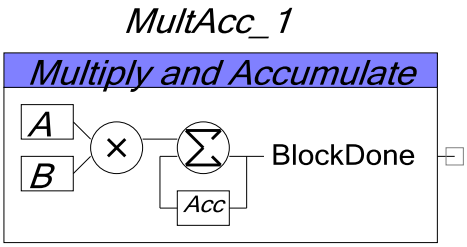
MultAcc
1.00



Features

- Hardware 24 x 24 bit Multiply and Accumulate block (MACC)
- Q23 fixed point input and output
- Accumulator overflow detection and saturation
- Dedicated processing engine requires zero CPU for multiplying / accumulating blocks of data
- Q23 to hex / floating point converter calculator and functions included
- Data arrays and DMA preconfigured for easy use
- Optional hardware “end of calculation” signal for triggering DMA or an interrupt
- ~ 18 cycles per MACC operation (100 elements takes ~1800 cycles)

General Description

A multiply and accumulate operation performs a multiplication of A and B and adds the result to an accumulator. The result of this operation on a block of data A_n and B_n looks like this:

$A_0*B_0+A_1*B_1+A_2*B_2+ A_3*B_3+...+A_n*B_n = \sum_{i=0}^{i=n} A_i * B_i$

Signal processing and digital control applications require the use of a multiply and accumulate operation on sets of data. A Finite Impulse Response (FIR) filter is constructed by multiplying historical input data by the impulse response of the filter and accumulating those results. Infinite Impulse Response (IIR) filters are constructed by taking historical inputs and outputs and multiplying / accumulating them with specific coefficients to achieve a particular filter response. Control systems take historical inputs and outputs and multiply / accumulate them with specific coefficients to control the input to a system to generate a specific response.

At their core, many complex signal processing applications (from FFTs to Correlation and Convolution) can be broken down into a series of multiply and accumulate operations. This component provides a hardware peripheral that performs multiply and accumulate operations on blocks of data from 1 to 1023 elements in size (from A_0*B_0 to $A_0*B_0+A_1*B_1+...+A_{1022}*B_{1022}$). The maximum size is limited by the maximum number of bytes that single DMA transaction descriptor can transfer.

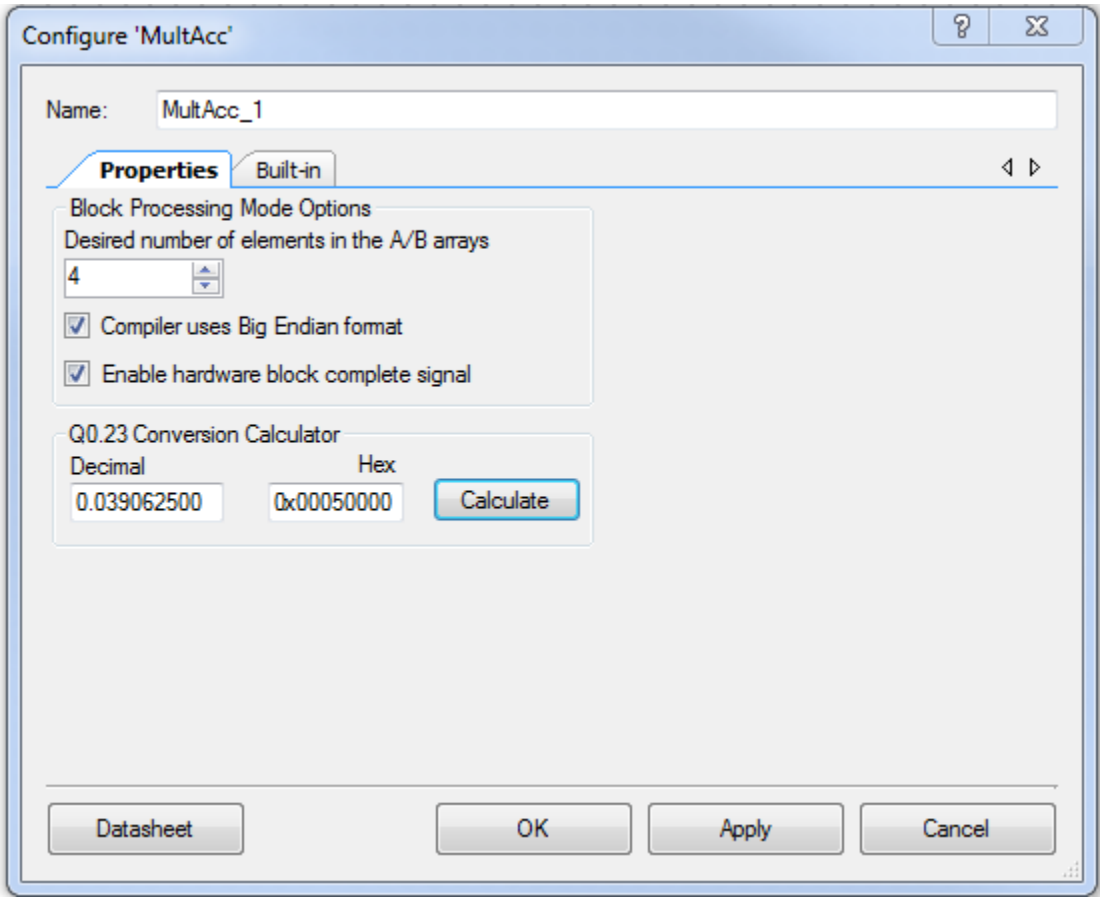
The MultAcc component automatically generates 2x 32 bit arrays of configurable size to hold the 24 bit A_n and B_n data, and configures 2 DMA channels to automatically move the data from the arrays into the multiply and accumulate hardware. The multiply and accumulate hardware computes the sum of the products and signals when it has completed the calculation on the blocks of data. The multiply and accumulate hardware operates on fixed point data in Q23 format (MSb has a value of -1 and the remaining 23 bits have magnitude of $\frac{1}{2^n}$ = ½, ¼, …). For simplicity, the component includes functions to convert a floating point value into it a Q23 number and back from a Q23 value into floating point number. The customizer also includes a Q23 converter that converts decimal numbers into their equivalent Q23 value and hexadecimal format. The converter can also change a hexadecimal value into its equivalent Q23 value. This calculator can improve efficiency by pre computing the hexadecimal value for a number and allowing the program to store and operate on the 32 bit hex value (as opposed to a floating point representation of the number).

Some of the features of the hardware include overflow detection and saturation protection. If the accumulator detects and overflow/underflow condition, it will saturate at the 24 bit numerical maximum/minimum value and set a flag on the calculation which can be checked at the end of the computation. The customizer also allows for selection of a Big Endian or Little Endian compiler to configure endian swapping in the DMA channels.

For reference:

- Keil, the default PSoC 3 compiler uses Big Endian
- GCC, the default PSoC 5 compiler uses Little Endian

Parameters and Setup



- **Desired number of elements in the A/B arrays**: this field allows you to specify the number of elements you would like to compute as a single block. The component will automatically generate 2 arrays of 32 bit values with the specified number of elements in each array and configure 2 DMA channels to move the data from the arrays into the multiply and accumulate hardware, one data pair (one element from array A and one element from array B) at a time. These arrays are called (assuming the component is called MultAcc_1)
    - `MultAcc_1_a.data[MultAcc_1_ARRAY_SIZE]`
    - `MultAcc_1_b.data[MultAcc_1_ARRAY_SIZE]`
- **Compiler uses Big Endian format**: checking this box enables endian swapping on the DMA channels moving the data from SRAM into the multiply and accumulate hardware. DMA endian swapping is required when the compiler accesses SRAM information in Big Endian format. The multiply and accumulate hardware is Little Endian. For compilers that use Little Endian format, no endian swapping is required.

- **Enable hardware block complete signal**: Checking this box enables an optional hardware signal that will assert when a block has finished processing. This terminal can be connected to an interrupt or to a DMA Channel*. The signal is de-asserted when the internal status register is read using the MultAcc_1_CheckStatus(…) API.

**Please read the DMA information section to understand caveats when using DMA to read the data from the multiply and accumulate hardware.*

Application Programming Interface

Application Programming Interface (API) routines allow you to configure the component using software. The following table lists and describes the interface to each function. The subsequent sections cover each function in more detail.

By default, PSoC Creator assigns the instance name "MultAcc_1" to the first instance of a component in a given design. You can rename it to any unique value that follows the syntactic rules for identifiers. The instance name becomes the prefix of every global function name, variable, and constant symbol.

Function	Description
Void MultAcc_1_Start(void)	Initializes and enables the multiply and accumulate hardware and DMA.
Void MultAcc_1_Init(void)	Initializes the multiply and accumulate hardware and the DMA. This function only needs to be called once after a reset.
Void MultAcc_1_Enable(void)	Enables the multiply and accumulate hardware and the DMA. MultAcc_1_Init() must be called before enabling the hardware.
Void MultAcc_1_Stop(void)	Disables the multiply and accumulate hardware and DMA. To re-enable the hardware, call MultAcc_1_Enable()
Void MultAcc_1_ProcessBlock(uint8 Reset)	Initiates a multiply and accumulate on the blocks of data stored in MultAcc_1_a and MultAcc_1_b. If the reset flag is passed, the accumulator is reset on the first multiply. If the reset flag is not passed, the previous accumulator contents are used.
Uint8 MultAcc_1_CheckStatus(uint8 Flag)	Reads the status of the Multiply and accumulate hardware and returns a non-zero value if the passed flag is set. Flags are: • Block complete • Accumulator was reset • Overflow occurred, saturation was enforced
Int32 MultAcc_1_GetResult(void)	Returns the most recent output of the accumulator in Q23 fixed point format. Only call this function when processing a block of data has completed.
Float MultAcc_1_Q23ToFloat(int32 Value)	Converts a Q23 fixed point value into its equivalent floating point number.
Int32 MultAcc_1_FloatToQ23(float Value)	Converts a floating point number between -1 and 0.999999880791 into an equivalent Q23 fixed point number.

Void MultAcc_1_Start(void)

Description: This function initializes and enables the multiply and accumulate hardware and DMA that moves the data stored in MultAcc_1_a and MultAcc_1_b into the MACC. After calling this function, the block is ready to process data.

Parameters: None

Return Value: None

Void MultAcc_1_Init(void)

Description: This function initializes the multiply and accumulate hardware and the DMA. This function only needs to be called once after a reset. It is safe to call this function multiple times, however only the first call will actually initialize the hardware.

Parameters: None

Return Value: None

void MultAcc_1_Enable(void)

Description: This function Enables the multiply and accumulate hardware and DMA. Only call this function after calling MultAcc_1_Init().

Parameters: None

Return Value: None

Void MultAcc_1_Stop(void)

Description: This function disables the multiply and accumulate hardware and disables the DMA. To re-enable the hardware and DMA, call MultAcc_1_Enable().

Parameters: None

Return Value: None

Void MultAcc_1_ProcessBlock(uint8 Reset)

Description: This function initiates processing of a block of data. After this function has been called, DMA will take each element from MultAcc_1_a and MultAcc_1_b (collectively called a block) and pass them to the multiply and accumulator hardware. The MACC will multiply the elements of A and the elements of B together and add each result to the accumulator. This function is non-blocking and will return almost immediately. To check the status of the MACC operation and determine when it is done, use the MultAcc_1_CheckStatus(…) function. The MultAcc_1_CheckStatus will return a non-zero value for MultAcc_1_BlockComplete when the block has finished processing.

If the value passed into MultAcc_1_ProcessBlock(…) is non-zero (MultAcc_1_RESET), then the accumulator is reset to zero before the first multiply and accumulate. Resetting the accumulator is useful when starting a new calculation. Resetting the accumulator will also reset the status of the MACC hardware. If you did not check on the status of any flags before resetting the accumulator, they will be lost.

If the parameter passed is zero (MultAcc_1_CONTINUE), the result of the accumulator is preserved from the previous calculation and the results will be added to this existing accumulator value. Preserving the accumulator value is useful when you need to process blocks of data longer than the maximum of 1023 elements, since the data arrays can be filled with new data and coefficients and the process can pick up where it left off. Using the continue functionality will not affect the status flags.

Refer to the MultAcc_1_CheckStatus(…) function for more information on the status flags and the meaning of each flag on the result of the MACC operation.

Parameters:

Reset: Indicates whether or not to preserve the accumulator value from the previous block calculation

Parameter Name	Description
MultAcc_1_RESET	Resets the accumulator to zero and clears all status flags before initiating a calculation

MultAcc_1_CONTINUE	Preserves the previous accumulator value and initiates a block calculation

Return Value: None

Void MultAcc_1_CheckStatus(uint8 Flag)

Description: When this function is called, it will read the current status of the MACC hardware and return a non-zero value if the bit specified in the Flag parameter is set. This function allows you to check the status of the result to determine if the hardware has:

- Completed a block calculation
- Cleared the accumulator and started a new calculation
- Detected an overflow condition during the computation and applied saturation to the accumulator

Each bit is sticky, and will only be cleared when it is specifically read using the passed Flag. For example, if you use the MultAcc_1_OVERFLOW_DETECT flag:

MultAcc_1_CheckStatus(MultAcc_1_OVERFLOW_DETECT)

The function will read the overflow detection bit, clear the bit if it is set and return a non-zero value if the bit was set.

Normally you would use this function for 2 purposes:

1. To poll the MACC hardware to determine if the MACC operation has finished
2. After the block has completed, check the status of the overflow detect to see if at any point during the block operation, the accumulator overflowed and saturation was applied to the accumulator. If saturation has occurred, the result is likely invalid.

You can also use the function to ensure that the result was either a new calculation (this will be set after initiating a block calculation and using the MultAcc_1_RESET parameter) or a continuation of a previous calculation by using the previous accumulator value.

Parameters: **Flag:** Indicates which status flags to check

Parameter Name	Description
MultAcc_1_BLOCK_COMPLETE	Checks the MACC hardware to determine if a block calculation has completed. If the calculation is complete, the returned value from the function will be non-zero
MultAcc_1_NEW_RESULT	This flag checks if the result from a recently completed block calculation is a “new” result, meaning that the accumulator was reset to zero at the beginning of the calculation. Checking this flag is not strictly required
MultAcc_1_OVERFLOW_DETECT	This flag checks if an overflow occurred at any time during the calculation. If this flag is set, the result is likely invalid because it

	means that at some point, the accumulator reached its numerical minimum/maximum and saturated at the extreme. If the calculation did overflow at some point, the return value will be non-zero

Return Value: The return value is non-zero if the flag passed into the function was set in the hardware.

Int32 MultAcc_1_GetResult(void)

Description: This function returns the most recent result from the accumulator. The returned value is in Q23 fixed point format. Call this function when MultAcc_1_CheckStatus(MultAcc_1_BLOCK_COMPLETE) returns a non-zero value, indicating the block has completed its calculation and the result in the accumulator is ready.

Parameters: None

Return Value: The accumulator value in Q23 fixed point format.

float MultAcc_1_Q23ToFloat(int32 value)

Description: This function converts a value in Q23 fixed point format into an equivalent floating point value. If the value exceeds the maximum (8388607, 0x007FFFFF, 0.999999880791) or minimum (-8388608, 0xFF800000, -1.0) of a Q23 value, it will be saturated to the appropriate minimum or maximum before conversion.

Parameters: None

Return Value: The floating point representation of the passed Q23 fixed point number

Int32 MultAcc_1_Q23ToFloat(float value)

Description: This function converts a value in floating point format into an equivalent Q23 fixed point value. If the value exceeds the maximum (0.999999880791) or minimum (-1.0) of a Q23 value, it will be saturated to the appropriate minimum or maximum before conversion.

Parameters: None

Return Value: The Q23 fixed point representation of the passed floating point number

Defines and Types

The size of each data array is provided as:

```
#define MultAcc_1_ARRAY_SIZE
```

This define is useful for writing loops to copy data into and out of the data arrays.

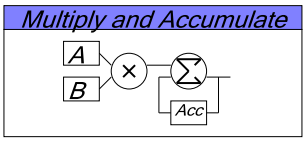
The A and B data is stored in a structure type called: `MultAcc_1_ARRAY`

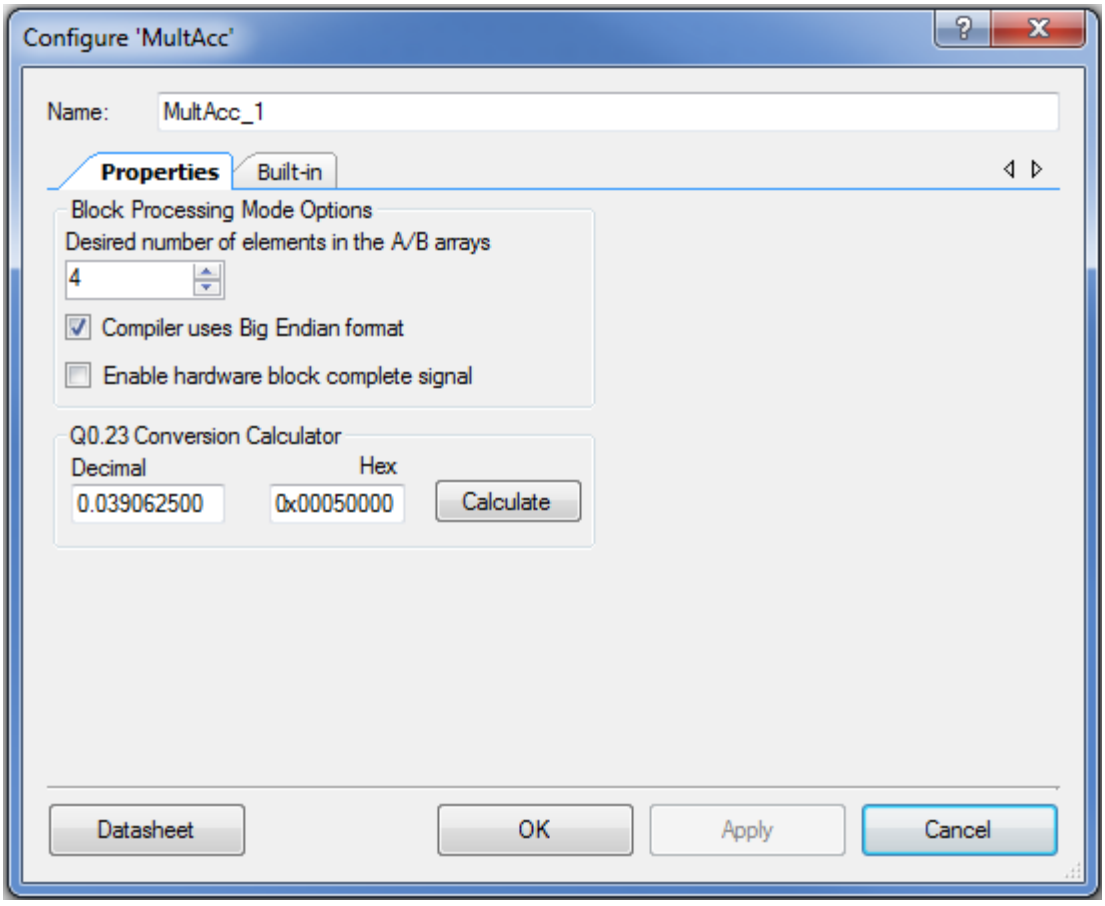
```
typedef struct
{
      uint32 dat[MultAcc_1_ARRAY_SIZE];
} MultAcc_1_ARRAY;
```

This structure allows the compiler to automatically write code for copying data stored in the same data type into the data structures. In the example firmware, this data type will be used to simplify copying coefficient data from FLASH into the data structure.

Sample Firmware Source Code

MultAcc_1





```
#include <device.h>


void main()
{
    /* Place your initialization/startup code here (e.g. MyInst_Start()) */

       uint8 i;
       int32 result;
       float resultf;

       // here we use the component defined structure type MultAcc_1_ARRAY
       // and the CYCODE keyword to define an array of coefficients that will be stored in
       // FLASH.  Storing the coefficients in flash will save space in RAM for large
       // set of coefficient data.  In this case, we are just defining 4 coefficients
       // of 0.2 represented in Q23 format.  Refer to the customizer calculator
       // pictured above to see the converter calculator
       MultAcc_1_ARRAY CYCODE my_coeff = {{0x0019999A, 0x0019999A, 0x0019999A, 0x0019999A}};

       // The big advantage to using the component defined type is that it
       // allows for very easy copying of the data from flash into the data data array
       // MultAcc_1_a.data[MultAcc_1_ARRAY_SIZE]
       MultAcc_1_a = my_coeff;

       // if you opt to not use the defined type, you can simply copy the data
       // into the structure element by element using a for loop
       // ash shown below.  Here we are loading the "B" data
       // with more values of 0.2.  We are also using the conversion
       // routines to convert a floating point number into an equivalent
       // Q23 representation
       for(i = 0; i < MultAcc_1_ARRAY_SIZE; i++)
       {
              MultAcc_1_b.dat[i] = MultAcc_1_FloatToQ23(0.2);
       }

       // initialize and enable the component
       MultAcc_1_Start();

    for(;;)
    {
              // initiate a block calculation, reset the accumulator on the first multiply
              MultAcc_1_ProcessBlock(MultAcc_1_RESET);

              // the MACC hardware is non-blocking so you can do anything you want
              // while the calculation is progressing, *except change the
              // data in the A and B arrays*!! This loop simply
              // checks the status of the BLOCK_COMPLETE flag and waits
              // until it is set, indicating that the block calculation
              // is complete
              while(!MultAcc_1_CheckStatus(MultAcc_1_BLOCK_COMPLETE)){}

              // check to see if the overflow hardware detected an overflow
              if(MultAcc_1_CheckStatus(MultAcc_1_OVERFLOW_DETECT))
              {
                     // oops!  we had an overflow
              }

              // pull the result from the accumulator
              result = MultAcc_1_GetResult();

              // convert the result into a floating point value for
```

```
            // easy reading in the debugger.  The expected
            // answer is 0.160000, although due to the limitations
            // of the Q23 format, the answer may be slightly different
            resultf = MultAcc_1_Q23ToFloat(result);

            // If you were working on a block of data that was larger
            // than the maximum number of elements supported in a single
            // block, you would load your A and B blocks with the next
            // set of data here and initiate another block calculation
            // but you would continue the calculation, preserving the accumulator

            // In this simple example, we are going to
            // initiate another block calculation on the same data
            // but this time, preserve the previous accumulator value
            MultAcc_1_ProcessBlock(MultAcc_1_CONTINUE);

            // the MACC hardware is non-blocking so you can do anything you want
            // while the calculation is progressing, *except change the
            // data in the A and B arrays*!! This loop simply
            // checks the status of the BLOCK_COMPLETE flag and waits
            // until it is set, indicating that the block calculation
            // is complete
            while(!MultAcc_1_CheckStatus(MultAcc_1_BLOCK_COMPLETE)){}

            // check to see if the overflow hardware detected an overflow
            if(MultAcc_1_CheckStatus(MultAcc_1_OVERFLOW_DETECT))
            {
                  // oops!  we had an overflow
            }


            result = MultAcc_1_GetResult();

            // The expected answer is 0.320000, although due to the limitations
            // of the Q23 format, the answer may be slightly different
            resultf = MultAcc_1_Q23ToFloat(result);
    }
}
```

DMA Information

If you intend to move data from the output of the MACC component, you need to understand that the DFB (the underlying core of the MACC component) output register is fixed at high byte coherency. You also need to understand that the output acts as a 2 element FIFO. The first output from the MACC is loaded on the output register, then stays there. The second, third, fourth … outputs all overwrite each other internally in the "last" entry of the output FIFO, waiting for the first output to be read. This is a bug in the hardware.

The result is that a read from the DFB output at the end of the block gives you the very first MACC result on the first read, and THEN gives you the final MACC result on the second read. The GetResult() API solves this problem by reading the high byte once (invalidating the first result and allowing the final result to move to the output register) and then reading the entire output. If you intend to use DMA, you must also read the output register TWICE in order to ensure that you get the most recent result from the accumulator. The first read will be the first result from the block.