sizes. These reference designs are made available in the KBA, "**How do I Download Demonstration and Reference Designs?**"[6]. A reference design user guide is also included.

Note

A support account is needed in order to access and download these reference designs. See the KBA, "**How Do I Register for an Achronix Support Account?**[7]"

One 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.

Reference Designs Available

Split MLP Shared BRAM Stack

This reference design, `split_mlp_shared_bram_stack`, shows how to share BRAM among MLPs. In this example, MLPs are configured as a number of parallel dot-product engines. A stack of MLPs with associated BRAMs are created but the BRAMs are shared so that nearly half of the device's BRAMs are free to be used for other purposes.

Dot Product

The dot-product suite of reference designs demonstrates how to perform vector multiplication, with the result being the sum of each of the elements of the vectors multiplied together, commonly known as the dot product. The MLP72 for the different designs are configured differently, demonstrating the range of numerical types and multipliers supported by the MLP72.

6 https://support.achronix.com/hc/en-us/articles/38065028969236-How-do-I-Download-Demonstration-and-Reference-Designs

7 https://support.achronix.com/hc/en-us/articles/4404014677268-How-Do-I-Register-for-an-Achronix-Support-Account

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.

dot_product_bfloat16_4mlp

This design demonstrates the dot product of a number of "brain floating point 16" (bfloat16) 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 bfloat16 multiplications is calculated.

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

dot_product_fp16_4mlp

This design demonstrates the dot product of a series of 16-bit floating-point values. The design consists of four MLP72s, connected using their cascade paths. Each MLP72 sums the result of two parallel multiplications, with each multiplication being the result of an `i_a` input multiplied by an `i_b` input (both being fp16 values). The sum from each MLP72 is cascaded up the column of MLP72s to the next block above. On each cycle, the sum of the eight parallel fp16 multiplications is calculated in the last MLP72.

The final result is the accumulated sum across a number of input cycles. The accumulation is controlled by the `i_first` and `i_last` inputs. The `i_first` input signal indicates the first set of inputs to sum and to zero the accumulation. The `i_last` signal indicated the last set of inputs to sum and add to the accumulation. The final value is available six cycles after `i_last` , and is qualified with `o_valid`.

Matrix Vector Multiply

The matrix vector multiply (MVM) design, `mvm_8mlp_16int8_earlyout`, multiplies a 256 × 256 matrix with a 256 × 1 vector, using int8 input values. The design uses 8 MLP72 and 8 BRAM72K instances. The 256 × 256 matrix is stored in the BRAMs, and can be used multiple times. No extra storage is needed for the 256 × 1 input vector: it is stored in the LRAMs that are closely coupled to the MLP72.

Input of data is at the bottom of the stack; the output of results is at the top. The design uses both the internal cascade connections and the direct connection between BRAMs and MLPs. The use of these internal connections greatly reduces the use of fabric routing resources, and simplifies timing as well.

Chapter 6 : MLP Cascade Paths

The MLP72, (and associated BRAM72K and LRAM2K), have dedicated high-speed cascade paths which allow connection of several primitives in series. These cascade paths enable complex, multi-element operations, such as systolic computation, 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. Alternatively, the BRAM address can be cascaded, allowing for different kernel and weights to be applied to each MLP72 while still sharing the same input data.

The MLP72 cascade paths are listed below

Table 5 • 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.
  - Drive the cascade path inputs directly from the fabric. Instead, the bottom MLP of a cascade chain can use its regular fabric inputs to drive its cascade outputs.
  - Use the cascade outputs in the fabric. Instead, the top MLP of a cascade chain should use its regular (non-cascade) output.

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 a 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 blocks and cascaded up to all MLP72 blocks in the column. This capability 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 path allows for a column of MLP72 blocks 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 registers enabled, the data is able to traverse the cascade chain at approximately 750 MHz (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 delay can have advantages in terms of collecting the results, and streaming them to either a following processing stage, or storing the results into memory.

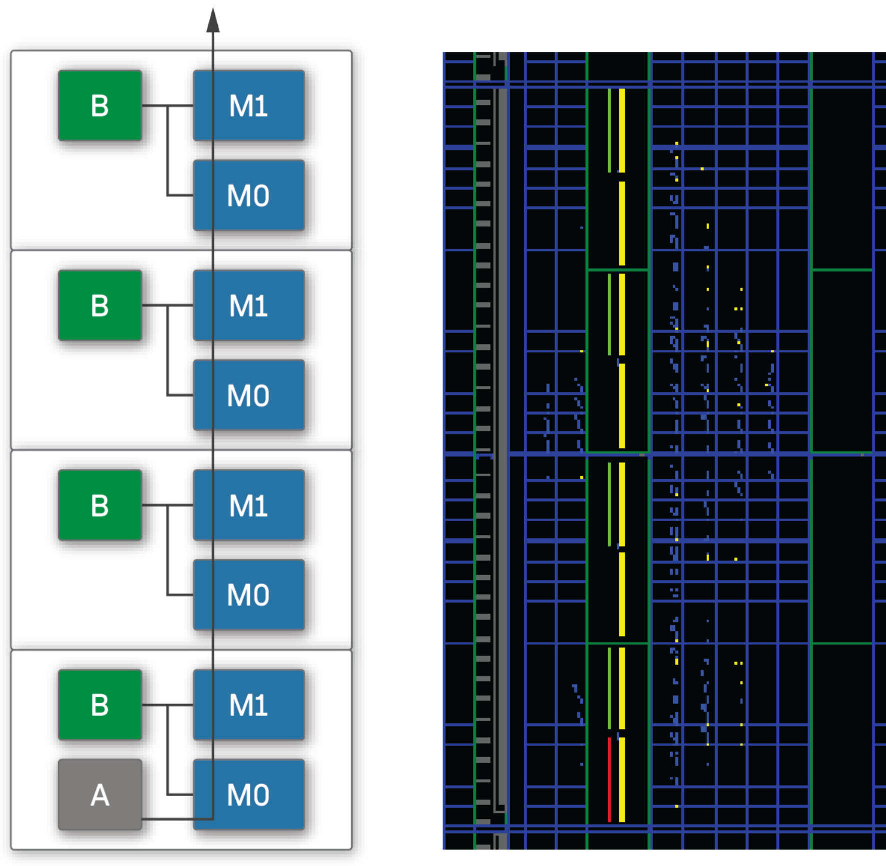

Figure 4 • Input Cascade and Floorplanner View. Data A is Read from the Bottom BRAM and Cascaded up to Each MLP72.

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 path 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.

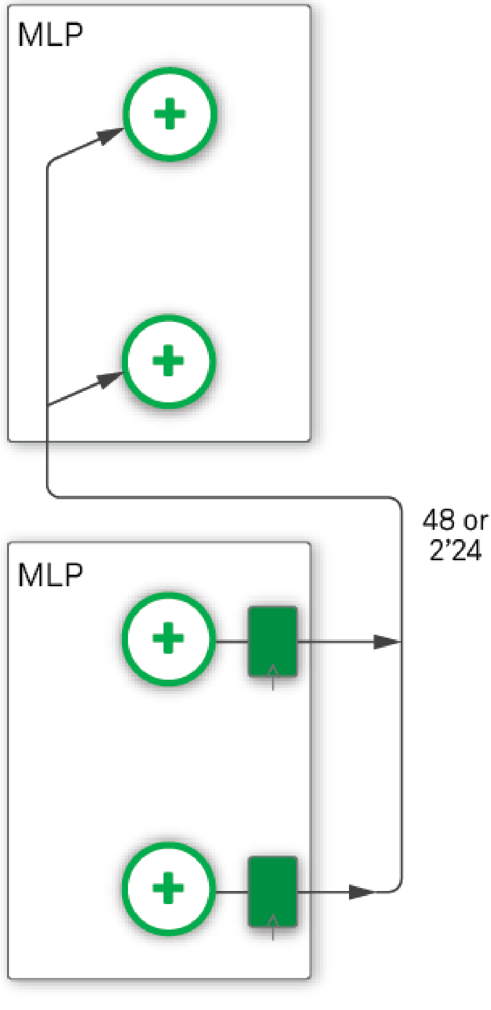

Figure 5 • Output Cascade Path

Example

Several designs in the Speedster7t MLP Reference Designs[8] uses the cascade paths. In the Matrix Vector Multiply reference design, both input and output cascade paths are utilized. In the design, vector data is distributed via the input cascade path and the MLP output cascade is used to collect and accumulate the output of each stage.

8 https://support.achronix.com/hc/en-us/articles/38065028969236-How-do-I-Download-Demonstration-and-Reference-Designs

Chapter 7 : Closely Coupled LRAM

The MLP72 has an integrated 2k-bit local 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, dependent upon application.

Modes

The LRAM has several different addressing modes. The address can be:

- Provided from the fabric – The LRAM can then be used as standalone RAM or as register file for the MLP72.
- Generated with internal counters – This is used when the LRAM is configured for FIFO operation, but can also be used for non-blocking iterative access.
- Provided from the adjacent BRAM72K – In this mode, the LRAM acts as extension of the BRAM address space.

Solutions

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

Several designs in the **Speedster7t MLP Reference Designs**[9] uses the MLP's LRAM. The dot-product examples store accumulated results in the LRAM FIFO. In the Matrix Vector Multiply example, the input vector is stored in LRAMs that are closely coupled to the MLP72.

Note

A support account is needed in order to access and download these reference designs. See the KBA, "**How Do I Register for an Achronix Support Account?**[10]"

9 https://support.achronix.com/hc/en-us/articles/38065028969236-How-do-I-Download-Demonstration-and-Reference-Designs
10 https://support.achronix.com/hc/en-us/articles/4404014677268-How-Do-I-Register-for-an-Achronix-Support-Account

Chapter 8 : Revision History

Version	Date	Description
1.0	21 Apr 2026	• Initial Achronix release.