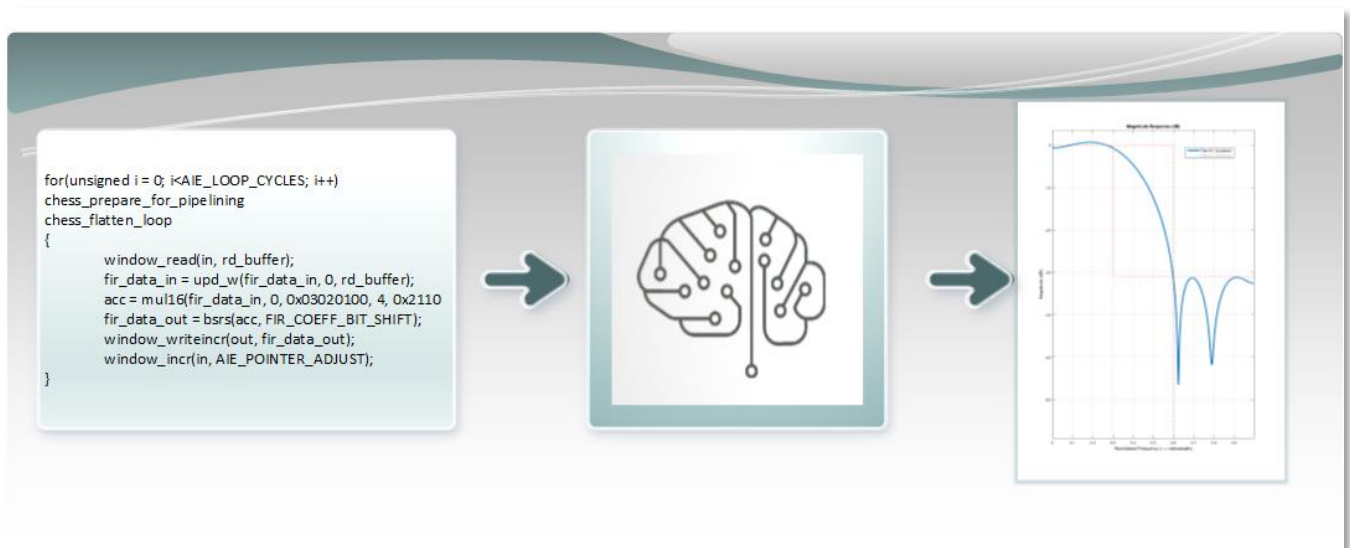
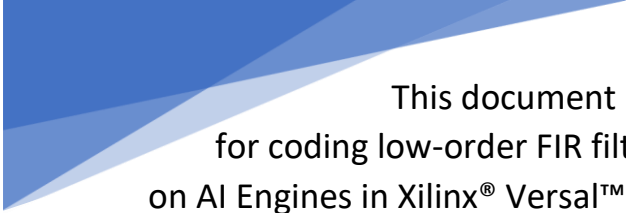


# Implementing Low-Order FIR Filters in Versal AI Engines

*Practical guidelines for kernel development*



A blue geometric graphic consisting of several overlapping triangles and quadrilaterals, creating a sense of depth and movement, positioned to the left of the introductory text.

This document provides guidance  
for coding low-order FIR filters to run  
on AI Engines in Xilinx® Versal™ ACAP devices.

The focus is on kernel code development in C using  
the low-level intrinsic API.

Example code is developed throughout the  
document, with detailed explanations of intrinsic  
concepts and parameterization including the  
permute squares.

A basic understanding of FIR filters, C language, and  
the Xilinx® Vitis™ tools is presumed.

© Copyright 2020 Xilinx, Inc. Xilinx, the Xilinx logo, Versal, Vitis, Vivado, and other designated brands included herein are trademarks of Xilinx in the United States and other countries. MATLAB and Simulink are registered trademarks of The MathWorks, Inc. All other trademarks are the property of their respective owners.

## Contents

Introduction .....	6
Low-Order FIR Filters .....	7
The Logic Implementation .....	7
The AIE Implementation .....	8
Design Steps.....	9
Example Specifications.....	9
Estimate Filter Taps.....	9
Select Data Types.....	10
Select the Number of Lanes.....	11
Vectors and Indexing .....	14
Indexing with Squares.....	17
Indexing for 16-Bit Real Data and Coefficients .....	19
Indexing for 16-Bit Real Data and 8-Bit Real Coefficients.....	23
Squares for 8-Bit Real Data and Coefficients .....	25
Coding .....	27
The First Intrinsic.....	27
The Vector Registers .....	28
Coefficients .....	29
The Second Intrinsic.....	32
Data Input and Output.....	32
The Pointer.....	33
The Accumulator and the Third Intrinsic .....	35
The Loop.....	36
Pragmas.....	36
Putting it All Together .....	38
Taking Advantage of Symmetry .....	40
8-bit Real Data and Coefficients.....	44
Testing.....	46
C Test Bench.....	46
Model Composer.....	47
Summary .....	48

Appendix A: Indexing for the General Scheme .....	49
Appendix B: Indexing for 16-Bit Real Data, 16-Bit Real Coefficients .....	50
Appendix C: Indexing for 16-Bit Real Data, 8-Bit Real Coefficients .....	51
Appendix D: Indexing for 8-Bit Real Data, 8-Bit Real Coefficients .....	52
Appendix E: Low-Order FIR Coding Flow .....	53

## Figures and Tables

Figure 1: Asymmetric FIR, direct form .....	7
Figure 2: Symmetric FIR, direct form .....	7
Figure 3: FIR filter implemented in a single AIE .....	8
Figure 4: Lanes in the AIE .....	11
Figure 5: FIR analysis for example design .....	13
Figure 6: FIR analysis for revised example design.....	14
Figure 7: Hardware representation of AIE intrinsic example.....	17
Figure 8: xstep for 16b x 16 real data .....	19
Figure 9: Lane offsets for 16b x 16b real data .....	19
Figure 10: Data x index values broken out for discussion .....	20
Figure 11: Permute square for 16b x 16b real data .....	20
Figure 12: Permute square offsets applied across the x plane.....	20
Figure 13: Comparing desired values to present values to derive xsquare .....	21
Figure 14: Values with xstep =x2110 .....	21
Figure 15: Parameter xstep for 16-bit data with permute square.....	21
Figure 16: Indexing for example equation, 16-bit real data and coefficients.....	22
Figure 17: xstep for 16b real data with 8b real coefficients .....	23
Figure 18: Lane offsets for 16b real data with 8b real coefficients .....	23
Figure 19: Permute square for 16b real data with 8b real coefficients.....	23
Figure 20: zstep for 8b coefficients with 16b real data .....	24
Figure 21: Permute square for 8b real coefficients with 16b real data .....	24
Figure 22: Lane offsets for 8b real coefficients with 16b real data .....	24
Figure 23: xstep for 8b real data with 8b real coefficients .....	25
Figure 24: Lane offsets for 8b real data with 8b real coefficients .....	25
Figure 25: Permute square for 8b real data with 8b real coefficients.....	25
Figure 26: zstep for 8b real coefficients with 8b real data .....	26
Figure 27: Lane offsets for 8b real coefficients with 8b data .....	26
Figure 28: Permute square for 8b real coefficients with 8b real data .....	26
Figure 29: Permute square for 8b real coefficients with 8b real data .....	26
Figure 30: FIR equations parameterized for mul8 .....	28
Figure 31: Filter response for minimum-phase asymmetric coefficients .....	31
Figure 32: Pointer movement during data input sequence .....	35
Figure 33: Symmetric FIR with additional y permute square .....	42
Figure 34: Permute square for the y values.....	42

Table 1: Accumulator types and processing capability for different numeric types (from UG1076) .....	10
Table 2: Variants of the mul intrinsic for asymmetric and symmetric FIRs .....	12
Table 3: Data and coefficient types that require the permute square .....	17

## Introduction

The Versal™ AI Engines (AIE) are well suited for vector math operations.

One specific application that warrants consideration is low-order FIR filters. For low-order symmetric and asymmetric FIR structures a single AIE can implement multiple filters, each running at the AIE domain clock rate which can be 1 GHz or more. And with an entire array of AIE many filters can be implemented in Versal™ ACAP.

This document provides guidelines for coding such a filter in the Vitis™ AI Engine tools: For the experienced programmer it provides a structured approach to code low-order FIR filters. For the less experienced it also provides some explanation of key concepts such as AIE vectors, intrinsics, permute squares, and data flow.

The code examples are esoteric with the assumption that you can restructure and build on them in your own coding style. Furthermore, you can expand the concepts and examples presented here to other applications including higher-order filters or other functions that rely on vector math.

A basic understanding of FIR filters, C language, and the Xilinx® Vitis™ tools is presumed.

## Low-Order FIR Filters

There is no fixed threshold for defining what makes a filter “low-order”, so for the purposes of this discussion it will be any filter whose order is low enough that the AIE can process the multiply/add/accumulate operations for that filter in a single clock cycle.

### The Logic Implementation

To clarify this, consider the direct-form symmetric and asymmetric FIR filter structures (shown for a 3<sup>rd</sup> order filter, i.e. four taps):

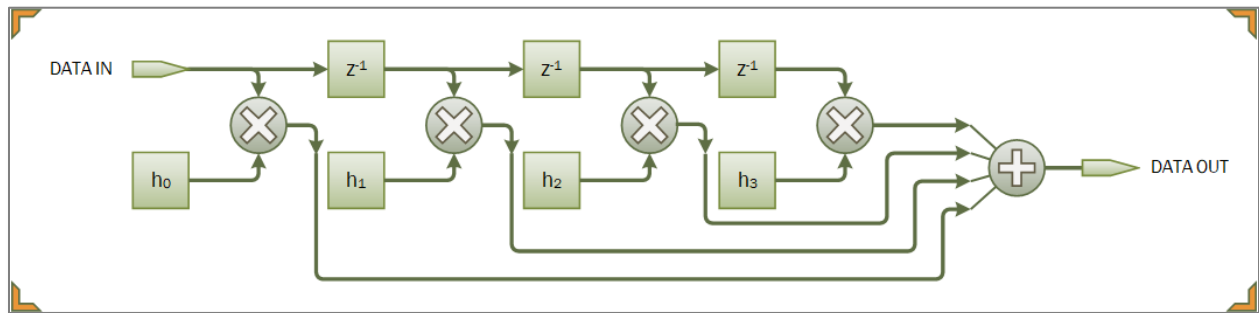


Figure 1: Asymmetric FIR, direct form

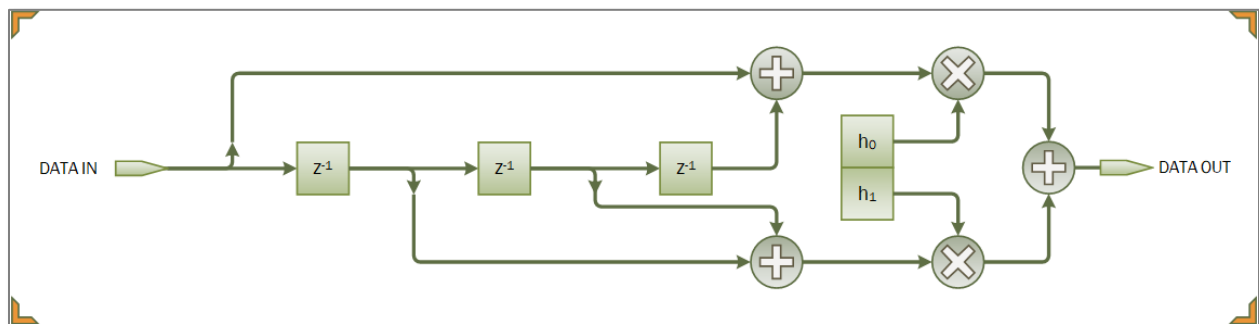


Figure 2: Symmetric FIR, direct form

Data is sequenced through the filter and at each tap is multiplied by a filter coefficient, or in the symmetric case two data taps are added then multiplied by their common coefficient to reduce the number of multipliers. The multiplicands are summed to generate the output, and the data progresses through taps sequentially on each clock. These familiar structures are readily implemented in programmable logic fabric or in dedicated blocks such as the DSP48 or DSP58.

The question is, how does this translate into a Versal™ AIE implementation?

## The AIE Implementation

The AIE performs vector processing to handle FIR calculations: Data and coefficients are packed into vectors and submitted to the processor's X and Z registers respectively, the VLIW processor calculates the outputs, and the results are stored in the AIE accumulator. The accumulator is split into lanes, so depending on the data type either 4, 8, or 16 FIR calculations can be performed in parallel (4 lanes are shown for illustration in figure 3). The accumulator data is then adjusted to the desired output data width and output as the FIR output vector.

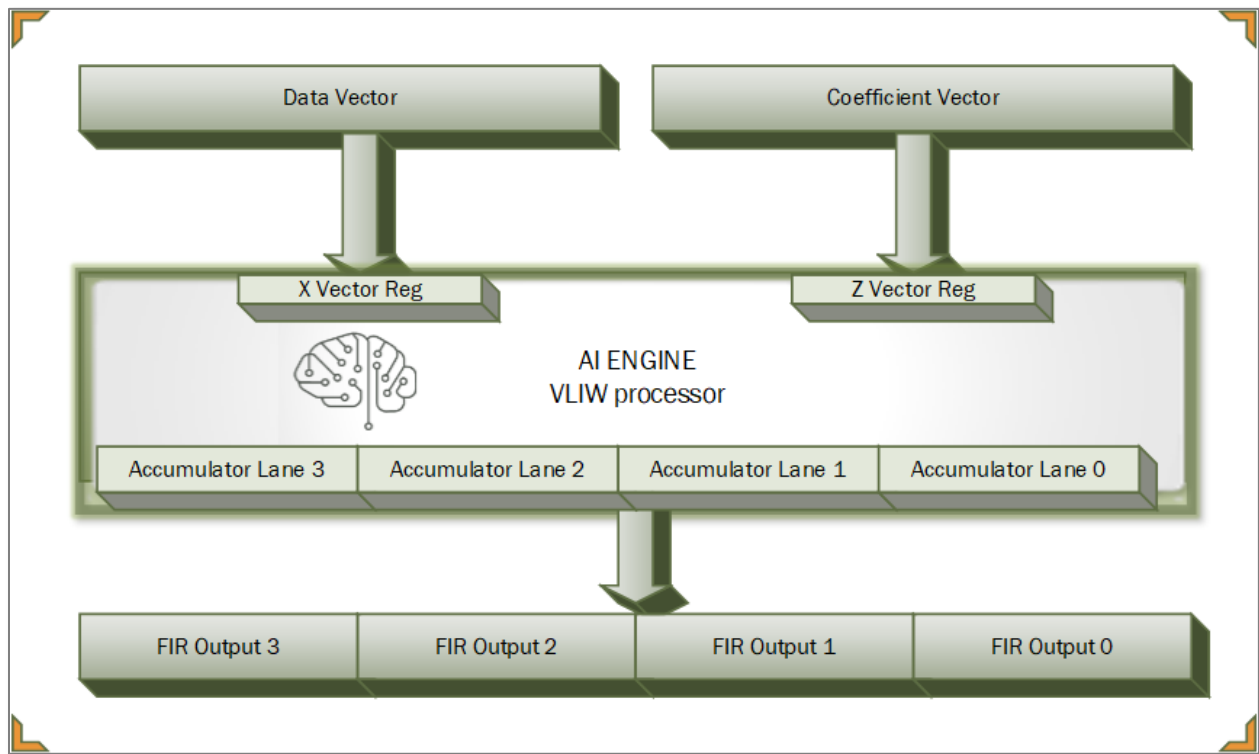


Figure 3: FIR filter implemented in a single AIE

This AIE process can handle both asymmetric and symmetric FIR filters, real and/or complex data and coefficients.

The design steps to coding this FIR for AIE will now be covered in further detail, but first make sure you have the following three documents<sup>1</sup> available for reference:

- ☐ UG1076 Versal ACAP AI Engine Programming Environment User Guide
- ☐ AM009 Versal ACAP AI Engine Architecture Manual
- ☐ AI Engine Intrinsics Documentation

Note that the last item may be available in the Vitis™ help menus. If you download it, make sure you get the version that matches your Vitis™ tool version.

<sup>1</sup> Available for download from Xilinx.com, or via the Xilinx® DocNav tool which installs with Vitis™ and Vivado™.



## Design Steps

To architect a FIR in the AIE you start with the data type. Specifically, the numeric format used to represent each data sample and coefficient. From there, the vectorization and accumulator types are selected. Finally, the vector input/output is selected. From that point it's a matter of applying and parameterizing the associated intrinsics into the code. These steps are detailed here, along with an example that is progressively developed throughout this document. And to seem more realistic the example is not chosen to neatly fit into the calculations, so real-world tradeoffs will be involved.

## Example Specifications

The example which will be used to illustrate each step will be as follows:

- ✚ LOW-PASS FIR, SAMPLING AT 1 GSPS WITH A CUTOFF FREQUENCY OF 300 MHz, A TRANSITION BAND OF 300 MHz, AND AT LEAST 30 dB OF REJECTION. INPUT DATA AND COEFFICIENTS ARE 16-BIT SIGNED FORMAT. COEFFICIENTS ARE ASYMMETRIC.

## Estimate Filter Taps

You may have already developed coefficients, in which case you already know the length of the filter you wish to implement. If not, a rule of thumb (provided by fred harris [sic]) that may help estimate the minimum required number of filter taps is

$$N = \frac{Fs}{\text{delta}(f)} * \frac{\text{atten}(dB)}{22}$$

where

- N = number of taps
- Fs = sampling frequency
- delta(f) = transition band of the filter
- atten(dB) = required attenuation in dB between the pass and stop bands.

- ✚ FOR OUR EXAMPLE FIR THE RULE OF THUMB GIVES AN ESTIMATED MINIMUM FILTER LENGTH OF

$$N = \frac{1e9}{300e6} * \frac{30}{22} \approx 5 \text{ TAPS}$$

## Select Data Types

With your estimated filter length in mind the process starts with the driving parameter – data type. There are three points to keep in mind regarding AIE data types:

- 1) The AIE processing capability depends on the data types involved, both the data and the coefficients.
- 2) Arbitrary precisions are not supported, so any data types that do not match the data types handled by the AIE must be extended or truncated to match one of the supported types. The supported data widths are 8, 16, and 32 bits.
- 3) The FIR data is loaded as the X operand of the AIE VLIW processor, and the coefficients are loaded as the Z operand.

With these points in mind, refer to the table “Supported Precision Width of the Vector Datapath” in the latest copy of UG1076. For convenience that table is reproduced here, but you should still verify in the original document:





 FIR data	 FIR coefficient	 Accumulator	 Capability
X Operand	Z Operand	Output	Number of MACs
8 real	8 real	48 real	128
16 real	8 real	48 real	64
16 real	16 real	48 real	32
16 real	16 complex	48 complex	16
16 complex	16 real	48 complex	16
16 complex	16 complex	48 complex	8
16 real	32 real	48/80 real	16
16 real	32 complex	48/80 complex	8
16 complex	32 real	48/80 complex	8
16 complex	32 complex	48/80 complex	4
32 real	16 real	48/80 real	16
32 real	16 complex	48/80 complex	8
32 complex	16 real	48/80 complex	8
32 complex	16 complex	48/80 complex	4
32 real	32 real	80 real	8
32 real	32 complex	80 complex	4
32 complex	32 real	80 complex	4
32 complex	32 complex	80 complex	2
32 SPFP	32 SPFP	32 SPFP	8

Table 1: Accumulator types and processing capability for different numeric types (from UG1076)

This table provides some key data points. Specifically, for any combination of data and coefficient you can derive the supported accumulator lane width (“Output” column) and the processing capability in MACs (“Number of MACs”). The latter is the key number for the next step, so make a note of the number of MACs for your data and operand types then continue.

✚ FOR OUR EXAMPLE FIR WE SPECIFIED 16-BIT REAL DATA AND COEFFICIENTS, SO THE PROCESSING CAPABILITY IS 32 MACS PER CLOCK.

### Select the Number of Lanes

Lanes warrant some explanation at this point. The lanes are independent processing paths that feed into an accumulator. Lanes are bound together in that they are instantiated by a common intrinsic and share common input vectors but are independent in that their results can be based on different elements of those input vectors.

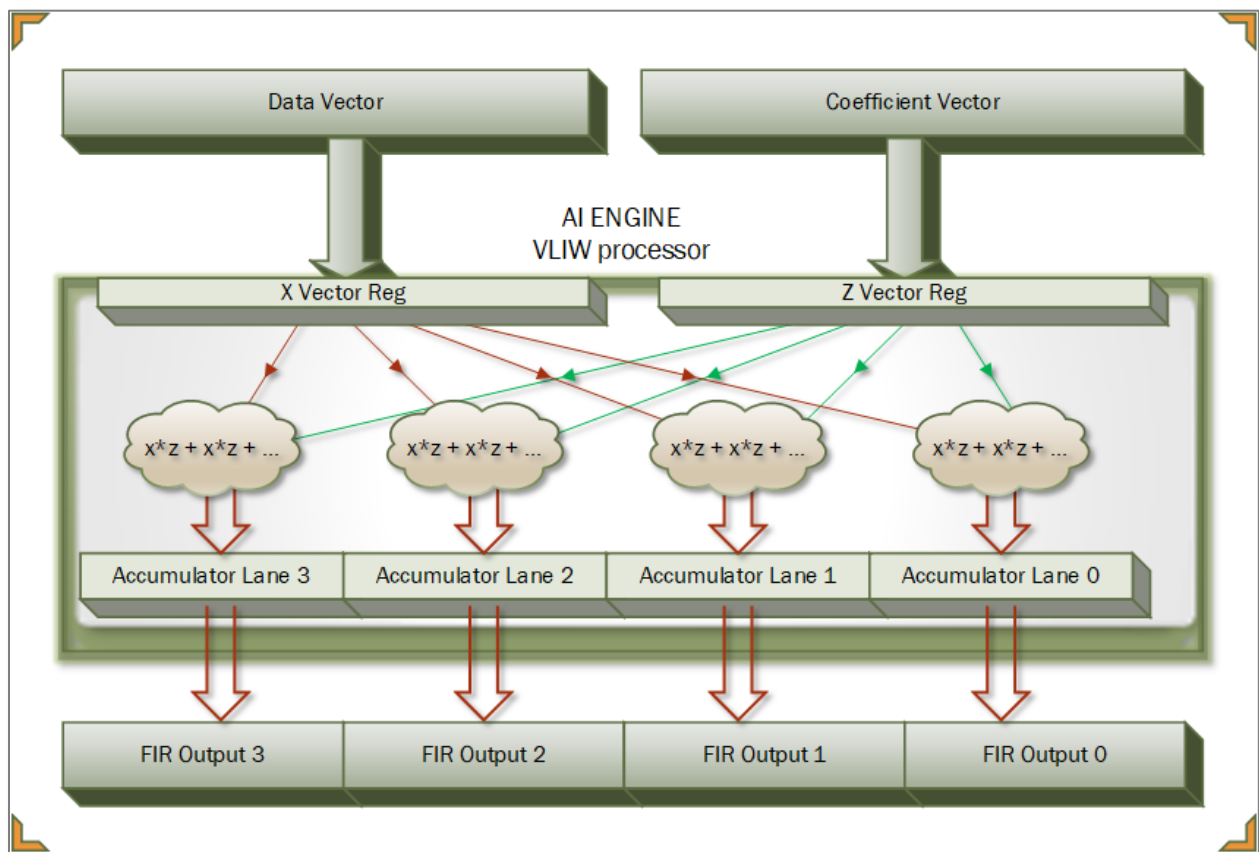


Figure 4: Lanes in the AIE

Depending on how the vector is indexed (more on that later) each lane can be a copy of the same FIR working on different data, different coefficients working on the same data, or different coefficients and data.

The two quintessential math intrinsics for implementing FIR calculations are the *mul* and *mac* (multiply and multiply-accumulate). For a low-order FIR, which we nominally defined as a FIR where all calculations could be done in a single clock cycle, there is no need to accumulate products across sequential operations since it is done in one. Therefore low-order FIRs will only use the *mul* intrinsic which will establish how the calculations (represented by the clouds in figure 4) are set up across all lanes. The situation is similar for higher order FIRs except they require a *mul* followed by one or more *mac* intrinsics to sequentially accumulate taps across multiple clock cycles until all taps are calculated.

So, for low-order FIR calculations you will need to understand *mul*. For starters, the intrinsic *mul* has separate variants for each supported lane configuration as well as for asymmetric and symmetric FIRs. The choices are as follows:

Number of lanes	Asymmetric FIR	Symmetric FIR
4	mul4	mul4_sym, mul4_sym_ct
8	mul8	mul8_sym, mul8_sym_ct
16	mul16	mul16_sym, mul16_sym_ct

Table 2: Variants of the *mul* intrinsic for asymmetric and symmetric FIRs

The *mul\_sym* implements symmetric FIRs with an even number of coefficients, while *mul\_sym\_ct* implements FIRs with an odd number of coefficients (where the single center tap requires one of the multipliers). In each case the total processing capability is split evenly across the number of lanes. The processing capability of each lane (i.e. each FIR) is easily calculated as

$$N = \frac{1}{4} * MACs \text{ (for 4 lanes)}$$

$$N = \frac{1}{8} * MACs \text{ (for 8 lanes)}$$

$$N = \frac{1}{16} * MACs \text{ (for 16 lanes)}$$

where

- N = achievable number of FIR coefficient multiply/add operations per lane
- MACs = processing capability as noted in table (from the previous step).

Note that N reflects the number of multiply/add operations per lane which is the number of taps for an asymmetric FIR. Symmetric FIR filters with an even number of coefficients will be able to support 2N taps, while symmetric FIR filters with an odd number of coefficients will be able to support (2N – 1) taps.

✚ FOR THE EXAMPLE FIR, WE RECORDED 32 MACs PER CLOCK FROM THE PREVIOUS STEP, SO THE POSSIBILITIES IN TERMS OF THE LANE OPTIONS (ASSUMING ASYMMETRY) ARE

- $32/4 = 8$  TAPS ACROSS 4 LANES
- $32/8 = 4$  TAPS ACROSS 8 LANES
- $32/16 = 2$  TAPS ACROSS 16 LANES.

At this point you now know the capability of each lane option. What you don't know yet is if all three options are viable choices for your filter. The available options depend on the data and coefficient numeric formats. To determine which choices are valid you must consult the intrinsic documentation:

- Open either the Vitis™ “Help / AIE Help” menu or the downloaded intrinsic documentation<sup>2</sup>.
- Select “Intrinsics and Datatypes / Vector Operations / MAC intrinsics / Multiplication / Advanced / Integer / Vector MAC “
- Select your data widths (16-bit x 16-bit, 16-bit x 32-bit, ...) then types (Real x Real, Complex x Complex, ...)
- Search for supported “mul” intrinsics and note which are present. Those are your valid choices.

✚ FOR THE EXAMPLE, FOLLOWING THE ABOVE STEPS SHOWS THAT FOR 16 BIT REAL X 16 BIT REAL THE SUPPORTED *MUL* INTRINSICS ARE *MUL8* AND *MUL16*; *MUL4* IS NOT PRESENT. COMBINED WITH THE PERFORMANCE NUMBERS CALCULATED IN THE PRIOR STEP THAT MEANS THE OPTIONS ARE 4 TAPS OR 2 TAPS. SINCE WE ESTIMATED A MINIMUM OF 5 TAPS IT APPEARS THAT NEITHER IS A VALID CHOICE, BUT SINCE 4 TAPS IS IN THE BALLPARK IT IS WORTH CALCULATING THE FILTER CHARACTERISTICS.

GENERATING COEFFICIENTS AND ANALYZING THE FILTER IN MATLAB® CONFIRMS THAT 4 TAPS IS NOT ENOUGH SINCE THE FILTER WOULD NOT MEET THE 30 DB STOP BAND REJECTION GOAL:

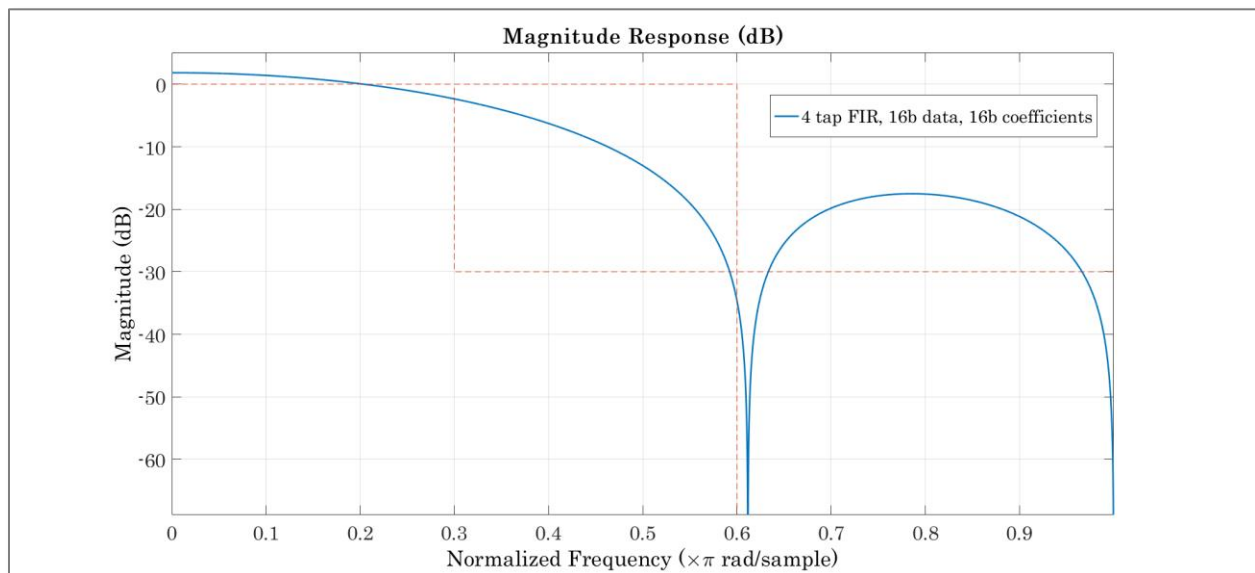


Figure 5: FIR analysis for example design

SO WHAT CAN BE DONE? REVISITING THE TABLE SHOWS THAT CHANGING THE COEFFICIENTS WIDTH FROM 16 TO 8 (16-BIT REAL X 8-BIT REAL ROW IN THE TABLE) WOULD DOUBLE PERFORMANCE FROM 32 TO 64 MACS PER CLOCK CYCLE. RECALCULATING THE LANE OPTIONS SHOWS THIS WOULD DOUBLE THE SUPPORTED TAP LENGTHS, AND CONSULTING THE INTRINSIC DOCUMENTATION CONFIRMS THAT THE SAME TWO INTRINSICS (*MUL8* AND *MUL16*) ARE SUPPORTED FOR 16-BIT REAL X 8-BIT REAL. SO NOW THE BEST OPTION IS A *MUL8* 8-TAP FIR WITH 16-BIT DATA AND 8-BIT COEFFICIENTS.

<sup>2</sup> In both cases the documentation is delivered in HTML format, with “index.html” as the starting point.

RERUNNING ANALYSIS WITH EIGHT TAPS AND DECREASED COEFFICIENT WIDTH SHOWS THAT THIS FILTER NOW MEETS THE ORIGINAL PERFORMANCE REQUIREMENTS:

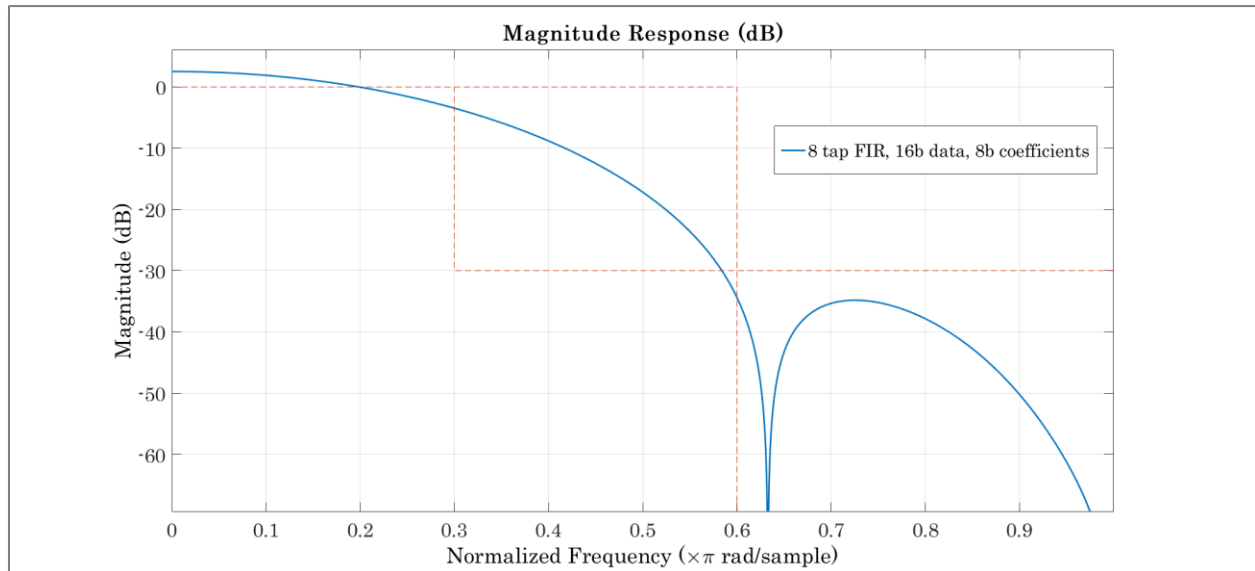


Figure 6: FIR analysis for revised example design

THE PROCESS CAN CONTINUE. THE SELECTED NUMBER OF LANES IS 8 (MUL8 INTRINSIC) AND THE DATA TYPES ARE NOW 16-BIT DATA AND 8-BIT COEFFICIENTS.

## Vectors and Indexing

With the number of lanes and data/coefficient types set, the *mul* intrinsic can now be parameterized. This is the start of coding for the AIE. But first, a quick word about the AIE vector registers and indexing.

The AIE operates on vectors stored in vector registers and delivers results in a vector accumulator. The FIR data and coefficients must thus be delivered in vector form.

Vectors are formed by packing C data types together into wider words. The native width for AIE vectors is 128 bits but there are also 256, 512, and 1024-bit vector registers; plus the accumulator can be 320, 384, 640, or 768 bits wide.

The naming convention for vectors is as follows:

`v{NumLanes}[c]{[u]int|acc}{SizeofElement}`

where

- 'v' denotes a vector
- NumLanes defines the number of lanes
- 'c' denotes complex data with real and imaginary parts packed together
- 'u' denotes unsigned.
- "int" defines an integer type
- "acc" defines an accumulator type
- SizeofElement defines the number of elements.

Here are a few examples:

- *V8acc48* would be a 384-bit accumulator holding 8 lanes of 48-bit real values.
- *v8int16* would define a 128-bit vector comprised of 8 lanes of 16-bit integers.
- *v16int8* would be a 128-bit vector holding 16 lanes of 8-bit integers.
- *V16uint8* would be a 128-bit vector holding 16 lanes of 8-bit unsigned integers.
- *v16int16* would be a 256-bit vector holding 16 lanes of 16-bit integers.
- *v16cint16* would be a 512-bit vector holding 16 lanes of 16-bit complex integers.

Note that the complex vectors have double the width of real counterparts since they contain both real and imaginary elements.

Within a vector, indexing is used to reference specific elements. Each lane has an independent index into each operand vector, so each lane can pick up different operands. All lanes perform the same math function, though.

Vectors are little endian, so a zero index will point to the lowest order (earliest in time for sequential samples) element of a vector. For intrinsics such as *mul* the indexing is done by specifying a starting value and an increment.

To illustrate, consider two input vectors *x* and *z* and an accumulator vector *acc*. To build a sum of products with those vectors we could specify a starting value of 0 and an increment of 1 to index the equation

$$acc = x_0 * z_0 + x_1 * z_1 + x_2 * z_2 + x_3 * z_3$$

Now to add some flexibility we might specify separate indexing for the *x* and *z* terms, and set *x* to start at 0 and increment by 2 while *z* starts at 1 and increments by 2:

$$acc = x_0 * z_1 + x_2 * z_3 + x_4 * z_5 + x_6 * z_7$$

And now, since we can multitask, we will perform four such calculations simultaneously. Again, to add flexibility we now add separate *x* and *z* increments for the vertical direction going down the rows. Let's set the *x* vertical increment at 4 and the *z* vertical at 0:

$$\begin{array}{lcl}
 \oplus \rightarrow & \text{Columns} & \\
 \downarrow & & \\
 \text{Lanes} & \text{acc}_0 = x_0 * z_1 + x_2 * z_3 + x_4 * z_5 + x_6 * z_7 & \\
 & \text{acc}_1 = x_4 * z_1 + x_6 * z_3 + x_8 * z_5 + x_{10} * z_7 & \\
 & \text{acc}_2 = x_8 * z_1 + x_{10} * z_3 + x_{12} * z_5 + x_{14} * z_7 & \\
 & \text{acc}_3 = x_{12} * z_1 + x_{14} * z_3 + x_{16} * z_5 + x_{18} * z_7 & 
 \end{array}$$

Refer to the different lines as “lanes”, and the product pairs as “columns”, with the term  $x_0 * z_1$  in the above example referenced as (lane 0, column 0). Moving across the top row the next term  $x_2 * z_3$  would be (lane 0, column 1). Moving down from there term  $x_6 * z_3$  would be (lane 1, column 1). This is the “general scheme” for AIE intrinsic indexing. Appendix A presents a graphical representation summarizing the indexing for the general scheme.

Since there are four lanes in the above example it correlates to the *mul4* intrinsic which has the following format:

$$acc = mul4(xbuff, xstart, xoffsets, xstep, zbuff, zstart, zoffsets, zstep);$$

The data is in vector *xbuff*, the coefficients are in vector *zbuff*, and the indexing is provided by giving each a starting value, offsets across lanes, and steps across rows. Essentially the same concept we just developed in the example equations. So, in AIE coding terms the example equations for  $acc_0$  through  $acc_3$  as previously expressed would be represented by the following parameterized intrinsic:

$$acc = mul4(x, 0, 0xC840, 2, z, 1, 0x0000, 2);$$

Note that the lane increments are expressed as a starting point and vector offset, while the column increment is simply expressed as a scalar increment, and that the placeholders “*xbuff*” and “*zbuff*” are replaced here with user variable names “*x*” (vectorized FIR data) and “*z*” (vectorized FIR coefficients) respectively:

- For the *x* values the starting value is 0 and the lane offset (starting with the least significant nibble in the hexadecimal offset vector 0xC840) is 0 for lane 0, 4 for lane 1, 8 for lane 2, and 12 for lane 3.
- Working across the columns the *x* increment is specified as 2, so the first lane has *x* indexed as [0,2,4,6].
- For the *z* values the starting value is 1 and the lane offsets are all 0, so *z* indexes start at 1 and remain 1 vertically down all the lanes.
- Across the columns the *z* increment is specified as 2, so the *z* index sequences as [1,3,5,7] horizontally.



This can be shown in terms of the hardware blocks as follows:

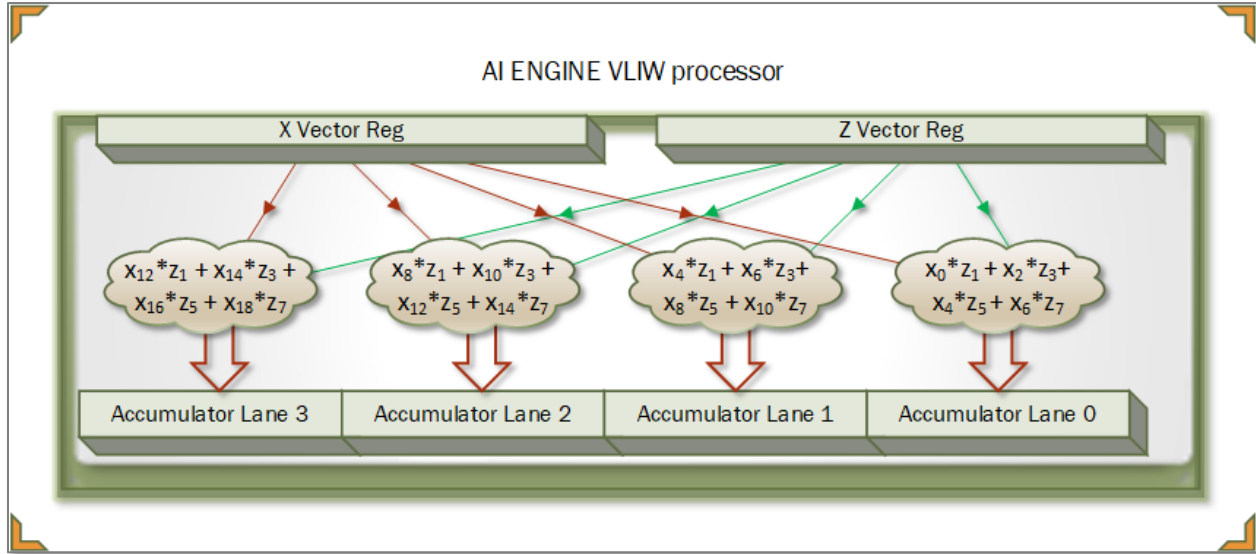


Figure 7: Hardware representation of AIE intrinsic example

## Indexing with Squares

So now that the general scheme for indexing in AIE intrinsics is clear, it's time to throw a little curve ball – the permute square.

For x and z (i.e. data and coefficient) types that are both real and 16-bits or 8-bits wide the indexing has a granularity that is coarser than the width of the data and/or coefficients. As a result, the start and offset values (*xstart/xstep/zstart/zstep*) are restricted to powers-of-2 or powers-of-4 and the finer-grained selection required to select smaller elements out of a vector is handled by a permute square (*xsquare, zsquare*). This is tabulated below:

Data Type	Coefficient Type	Permute Square(s)	xstart, xstep	zstart, zstep
16-bit real	16-bit real	xsquare	multiple of 2	-
16-bit real	8-bit real	xsquare zsquare	multiple of 2	multiple of 2
8-bit real	8-bit real	xsquare zsquare	multiple of 4	multiple of 2

Table 3: Data and coefficient types that require the permute square

Any other combinations of data/coefficient types use the previously discussed general scheme without the square. This will be evident when you view the intrinsic documentation for you selected *mul*. For instance, the *mul8* for 16-bit x 16-bit real (int16) data types has the *xsquare* parameter while the *mul8* for 16-bit x 16-bit complex types (cint16) does not as shown in the intrinsic documentation:

v8acc48	mul8 (v32int16 xbuff, int xstart, unsigned int xoffsets, int xstep, unsigned int xsquare, v16int16 zbuff, int zstart, unsigned int zoffsets, int zstep)
v8cacc48	mul8 (v32cint16 xbuff, int xstart, unsigned int xoffsets, v8cint16 zbuff, int zstart, unsigned int zoffsets)

With smaller real numeric formats the permute square fine tunes the element selection within a vector register. The previously described start/offset/step approach gets down to the block of data but not the specific word in the block. A 2x2 or 2x4 square is applied across the matrix to enable selecting any element within the x and z vector registers.

Since the permute square functions differently for the x and z coefficients as well as for different combinations of coefficient and data types each will be discussed in turn. For each discussion the following FIR equations for a *mul4* will be used as the end goal:

$$\begin{array}{lcl}
 \oplus \rightarrow & \text{Columns} & \\
 \downarrow & & \\
 \text{Lanes} & \text{acc}_0 = x_0 * z_0 + x_1 * z_1 + x_2 * z_2 + x_3 * z_3 & \\
 & \text{acc}_1 = x_1 * z_0 + x_2 * z_1 + x_3 * z_2 + x_4 * z_3 & \\
 & \text{acc}_2 = x_2 * z_0 + x_3 * z_1 + x_4 * z_2 + x_5 * z_3 & \\
 & \text{acc}_3 = x_3 * z_0 + x_4 * z_1 + x_5 * z_2 + x_6 * z_3 & 
 \end{array}$$

In the general scheme this can be expressed as a *mul4* intrinsic with parameters

$$\begin{aligned}
 xstart &= 0, xoffsets = 0x3210, xstep = 1 \\
 zstart &= 0, zoffsets = 0x0000, zstep = 1.
 \end{aligned}$$

The specifics of permute squares for each of the three cases listed in the table will be addressed now.

## Indexing for 16-Bit Real Data and Coefficients

For 16-bit real data and 16-bit real coefficients,

- The first row of table 3 shows the  $xstart$  and  $xstep$  must be multiples of 2. The steps are applied to every other column, with odd columns assuming the value of the preceding even column as shown here.
- The lane offsets are doubled before being applied, so adjust offsets to half the offset that would have been used in the general scheme. These can be even or odd.
- Offsets are also distributed differently; a 4-bit even offset applies to both its associated lane plus the following odd lane, while a 4-bit odd offset applies only to odd columns in the associated odd lane and is applied in addition to the even offset as shown here.

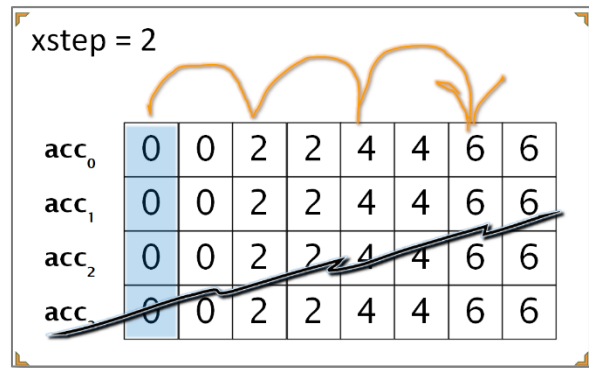


Figure 8: xstep for 16b x 16 real data

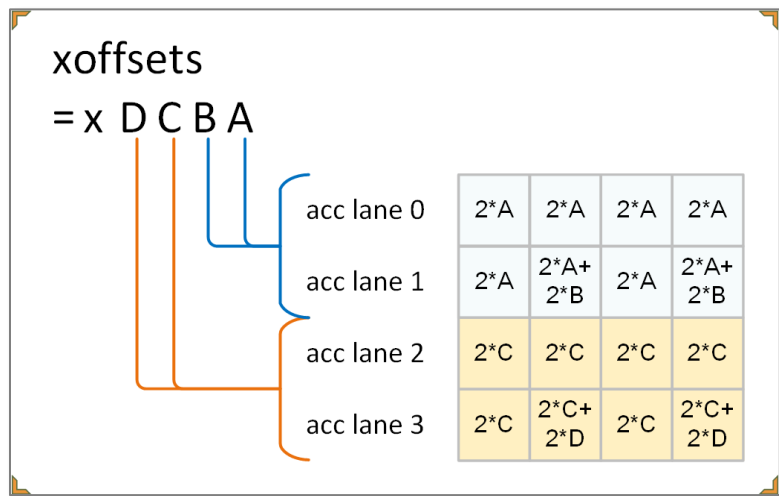


Figure 9: Lane offsets for 16b x 16b real data

So for the previous *mul4* example the parameters must be adjusted as follows to comply with these rules (changes highlighted in red):

$$xstart = 0, xoffsets = 0x0100, xstep = 0$$

$$zstart = 0, zoffsets = 0x0000, zstep = 1.$$

Note that the 'z' values remain unchanged by 'x'. Based on these new values the equations are now as follows (where the values that need further updating are highlighted in red):

⊕ → Columns

↓

Lanes

$$acc_0 = x_0 * z_0 + x_0 * z_1 + x_0 * z_2 + x_0 * z_3$$

$$acc_1 = x_0 * z_0 + x_0 * z_1 + x_0 * z_2 + x_0 * z_3$$

$$acc_2 = x_2 * z_0 + x_2 * z_1 + x_2 * z_2 + x_2 * z_3$$

$$acc_3 = x_2 * z_0 + x_2 * z_1 + x_2 * z_2 + x_2 * z_3$$

To simplify, let's disregard the 'z' coefficients and focus on the 'x' data indices which can be tabulated as shown to the right. In this case the permute square provides the finer-grained resolution to select values that are not factors of two.

0	0	0	0
0	0	0	0
2	2	2	2
2	2	2	2

Figure 10: Data x index values broken out for discussion

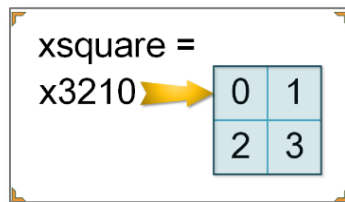


Figure 11: Permute square for 16b x 16b real data

The permute square for 16-bit data is a 2x2 square with offsets specific to each position. The intrinsic parameter for the square is *xsquare*, which defaults to 0x3210 with the values arranged inside the square as shown to the left.

To visualize this, picture the square offset values added to the values set by the *xstart*/*xoffsets*/*xstep* parameters starting with the upper left two lanes and two columns then repeating across the entire matrix of data:

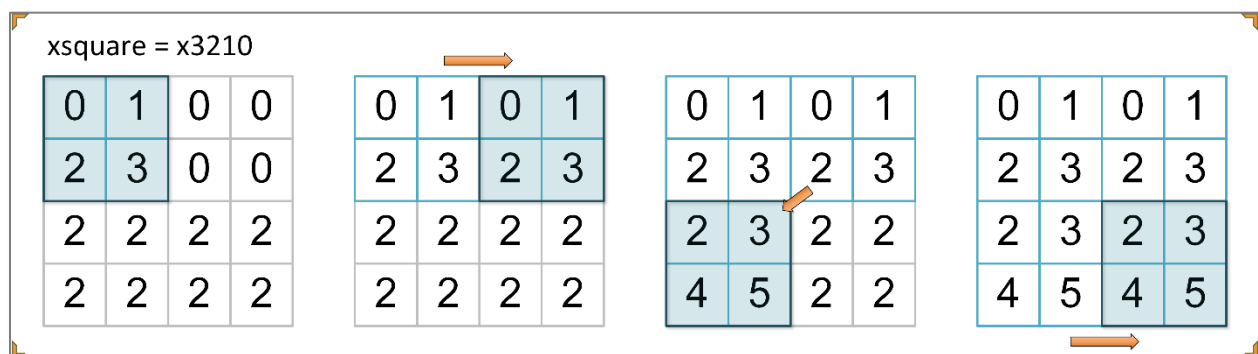


Figure 12: Permute square offsets applied across the x plane

Note that this is a build-time parameter, so the sequence is compiled into the machine code as indexing to simultaneously select elements out of the data vector for processing in the AIE; the sequential steps shown here are for illustration only.

Also note that the default square value of x3210 does not produce the desired pattern so to implement the equations a different *xsquare* value needs to be applied. To calculate new values, start with the upper four squares and compare the starting point (original *xstart* and *xstep* adjusted to comply with the multiple-of-two restriction) of those four values to the desired values:

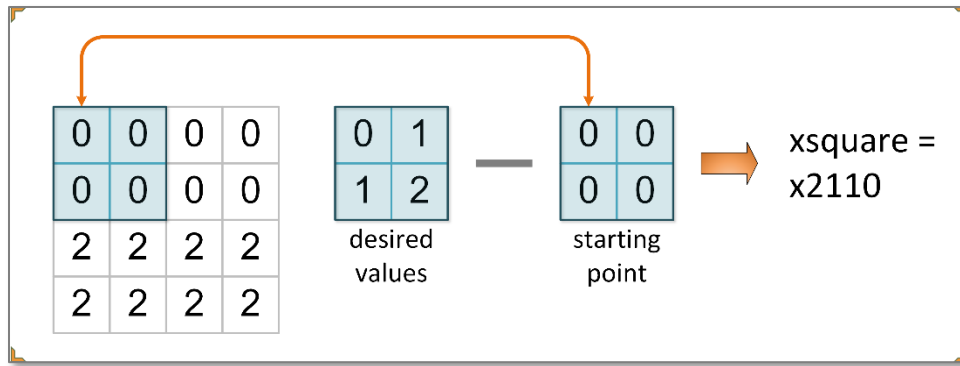


Figure 13: Comparing desired values to present values to derive *xsquare*

Applying the new permute square value x2110 across the matrix yields the values shown to the right. This is getting closer to the desired values; the first two columns now match the desired data coefficients. Moving to the right the values do not increment, though, so it's time to work with the *xstep* value.

The *xstep* value which increments across every column in the general scheme behaves differently when the permute square is required. It applies to even columns (highlighted in yellow). The odd columns assume the value of the even column to their left plus associated offsets (from the permute square and lane offsets). As indicated in table 3 the *xstep* value is also limited to factors of two.

0	1	0	1
1	2	1	2
2	3	2	3
3	4	3	4

Figure 14: Values with *xstep* = x2110

To determine the proper value for *xstep* you can visualize the current values and the desired values next to each other and then compare the even rows as shown here:

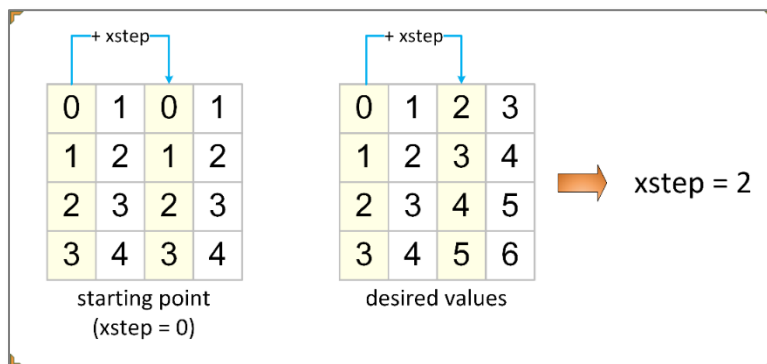


Figure 15: Parameter *xstep* for 16-bit data with permute square

As shown in table 3, the coefficients follow the general scheme so the values originally set can be reused. With that final step the parameterization to achieve the desired 16-bit real FIR coefficient and data indexing using a permute square is complete as shown on the next page.

For additional clarification, Appendix B provides a graphical representation of how these parameters work together for 16-bit real data and coefficients.

xstart = 0, xoffsets = 0x0200, xsquare = 0x2110, xstep = 2  
 zstart = 0, zoffsets = 0x0000, zstep = 1.

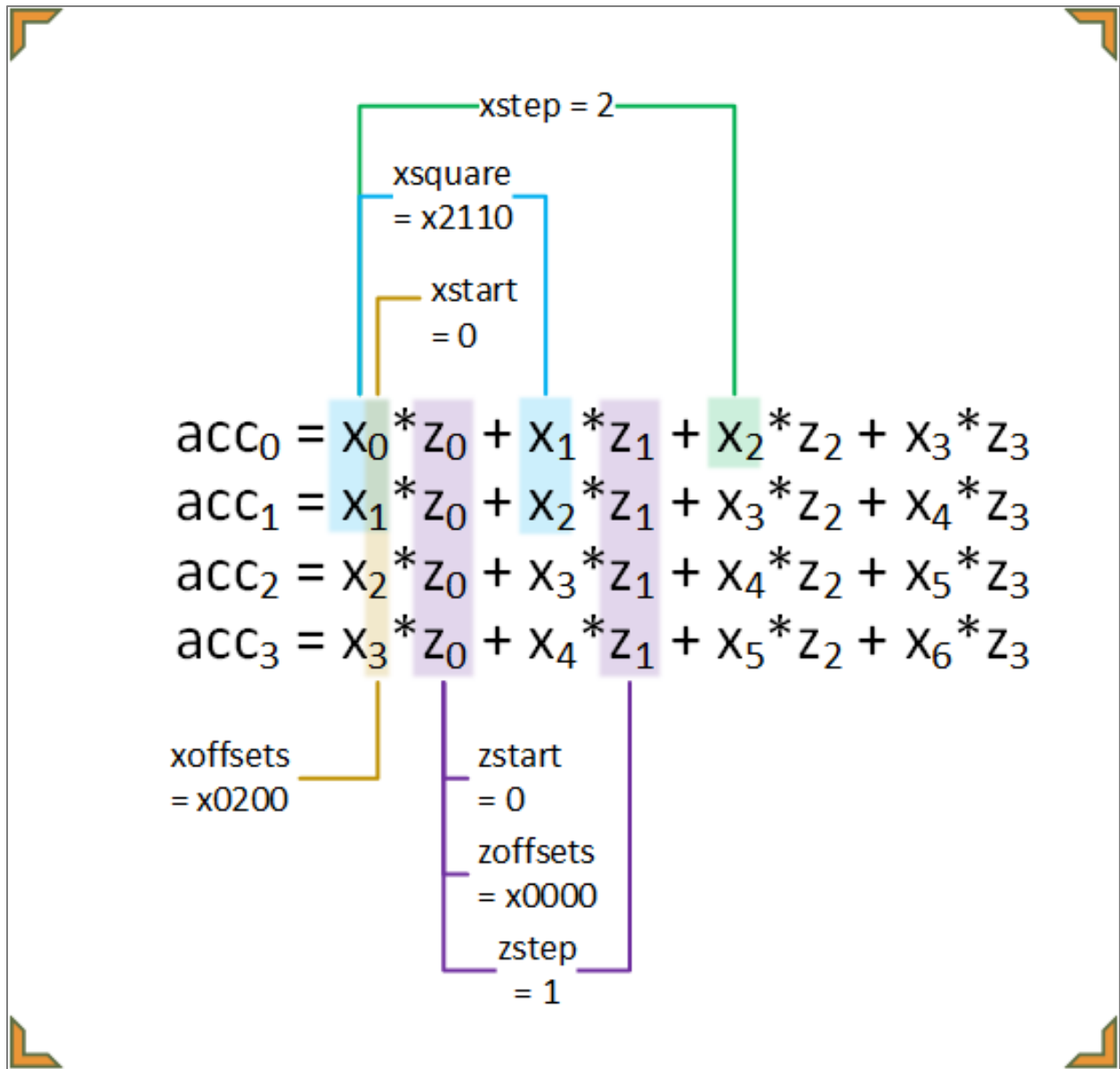


Figure 16: Indexing for example equation, 16-bit real data and coefficients

## Indexing for 16-Bit Real Data and 8-Bit Real Coefficients

For 16-bit real data and 8-bit real coefficients the data can be handled identically to the 16-bit x 16-bit real case previously covered, while the coefficients introduce a new set of rules.

For the 16-bit real data,

- The second row of table 3 shows the  $xstart$  and  $xstep$  must be multiples of 2. The steps are applied to every other column, with odd columns assuming the value of the preceding even column as shown here.
- The lane offsets are doubled before being applied, so adjust offsets to half the offset that would have been used in the general scheme. These can be even or odd.
- A 4-bit even offset applies to both its associated lane plus the following odd lane, while a 4-bit odd offset ("B" and "D" in the figure) applies only to odd columns in the associated odd lane, and is applied in addition to the even offset as shown above. Typically the odd offsets will be zero.

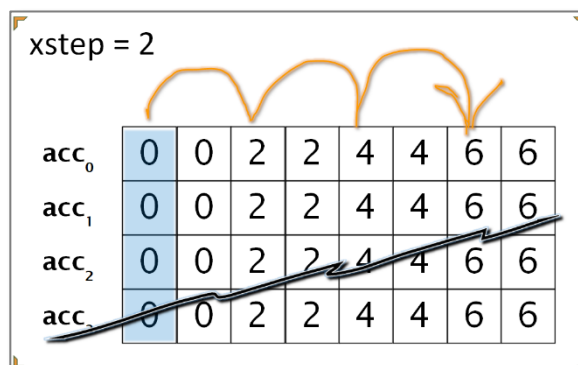


Figure 17: xstep for 16b real data with 8b real coefficients

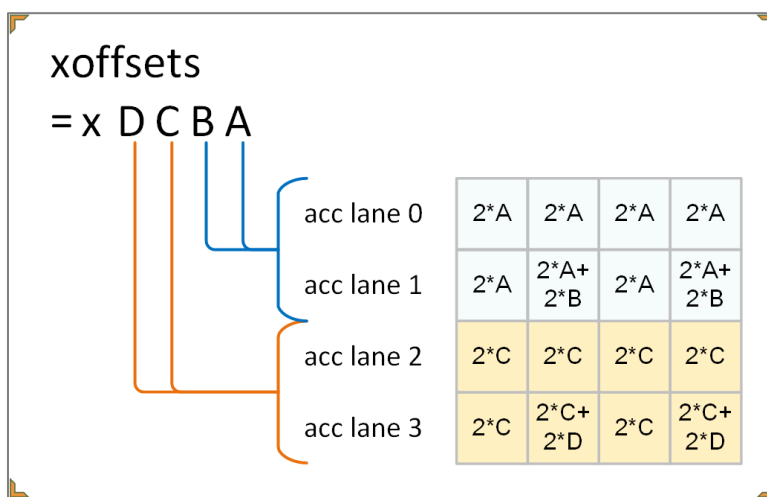


Figure 18: Lane offsets for 16b real data with 8b real coefficients

- The permute square for 16-bit data is a 2x2 square with offsets specific to each position. The intrinsic parameter for the square is  $xsquare$ , which defaults to 0x3210 with the values arranged inside the square as shown to the right.

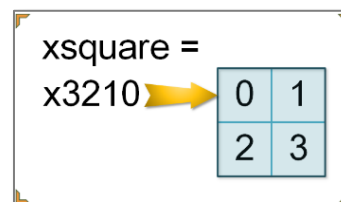


Figure 19: Permute square for 16b real data with 8b real coefficients

For the 8-bit real coefficients,

- a) The second row of table 3 shows the *zstart* and *zstep* must be multiples of 2. The steps are applied to every other column, with odd columns assuming the value of the preceding even column as shown here.

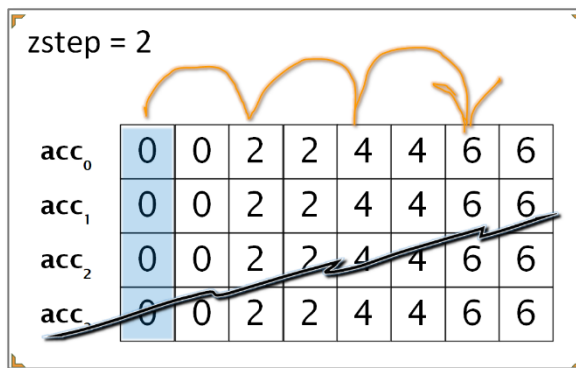


Figure 20: *zstep* for 8b coefficients with 16b real data

- b) Each even offset applies to both its associated lane plus the following odd lane, while the odd offsets ("B" and "D" in the figure) are not applied and can be set to zero.

- c) Since the *zoffsets*, *zstart*, and *zstep* values are all constrained to even numbers, the permute *zsquare* (below) must be used to select coefficient values that do not fall on even boundaries.

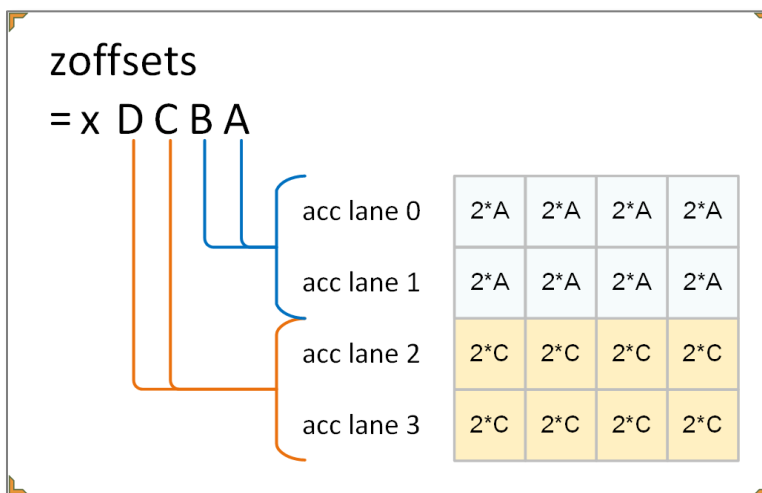


Figure 22: Lane offsets for 8b real coefficients with 16b real data

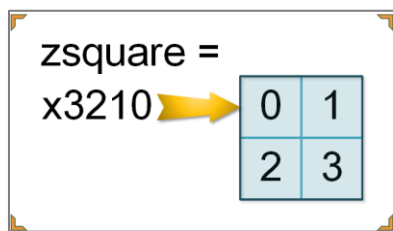


Figure 21: Permute square for 8b real coefficients with 16b real data

For additional clarification, Appendix C provides an alternative perspective showing how these parameters work together for 16-bit real data with 8-bit real coefficients.



## Squares for 8-Bit Real Data and Coefficients

Moving on to the third and final row in table 3, the coefficient parameterization for a FIR with 8-bit real data and coefficients has its own unique set of requirements.

For the 8-bit real data,

- The  $xstart$  and  $xstep$  values must be multiples of 4. The steps are applied to every other column, with odd columns assuming the value of the preceding even column as shown here.
- The lane offsets are quadrupled before being applied, so adjust offsets to a quarter the offset that would have been used in the general scheme. These can be even or odd.
- Offsets are distributed across four consecutive lanes with the odd lanes additionally incremented by 1. The 4-bit odd offsets ("B" and "D" in the figure) are not used and can be set to zero.
- The  $xsquare$  permute square for 8-bit data is a 4x2 square with the four offsets spread across eight positions as shown below. Values are doubled.

$xstep = 4$

$acc_0$	0	0	4	4	8	8	12	12
$acc_1$	0	0	4	4	8	8	12	12
$acc_2$	0	0	4	4	8	8	12	12
$acc_3$	0	0	4	4	8	8	12	12

Figure 23:  $xstep$  for 8b real data with 8b real coefficients

$xoffsets = x \ D \ C \ B \ A$

acc lane 0	$4*A$	$4*A$	$4*A$	$4*A$
acc lane 1	$4*A + 1$	$4*A + 1$	$4*A + 1$	$4*A + 1$
acc lane 2	$4*A$	$4*A$	$4*A$	$4*A$
acc lane 3	$4*A + 1$	$4*A + 1$	$4*A + 1$	$4*A + 1$
acc lane 4	$4*C$	$4*C$	$4*C$	$4*C$
acc lane 5	$4*C + 1$	$4*C + 1$	$4*C + 1$	$4*C + 1$
acc lane 6	$4*C$	$4*C$	$4*C$	$4*C$
acc lane 7	$4*C + 1$	$4*C + 1$	$4*C + 1$	$4*C + 1$

Figure 24: Lane offsets for 8b real data with 8b real coefficients

$xsquare =$

$x3210 \rightarrow$

0	1
0	1
2	3
2	3

Figure 25: Permute square for 8b real data with 8b real coefficients

For the 8-bit real coefficients,

- a) The third row of table 3 shows the *zstart* and *zstep* must be multiples of 2. The steps are applied to every other column, with odd columns assuming the value of the preceding even column as shown here.
- b) The lane offsets are doubled before being applied, so adjust offsets to half the offset that would have been used in the general scheme. These can be even or odd.

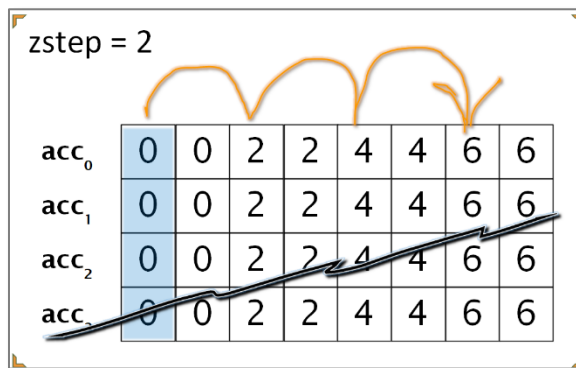


Figure 26: *zstep* for 8b real coefficients with 8b real data

- c) Offsets apply to four consecutive lanes. The odd offsets ("B" and "D" in the figure) are not applied and can be set to zero.
- d) Since the *zoffsets*, *zstart*, and *zstep* values are all constrained to even numbers, the permute *zsquare* (below) must be used to select coefficient values that do not fall on even boundaries. One notable issue exists, though: Only the least significant bit of each square offset is applied, so all even numbers are treated as 0 while all odd numbers become 1.

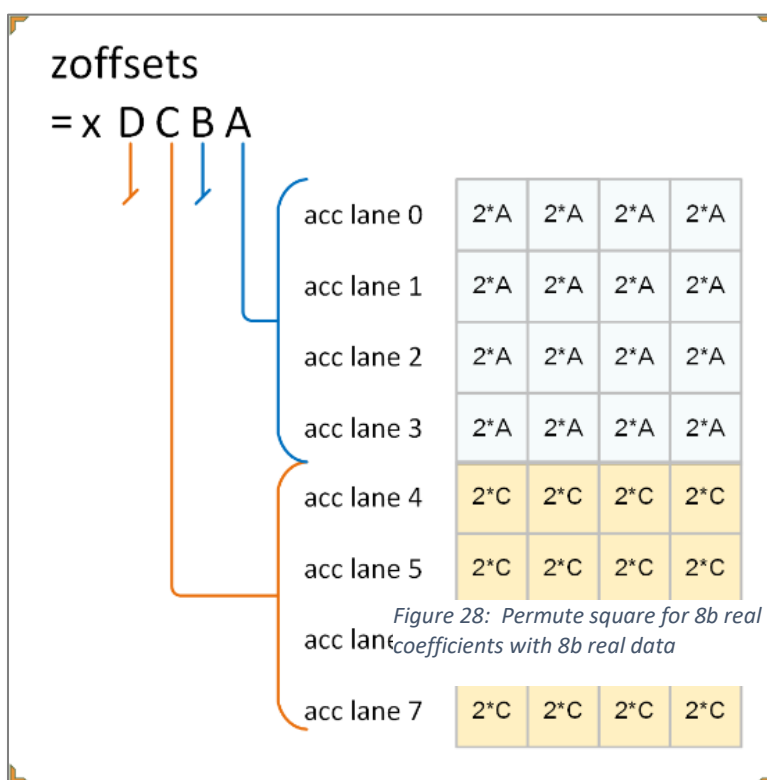


Figure 28: *Permute square* for 8b real coefficients with 8b real data

Figure 27: *Lane offsets* for 8b real coefficients with 8b data

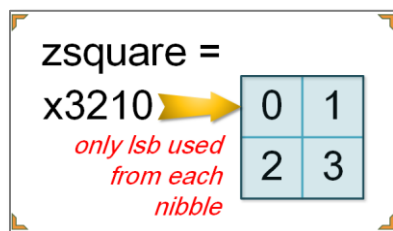


Figure 29: *Permute square* for 8b real coefficients with 8b real data

For additional clarification, Appendix D provides an alternative perspective showing how these parameters work together for 8-bit real data with 8-bit real coefficients.

## Coding

Coding in a sense builds around the *mul* intrinsic. That intrinsic sets the types for the data, coefficients, and accumulator output as well as the number of lanes and calculations per clock cycle. It makes a logical starting point for coding. To expound on the process we'll resume work on the example design, repeated here for reference:

- ❏ LOW-PASS FIR, SAMPLING AT 1 GSPS WITH A CUTOFF FREQUENCY OF 300 MHz, A TRANSITION BAND OF 300 MHz, AND AT LEAST 30 dB OF REJECTION. INPUT DATA IS 16-BIT SIGNED INTEGER FORMAT, AND COEFFICIENTS ARE NOW 8-BIT SIGNED INTEGER FORMAT (REMEMBER THE TRADEOFF FROM 16 TO 8 BITS TO GAIN TAPS AND MEET PERFORMANCE).

## The First Intrinsic

We previously determined that for the data types involved (see table 1) the AIE could process at 64 MACs and that the *mul8* intrinsic was the best fit, supporting  $64/8 = 8$  taps per lane across 8 lanes. How we use those lanes is determined by how we arrange the indexing.

At this point let's set down eight lanes of FIR equations as a target:

$$\begin{array}{l} \oplus \rightarrow \text{Columns} \\ \downarrow \\ \text{Lanes} \end{array} \quad \begin{array}{l} \text{acc}_0 = x_0 * z_0 + x_1 * z_1 + x_2 * z_2 + x_3 * z_3 + x_4 * z_4 + x_5 * z_5 + x_6 * z_6 + x_7 * z_7 \\ \text{acc}_1 = x_1 * z_0 + x_2 * z_1 + x_3 * z_2 + x_4 * z_3 + x_5 * z_4 + x_6 * z_5 + x_7 * z_6 + x_8 * z_7 \\ \text{acc}_2 = x_2 * z_0 + x_3 * z_1 + x_4 * z_2 + x_5 * z_3 + x_6 * z_4 + x_7 * z_5 + x_8 * z_6 + x_9 * z_7 \\ \text{acc}_3 = x_3 * z_0 + x_4 * z_1 + x_5 * z_2 + x_6 * z_3 + x_7 * z_4 + x_8 * z_5 + x_9 * z_6 + x_{10} * z_7 \\ \text{acc}_4 = x_4 * z_0 + x_5 * z_1 + x_6 * z_2 + x_7 * z_3 + x_8 * z_4 + x_9 * z_5 + x_{10} * z_6 + x_{11} * z_7 \\ \text{acc}_5 = x_5 * z_0 + x_6 * z_1 + x_7 * z_2 + x_8 * z_3 + x_9 * z_4 + x_{10} * z_5 + x_{11} * z_6 + x_{12} * z_7 \\ \text{acc}_6 = x_6 * z_0 + x_7 * z_1 + x_8 * z_2 + x_9 * z_3 + x_{10} * z_4 + x_{11} * z_5 + x_{12} * z_6 + x_{13} * z_7 \\ \text{acc}_7 = x_7 * z_0 + x_8 * z_1 + x_9 * z_2 + x_{10} * z_3 + x_{11} * z_4 + x_{12} * z_5 + x_{13} * z_6 + x_{14} * z_7 \end{array}$$

Consulting the intrinsic documentation and searching for *mul8* intrinsic we note two options for int16 data and int8 coefficients:

<a href="#"><u>v8acc48</u></a>	<a href="#"><u>mul8</u></a> ( <a href="#"><u>v64int16</u></a> xbuff, int xstart, unsigned int xoffsets, int xstep, unsigned int xsquare, <a href="#"><u>v32int8</u></a> zbuff, int zstart, unsigned int zoffsets, int zstep, unsigned int zsquare)
	Multiply intrinsic function.
<a href="#"><u>v8acc48</u></a>	<a href="#"><u>mul8</u></a> ( <a href="#"><u>v32int16</u></a> xbuff, int xstart, unsigned int xoffsets, int xstep, unsigned int xsquare, <a href="#"><u>v32int8</u></a> zbuff, int zstart, unsigned int zoffsets, int zstep, unsigned int zsquare)
	Multiply intrinsic function using small X input buffer.

We know by design that the math for our low-order FIR can be calculated in a single clock cycle but depending on the vector loads we must process to deliver data and coefficients to the AIE it could

extend to multiple clock cycles. Each AIE vector processor can perform up to 7 operation per cycle utilizing the VLIW SIMD architecture<sup>3</sup>, however it is limited to two vector loads and one vector store per clock cycle so we need to structure the vectorized input/output accordingly.

Referencing the FIR equations shows that to support our data vector we need  $x_0$  thru  $x_{14}$  - a total of fifteen 16-bit words. For the coefficients we need  $z_0$  thru  $z_7$ , a total of eight 8-bit words. We should select the intrinsic with vector buffers closest to those needs, which is the one designated “small X input buffer” which uses a `v32int16` for the data and `v32int8` for coefficients.

Referencing table 3 we also note that with 16-bit data and 8-bit coefficients we will need permute squares on both the x and z. We can follow the guidelines for parameterizing to arrive with the following:

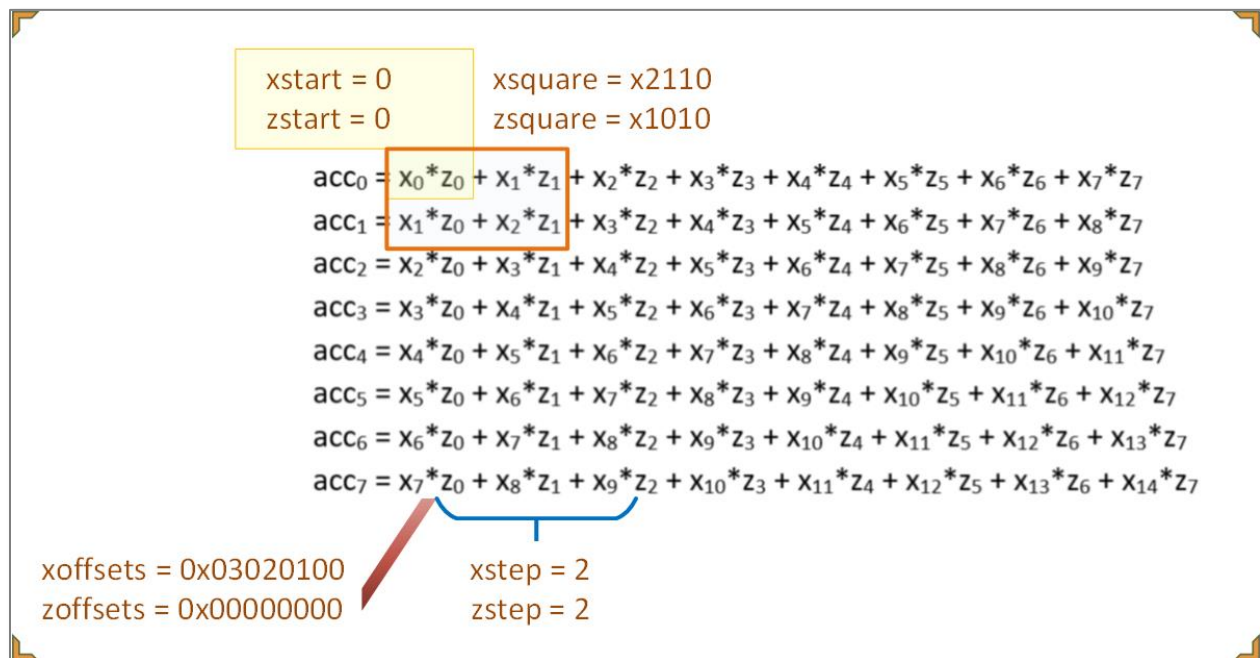


Figure 30: FIR equations parameterized for mul8

Using the prototype for the intrinsic we can code it as follows:

```
acc = mul8(fir_data_in, 0, 0x03020100, 2, 0x2110, coeffs, 0, 0x00000000, 2, 0x1010);
```

## The Vector Registers

Referencing the intrinsic document for the selected mul8 function we now

- Define a user variable “acc” matching the type generated by the intrinsic (v8acc48).

<sup>3</sup> The AIE Very Long Instruction Word (VLIW), Single Instruction Multiple Data (SIMD) architecture is covered in Xilinx documents AM009 and UG1076.

- Define a user variable “rd\_buffer” to load 128-bit or 256-bit vectors for data (select 256 in this case for v16int16 to load the required data,  $x_0$  thru  $x_{14}$ ).
- Define a user variable “fir\_data\_in” matching the type used for *xbuff* in the intrinsic (v32int16).
- Define a user variable “fir\_data\_out” matching the number of lanes (8) and desired output data type (int16) to serve as the output vector (v8int16).
- Define a user variable “coeffs” matching the type used for *zbuff* in the intrinsic (v32int8).

These can be declared and initialized in code as

```
v8acc48 acc;
v16int16 rd_buffer = undef_v16int16();
v32int16 fir_data_in = undef_v32int16();
v8int16 fir_data_out = undef_v8int16();
v32int8 coeffs = *(v32int8 *)taps;
```

Note that while the vector buffer register *rd\_buffer* should be either 128-bits or 256-bits wide to match the vector load capability of the AIE processor, all other variable types are set to match the requirements of the selected intrinsic (the *mul8* with small x buffer in this case). You will find that it helps to keep the intrinsic documentation handy while coding.

### Coefficients

Also note that the coefficients are initialized from *taps*; they can be set statically within the kernel by initializing *taps* as

```
alignas(8) int8_t taps[32] = { 1, 2, 3, 4, 5, 6, ..., 32};
```

where the left most value (1) will become the first coefficient in the FIR and the rightmost (32) will be the last. Those numbers are of course placeholders; let’s implement the real values now.

From filter design software, the coefficients for our example filter are generated as floating point numbers:

```
-0.00134277343750000, 0.0599365234375000, 0.224975585937500, 0.381958007812500,
0.381958007812500, 0.224975585937500, 0.0599365234375000, -0.00134277343750000 .
```

To convert into int8 format they are first converted from floating point into signed binary fractional then scaled. Normal rules for scaling FIR coefficients apply, with the additional consideration that scaling by a power of two allows the introduced gain to be factored out without performance or resource penalty within the AIE. This is because the AIE accumulator lanes can be stored to the output vector with a bit shift, allowing power of two coefficient scaling to be divided out to maintain unity gain through the filter (neglecting any fixed-point rounding error introduced). The bit shifting does not require any additional cycles since it relies on dedicated accumulator permute logic.

For this example we will scale the coefficients by  $2^7$ . This is reflected by the parameter `FIR_COEFF_BIT_SHIFT` in the header, which will be set to seven. Since the vector format is `v32int8`, the 24 unused elements can be set to zero yielding the scaled fixed-point coefficients:

```
alignas(8) int8_t taps[32] = { 0, 8, 29, 49, 49, 29, 8, 0,
```

The astute reader will notice that the eight coefficients are symmetric (since they were generated for linear phase). If you know your coefficients will be symmetric then you can utilize one of the “\_sym” intrinsics as listed in table 2 to further optimize the implementation. This will be shown in an upcoming example.

Alternately, we could generate asymmetric coefficients to make this a minimum-phase FIR filter. Since the minimum-phase FIR requires an even order we select a filter order of six which will generate seven coefficients. The unused eighth coefficient can be set to zero. And since all FIR calculations are performed in one clock cycle the zero coefficient will not impact the delay through the AIE-implemented filter.

The only code change to switch from the 8-tap symmetric filter to a 7-tap minimum-phase asymmetric variant would be to replace the *taps* assignment with the seven asymmetric coefficients, scaled as before by  $2^7$  to fit into the int8 numeric format. The new coefficients and the minimum-phase filter response are as follows:

```
alignas(8) int8_t taps[32] = { 18, 44, 54, 29, -3, -16, -7, 0,  
0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0};
```

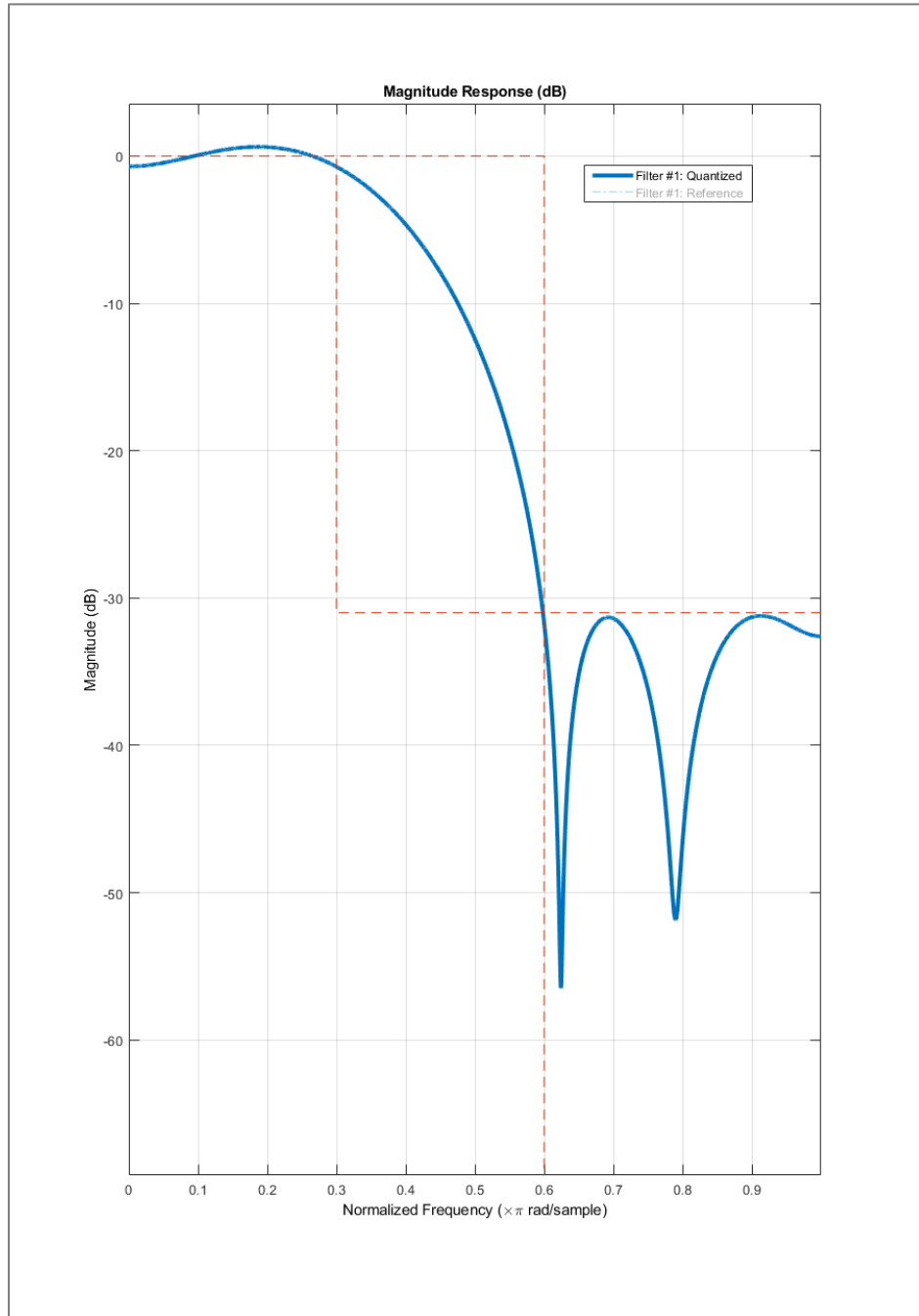


Figure 31: Filter response for minimum-phase asymmetric coefficients

## The Second Intrinsic

The next intrinsic essential to FIR coding is the update intrinsic which comes in two widths,

- `upd_v` = 128-bit vector update
- `upd_w` = 256-bit vector update.

These intrinsics update a vector register in indexed portions. For the application at hand the *fir\_data\_in* vector is 512-bits wide and is updated from the 256-bit *rd\_buffer* vector, which can be coded as

```
fir_data_in = upd_w(fir_data_in, 0, rd_buffer);
```

Since a 512-bit register is being updated in 256-bit increments the index can be either 0 (lower-order half of the vector) or 1 (upper half). In the above code the lower 256-bits of the wider vector, sized to match the *mul8*'s required buffer size, will carry all the data input required to implement the FIR equations since it has a capacity of sixteen 16-bit words and the FIR equations only needed fifteen ( $x_0$  thru  $x_{14}$ ). A single update thus brings in enough data to support the FIR processing. In some FIRs you may need to do multiple updates to fill a vector sufficiently for processing.

## Data Input and Output

Since the kernel consumes and produces data there needs to be a mechanism to move data vectors. This is handled by calls to the window and streaming data API. The API handles the details of synchronizing data, allowing the kernel developer to focus on the data.

The actual transport mechanism can be

- 1) AXI4 streams through the streaming interconnects and switches in the AIE array,
- 2) windowed through the AIE memory blocks and DMA hardware,
- 3) streaming across the dedicated cascade interface from/to an adjacent AIE,
- 4) or packetized streams to share a common path.

The type declared for the kernel's inputs and outputs is what establishes the type of interface. For the example design, set our FIR up with window inputs/outputs as follows:

```
void fir_asym_4t_16x8real(input_window_int16 * restrict in,  
    output_window_int16 * restrict out)
```

Likewise, we could have declared the same inputs/outputs as AXI4 streams using

```
void fir_asym_4t_16x8real(input_stream_int16 * restrict in,  
    output_stream_int16 * restrict out).
```



The API calls to move data in and out of the kernel are essentially the same once the type is declared. See UG1076 and AM009 for further details on interconnects and the associated API calls.

To bring data into the kernel use

***window\_readincr(in, rd\_buffer);***

This will read in the amount of incoming data to fill the target register, which in this case is the v16int16 (16 x 16-bit) *rd\_buffer*. To write data out of the kernel use

***window\_writeincr(out, fir\_data\_out);***

This will write out the amount of data in the vector register *fir\_data\_out* which was declared as a v8int16 (i.e. 8 x 16-bits).

### The Pointer

These API calls for windowed kernel reads and writes track the current position within the vectorized data using a pointer. The API handles pointer initialization, incrementing, and decrementing as data moves. The kernel developer does not need to be concerned with the actual value of the pointer, just the relative motion. To see this in action consider the current FIR: Input to the kernel is loaded into *rd\_buffer* as a v16int16 vector, and out of that vector the FIR equations use elements 0 through 14. Since we loaded 16 elements, the pointer advances 16 regardless of which elements were used or not used. Reviewing the FIR equations of figure 25 shows that that vector calculated eight FIR outputs, and that the next sequence should be indexed as follows:

$\oplus \rightarrow$  Columns  
 $\downarrow$   
 Lanes

$$\begin{aligned}
 \text{acc}_0 &= x_0 * z_0 + x_1 * z_1 + x_2 * z_2 + x_3 * z_3 + x_4 * z_4 + x_5 * z_5 + x_6 * z_6 + x_7 * z_7 \\
 \text{acc}_1 &= x_1 * z_0 + x_2 * z_1 + x_3 * z_2 + x_4 * z_3 + x_5 * z_4 + x_6 * z_5 + x_7 * z_6 + x_8 * z_7 \\
 \text{acc}_2 &= x_2 * z_0 + x_3 * z_1 + x_4 * z_2 + x_5 * z_3 + x_6 * z_4 + x_7 * z_5 + x_8 * z_6 + x_9 * z_7 \\
 \text{acc}_3 &= x_3 * z_0 + x_4 * z_1 + x_5 * z_2 + x_6 * z_3 + x_7 * z_4 + x_8 * z_5 + x_9 * z_6 + x_{10} * z_7 \\
 \text{acc}_4 &= x_4 * z_0 + x_5 * z_1 + x_6 * z_2 + x_7 * z_3 + x_8 * z_4 + x_9 * z_5 + x_{10} * z_6 + x_{11} * z_7 \\
 \text{acc}_5 &= x_5 * z_0 + x_6 * z_1 + x_7 * z_2 + x_8 * z_3 + x_9 * z_4 + x_{10} * z_5 + x_{11} * z_6 + x_{12} * z_7 \\
 \text{acc}_6 &= x_6 * z_0 + x_7 * z_1 + x_8 * z_2 + x_9 * z_3 + x_{10} * z_4 + x_{11} * z_5 + x_{12} * z_6 + x_{13} * z_7 \\
 \text{acc}_7 &= x_7 * z_0 + x_8 * z_1 + x_9 * z_2 + x_{10} * z_3 + x_{11} * z_4 + x_{12} * z_5 + x_{13} * z_6 + x_{14} * z_7
 \end{aligned}$$

*(end of first AIE math cycle)*

*(start of next AIE math cycle)*

$$\begin{aligned}
 \text{acc}_0 &= x_8 * z_0 + x_9 * z_1 + x_{10} * z_2 + x_{11} * z_3 + x_{12} * z_4 + x_{13} * z_5 + x_{14} * z_6 + x_{15} * z_7 \\
 \text{acc}_1 &= x_9 * z_0 + x_{10} * z_1 + x_{11} * z_2 + x_{12} * z_3 + x_{13} * z_4 + x_{14} * z_5 + x_{15} * z_6 + x_{16} * z_7 \\
 \text{acc}_2 &= x_{10} * z_0 + x_{11} * z_1 + x_{12} * z_2 + x_{13} * z_3 + x_{14} * z_4 + x_{15} * z_5 + x_{16} * z_6 + x_{17} * z_7 \\
 \text{acc}_3 &= x_{11} * z_0 + x_{12} * z_1 + x_{13} * z_2 + x_{14} * z_3 + x_{15} * z_4 + x_{16} * z_5 + x_{17} * z_6 + x_{18} * z_7 \\
 \text{acc}_4 &= x_{12} * z_0 + x_{13} * z_1 + x_{14} * z_2 + x_{15} * z_3 + x_{16} * z_4 + x_{17} * z_5 + x_{18} * z_6 + x_{19} * z_7 \\
 \text{acc}_5 &= x_{13} * z_0 + x_{14} * z_1 + x_{15} * z_2 + x_{16} * z_3 + x_{17} * z_4 + x_{18} * z_5 + x_{19} * z_6 + x_{20} * z_7 \\
 \text{acc}_6 &= x_{14} * z_0 + x_{15} * z_1 + x_{16} * z_2 + x_{17} * z_3 + x_{18} * z_4 + x_{19} * z_5 + x_{20} * z_6 + x_{21} * z_7 \\
 \text{acc}_7 &= x_{15} * z_0 + x_{16} * z_1 + x_{17} * z_2 + x_{18} * z_3 + x_{19} * z_4 + x_{20} * z_5 + x_{21} * z_6 + x_{22} * z_7
 \end{aligned}$$

This poses an issue in that the next cycle of FIR calculation work on data  $x_8$  thru  $x_{22}$ , but the pointer is on  $x_{16}$  and will load  $x_{16}$  thru  $x_{31}$  on the next read. The pointer needs to be decremented back to  $x_8$ . Again, there are API calls for that – the increment/decrement functions. The following code will accomplish the desired pointer adjustment:

***window\_decr\_v8(in, 1);***

This adjusts the pointer backwards an amount equivalent to a `v8int16` vector, where the “v8” portion was built into the function name and the “int16” was derived based on the type of the referenced vector register *in*. The “1” indicates that this adjustment should be done only once; a value of “2” would put the pointer back to its initial position.

The pointer sequence for data input in the example FIR can be visualized as follows:

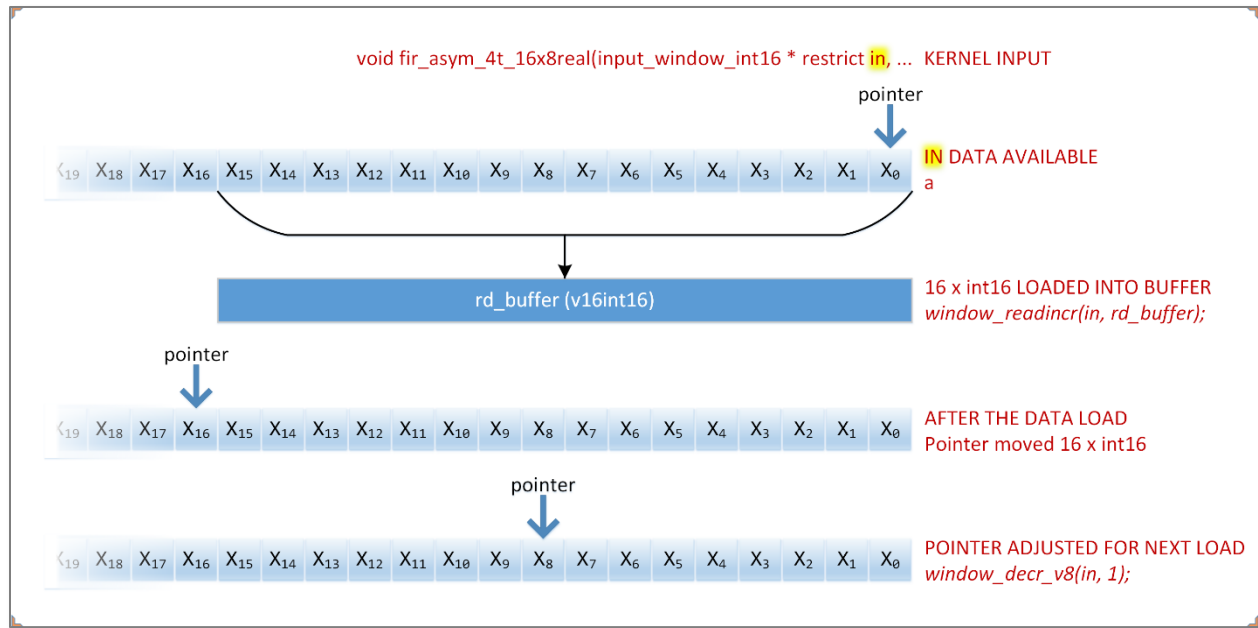


Figure 32: Pointer movement during data input sequence

Now it's time to step back and look at the overall code with respect to the input buffer pointer. Since we're doing a read-with-increment followed later by a decrement we can simplify this to a read (without increment) followed by an increment:

<u>Original</u>	<u>Simplified</u>
<code>window_readincr(in, rd_buffer);</code>	<code>window_read(in, rd_buffer);</code>
<code>: (other code)</code>	<code>: (other code)</code>
<code>window_decr_v8(in, 1);</code>	<code>window_incr(in, 8);</code>

Intuitively, incrementing by 16 then decrementing by 8 is equivalent to just incrementing by 8. Note that either is correct, the idea in the simplified case is that two pointer manipulations have been reduced to one. In general, the compiler will optimize the pointer increment/decrement operations such that the two options yield identical results, so it becomes a matter of coding style preference. Consult UG1076 for a complete listing of the pointer intrinsics.

For most AIE applications these adjustments to pointer position will be required on the input but not on the output.

### The Accumulator and the Third Intrinsic

The results of AIE calculations end up in the accumulator. Each `mul` intrinsic has a set accumulator size with a defined number of lanes and a width for each lane. For the `mul8` employed in the example FIR the accumulator type is `v8acc48`, so it will generate 8 lanes of 48-bit width. The desired kernel output is vectors of `int16` so we need to extract the appropriate bits out of the wider accumulator field and store

them in the output buffer. Hence the receiving v8int16 buffer *fir\_data\_out*. This is also the opportunity to account for the coefficient scaling. Since all coefficients were multiplied by  $2^7$  when converting from floating point into int8 format all FIR accumulator results will be shifted by 7 bit positions. When extracting the accumulator lanes into the receiving register (*fir\_data\_out*) that shifting can be reversed without any performance penalty. This is coded using the *srs* intrinsic:

```
fir_data_out = srs(acc, 7);
```

The bit shift is specified as 7, thereby accounting for coefficient scaling. The fact that we want 16 bits out of the 48-bit accumulator is extrapolated from the receiving buffer's type.

### The Loop

The AIE architecture is optimized to execute vector operations in pipelined loops. The loop loads data, executes the vector operations, writes out the vectorized results, then repeats the process. For our FIR application the calculations that should be inside the loop are the *acc<sub>0</sub>* thru *acc<sub>7</sub>* equations. As that loop iterates it will consume incoming data, position the pointer for the next data load, and write out the results. The accumulator calculations are performed in one clock cycle, and if the vector read/write operations are within the 7-way VLIW processor's two loads plus one store capability then the data inputs and outputs can be executed in that same clock.

To code the loop, the math intrinsic and the supporting intrinsics for reading vectorized data, extracting accumulator outputs, writing vectorized results plus any supporting pointer increment/decrement intrinsics should all be placed inside the loop. The code format is as follows:

```
for(unsigned i = 0; i < AIE_LOOP_CYCLES; i++)  
chess_prepare_for_pipelining  
{  
    // data reading API function  
    // mul intrinsic for the calculations  
    // srs intrinsic for accumulator bit extraction/shifting  
    // data writing API function  
    // pointer adjustment intrinsic  
}
```

### Pragmas

Note the "chess" pragma that informs the compiler that the pipelined loop is about to start. Also note that the loop limit is set here as *AIE\_LOOP\_CYCLES*; that defines how many loop cycles will run each time the kernel is run (under system control via the API). If you have predefined lengths of data then you can set the number of iterations based on the data length divided by how many data words are consumed per cycle. For instance, in our present example the pointer increments by 8 every loop cycle so to process 1024 words through the FIR you could set the loop count as

$$ITERATIONS = \frac{1024}{8}$$

One additional pragma to consider is “chess\_flatten\_loop”. This may be suggested by the tool if you successfully achieve single-cycle execution for the entire loop (both the math intrinsic and the vector load/stores). In some cases adding that pragma can result in more optimal code. The suggested approach is to leave it out initially, then once you achieve successful compilation and functionality consider adding to determine if it improves cycle times for implementing your kernel operation. You can see this pragma applied in some of the example code. For further information on pragmas reference the *ASIP Programmer Chess Compiler User Manual*.

## Putting it All Together

The portions AIE code we just discussed can be assembled into a C header file “include.h” and kernel source file shown here. The header contains parameters for both the test code (to be discussed) and the kernel code. The kernel code includes standard C plus the AIE intrinsics which are defined in the included “cardano.h” file.

Coefficients are set statically in the beginning, the inputs and outputs are declared, vector registers are declared, then the loop starts. Within the loop data is loaded into the kernel, the intrinsic executes the

math equations that define the FIR processing, and the output data from the accumulator lanes is adjusted and stored into a vector register to be written as kernel output. Finally, the pointer is adjusted then the loop cycles.

```
#ifndef INCLUDE_H
#define INCLUDE_H

// Kernel-specific based on intrinsics
#define AIE_MUL_LANES      8
#define AIE_POINTER_ADJUST 8

// FIR-specific
#define FIR_COEFF_BIT_SHIFT 7

// Simulation parameters
#define NUM_INPUT_WORDS    32
#define NUM_OUTPUT_WORDS  (NUM_INPUT_WORDS)
#define AIE_LOOP_CYCLES   (NUM_INPUT_WORDS / AIE_MUL_LANES)
#define INPUT_FILE         "./SimInputs.txt"

// Kernel prototype
void fir_asym_4t_16x8real(input_window_int16 *a, output_window_int16 *c);

#endif // INCLUDE_H
```

```
#include <cardano.h>
#include <stdio.h>
#include "include.h"

alignas(8) int8_t taps[32] = { 0, 8, 29, 49, 49, 29, 8, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0 }; // from lowest order to highest, only first 8 used

void fir_asym_4t_16x8real(input_window_int16 * restrict in, output_window_int16 * restrict out)
{
    v8acc48 acc;
    v16int16 rd_buffer = undef_v16int16();
    v32int16 fir_data_in = undef_v32int16();
    v8int16 fir_data_out = undef_v8int16();
    v32int8 coeffs = *(v32int8 *)taps;

    for(unsigned i = 0; i<AIE_LOOP_CYCLES; i++)
    {
        chess_prepare_for_pipelining
        {
            window_read(in, rd_buffer);
            fir_data_in = upd_w(fir_data_in, 0, rd_buffer);
            acc = mul8(fir_data_in, 0, 0x03020100, 2, 0x2110, coeffs, 0, 0x00000000, 2, 0x1010);
            fir_data_out = srs(acc, FIR_COEFF_BIT_SHIFT);
            window_writeincr(out, fir_data_out);
            window_incr(in, AIE_POINTER_ADJUST);
        }
    }
}
```

Cardano dataflow library, include file for for intrinsics and vector types (change to “adf.h” in Vitis 2020.2)

FIR coefficients

vector registers

kernel inputs and outputs

read in data

math intrinsic, implements the FIR equations

write out data

data input pointer adjustment

pipelined loop

These files are included in the GitHub repository associated with this document in the folder “fir\_asym\_8t\_16x8real”.

## Taking Advantage of Symmetry

It was previously noted that the coefficients for the example filter demonstrated symmetry. That means that we have the option of using one of the *mul\_sym* or *mul\_sym\_ct* intrinsics to implement multipliers with preadders, thereby increasing the number of multiply-add operations supported for the selected data and coefficient types. That increase can potentially be applied to

- a) keep the same number of lanes while doubling the number of coefficients (or doubling minus one for an odd number of symmetric coefficients),
- b) keep the same number of coefficients while increasing the number of lanes, or
- c) increasing bit width.

Note the word “potentially” since these options are conditional on intrinsic support. To clarify this, let’s explore these options for our example.

For option (a), the current code employed a *mul8* to process 8 coefficients, so a *mul8\_sym* would support 16 symmetric coefficients. Consulting the intrinsic documentation for the types involved shows that there are no symmetric *mul\_sym* options that support 16-bit x 8-bit numeric formats. This option is not available for those bit widths.

For (b), the same issue will apply – that there are no *mul\_sym* intrinsics that support 16-bit x 8-bit math.

For (c), we could switch to 16-bit coefficients (the original plan). This highlights a good point; when planning your filter another key data point is coefficient symmetry.

To set about changing the current design to 16-bit data and coefficients, start with the intrinsic:

- Open either the Vitis™ “Help / AIE Help” menu or the downloaded intrinsic documentation<sup>4</sup>.
- Select “Intrinsics and Datatypes / Vector Operations / MAC intrinsics / Multiplication / Advanced / Integer / Vector MAC with pre-adding “
- Select the new configuration “16 bit Real x 16 bit Real”.
- Search for supported “mul” intrinsics and note which are present. Those are your valid choices.

Noting that there are two choices for *mul8\_sym*,

---

<sup>4</sup> In both cases the documentation is delivered in HTML format, with “index.html” as the starting point.



<a href="#"><u>v8acc48</u></a>	<a href="#"><u>mul8_sym</u></a> ( <a href="#"><u>v64int16</u></a> xbuff, int xstart, unsigned int xoffsets, int xstep, unsigned int xsquare, int ystart, unsigned int ysquare, <a href="#"><u>v16int16</u></a> zbuff, int zstart, unsigned int zoffsets, int zstep)
	Symetric multiply intrinsic function with pre-add from x input buffer .

<a href="#"><u>v8acc48</u></a>	<a href="#"><u>mul8_sym</u></a> ( <a href="#"><u>v32int16</u></a> xbuff, int xstart, unsigned int xoffsets, int xstep, unsigned int xsquare, <b>int ystart, unsigned int ysquare</b> , <a href="#"><u>v16int16</u></a> zbuff, int zstart, unsigned int zoffsets, int zstep)
	Symetric multiply intrinsic function with pre-add from x input buffer using small X input buffer.

Both allow pre-adding of values from the x buffer, so we select the smallest buffer sizes that meet our requirements (the second). Compared to the previous *mul8* there are two changes – the addition of *ystart* and *ysquare* for selecting data pairs for adding, plus the switch from *zsquare* to the general scheme (*zstart*, *zoffsets*, *zstep* without a permute square) for coefficients. Note that this aligns with table 3.

Recalculating for 16-bit coefficients allows the filter to be retuned for better pass band ripple and/or stopband rejection. A balance of the two provides recalculated coefficients

$$z_0 = -1371, z_1 = -63, z_2 = 6005, z_3 = 12679, z_4 = -12679, z_5 = 6005, z_6 = -63, z_7 = -1371.$$

The symmetry pairing for the coefficients is

$$(z_0 = z_7), (z_1 = z_6), (z_2 = z_5), (z_3 = z_4).$$

Pre-adding data across the symmetry pairs yields the optimized FIR equation

$$acc_0 = (x_0 + x_7) * z_0 + (x_1 + x_6) * z_1 + (x_2 + x_5) * z_2 + (x_3 + x_4) * z_3.$$

This illustrates the fact that now four multipliers can support eight data taps with symmetry. Extending across all lanes, the previously implemented FIR can be restructured to take advantage of symmetry and increase coefficient width as follows:

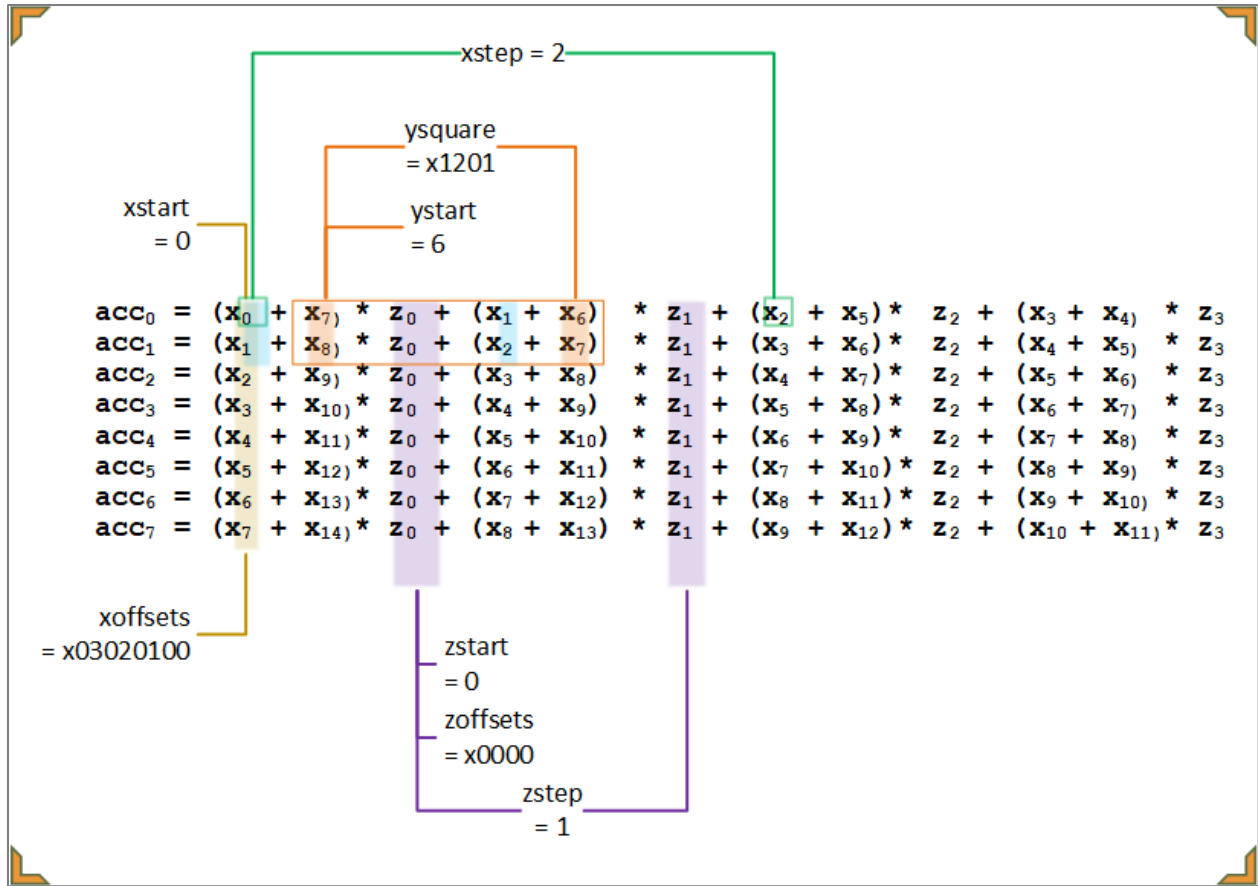


Figure 33: Symmetric FIR with additional y permute square

Note that the “x” indexes ( $xstart$ ,  $xstep$ ,  $xoffsets$ ) apply to the leftmost x value in the preadded pair, while the “y” indexes ( $ystart$ ,  $ysquare$ ) apply to the right value. The  $ysquare$  values follow the 2x2 permute square patterns shown here, and like the other 2x2 squares the default value of x3210 applies as shown. Since the data start values (both “x” and “y”) must be multiples of two the  $ystart$  value of six combined with the square values shown in the figure above generate the first “y” square (highlighted in orange). There is no step for the “y” values since they will be sequenced for symmetry based on the established starting values.

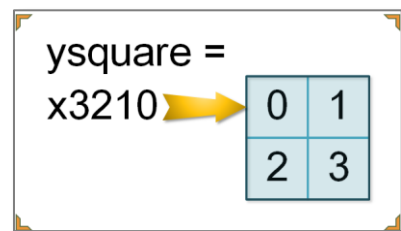


Figure 34: Permute square for the y values

Adding the new “y” parameters and removing the permute square for the “z” values then applying the updated parameter values yields the multiply code:

```
acc = mul8_sym(fir_data_in, 0, 0x03020100, 2, 0x2110, 6, 0x1201, coeffs, 0, 0x00000000, 1);
```

Building out from that intrinsic following the same process covered in the previous section yields the following:

```

#ifndef INCLUDE_H
#define INCLUDE_H

// Kernel-specific based on intrinsics
#define AIE_RD_BUFFER_V      16
#define AIE_MUL_LANES        8
#define AIE_POINTER_ADJUST   (AIE_RD_BUFFER_V - AIE_MUL_LANES)

// FIR-specific
#define FIR_COEFF_BIT_SHIFT   15

// Simulation parameters
#define NUM_INPUT_WORDS       32
#define NUM_OUTPUT_WORDS      (NUM_INPUT_WORDS)
#define AIE_LOOP_CYCLES      (NUM_INPUT_WORDS / AIE_MUL_LANES)
#define INPUT_FILE            "./SimInputs.txt"

// Kernel prototype
void fir_sym_8t_16x16real(input_window_int16 *a, output_window_int16 *c);

#endif // INCLUDE_H

```

Note that the FIR\_COEFF\_BIT\_SHIFT increased to 15, reflecting the scaling factor of  $2^{15}$  applied to the coefficients.

This code and the associated test code are in the folder "fir\_sym\_8t\_16x16real".

```

#include <cardano.h>
#include <stdio.h>
#include "include.h"

alignas(16) int16_t taps[16] = { -1371, -63, 6005, 12679, 12679, 6005, -63, -1371,
                                0, 0, 0, 0, 0, 0, 0, 0 }; // from lowest order to highest, only first 4 used due to symmetry

void fir_sym_8t_16x16real(input_window_int16 * restrict in, output_window_int16 * restrict out)
{
    v8acc48 acc;
    v16int16 rd_buffer = undef_v16int16();
    v32int16 fir_data_in = undef_v32int16();
    v8int16 fir_data_out = undef_v8int16();
    v16int16 coeffs = *(v16int16 *)taps;

    for(unsigned i = 0; i<AIE_LOOP_CYCLES; i++)
    chess_prepare_for_pipelining
    {
        window_read(in, rd_buffer);
        fir_data_in = upd_w(fir_data_in, 0, rd_buffer);
        acc = mul8_sym(fir_data_in, 0, 0x03020100, 2, 0x2110, 6, 0x1201, coeffs, 0, 0x00000000, 1);
        fir_data_out = srs(acc, FIR_COEFF_BIT_SHIFT);
        window_writeincr(out, fir_data_out);
        window_incr(in, AIE_POINTER_ADJUST);
    }
}

```

Cardano dataflow library, include file for for intrinsics and vector types (change to "adf.h" in Vitis 2020.2)

FIR coefficients

kernel inputs and outputs

math intrinsic, implements the FIR equations

read in data

write out data

data input pointer adjustment

pipelined loop

vector registers

## 8-bit Real Data and Coefficients

The first example developed throughout this text worked on 16-bit real data and 8-bit real coefficients. The next example took advantage of coefficient symmetry which allowed the coefficients to grow to 16-bits. That leaves one remaining row in table 3 that has not been covered, 8-bit real data and coefficients.

For 8-bit real math table 1 shows the processing capability as 128 MACs. Consulting the intrinsic documentation the available *mul* intrinsics include *mul8* and *mul16* which yield 16 or 8 taps (derived as 128/8 or 128/16) of asymmetric FIR processing respectively. Since we've been working with an 8-tap filter the *mul16* is selected, providing 16-lanes of processing in a single clock cycle.

In the prior examples the lanes generate successive outputs, for instance the 16-bit data with 8-bit coefficient code produces 8 outputs per clock cycle equivalent to 8 sequential outputs of a standard FIR filter. For this example we will vary the scheme and utilize the 16 lanes as two separate FIR filters, one for the even vector elements and the other for the odd vector elements. In this manner the single AIE can implement two 8-tap FIR filters which each generate 8 outputs per clock cycle. For this example we will utilize the previously generated minimum-phase asymmetric coefficients for both the even and odd filters. The equations to represent this can be expressed in equations as follows:

$$\begin{array}{l} \oplus \rightarrow \text{Columns} \\ \downarrow \\ \text{Lanes} \end{array} \begin{array}{l} \text{acc}_0 = x_0 * z_0 + x_2 * z_1 + x_4 * z_2 + x_6 * z_3 + x_8 * z_4 + x_{10} * z_5 + x_{12} * z_6 + x_{14} * z_7 \\ \text{acc}_1 = x_1 * z_0 + x_3 * z_1 + x_5 * z_2 + x_7 * z_3 + x_9 * z_4 + x_{11} * z_5 + x_{13} * z_6 + x_{15} * z_7 \\ \text{acc}_2 = x_2 * z_0 + x_4 * z_1 + x_6 * z_2 + x_8 * z_3 + x_{10} * z_4 + x_{12} * z_5 + x_{14} * z_6 + x_{16} * z_7 \\ \text{acc}_3 = x_3 * z_0 + x_5 * z_1 + x_7 * z_2 + x_9 * z_3 + x_{11} * z_4 + x_{13} * z_5 + x_{15} * z_6 + x_{17} * z_7 \\ \text{acc}_4 = x_4 * z_0 + x_6 * z_1 + x_8 * z_2 + x_{10} * z_3 + x_{12} * z_4 + x_{14} * z_5 + x_{16} * z_6 + x_{18} * z_7 \\ \text{acc}_5 = x_5 * z_0 + x_7 * z_1 + x_9 * z_2 + x_{11} * z_3 + x_{13} * z_4 + x_{15} * z_5 + x_{17} * z_6 + x_{19} * z_7 \\ \text{acc}_6 = x_6 * z_0 + x_8 * z_1 + x_{10} * z_2 + x_{12} * z_3 + x_{14} * z_4 + x_{16} * z_5 + x_{18} * z_6 + x_{20} * z_7 \\ \text{acc}_7 = x_7 * z_0 + x_9 * z_1 + x_{11} * z_2 + x_{13} * z_3 + x_{15} * z_4 + x_{17} * z_5 + x_{19} * z_6 + x_{21} * z_7 \\ \text{acc}_8 = x_8 * z_0 + x_{10} * z_1 + x_{12} * z_2 + x_{14} * z_3 + x_{16} * z_4 + x_{18} * z_5 + x_{20} * z_6 + x_{22} * z_7 \\ \text{acc}_9 = x_9 * z_0 + x_{11} * z_1 + x_{13} * z_2 + x_{15} * z_3 + x_{17} * z_4 + x_{19} * z_5 + x_{21} * z_6 + x_{23} * z_7 \\ \text{acc}_{10} = x_{10} * z_0 + x_{12} * z_1 + x_{14} * z_2 + x_{16} * z_3 + x_{18} * z_4 + x_{20} * z_5 + x_{22} * z_6 + x_{24} * z_7 \\ \text{acc}_{11} = x_{11} * z_0 + x_{13} * z_1 + x_{15} * z_2 + x_{17} * z_3 + x_{19} * z_4 + x_{21} * z_5 + x_{23} * z_6 + x_{25} * z_7 \\ \text{acc}_{12} = x_{12} * z_0 + x_{14} * z_1 + x_{16} * z_2 + x_{18} * z_3 + x_{20} * z_4 + x_{22} * z_5 + x_{24} * z_6 + x_{26} * z_7 \\ \text{acc}_{13} = x_{13} * z_0 + x_{15} * z_1 + x_{17} * z_2 + x_{19} * z_3 + x_{21} * z_4 + x_{23} * z_5 + x_{25} * z_6 + x_{27} * z_7 \\ \text{acc}_{14} = x_{14} * z_0 + x_{16} * z_1 + x_{18} * z_2 + x_{20} * z_3 + x_{22} * z_4 + x_{24} * z_5 + x_{26} * z_6 + x_{28} * z_7 \\ \text{acc}_{15} = x_{15} * z_0 + x_{17} * z_1 + x_{19} * z_2 + x_{21} * z_3 + x_{23} * z_4 + x_{25} * z_5 + x_{27} * z_6 + x_{29} * z_7 \end{array}$$

The input and output data vectors are now channelized, with even/odd elements carrying data for separate filters. That doesn't change the coding process, though; proceeding as before we declare the vector registers, the coefficients, and the input/output to generate the code:

```

#ifndef INCLUDE_H
#define INCLUDE_H

// Kernel-specific based on intrinsics
#define AIE_MUL_LANES          16
#define AIE_POINTER_ADJUST    16

// FIR-specific
#define FIR_COEFF_BIT_SHIFT    7

// Simulation parameters
#define NUM_INPUT_WORDS        64
#define NUM_OUTPUT_WORDS      (NUM_INPUT_WORDS)
#define AIE_LOOP_CYCLES       (NUM_INPUT_WORDS / AIE_MUL_LANES)
#define INPUT_FILE             "./SimInputs.txt"

// Kernel prototype
void fir_asym_dual8t_8x8real(input_window_int8 *a, output_window_int8 *c);

#endif // INCLUDE_H

```

Note that the FIR\_COEFF\_BIT\_SHIFT is 7, reflecting the scaling factor of  $2^7$  applied to the coefficients.

This code and the associated test code are in the folder "fir\_asym\_dual8t\_8x8real".

```

#include <cardano.h>
#include <stdio.h>
#include "include.h"

alignas(8) int8_t taps[32] = { 18, 44, 54, 29, -3, -16, -7, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0,
    0, 0, 0, 0, 0, 0, 0, 0 }; // from lowest order to highest, only first 8 used

void fir_asym_8t_8x8real(input_window_int8 * restrict in, output_window_int8 * restrict out)
{
    v16acc48 acc;
    v32int8 rd_buffer = undef_v32int8();
    v64int8 fir_data_in = undef_v64int8();
    v16int8 fir_data_out = undef_v16int8();
    v32int8 coeffs = *(v32int8 *)taps;

    for(unsigned i = 0; i<AIE_LOOP_CYCLES; i++)
        chess_prepare_for_pipelining
        chess_flatten_loop
        {
            window_read(in, rd_buffer);
            fir_data_in = upd_w(fir_data_in, 0, rd_buffer);
            acc = mull6(fir_data_in, 0, 0x03020100, 4, 0x2110, coeffs, 0, 0x00000000, 2, 0x1010);
            fir_data_out = bsrs(acc, FIR_COEFF_BIT_SHIFT);
            window_writeincr(out, fir_data_out);
            window_incr(in, AIE_POINTER_ADJUST);
        }
}

```

Cardano dataflow library, include file for for intrinsics and vector types (change to "adf.h" in Vitis 2020.2)

FIR coefficients

kernel inputs and outputs

vector registers

pipelined loop

read in data

math intrinsic, implements the FIR equations

write out data

data input pointer adjustment

## Testing

Once you have compiled your kernel code in Vitis™ (or XChessDE for 2020.1) you will need to test it. Two flows are worth considering -- coding a C test bench and/or instantiating the kernel in Model Composer. Both allow you to verify the functionality of your kernel prior to integrating into a larger design.

### C Test Bench

This flow requires you code stimulus and read the response in C/C++. The advantage of this approach is that you work in the same native tool flow used for kernel development. This allows cross-probing between the test code and the kernel source code as well as breakpoints, single-stepping and register watches to support debugging. The disadvantage is the limited options for generating and monitoring signals. For the current example a simple test would be as follows:

```
#include <stdio.h>
#include <stdlib.h>
#include <cardano.h>
#include "include.h" ← Same file as referenced by kernel code

alignas(16) int16_t buffer_aie_in[NUM_INPUT_WORDS + AIE_POINTER_ADJUST];
alignas(16) int16_t buffer_aie_out[NUM_OUTPUT_WORDS];

int main()
{
    // Setup and initialize windowed inputs and outputs
    window_internal_aie_input;
    window_internal_aie_output;
    window_init(&aie_input, 1, buffer_aie_in, (NUM_INPUT_WORDS + AIE_POINTER_ADJUST));
    window_init(&aie_output, 1, buffer_aie_out, NUM_OUTPUT_WORDS);

    // Read test inputs from source file
    int file_buffer;
    FILE *fp = fopen(INPUT_FILE, "r");

    for (int rd_iter = 0; rd_iter < (NUM_INPUT_WORDS + AIE_POINTER_ADJUST); rd_iter++)
    {
        fscanf(fp, "%d\n", &file_buffer);
        window_writeincr((output_window_int16 *)&aie_input, file_buffer);
    }

    // Send the window of data to the kernel
    fir_asym_4t_16x8real(get_input_window_int16(&aie_input), get_output_window_int16(&aie_output));

    // Read the output from the kernel
    for (int out_iter = 0; out_iter < NUM_OUTPUT_WORDS; out_iter++)
    {
        if (out_iter % 8 == 0)
        {
            printf("\n");
        }
        printf("aie_output[%d] = %d\n", out_iter, buffer_aie_out[out_iter]);
    }

    // Close the test data file
    fclose(fp);
    return 0;
}
```

Annotations in the code block:

- kernel input and output windows**: Points to the `window_init` and `window_writeincr` calls.
- read stimulus from file and vectorize into window**: Points to the `fscanf` and `window_writeincr` calls.
- the kernel undergoing testing**: Points to the `fir_asym_4t_16x8real` call.
- log kernel responses to console**: Points to the `printf` calls.

### The test

- 1) reads stimulus from a previously generated text file

- 2) vectorizes those inputs
- 3) runs them through an instance of the kernel
- 4) reads kernel responses
- 5) logs the responses to the console for review.

This approach is suitable for short data sets and the simple responses typical of low-order FIR filters. The expected outputs can be readily calculated in a spreadsheet then compared to the logged responses.

Potential enhancements from this starting point include writing kernel output to a file for post-analysis, longer data sets, and/or adding a reference function to generate expected results for comparison in order to issue a pass-fail verdict.

For the examples associated with this document the test code can be found in the same folder as the kernel source code in a file named “test.cc”.

### Model Composer

For Model Composer you instantiate the “AIE” block from the Model Composer blockset then import your kernel source into that block and specify the input/output types. The block will then show your inputs and outputs, and you can utilize the Simulink® sources and sinks to test the kernel.

You must initially get your code to the point where it compiles in Vitis™ (or XChessDE for releases prior to v2020.2) before working in Model Composer. So a good flow is to develop your filters as detailed here, then instantiate it in a Simulink® model using Model Composer for further simulation.

The key advantage of this approach is the simulation capabilities that Simulink® and Model Composer add. Since the focus of this document is code development, the details of simulating your kernel code in Model Composer will be left to other documents.

## Summary

This document introduced a structured approach to code low-order FIR filters. It also covered key concepts such as AIE vectors, intrinsics, permute squares, and data flow.

Along the way code was developed for three different FIR filters:

FIR Type	Data/Coefficient Format	Folder Name
Asymmetric, 8-tap	16-bit real / 8-bit real	Fir_asym_8t_16x8real
Symmetric, 8-tap	16-bit real / 16-bit real	Fir_sym_8t_16x16real
Asymmetric, 2-channel, 8-tap	8-bit real / 8-bit real	Fir_asym_dual8t_8x8real

Each of these can be implemented in a single AI Engine and can run at the full AIE clock rate<sup>5</sup>. Each folder named above contains

- ✓ kernel source code (same name as the folder with extension “.cc”)
- ✓ test source code (file “test.cc”)
- ✓ header code referenced by both the kernel and test code (file “include.h”)
- ✓ simulation data input (file “SimInputs.txt”)

These examples provide some of the first available reference code for the three cases covered in table 3 which utilize permute squares. Other data/coefficient types that utilize the general scheme follow the same development flow but are simpler since the general scheme employs straightforward indexing without permute squares.

These concepts can be extended to build higher-order filters or even other functions that rely on vectorized math.

And finally, Appendix E captures the flow into a convenient worksheet that you can utilize to assist in your own kernel code development.

### About the author

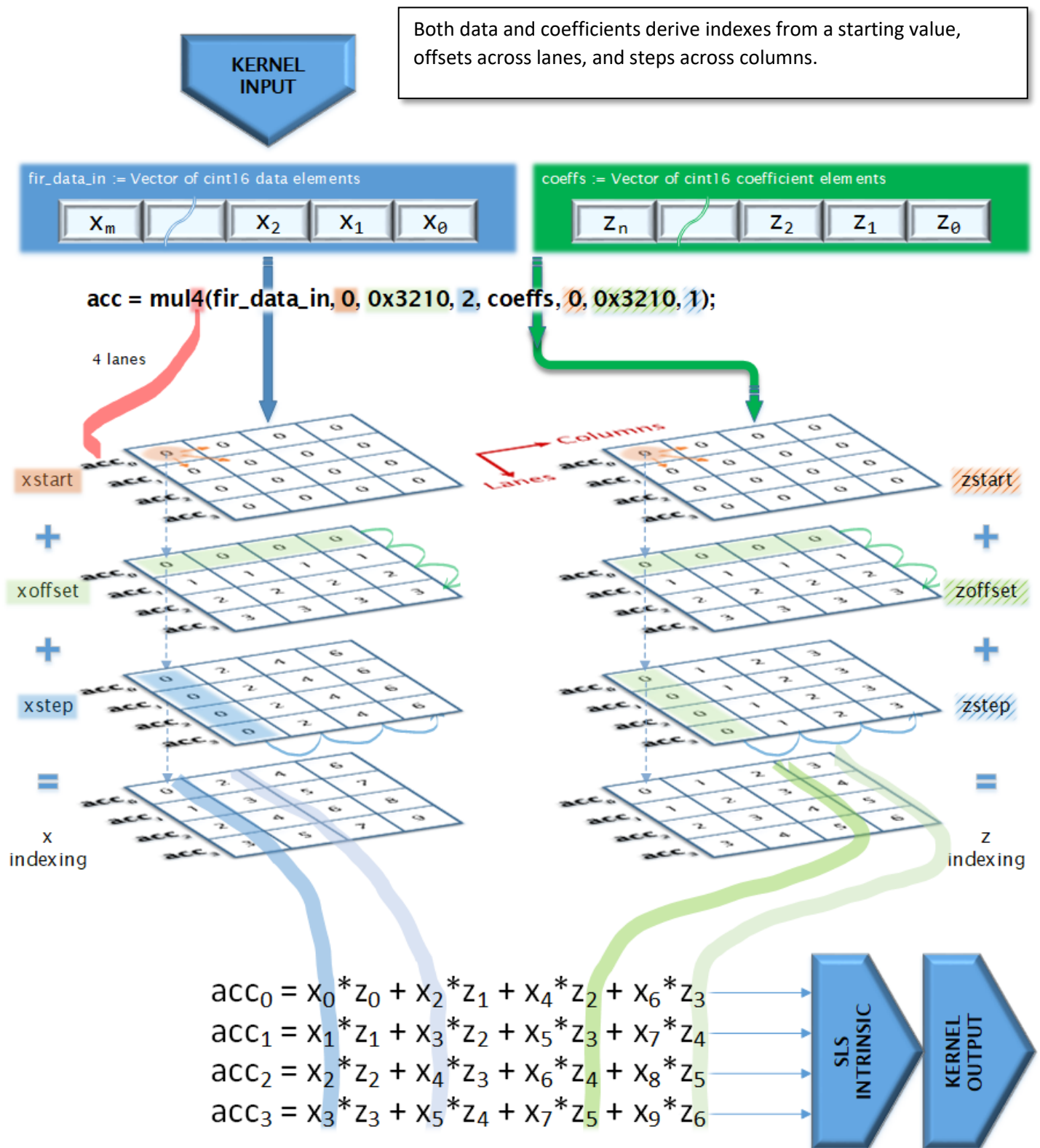
Craig Holmberg is a DSP Field Applications Engineer for Xilinx. He has over 30 years of R&D experience designing electronics for telemetry, defense, digital communications, office imaging, video projection, and medical research. His core areas of expertise are FPGA design, PCB design, and signal processing. Leadership experience includes Director of R&D at a large company and multiple project leadership roles at other companies. He received a BS in Electrical Engineering from Virginia Tech in 1984.

---

<sup>5</sup> 1 GHz for the slowest -2L grade Versal™ device



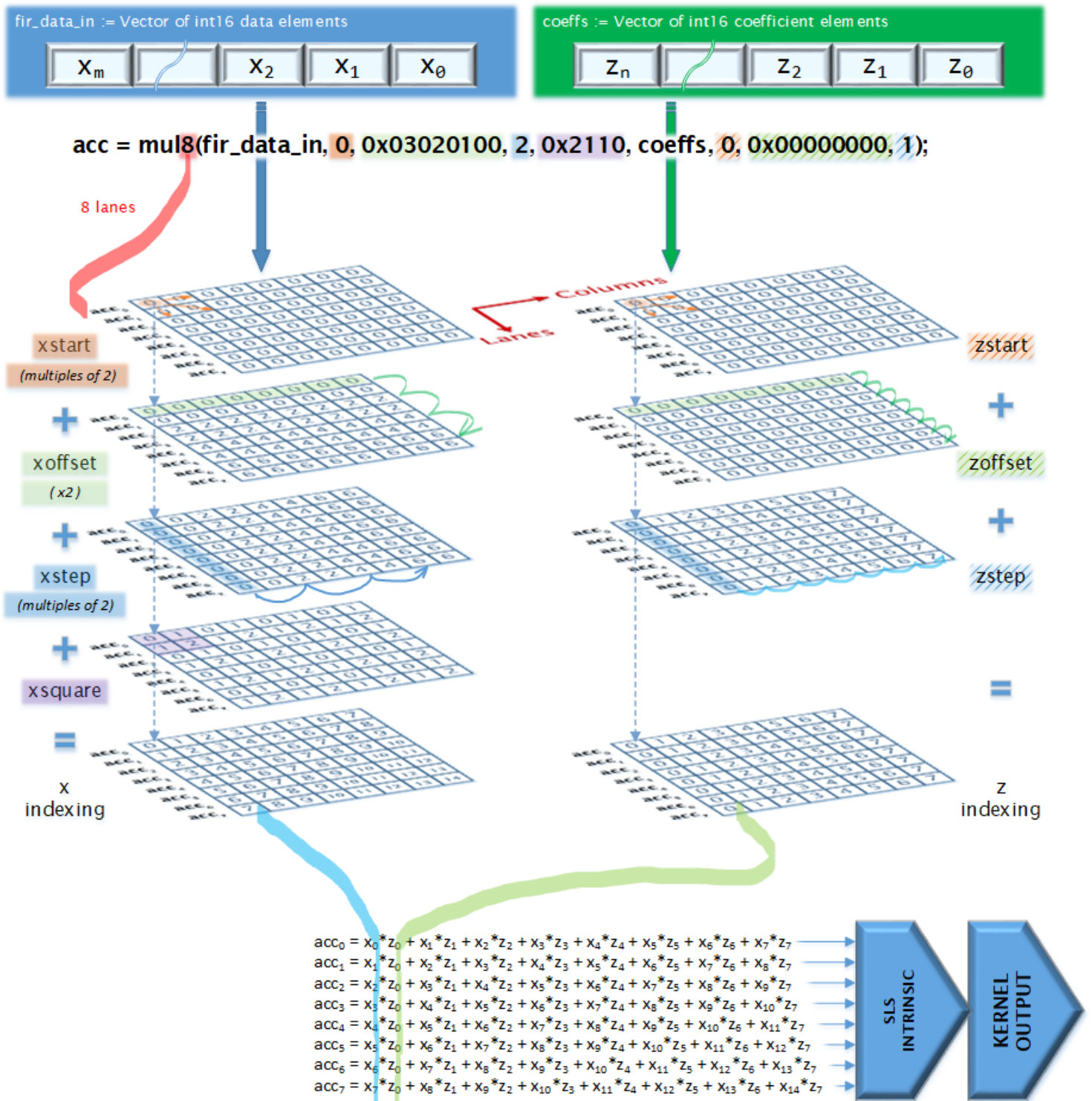
## Appendix A: Indexing for the General Scheme



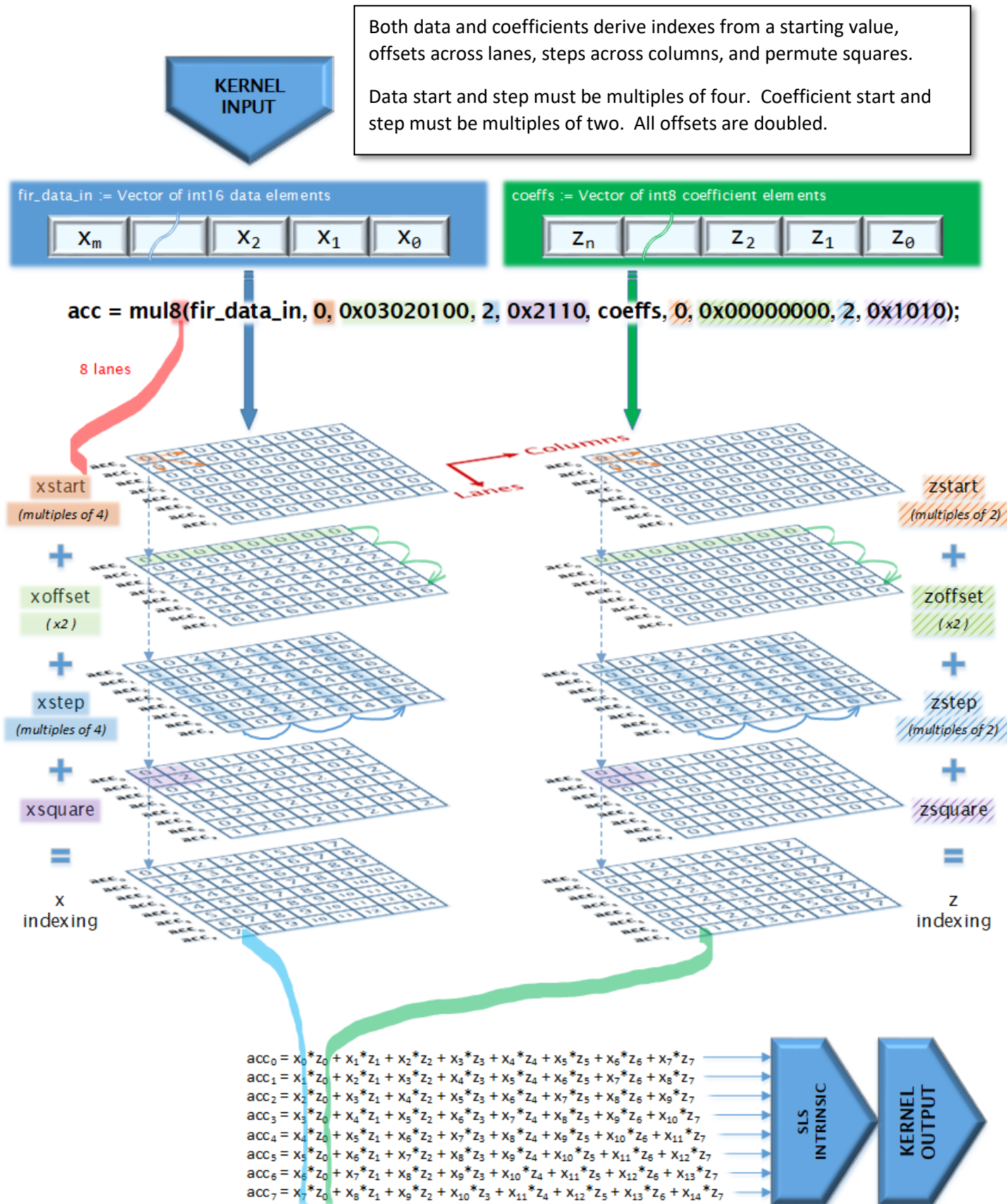
## Appendix B: Indexing for 16-Bit Real Data, 16-Bit Real Coefficients

Both data and coefficients derive indexes from a starting value, offsets across lanes, and steps across columns.

Data start and step must be multiples of two. Data offsets are doubled. Data requires a permute square. Coefficients follow the general scheme.

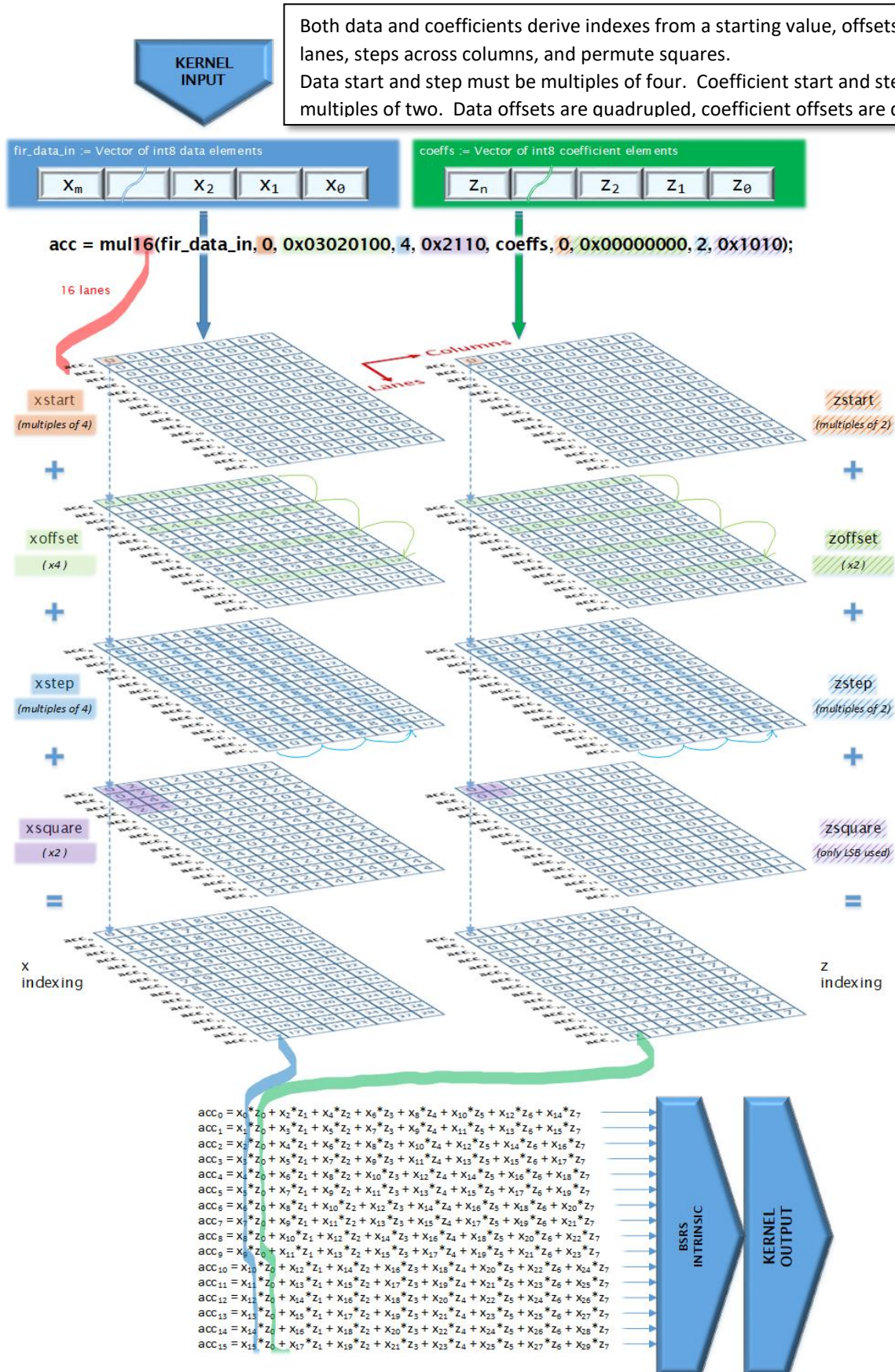


## Appendix C: Indexing for 16-Bit Real Data, 8-Bit Real Coefficients





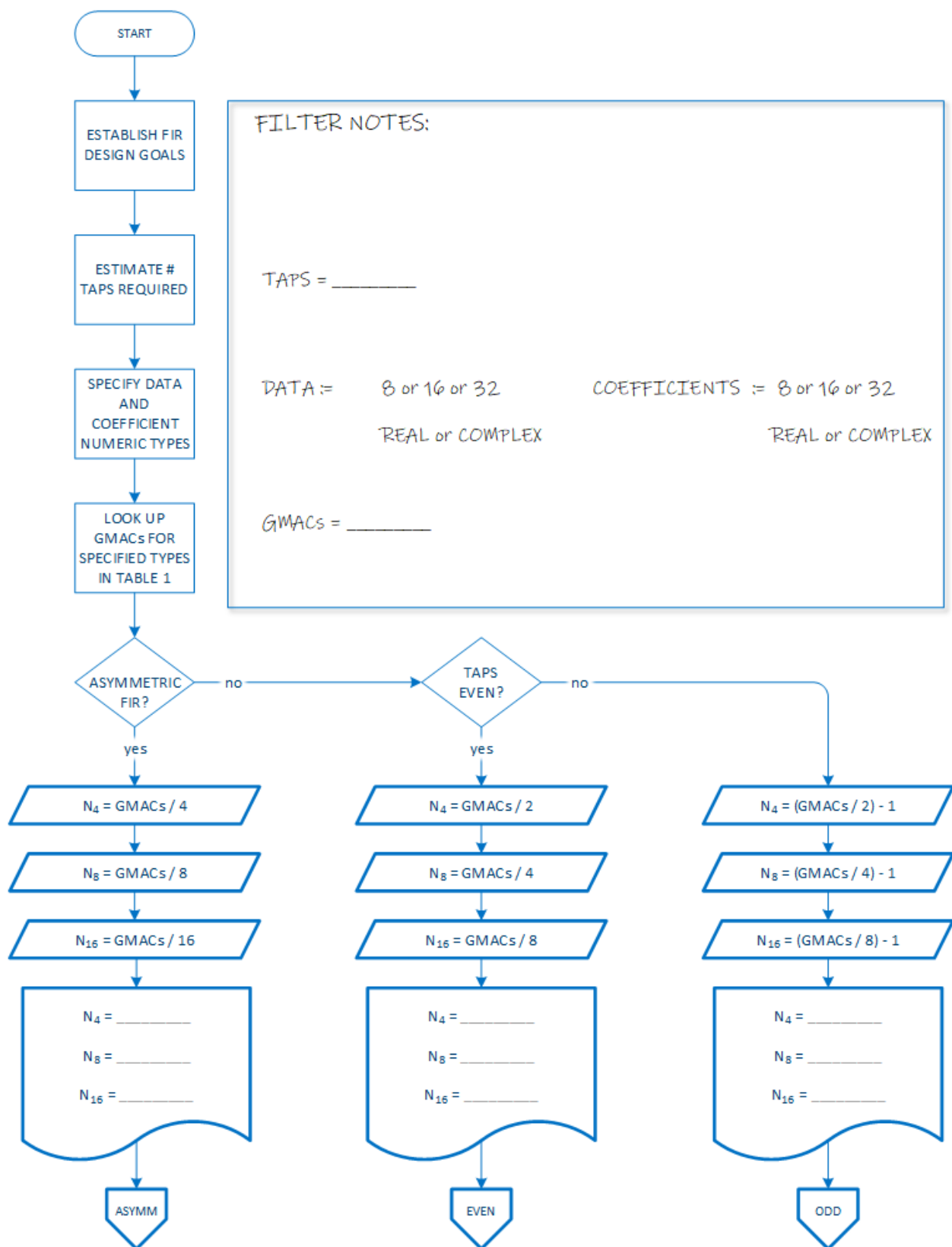
## Appendix D: Indexing for 8-Bit Real Data, 8-Bit Real Coefficients

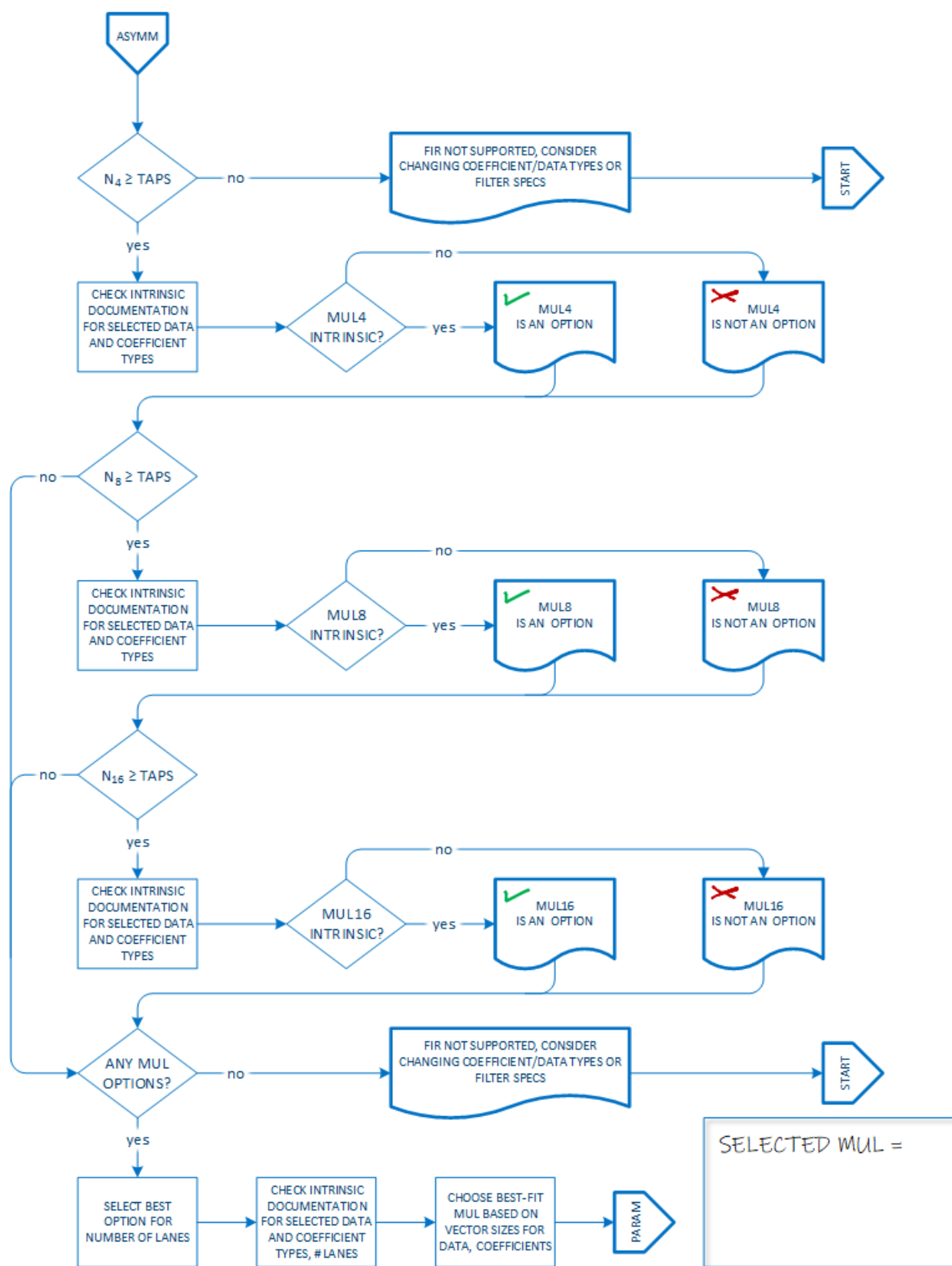


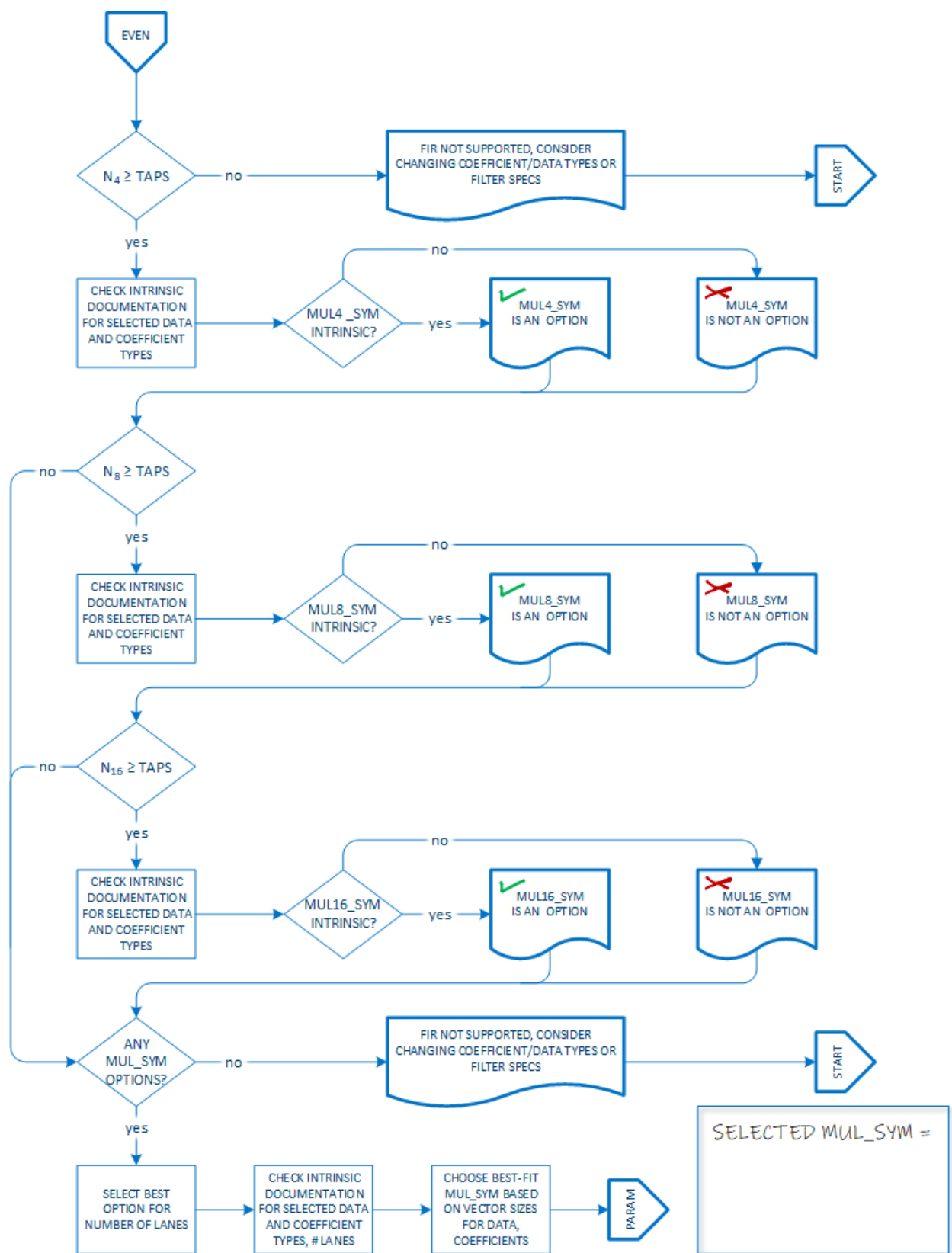
## Appendix E: Low-Order FIR Coding Flow

The flowcharts on the following pages are based on the design flow developed throughout this document.

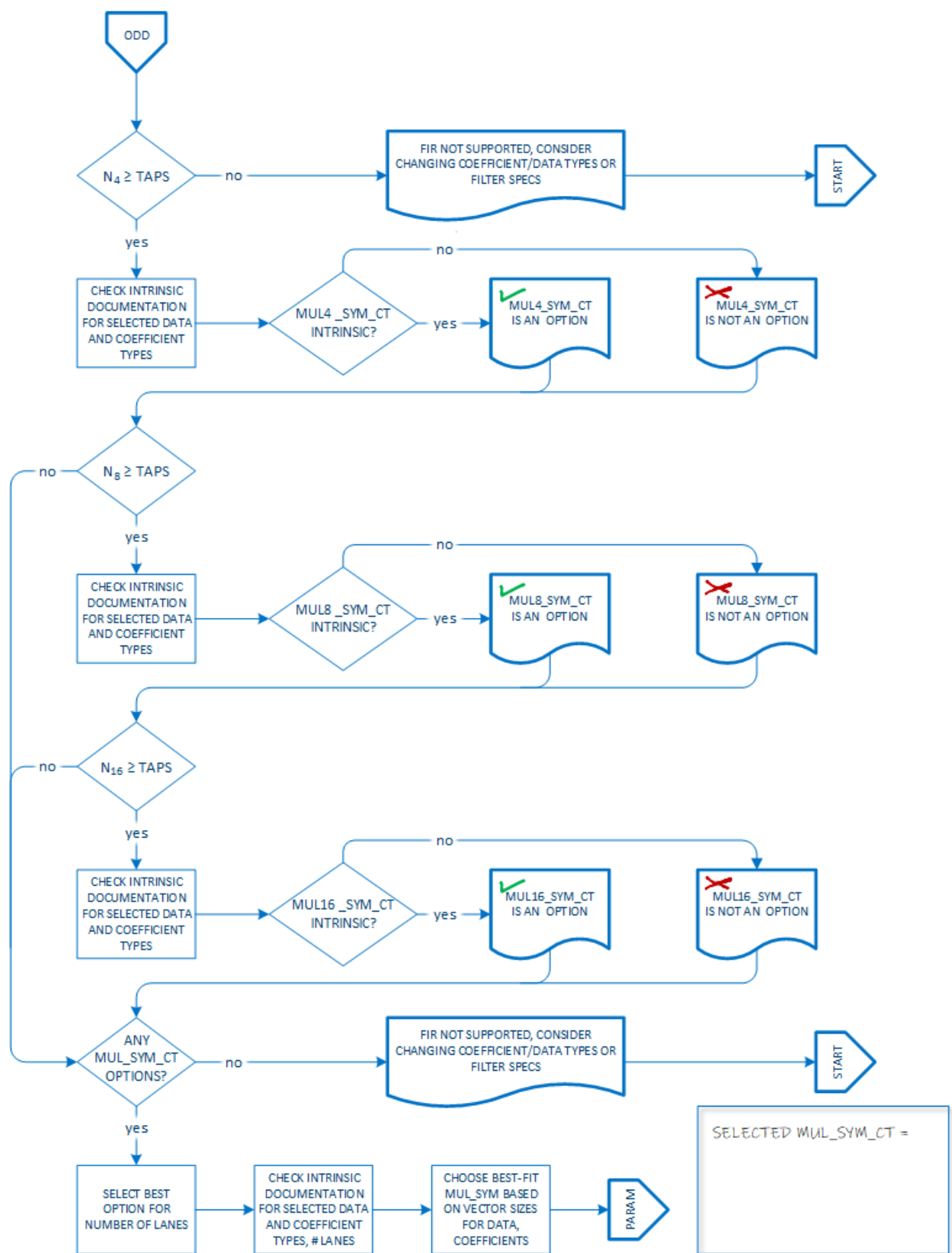
The intent is to provide a guideline for code development, and as such does not include details on each step. For details refer to the document.

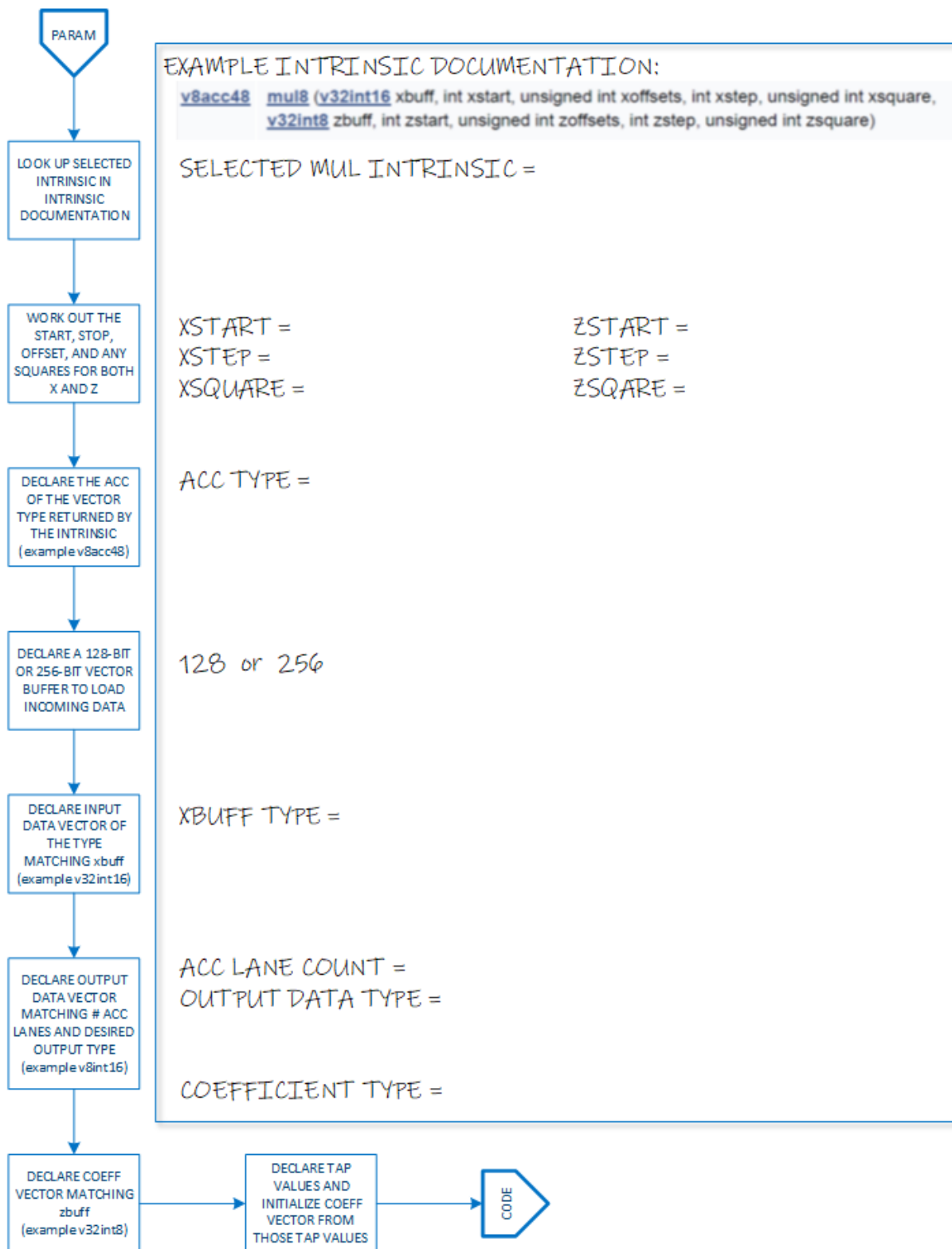






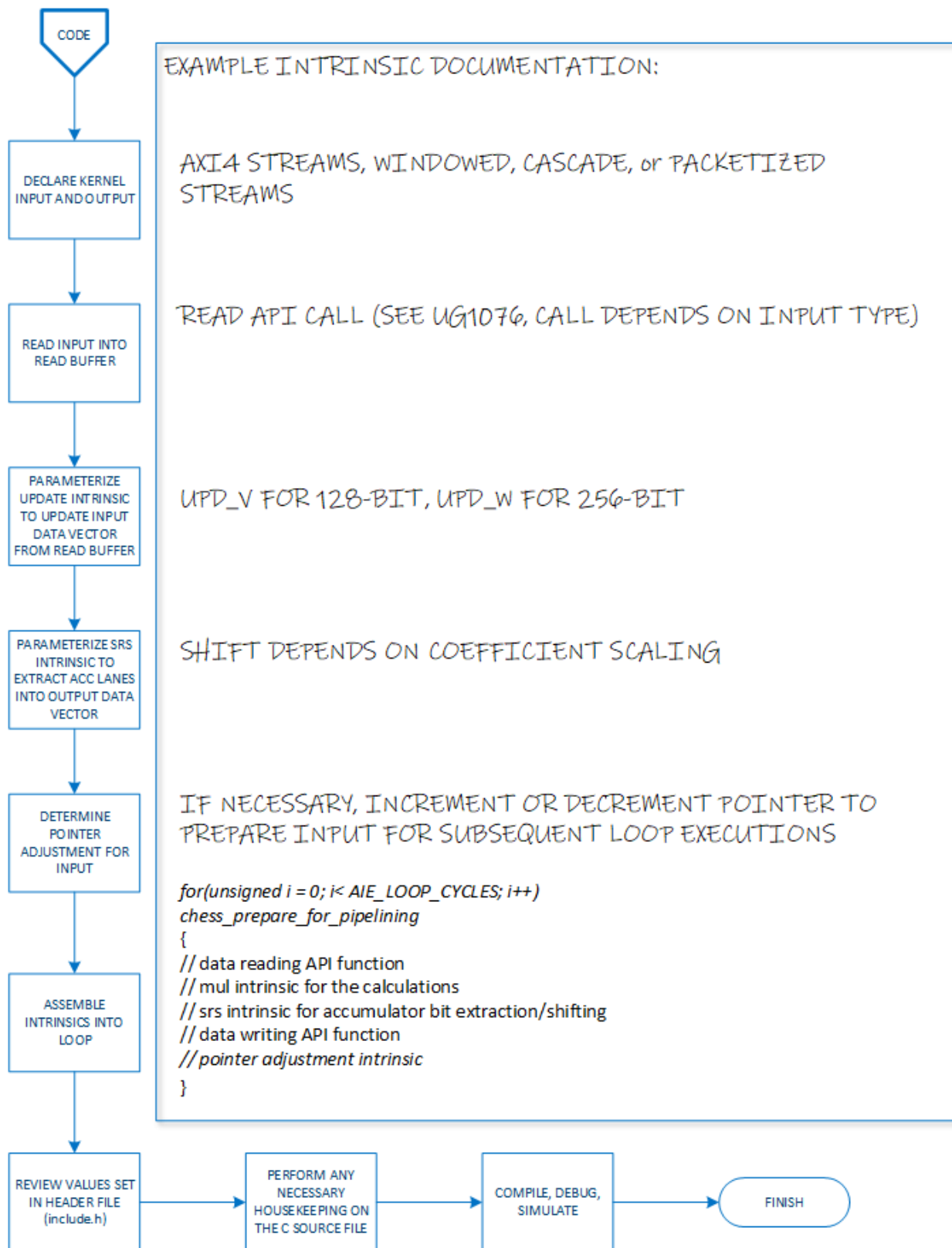






#### EXAMPLE PARAMETERIZED INTRINSIC:

```
acc = mul8(fir_data_in, 0, 0x03020100, 2, 0x2110, coeffs, 0, 0x00000000, 2, 0x1010);
```



## **Version History**

09/15/2020 = Initial draft

10/27/2020 = version 1.0, first publication, aligns with 2020.1 tools

12/03/2020 = first public publication, removed confidential markings, minor wording changes