AI Engine Kernel Coding

Best Practices Guide

UG1079 (v2020.2) February 4, 2021

XILINX®

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
02/04/2021 Version 2020.2	
Initial release.	N/A

Table of Contents

Overview

The Versal™ AI Core series delivers breakthrough artificial intelligence (AI) inference acceleration with AI Engines that deliver over 100x greater compute performance than current server-class of CPUs. This series is designed for a breadth of applications, including cloud for dynamic workloads and network for massive bandwidth, all while delivering advanced safety and security features. AI and data scientists, as well as software and hardware developers, can all take advantage of the high compute density to accelerate the performance of any application. Given the AI Engine's advanced signal processing compute capability, it is well-suited for highly optimized wireless applications such as radio, 5G, backhaul, and other high-performance DSP applications.

AI Engines are an array of very-long instruction word (VLIW) processors with single instruction multiple data (SIMD) vector units that are highly optimized for compute-intensive applications, specifically digital signal processing (DSP), 5G wireless applications, and AI technology such as machine learning (ML).

The AI Engine array supports three levels of parallelism:

- **Instruction Level Parallelism (ILP):** Through the VLIW architecture allowing multiple operations to be executed in a single clock cycle.
- **SIMD:** Through vector registers allowing multiple elements (for example, eight) to be computed in parallel.
- **Multicore:** Through the AI Engine array, allowing up to 400 AI Engines to execute in parallel.

Instruction-level parallelism includes a scalar operation, up to two moves, two vector reads (loads), one vector write (store), and one vector instruction that can be executed—in total, a 7-way VLIW instruction per clock cycle. Data-level parallelism is achieved via vector-level operations where multiple sets of data can be operated on a per-clock-cycle basis.

Each AI Engine contains both a vector and scalar processor, dedicated program memory, local 32 KB data memory, access to local memory in itself and three neighboring AI Engines with the direction depending on the row it is in. It also has access to DMA engines and AXI4 interconnect switches to communicate via streams to other AI Engines or to the programmable logic (PL) or the DMA. Refer to the *Versal ACAP AI Engine Architecture Manual* (AM009) for specific details on the AI Engine array and interfaces.

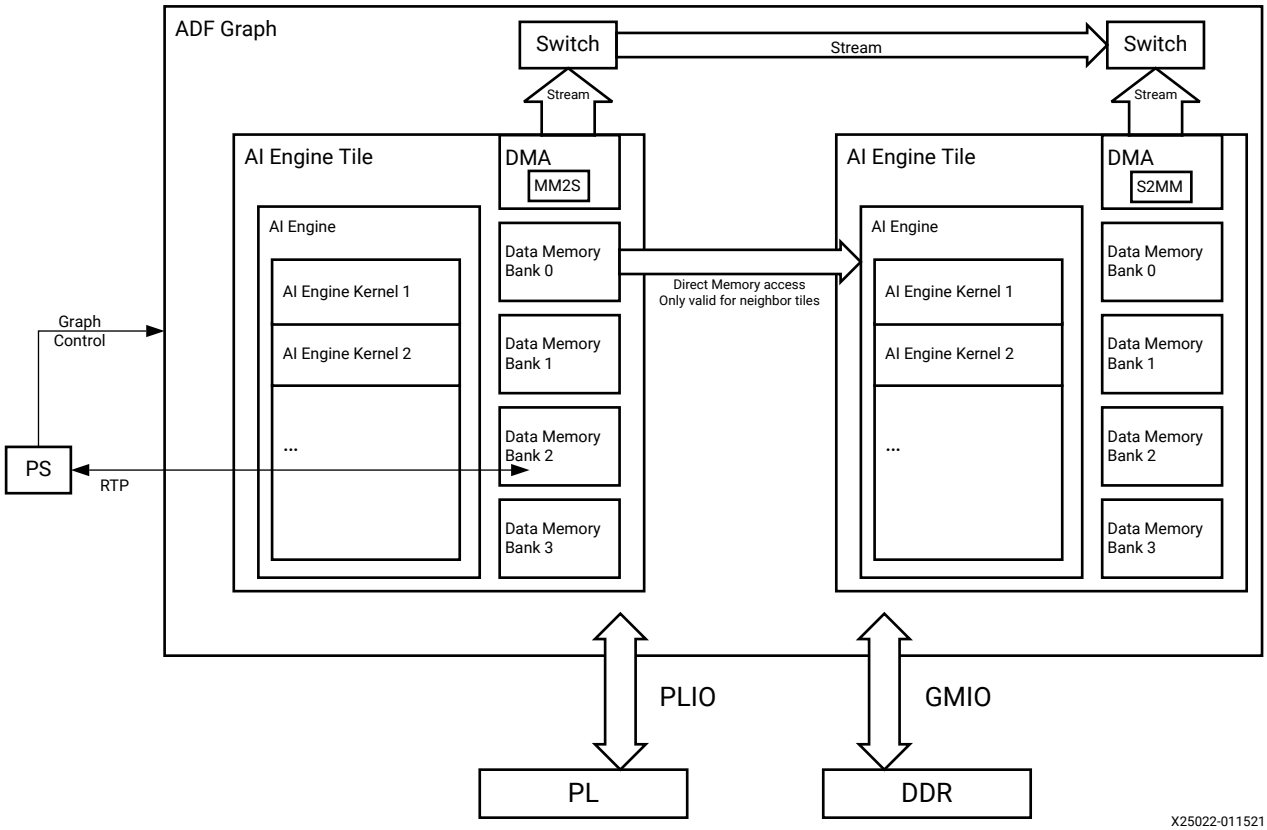
While most standard C code can be compiled for the AI Engine, the code might need restructuring to take full advantage of the parallelism provided by the hardware. The power of an AI Engine is in its ability to execute a multiply-accumulate (MAC) operation using two vectors, load two vectors for the next operation, store a vector from the previous operation, and increment a pointer or execute another scalar operation in each clock cycle. Specialized functions called intrinsics allow you to target the AI Engine vector and scalar processor and provides implementation of several common vector and scalar functions, so you can focus on the target algorithm. In addition to its vector unit, an AI Engine also includes a scalar unit which can be used for non-linear functions and data type conversions.

An AI Engine program consists of a data-flow graph (adaptable data flow graph) specification that is written in C++. This specification can be compiled and executed using the AI Engine compiler. An adaptive data flow (ADF) graph application consists of nodes and edges where nodes represent compute kernel functions, and edges represent data connections. Kernels in the application can be compiled to run on the AI Engines, and are the fundamental building blocks of an ADF graph specification. ADF graph is a Kahn process network with the AI Engine kernels operating in parallel. AI Engine kernels operate on data streams. These kernels consume input blocks of data and produce output blocks of data. Kernels can also have static data or run-time parameter (RTP) arguments that can be either asynchronous or synchronous.

The following figure shows the conceptual view of the ADF graph and its interfaces with the processing system (PS), programmable logic (PL), and DDR memory. It consists of the following.

- **AI Engine :** Each AI Engine is a VLIW processor containing a scalar unit, a vector unit, two load units, and a single store unit.
- **AI Engine Kernel:** Kernels are written in C/C++ running in an AI Engine.
- **ADF Graph:** ADF graph is a network with a single AI Engine kernel or multiple AI Engine kernels connected by data streams. It interacts with the PL, global memory, and PS with specific constructs like PLIO (port attribute in the graph programming that is used to make stream connections to or from the programmable logic), GMIO (port attribute in the graph programming that is used to make external memory-mapped connections to or from the global memory), and RTP.

Figure 1: **Conceptual Overview of the ADF Graph**



This document focuses on AI Engine kernel programming and covers some aspects beyond single kernel programming, like data communication between kernels, which are essential concepts for partitioning the application into multiple kernels to achieve overall system performance.

For additional details about constructing graph, compiling and simulating graph, and hardware flow, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Navigating Content by Design Process

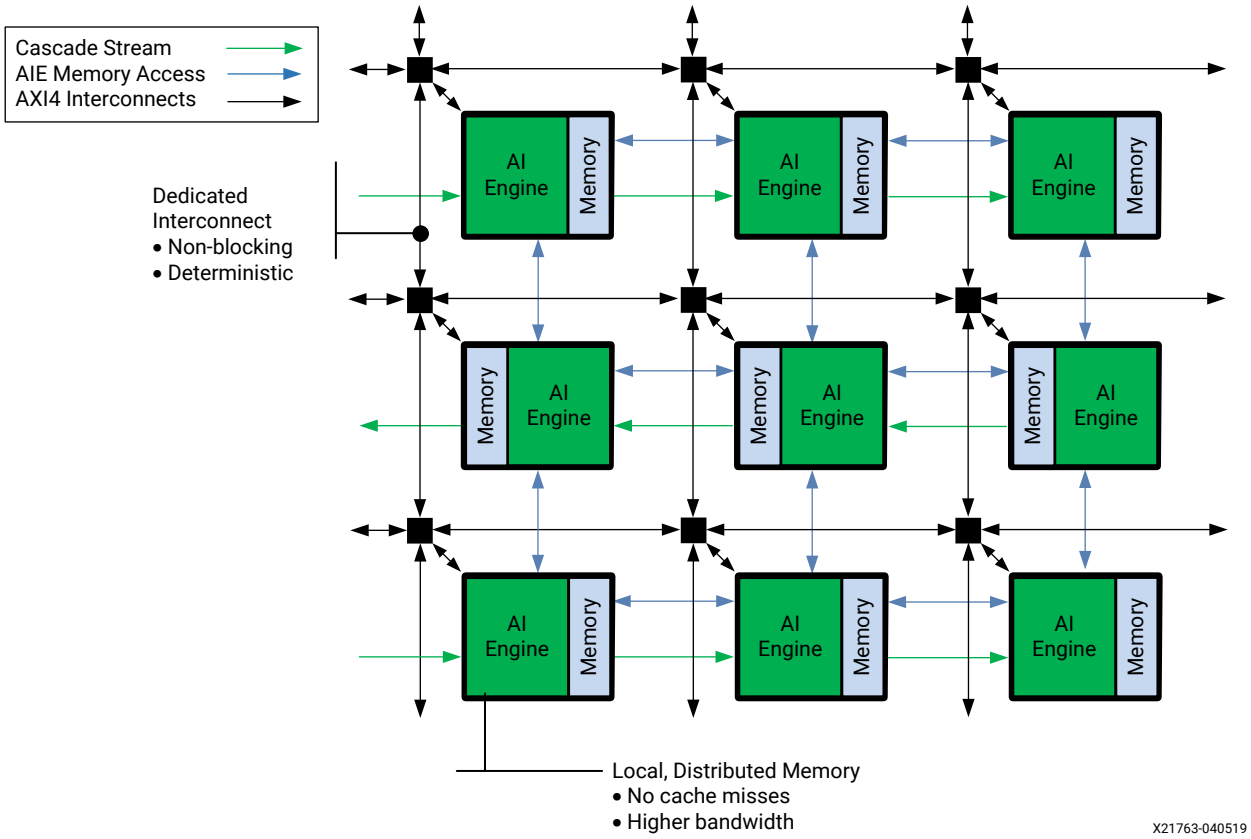
Xilinx® documentation is organized around a set of standard design processes to help you find relevant content for your current development task. This document covers the following design processes:

- **AI Engine Development:** Creating the AI Engine graph and kernels, library use, simulation debugging and profiling, and algorithm development. Also includes the integration of the PL and AI Engine kernels.

AI Engine Architecture Overview

The AI Engine array consists of a 2D array of AI Engine tiles, where each AI Engine tile contains an AI Engine, memory module, and tile interconnect module. An overview of such a 2D array of AI Engine tiles is shown in the following figure.

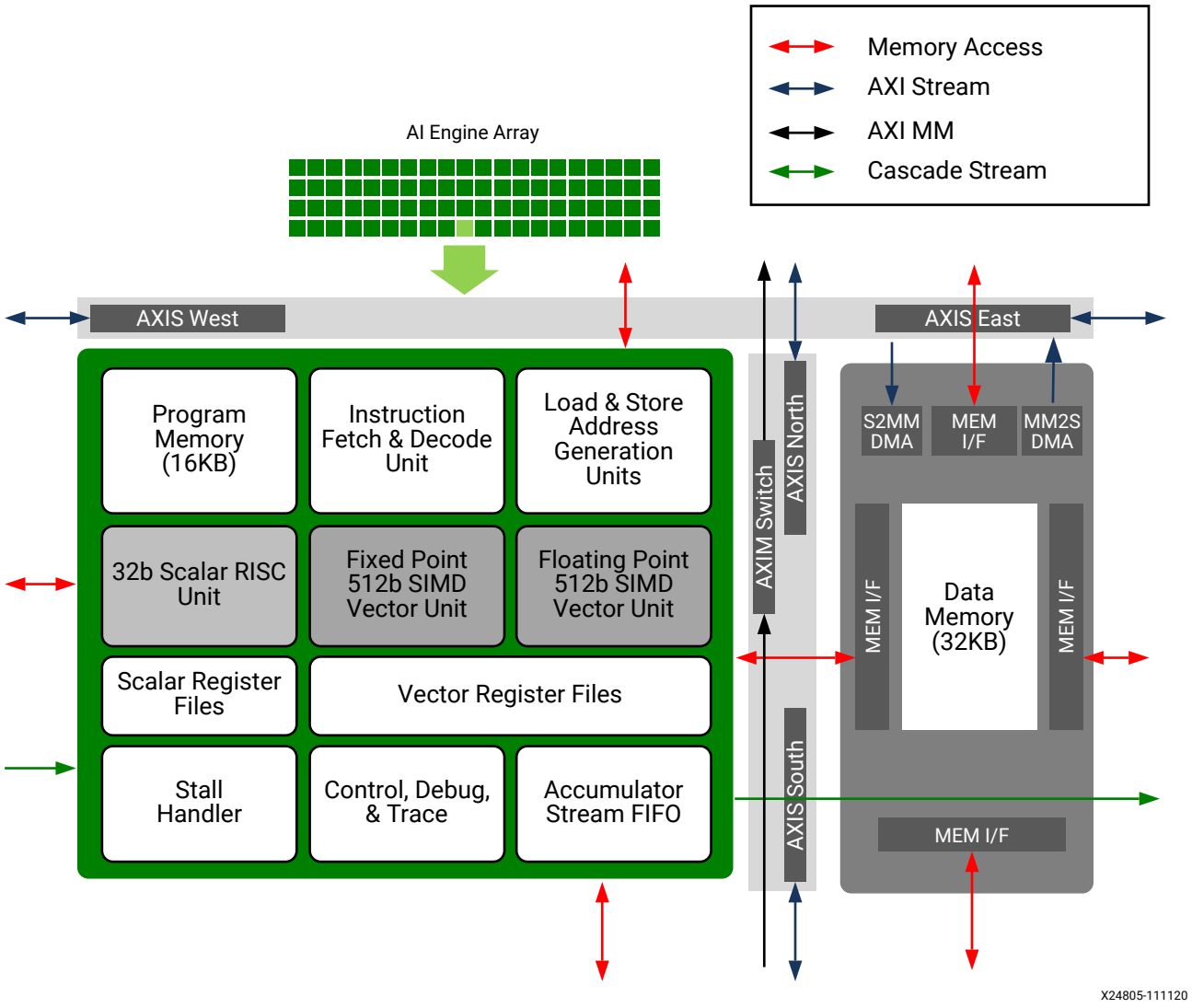
Figure 2: **AI Engine Array**



The memory module is shared between its north, south, east, or west AI Engine neighbors, depending on the location of the tile within the array. An AI Engine can access its north, south, east, or west, and its own memory module. Those neighboring memory modules are accessed by AI Engine through dedicated memory access interfaces, and each of the access can be at most 256-bit wide. AI Engine can also send or receive cascade streaming data from neighboring AI Engine. The cascade stream is one-way stream from left to right or right to left in a horizontal manner which wraps around when moving to the next row. The AXI4 interconnect module provides streaming connections between AI Engine tiles and provides stream to memory (s2mm) or memory to stream (mm2s) connections between streaming interfaces and the memory module. In addition, the interconnect modules are also connected to the neighboring interconnect module to provide flexible routing capability in a grid like fashion.

The following illustration is the architecture of a single AI Engine tile.

Figure 3: **AI Engine Tile Details**

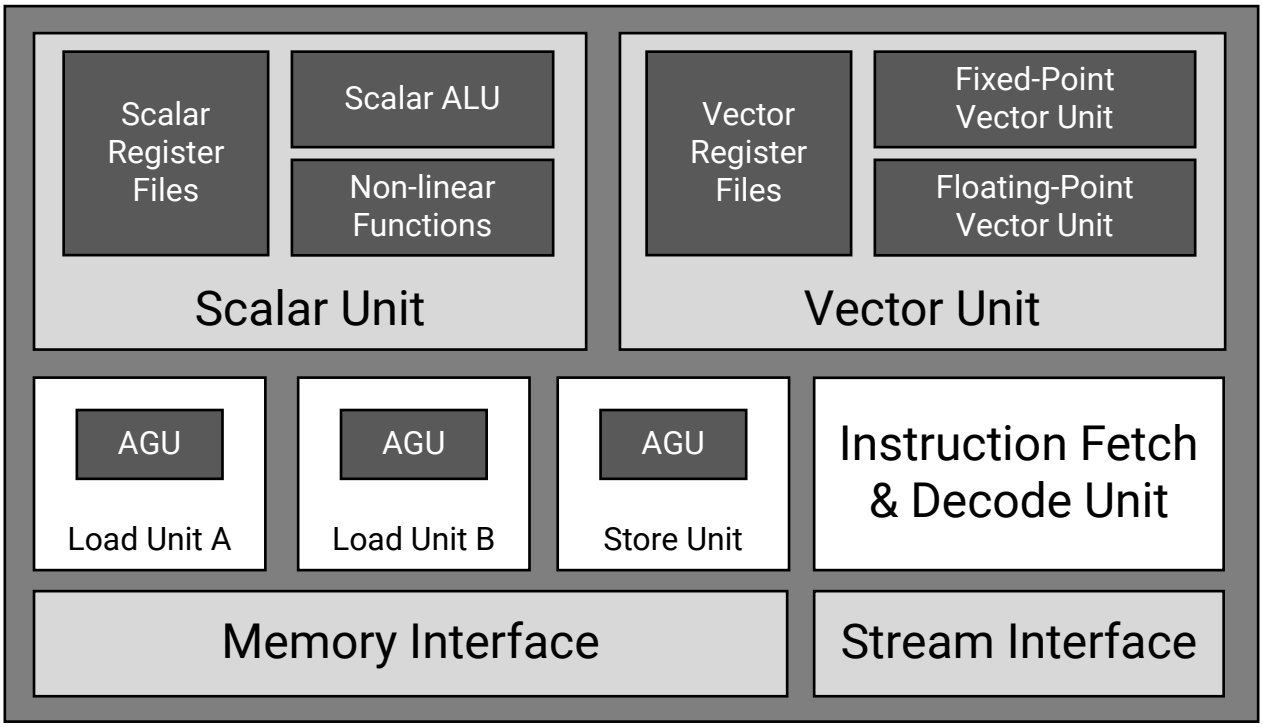


Each AI Engine tile has an AXI4-Stream switch that is a fully programmable 32-bit AXI4-Stream crossbar. It supports both circuit-switched and packet-switched streams with back-pressure. Through MM2S DMA and S2MM DMA, the AXI4-Stream switch provides stream access from and to AI Engine data memory. The switch also contains two 16-deep 33-bit (32-bit data + 1-bit TLAST) wide FIFOs, which can be chained to form a 32-deep FIFO by circuit-switching the output of one of the FIFOs to the other FIFO's input.

As shown in the following figure, the AI Engine is a highly-optimized processor featuring a single-instruction multiple-data (SIMD) and very long instruction word (VLIW) processor containing a scalar unit, a vector unit, two load units, a single store unit, and an instruction fetch and decode unit. One VLIW instruction can support a maximum of two loads, one store, one scalar operation, one fixed-point or floating-point vector operation, and two move instructions.

The AI Engine also has three address generator units (AGUs) to support multiple addressing modes. Two of the AGUs are dedicated for the two load units and one AGU is dedicated for the store unit.

Figure 4: **AI Engine**

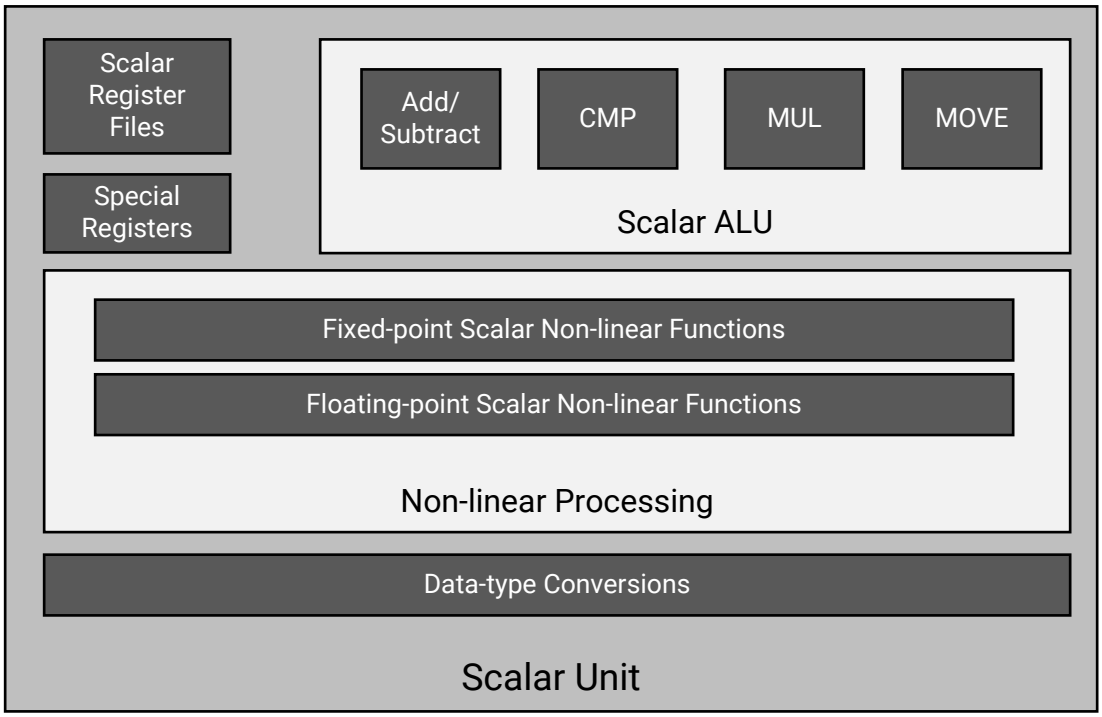


Additional details about vector processing unit, AI Engine memory, and AI Engine tile interface can be found in the following sections.

Scalar Processing Unit

The following figure shows the sub-components of the scalar unit. The scalar unit is used for program control (branch, comparison), scalar math operations, non-linear functions, and data type conversions much like a general-purpose processor. Similar to a general-purpose processor, generic C/C++ code can be used.

Figure 5: **Scalar Processing Unit**



The register files are used to store input and output. There are dedicated registers for pointer arithmetic, as well as for general-purpose usage and configuration. Special registers include stack pointers, circular buffers, and zero overhead loops. Two types of scalar elementary non-linear functions are supported in the AI Engine, fixed-point and floating-point precisions.

Fixed-point, non-linear functions include:

- Sine and cosine
- Absolute value (ABS)
- Count leading zeros (CLZ)
- Comparison to find minimum or maximum (lesser than (LG)/greater than (GT))
- Square root
- Inverse square root and inverse

Floating-point, non-linear functions include:

- Square root
- Inverse square root
- Inverse
- Absolute value (ABS)

- Comparison to find minimum or maximum (lesser than (LG)/greater than (GT))

The arithmetic logic unit (ALU) in the AI Engine manages the following operations with an issue rate of one instruction per cycle.

- Integer addition and subtraction of 32 bits. The operation has a one-cycle latency.
- Bit-wise logical operation on 32-bit integer numbers (BAND, BOR, and BXOR). The operation has a one-cycle latency.
- Integer multiplication: 32 x 32-bit with an output result of 32 bits stored in the R register file. The operation has a three-cycle latency.
- Shift operation: Both left and right shift are supported. The operation has a one-cycle latency.

Data type conversion can be done using `float2fix` and `fix2float`. This conversion can also support sqrt, inv, and inv_sqrt fixed-point operations.

Scalar Programming

The compiler and scalar unit provide the programmer the ability to use standard 'C' data types. The following table shows standard C data types with their precisions. All types except float and double support signed and unsigned prefixes.

Table 1: **Scalar data types**

Data Type	Precision	Comment
char	8-bit signed	
short	16-bit signed	
int	32-bit signed	Native support
long	64-bit signed	
float	32-bit	
double	64-bit	Emulated using softfloat library. Scalar proc does not contain FPU.

It is important to remember that control flow statements such as branching are still handled by the scalar unit even in the presence of vector instructions. This concept is critical to maximizing the performance of the AI Engine.

Vector Processing Unit

The vector unit contains a fixed-point unit with 128 8-bit fixed-point multipliers and a floating-point unit with eight single-precision floating-point multipliers. The vector registers and permute network are shared between the fixed-point and floating-point multipliers. The peak performance depends on the size of the data types used by the operands. The following table provides the number of MAC operations that can be performed by the vector processor per instruction.

Table 2: **AI Engine Vector Precision Support**

X Operand	Z Operand	Output	Number of MACs/Clock
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

The X operand is 1024 bits wide and the Z operand is 256 bits wide. In terms of component use, consider the first row in the previous table. The multiplier operands come from the same 1024-bit and 256-bit input registers but some values are broadcast to multiple multipliers. There are 128 8-bit single multipliers and results are post-added and accumulated into 16 or 8 accumulator lanes of 48 bits each.

To calculate the maximum performance for a given datapath, it is necessary to multiply the number of MACs per instruction with the clock frequency of the AI Engine kernel. For example, with 16-bit input vectors X and Z, the vector processor can achieve 32 MACs per instruction. Using the clock frequency for the slowest speed grade device results in:

32 MACs * 1 GHz clock frequency = 32 Giga MAC operations/second

AI Engine Memory

Each AI Engine has 16 KB of program memory, which allows storing 1024 instructions of 128-bit each. The AI Engine instructions are 128-bit (maximum) wide and support multiple instruction formats, as well as variable length instructions to reduce the program memory size. Many instructions outside of the optimized inner loop can use the shorter formats.

Each AI Engine tile has eight data memory banks, where each memory bank (single bank) is a 256 word x 128-bit single-port memory (for a total of 32 KB). Each AI Engine can access the memory from its north, south, and east or west neighboring tiles, for a total of 128 KB data memory, including its own data memory. The stack is a subset of the data memory. The default value for stack size and heap size is 1 KB. This value can be changed using compiler options.

In a logical representation, the 128 KB memory can be viewed as one contiguous 128 KB block or four 32 KB blocks, and each block can be divided into four odd and four even banks. One even bank and one odd bank are interleaved to comprise a double bank. AI Engines on the edges of the AI Engine array have fewer neighbors and correspondingly less memory available.

Each memory port operates in 256-bit/128-bit vector register mode or 32-bit/16-bit/8-bit scalar register mode. The 256-bit port is created by an even and odd pairing of the memory banks. The 8-bit and 16-bit stores are implemented as read-modify-write instructions. Concurrent operation of all three ports is supported if each port is accessing a different bank.

Data stored in memory is in little endian format.

RECOMMENDED: *It is recommended to access data memory on a 128-bit boundary with vector operations.*

Each AI Engine has a DMA controller that is divided into two separate modules, S2MM to store stream data to memory (32-bit data) and MM2S to write the contents of the memory to a stream (32-bit data). Both S2MM and MM2S have two independent data channels.

AI Engine Tile Interface

Data memory interfaces, stream interfaces, and cascade stream interfaces are the primary I/O interfaces that read and write data for compute to and from the AI Engine.

- The data memory interface sees one contiguous memory consisting of the data memory modules in all four directions with a total capacity of 128 KB. The AI Engine has two 256-bit wide load units and one 256-bit wide store unit. Each load or store can access the data in 128-bit or 256-bit width using 128-bit alignment.
- The AI Engine has two 32-bit input AXI4-Stream interfaces and two 32-bit output AXI4-Stream interfaces. Each stream is connected to a FIFO on both the input and output side, allowing the AI Engine to have a 128-bit access every four clock cycles or 32-bit wide access per cycle on a stream.
- The 384-bit accumulator data from one AI Engine can be forwarded to another by using the dedicated cascade stream interfaces to form a chain. There is a small, two deep, 384-bit wide FIFO on both the input and output streams that allows storing up to four values between AI Engines. In each cycle, 384-bits can be received and sent by the chained AI Engines. The cascade stream chain provides a relative tight coupling of multiple kernels operating at the same throughput.

When programming for the AI Engine, it is important to note that each AI Engine has the capability to access two 32-bit AXI4-Stream inputs, two 32-bit AXI4-Stream outputs, one 384-bit cascade stream input, one 384-bit cascade stream output, two 256-bit data loads, and one 256-bit data store. However, due to the length of the instruction, not all of these operations can be performed during the same cycle.

Tools

Vitis Integrated Design Environment

The Vitis™ integrated design environment (IDE) can be used to target system programming of Xilinx devices including, Versal devices. It supports development of single and multiple AI Engine kernel applications. The following features are available in the tool.

- An optimizing C/C++ compiler that compiles the kernels and graph code making all of the necessary connections, placements, and checks to ensure proper functioning on the device.
- A cycle accurate simulator, accelerated functional simulator, and profiling tools.
- A powerful debugging environment that works in both simulation and hardware environments. Various views, such as, variables view, disassembly view, memory view, register view, and pipeline view are available.

Vitis Command Line Tools

Command line tools are available to build, simulate, and generate output files and reports.

- The AI Engine compiler compiles kernels and graph code into ELF files that are run on the AI Engine processors.

- The AI Engine simulator and *x86simulator* are tools for cycle approximate simulation and functional simulation respectively.
- The cross compiler for Arm® Core is provided for PS code compilation.
- The Vitis compiler is the system compilation and linking tool for integrating whole system together.
- The Vitis Analyzer IDE is available for report viewing and analysis of the output files and reports generated by the command line tools.

The *Versal ACAP AI Engine Programming Environment User Guide* (UG1076) contains a wealth of information on the design flow and tools' usage.

Single Kernel Programming

An AI Engine kernel is a C/C++ program which is written using native C/C++ language and specialized intrinsic functions that target the VLIW scalar and vector processors. The AI Engine kernel code is compiled using the AI Engine compiler (aiecompiler) that is included in the Vitis™ core development kit. The AI Engine compiler compiles the kernels to produce ELF files that are run on the AI Engine processors.

For more information on intrinsic functions, see the *Versal ACAP AI Engine Intrinsics Documentation* (UG1078). AI Engine compiler and simulator are covered in the first few sections of this chapter.

AI Engine supports specialized data types and intrinsic functions for vector programming. By restructuring the scalar application code with these intrinsic functions and vector data types as needed, you can implement the vectorized application code. The AI Engine compiler takes care of mapping intrinsic functions to operations, vector or scalar register allocation and data movement, automatic scheduling, and generation of microcode that is efficiently packed in VLIW instructions.

The following sections introduce the data types supported and registers available for use by the AI Engine kernel. In addition, the vector intrinsic functions that initialize, load, and store, as well as operate on the vector registers using the appropriate data types are also described.

To achieve the highest performance on the AI Engine, the primary goal of single kernel programming is to ensure that the usage of the vector processor approaches its theoretical maximum. Vectorization of the algorithm is important, but managing the vector registers, memory access, and software pipelining are also required. The programmer must strive to make the data for the new operation available while the current operation is executing because the vector processor is capable of an operation every clock cycle. Optimizations using software pipelining in loops is available using pragmas. For instance, when the inner loop has sequential or loop carried dependencies it might be possible to unroll an outer loop and compute multiple values in parallel. The following sections go over these concepts as well.

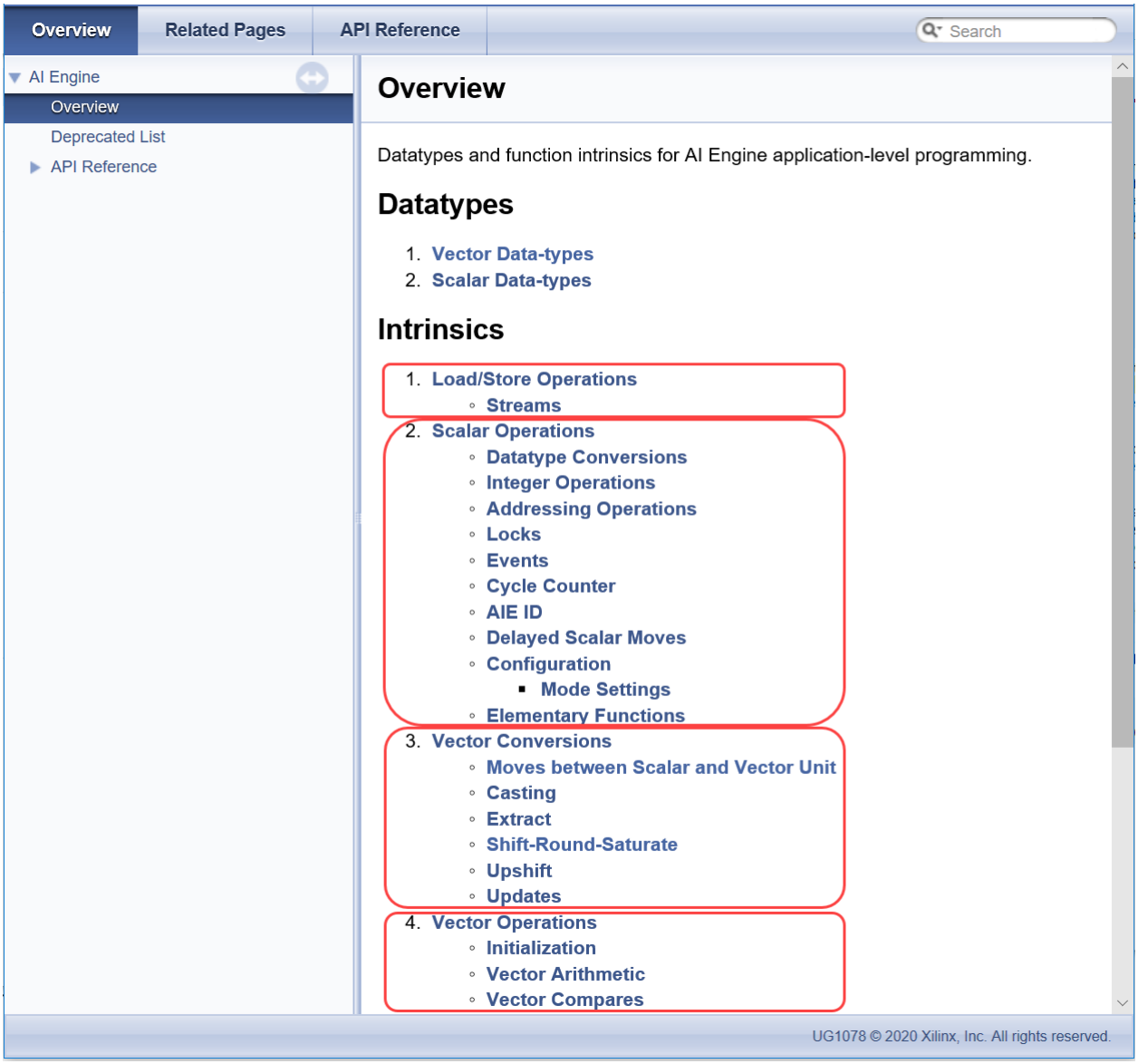
Intrinsics

For intrinsic function documentation, see the *Versal ACAP AI Engine Intrinsics Documentation* (UG1078).

The intrinsic function documentation is organized by the different types of operations and data types. The function calls at the highest level are organized in the documentation as follows:

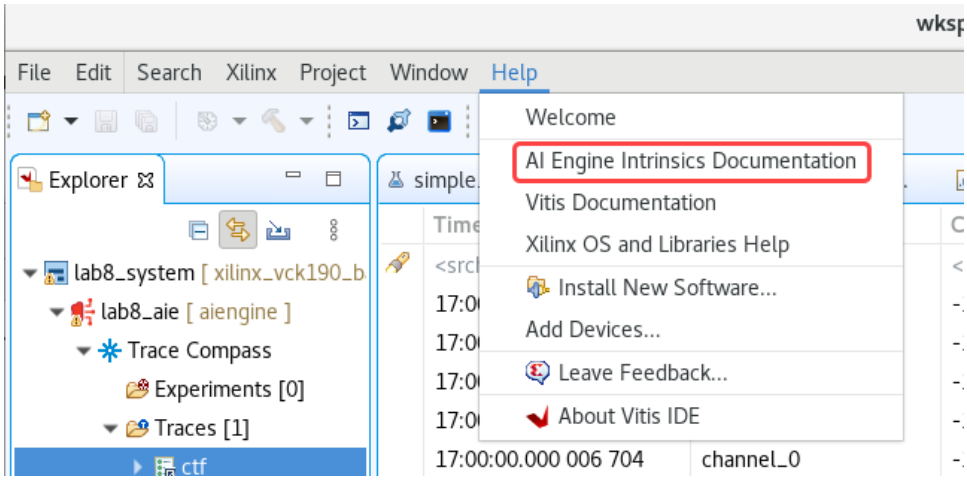
- **Load /Store Operations:** About pointer dereferencing and pointer arithmetic, as well as operations on streams.
- **Scalar Operations:** Operations on integer and floating point scalar values, configuration, conversions, addressing, locks and events.
- **Vector Conversions:** Handling of vectors with different sizes and precisions.
- **Vector Operations:** Arithmetic operations performed on vectors.
- **Application Specific Intrinsics:** Intrinsics assisting in the implementation of a specific application.

Figure 6: **Intrinsic Function Documentation**



It is also available through the Vitis™ IDE.

Figure 7: **Intrinsics Documentation through the Vitis IDE**



Kernel Pragmas

The AI Engine compiler supports dedicated directives for efficient loop scheduling. Additional pragmas for reducing memory dependencies and removing function hierarchy while further optimizing kernel performance is also included. Examples of the use of those pragmas can be found in this document.

The Chess Compiler User Manual has a list of all the pragmas and functions used in kernel coding. This manual can be found in the AI Engine lounge.

Kernel Compiler

The AI Engine compiler is used to compile AI Engine kernel code. Compiling an AI Engine Graph Application in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416) describes in detail the AI Engine compiler usage, options available, input files that can be passed in, and expected output.

Kernel Simulation

To simulate an AI Engine kernel you need to write an AI Engine graph application that consists of a data-flow graph specification which is written in C++. This graph will contain just the AI Engine kernel, with test bench data being provided as input(s) to the kernel. The data output(s) from the kernel can be captured as the simulation output and compared against golden data. This specification can be compiled and executed using the AI Engine compiler. The application can be simulated using the AI Engine simulator. For additional details on the simulator, see Simulating an AI Engine Graph Application in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Kernel Inputs and Outputs

AI Engine kernels operate on either streams or blocks of data. AI Engine kernels operate on data of specific types, for example, int32 and cint32. A block of data used by a kernel is called a window of data. Kernels consume input stream or window of data and produce output stream or window of data. Kernels access data streams in a sample-by-sample fashion. For additional details on the window and stream APIs, see Window and Streaming Data API in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

AI Engine kernels can also have RTP ports to be updated or read by PS. For more information about RTP, see Run-Time Graph Control API in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Introduction to Scalar and Vector Programming

This section provides an overview of the key elements of kernel programming for scalar and vector processing elements. The details of each element and optimization skills will be seen in following sections.

The following example uses only the scalar engine. It demonstrates a for loop iterating through 512 int32 elements. Each loop iteration performs a single multiply of int32 a and int32 b storing the result in c and writing it to an output window. The scalar_mul kernel operates on two input blocks (window) of data `input_window_int32` and produces an output window of data `output_window_int32`.

The APIs `window_readincr` and `window_writeincr` are used to read and write to the circular buffers outside the kernel. For additional details on the window APIs, see Window and Streaming Data API in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

```
void scalar_mul(input_window_int32* data1,
            input_window_int32* data2,
            output_window_int32* out){
    for(int i=0;i<512;i++)
    {
        int32 a=window_readincr(data1);
        int32 b=window_readincr(data2);
        int32 c=a*b;
        window_writeincr(out,c);
    }
}
```

The following example is a vectorized version for the same kernel.

```
void vect_mul(input_window_int32* __restrict data1,
            input_window_int32* __restrict data2,
            output_window_int32* __restrict out){
    for(int i=0;i<64;i++)
    chess_prepare_for_pipelining
    {
        v8int32 va=window_readincr_v8(data1);
        v8int32 vb=window_readincr_v8(data2);
        v8acc80 vt=mul(va,vb);
        v8int32 vc=srs(vt,0);

        window_writeincr(out,vc);
    }
}
```

Note the data types v8int32 and v8acc80 used in the previous kernel code. The window API `window_readincr_v8` returns a vector of 8 int32s and stores them in variables named `va` and `vb`. These two variables are vector type variables and they are passed to the intrinsic function `mul` which outputs `vt` which is a v8acc80 data type. The v8acc80 type is reduced by a *shift round saturate* function `srs` that allows a v8int32 type, variable *vc*, to be returned and then written to the output window. Additional details on the data types supported by the AI Engine are covered in the following sections.

The `__restrict` keyword used on the input and output parameters of the `vect_mul` function, allows for more aggressive compiler optimization by explicitly stating independence between data.

`chess_prepare_for_pipelining` is a compiler pragma that directs kernel compiler to achieve optimized pipeline for the loop.

The scalar version of this example function takes 1055 cycles while the vectorized version takes only 99 cycles. As you can see there is more than 10 times speedup for vectorized version of the kernel. Vector processing itself would give 8x the throughput for int32 multiplication but has a higher latency and would not get 8x the throughput overall. However, with the loop optimizations done, it can get close to 10x. The sections that follow describe in detail the various data types that can be used, registers available, and also the kinds of optimizations that can be achieved on the AI Engine using concepts like software pipelining in loops and keywords like `__restrict`.

Related Information

Software Pipelining of Loops
Restrict Keyword

AI Engine Data Types

The AI Engine scalar unit supports signed and unsigned integers in 8, 16, and 32-bit widths, along with some single-precision floating-point for specific operations.

The AI Engine vector unit supports integers and complex integers in 8, 16, and 32-bit widths, along with real and complex single-precision floating-point numbers. It also supports accumulator vector data types, with 48 and 80-bit wide elements. Intrinsic functions such as absolute value, addition, subtraction, comparison, multiplication, and MAC operate using these vector data types. Vector data types are named using a convention that includes the number of elements, real or complex, vector type or accumulator type, and bit width as follows:

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

Optional specifications include:

- **`NumLanes`:** Denotes the number of elements in the vector which can be 2, 4, 8, 16, 32, 64, or 128.
- **`c`:** Denotes complex data with real and imaginary parts packed together.
- **`int`:** denotes integer vector data values.
- **`float`:** Denotes single precision floating point values.

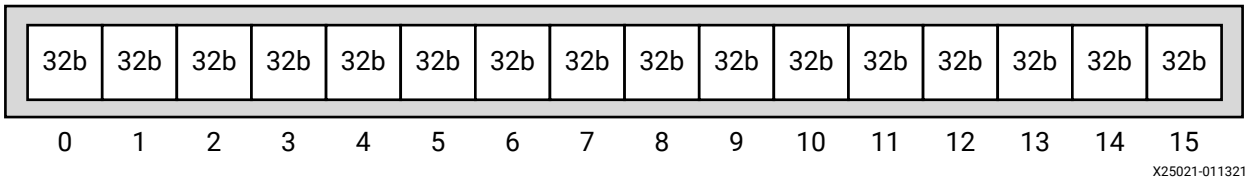
  Note: There are no accumulator registers for floating-point vectors.
- **`acc`:** Denotes accumulator vector data values.
- **`u`:** Denotes unsigned. Unsigned only exists for int8 vectors.
- **`SizeofElement`:** Denotes the size of the vector data type element.

- 1024-bit integer vector types are vectors of 8-bit, 16-bit, or 32-bit vector elements. These vectors have 16, 32, 64, or 128 lanes.
- 512-bit integer vector types are vectors of 8-bit, 16-bit, 32-bit, or 64-bit vector elements. These vectors have 4, 8, 16, 32, or 64 lanes.
- 256-bit integer vector types are vectors of 8-bit, 16-bit, 32-bit, 64-bit, or 128-bit vector elements. These vectors have 1, 2, 4, 8, 16, or 32 lanes.
- 128-bit integer vector types are vectors of 8-bit, 16-bit, or 32-bit vector elements. These vectors have 2, 4, 8, or 16 lanes.
- Accumulator data types are vectors of 80-bit or 48-bit elements These vectors have 2, 4, 8, or 16 lanes.

The total data-width of the vector data-types can be 128-bit, 256-bit, 512-bit, or 1024-bit. The total data-width of the accumulator data-types can be 320/384-bit or 640/768-bit.

For example, v16int32 is a sixteen element vector of integers with 32 bits. Each element of the vector is referred to as a *lane*. Using the smallest bit width necessary can improve performance by making good use of registers.

Figure 8: **v16int32**



Vector Registers

All vector intrinsic functions require the operands to be present in the AI Engine vector registers. The following table shows the set of vector registers and how smaller registers are combined to form large registers.

Table 3: **Vector Registers**

128-bit	256-bit	512-bit	1024-bit	
vrl0	wr0	xa	ya	N/A
vrh0				
vrl1	wr1			
vrh1				
vrl2	wr2	xb		yd (msbs)
vrh2				
vrl3	wr3			
vrh3				
vcl0	wc0	xc	N/A	N/A
vch0				
vcl1	wc1			
vch1				
vdl0	wd0	xd	N/A	yd (lsbs)
vdh0				
vdl1	wd1			
vdh1				

The underlying basic hardware registers are 128-bit wide and prefixed with the letter v. Two v registers can be grouped to form a 256-bit register prefixed with w. wr, wc, and wd registers are grouped in pairs to form 512-bit registers (xa, xb, xc, and xd). xa and xb form the 1024-bit wide ya register, while xd and xb form the 1024-bit wide yd register. This means the xb register is shared between ya and yd registers. xb contains the most significant bits (MSBs) for both ya and yd registers.

The vector register name can be used with the `chess_storage` directive to force vector data to be stored in a particular vector register. For example:

```
v8int32 chess_storage(wr0) bufA;
v8int32 chess_storage(WR) bufB;
```

When upper case is used in the `chess_storage` directive, it means register files (for example, any of the four wr registers), whereas lower case in the directive means just a particular register (for example, wr0 in the previous code example) will be chosen.

Vector registers are a valuable resource. If the compiler runs out of available vector registers during code generation, then it generates code to spill the register contents into local memory and read the contents back when needed. This consumes extra clock cycles.

The name of the vector register used by the kernel during its execution is shown for vector load/store and other vector-based instructions in the kernel microcode. This microcode is available in the disassembly view in Vitis IDE. For additional details on Vitis IDE usage, see Using Vitis IDE and Reports.

Many intrinsic functions only accept specific vector data types but sometimes not all values from the vector are required. For example, certain intrinsic functions only accept 512-bit vectors. If the kernel code has smaller sized data, one technique that can help is to use the `concat()` intrinsic to concatenate this smaller sized data with an undefined vector (a vector with its type defined, but not initialized).

For example, the `lmul8` intrinsic only accepts a `v16int32` or `v32int32` vector for its `xbuff` parameter. The intrinsic prototype is:

```
v8acc80 lmul8    (    v16int32     xbuff,
    int     xstart,
    unsigned int     xoffsets,
    v8int32     zbuff,
    int     zstart,
    unsigned int     zoffsets
)
```

The `xbuff` parameter expects a 16 element vector (v16int32). In the following example, there is an eight element vector (`v8int32`) rva. The `concat()` intrinsic is used to upgrade it to a 16 element vector. After concatenation, the lower half of the 16 element vector has the contents of rva. The upper half of the 16 element vector is uninitialized due to concatenation with the undefined `v8int32` vector.

```
int32 a[8] = {1, 2, 3, 4, 5, 6, 7, 8};
v8int32 rva = *((v8int32*)a);
acc = lmul8(concat(rva,undef_v8int32()),0,0x76543210,rvb,0,0x76543210);
```

For more information about how vector-based intrinsic functions work, refer to Vector Register Lane Permutations.

Accumulator Registers

The accumulation registers are 384 bits wide and can be viewed as eight vector lanes of 48 bits each. The idea is to have 32-bit multiplication results and accumulate over those results without bit overflows. The 16 guard bits allow up to 2^{16} accumulations. The output of fixed-point vector MAC and MUL intrinsic functions is stored in the accumulator registers. The following table shows the set of accumulator registers and how smaller registers are combined to form large registers.

Table 4: **Accumulator Registers**

384-bit	768-bit
aml0	bm0
amh0	

Table 4: **Accumulator Registers** *(cont'd)*

384-bit	768-bit
aml1	bm1
amh1	
aml2	bm2
amh2	
aml3	bm3
amh3	

The accumulator registers are prefixed with the letters 'am'. Two of them are aliased to form a 768-bit register that is prefixed with 'bm'.

The shift-round-saturate `srs()` intrinsic is used to move a value from an accumulator register to a vector register with any required shifting and rounding.

```
v8int32 res = srs(acc, 8); // shift right 8 bits, from accumulator register
to vector register
```

The upshift `ups()` intrinsic is used to move a value from an vector register to an accumulator register with upshifting:

```
v8acc48 acc = ups(v, 8); //shift left 8 bits, from vector register to
accumulator register
```

The `set_rnd()` and `set_sat()` instrinsics are used to set the rounding and saturation mode of the accumulation result, while `clr_rnd()` and `clr_sat()` intrinsics are used to clear the rounding and saturation mode, that is to truncate the accumulation result.

Note that only when operations are going through the shift-round-saturate data path, the shifting, rounding, or saturation mode will be effective. Some intrinsics only use the vector pre-adder operations, where there will be no shifting, rounding, or saturation mode for configuration. Such operations are adds, subs, abs, vector compares, or vector selections/shuffles. It is possible to choose MAC intrinsics instead to do subtraction with shifting, rounding, or saturation mode configuration. The following code performs subtraction between `va` and `vb` with `mul` instead of `sub` intrinsics.

```
v16cint16 va, vb;
int32 zbuff[8]={1,1,1,1,1,1,1,1};
v8int32 coeff=*(v8int32*)zbuff;
v8acc48 acc = mul8_antisym(va, 0, 0x76543210, vb, 0, false, coeff, 0 ,
0x76543210);
v8int32 res = srs(acc,0);
```

Floating-point intrinsic functions do not have separate accumulation registers and instead return their results in a vector register.

Casting and Datatype Conversion

Casting intrinsic functions (`as_[Type]()`) allow casting between vector types or scalar types of the same size. The casting can work on accumulator vector types too. Generally, using the smallest data type possible will reduce register spillage and improve performance. For example, if a 48-bit accumulator (acc48) can meet the design requirements then it is preferable to use that instead of a larger 80-bit accumulator (acc80).

***Note*:** The `acc80` vector data type occupies two neighboring 48-bit lanes.

Standard C casts can be also used and works identically in almost all cases as shown in the following example.

```
v8int16 iv;
v4cint16 cv=as_v4cint16(iv);
v4cint16 cv2=*(v4cint16*)&iv;
v8acc80 cas_iv;
v8cacc48 cas_cv=as_v8cacc48(cas_iv);
```

There is hardware support built-in for floating-point to fixed-point (`float2fix()`) and fixed-point to floating-point (`fix2float()`) conversions. For example, the fixed-point square root, inverse square root, and inverse are implemented with floating point precision and the `fix2float()` and `float2fix()` conversions are used before and after the function. The scalar engine is used in this example because the square root and inverse functions are not vectorizable. This can be verified by looking at the function prototype's input data types:

```
float _sqrtf(float a) //scalar float operation
int sqrt(int a,...) //scalar integer operation
```

Note that the input data types are scalar types (int) and not vector types (vint).

The conversion functions (`fix2float`, `float2fix`) can be handled by either the vector or scalar engines depending on the function called. Note the difference in data return type and data argument types:

```
float fix2float(int n,...) //Uses the scalar engine
v8float fix2float(v8int32 ivec,...) //Uses the vector engine
```

***Note*:** For `float2fix`, there are two types of implementations, `float2fix_safe` (default) and `float2fix_fast` with the `float2fix_safe` implementation offering a more strict data type check. You can define the macro `FLOAT2FIX_FAST` to make `float2fix` choose the `float2fix_fast` implementation.

Vector Initialization, Load, and Store

Vector registers can be initialized, loaded, and saved in a variety of ways. For optimal performance, it is critical that the local memory that is used to load or save the vector registers be aligned on 16-byte boundaries.

Alignment

The `alignas` standard C specifier can be used to ensure proper alignment of local memory. In the following example, the `reals` is aligned to 16 byte boundary.

```
alignas(16) const int32 reals[8] =
       {32767, 23170, 0, -23170, -32768, -23170, 0, 23170};
       //align to 16 bytes boundary, equivalent to "alignas(v4int32)"
```

Initialization

The following functions can be used to initialize vector registers as undefined, all 0's, with data from local memory, or with part of the values set from another register and the remaining part are undefined. Initialization using the `undef_type()` initializer ensures that the compiler can optimize regardless of the undefined parts of the value.

```
v8int32 v;
v8int32 uv = undef_v8int32(); //undefined
v8int32 nv = null_v8int32(); //all 0's
v8int32 iv = *(v8int32 *) reals; //re-interpret "reals" as "v8int32"
pointer and load value from it
v16int32 sv = xset_w(0, iv); //create a new 512-bit vector with lower 256-
bit set with "iv"
```

In the previous example, vector set intrinsic functions `[T]set_[R]` allow creating a vector where only one part is initialized and the other parts are undefined. Here [T] indicates the target vector register to be set, w for W register (256-bit), x for X register (512-bit), and y for Y register (1024-bit). [R] indicates where the source value comes from, v for V register (128-bit), w for W register (256-bit), and x for X register (512-bit). Note that [R] width is smaller than [T] width. The valid vector set intrinsic functions are, `wset_v`, `xset_v`, `xset_w`, `yset_v`, `yset_w`, and `yset_x`.

Load and Store

Load and Store From Vector Registers

The compiler supports standard pointer de-referencing and pointer arithmetic for vectors. Post increment of the pointer is the most efficient form for scheduling. No special intrinsic functions are needed to load vector registers.

```
v8int32 * ptr_coeff_buffer = (v8int32 *)ptr_kernel_coeff;
v8int32 kernel_vec0 = *ptr_coeff_buffer++; // 1st 8 values (0 .. 7)
v8int32 kernel_vec1 = *ptr_coeff_buffer;   // 2nd 8 values (8 .. 15)
```

Load and Store From Memory

AI Engine APIs provide access methods to read and write data from data memory, streaming data ports, and cascade streaming ports which can be used by AI Engine kernels. For additional details on the window and stream APIs, see Window and Streaming Data API in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416). In the following example, the window `readincr` (`window_readincr_v8(din)`) API is used to read a window of complex int16 data into the data vector. Similarly, `readincr_v8(cin)` is used to read a sample of int16 data from the `cin` stream. `writeincr_v4 (cas_out, v)` is used to write data to a cascade stream output.

```
void func(input_window_cint16 *din,
          input_stream_int16 *cin,
          output_stream_cacc48 *cas_out){
    v8cint16 data=window_readincr_v8(din);
    v8int16 coef=readincr_v8(cin);
    v4cacc48 v;
    …
    writeincr_v4(cas_out, v);
}
```

Load and Store Using Pointers

It is mandatory to use the window API in the kernel function prototype as inputs and outputs. However, in the kernel code, it is possible to use a direct pointer reference to read/write data.

```
void func(input_window_int16 *w_input,
          output_window_cint16 *w_output){
    .....
    v16int16 *ptr_in  = (v16int16 *)w_input->ptr;
    v8cint16 *ptr_out = (v8cint16 *)w_output->ptr;
    ......
}
```

The window structure is responsible for managing buffer locks tracking buffer type (ping/pong) and this can add to the cycle count. This is especially true when load/store are out-of-order (scatter-gather). Using pointers may help reduce the cycle count required for load and store.

***Note*:** If using pointers to load and store data, it is the designer's responsibility to avoid out-of-bound memory access.

Load and Store Using Streams

Vector data can also be loaded from or stored in streams as shown in the following example.

```
void func(input_stream_int32 *s0, input_stream_int32 *s1, …){
    for(…){
        data0=readincr(s0);
        data1=readincr(s1);
        …
    }
}
```

For more information about window and streaming data API usage, see Window and Streaming Data API in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Update, Extract, and Shift

To update portions of vector registers, the `upd_v()`, `upd_w()`, and `upd_x()` intrinsic functions are provided for 128-bit (v), 256-bit (w), and 512-bit (x) updates.

***Note*:** The updates overwrite a portion of the larger vector with the new data while keeping the other part of the vector alive. This alive state of the larger vector persists through multiple updates. If too many vectors are kept unnecessarily alive, register spillage can occur and impact performance.

Similarly, `ext_v()`, `ext_w()`, and `ext_x()` intrinsic functions are provided to extract portions of the vector.

To update or extract individual elements, the `upd_elem()` and `ext_elem()` intrinsic functions are provided. These must be used when loading or storing values that are not in contiguous memory locations and require multiple clock cycles to load or store a vector. In the following example, the 0th element of vector `v1` is updated with the value of `a` - which is 100.

```
int a = 100;
v4int32 v1 = upd_elem(undef_v4int32(), 0, a);
```

Another important use is to move data to the scalar unit and do an inverse or sqrt. In the following example, the 0th element of vector `vf` is extracted and stored in the scalar variable `f`.

```
v4float vf;
float f=ext_elem(vf,0);
float i_f=invsqrt(f);
```

The `shft_elem()` intrinsic function can be used to update a vector by inserting a new element at the beginning of a vector and shifting the other elements by one.

Vector Register Lane Permutations

The AI Engine fixed point vector units datapath consists of the following three separate and largely independently usable paths:

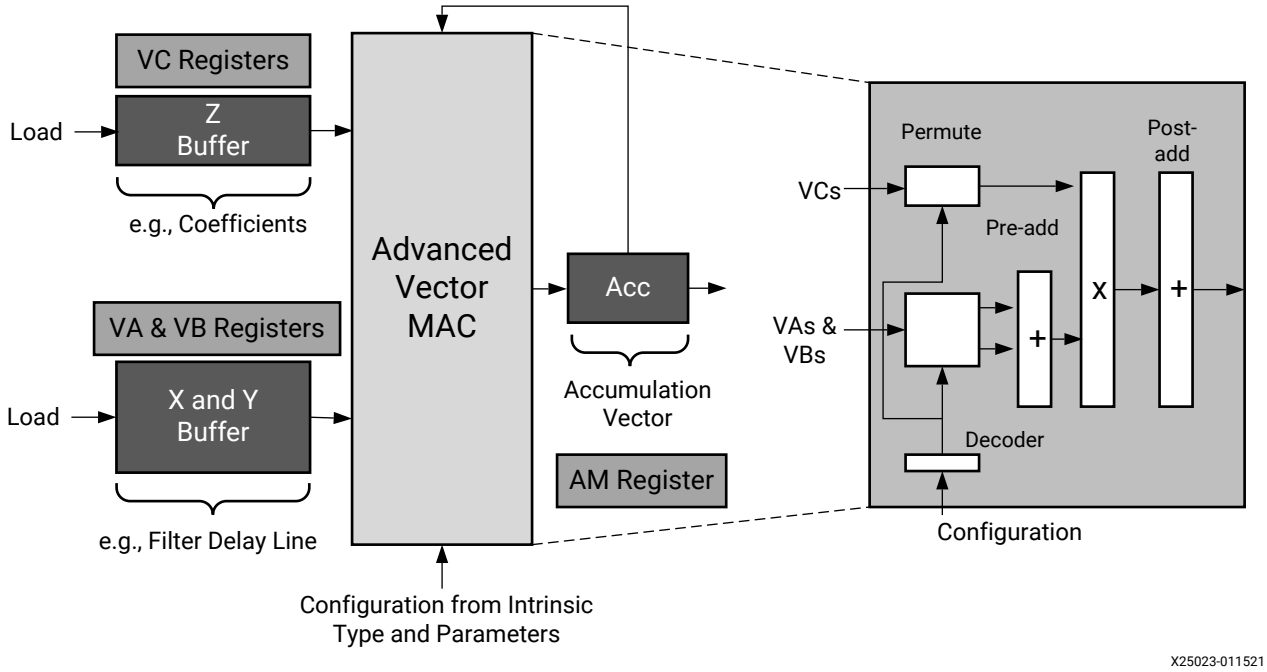
- Main MAC datapath
- Shift-round-saturate path
- Upshift path

The main multiplication path reads values from vector registers, permutes them in a user controllable fashion, performs optional pre-adding, multiplies them, and after some post-adding accumulates them to the previous value of the accumulator register.

While the main datapath stores to the accumulator, the shift-round-saturate path reads from the accumulator registers and stores to the vector registers or the data memory. In parallel to the main datapath runs the upshift path. It does not perform any multiplications but simply reads vectors, upshifts them and feeds the result into the accumulators. For details on the Fixed point and Floating point data paths refer to *Versal ACAP AI Engine Architecture Manual* (AM009). Details on the intrinsic functions that can be used to exercise these data paths can be found in the *Versal ACAP AI Engine Intrinsics Documentation* (UG1078).

As shown in the following figure, the basic functionality of MAC data path consists of vector multiply and accumulate operations between data from the X and Z buffers. Other parameters and options allow flexible data selection within the vectors and number of output lanes and optional features allow different input data sizes and pre-adding. There is an additional input buffer, the Y buffer, whose values can be pre-added with those from the X buffer before the multiplication occurs. The result from the intrinsic is added to an accumulator.

Figure 9: **Functional Overview of the MAC Data Path**



The operation can be described using *lanes* and *columns*. The number of lanes corresponds to the number of output values that will be generated from the intrinsic call. The number of columns is the number of multiplications that will be performed per output lane, with each of the multiplication results being added together. For example:

```
acc0 += z00*(x00+y00) + z01*(x01+y01) + z02*(x02+y02) + z03*(x03+y03)
acc1 += z10*(x10+y10) + z11*(x11+y11) + z12*(x12+y12) + z13*(x13+y13)
acc2 += z20*(x20+y20) + z21*(x21+y21) + z22*(x22+y22) + z23*(x23+y23)
acc3 += z30*(x30+y30) + z31*(x31+y31) + z32*(x32+y32) + z33*(x33+y33)
```

In this case, four outputs are being generated, so there are four lanes and four columns for each of the outputs with pre-addition from the X and Y buffers.

The parameters of the intrinsics allow for flexible data selection from the different input buffers for each lane and column, all following the same pattern of parameters. The following section introduces the data selection (or data permute) schemes with detailed examples that include `shuffle` and `select` intrinsics. Details around the `mac` intrinsic and its variants are also discussed in the following sections.

Data Selection

AI Engine intrinsic functions support various types of data selection. The details around the shuffle and select intrinsic are as follows.

Data Shuffle

The AI Engine shuffle intrinsic function selects data from a single input data buffer according to the start and offset parameters. This allows for flexible permutations of the input vector values without needing to rearrange the values. `xbuff` is the input data buffer, with `xstart` indicating the starting position offset for each lane in the `xbuff` data buffer and `xoffset` indicating the position offset applied to the data buffer. The shuffle intrinsic function is available in 8, 16, and 32 lane variants (`shuffle8`, `shuffle16`, and `shuffle32`). The main permute for data (`xoffsets`) is at 32-bit granularity and `xsquare` allows a further 16-bit granularity mini permute after main permute. Thus, the 8-bit and 16-bit vector intrinsic functions can have additional square parameter- for more complex permutations.

For example, a `shuffle16` intrinsic has the following function prototype.

```
v16int32 shuffle16    (    v16int32     xbuff,
    int     xstart,
    unsigned int     xoffsets,
    unsigned int     xoffsets_hi
)
```

The data permute performs in 32 bits granularity. When the data size is 32 bits or 64 bits, the start and offsets are relative to the full data width, 32 bits or 64 bits. The lane selection follows the regular lane selection scheme.

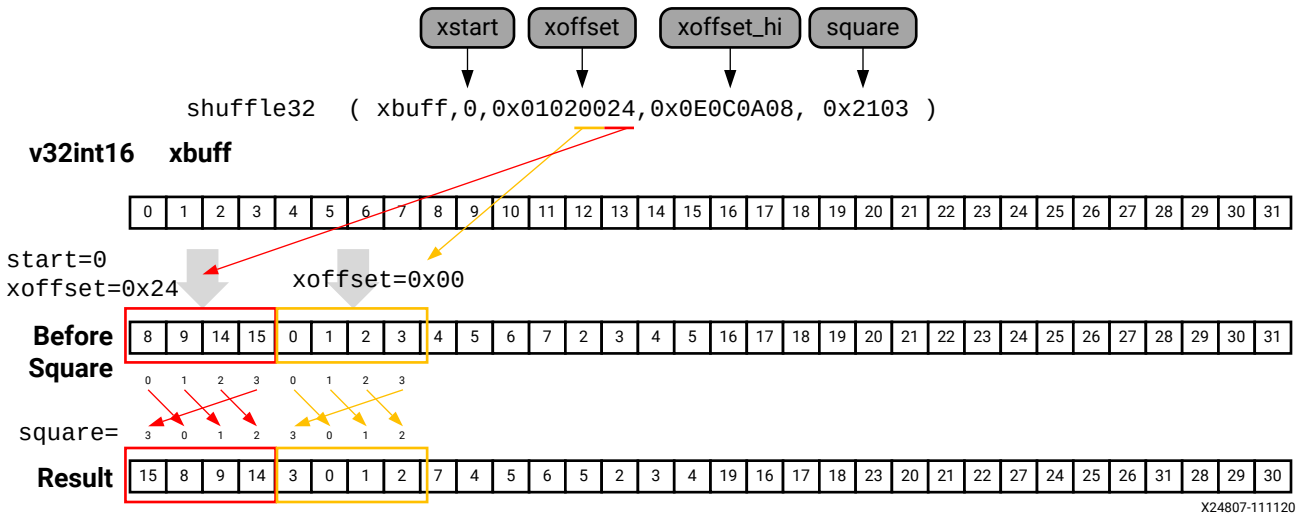
```
f: result [lane number] = (xstart + xbuff [lane number]) Mod input_samples
```

The following example shows how shuffle works on the `v16int32` vector. `xoffset` and `xoffset_hi` have 4 bits for each lane. This example moves the even and odd elements of the buffer into lower and higher parts of the buffer.

Figure 10: **Data Shuffle on int32 Type**

When data permute is on 16 bits data, the intrinsic function includes another parameter, `xsquare`, allowing flexibility to perform data selection in each 4 x 16 bits block of data. The `xoffset` comes in pairs. The first hex value is an absolute 32 bits offset and picks up 2 x 16 bits values (index, index+1). The second hex value is offset from first value + 1 (32 bits offset) and picks up 2 x 16 bits values. For example, `0x00` selects index 0, 1, and index 2, 3. `0x24` selects index 8, 9, and index 14, 15. Following is a shuffle example on the `v32int16` vector.

Figure 11: **Data Shuffle on int16 Type**



Data Select

The select intrinsic selects between the first set of lanes or the second one according to the value of the `select` parameter. If the lane corresponding bit in `select` is zero, it returns the value in the first set of lanes. If the bit is one, it returns the value in the second set of lanes. For example, a `select16` intrinsic function has the following function prototype.

```
v16int32 select16    (    unsigned int     select,
    v16int32     xbuff,
    int     xstart,
    unsigned int     xoffsets,
    unsigned int     xoffsets_hi,
    v16int32     ybuff,
    int     ystart,
    unsigned int     yoffsets,
    unsigned int     yoffsets_hi
)
```

For each bit of `select` (from low to high), it will select a lane either from `xbuff` (if the `select` parameter bit is 0) or from `ybuff` (if the `select` parameter bit is 1). Data permute on the resulting lane of `xbuff` or `ybuff` is achieved by a `shuffle` with corresponding bits in `xoffsets` or `yoffsets`. Following is the pseudo C-style code for `select`.

```
for (int i = 0; i < 16; i++){
    idx = f( xstart, xoffsets[i]); //i'th 4 bits of offsets
    idy = f( ystart, yoffsets[i]);
    o[i] = select[i] ? y[idy]:x[idx];
}
```

For information about how `f` works in previous code, refer to the regular lane selection scheme equation listed at the beginning of this section.

When working on the int16 data type, the `select` intrinsic has an additional `xsquare` parameter which allows a further 16-bit granularity mini permute after main permute. For example, a `select32` intrinsic function has the following function prototype.

```
v32int16 select32    (    unsigned int     select,
    v64int16     xbuff,
    int     xstart,
    unsigned int     xoffsets,
    unsigned int     xoffsets_hi,
    unsigned int     xsquare,
    int     ystart,
    unsigned int     yoffsets,
    unsigned int     yoffsets_hi,
    unsigned int     ysquare
)
```

Following is the pseudo C-style code for `select`.

```
for (int i = 0; i < 32; i++){
    idx = f( xstart, xoffsets[i], xsquare);
    idy = f( ystart, yoffsets[i], ysquare);
    o[i] = select[i] ? y[idy]:x[idx];
}
```

The following example uses `select32` to interleave first 16 elements of `A` and `B` (A first).

```
int16 A[32]={0,1,2,3,4,5,6,7,8,9,10,11,12,13,14,15,
        16,17,18,19,20,21,22,23,24,25,26,27,28,29,30,31
};
int16 B[32]={32,33,34,35,36,37,38,39,40,41,42,43,44,45,46,47,
        48,49,50,51,52,53,54,55,56,57,58,59,60,61,62,63
};
v32int16 *pA=(v32int16*)A;
v32int16 *pB=(v32int16*)B;
v32int16 C = select32(0xAAAAAAAA, concat(*pA,*pB),
        0, 0x03020100, 0x07060504, 0x1100,
        32, 0x03020100, 0x07060504, 0x1100);
```

The output C for the previous code is as follows.

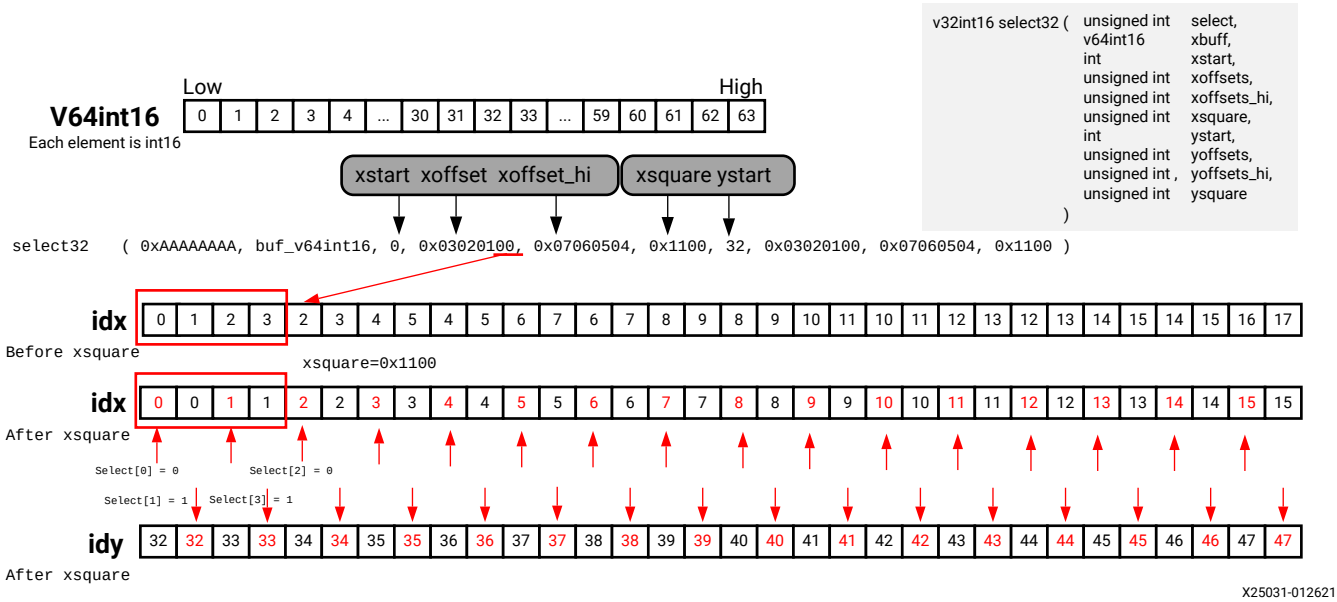
```
{0,32,1,33,2,34,3,35,4,36,5,37,6,38,7,39,8,40,9,41,10,42,11,43,12,44,13,45,1
4,46,15,47
}
```

This can also be done using the `shuffle32` intrinsic.

```
v32int16 C = shuffle32(concat(*pA,*pB),
    0, 0xF3F2F1F0, 0xF7F6F5F4, 0x3120);
```

The following figure shows how the previous `select32` intrinsic works.

Figure 12: **Data Select on int16 Type**



MAC Intrinsics

MAC intrinsics perform vector multiply and accumulate operations between data from two buffers, the X and Z buffers, with the other parameters and options allowing flexibility (data selection within the vectors, number of output lanes) and optional features (different input data sizes, pre-adding, etc). There is an additional input buffer, the Y buffer, whose values can be pre-added with those from the X buffer before the multiplication occurs. The result from the intrinsic is added to an accumulator.

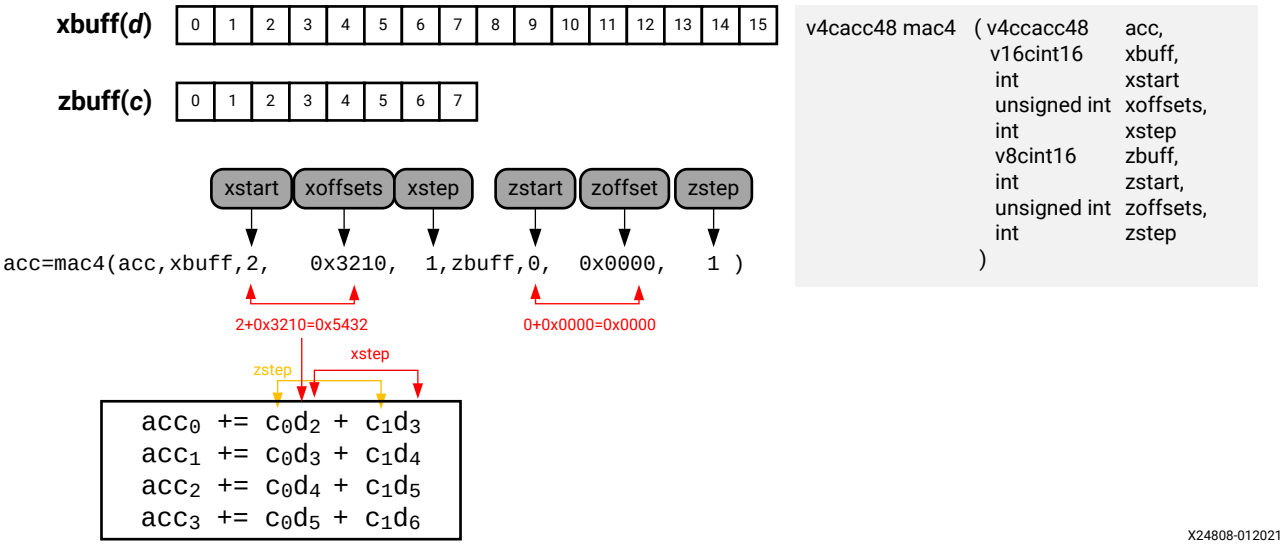
The parameters of the intrinsics allow for flexible data selection from the different input buffers for each lane and column, all following the same pattern of parameters. A starting point in the buffer is given by the (x/y/z) start parameter which selects the first element for the first row as well as first column. To allow flexibility for each lane, (x/y/z) offsets provides an offset value for each lane that will be added to the starting point. Finally, the (x/y/z) step parameter defines the step in data selection between each column based on the previous position. It is worth noticing that when the ystep is not specified in the intrinsic it will be the symmetric of the xstep.

Main permute granularity for x/y and z buffers is 32 bits and 16 bits, respectively. Complex numbers are considered as one entity for the permute (for example, cint16 as 32 bits for permute). Parameter `zstart` must be a compile time constant. 8-bit and 16-bit permute granularity in x/y and 8-bit permute granularity in z have certain limitations as addressed towards the end of this section. The following sections covers the different data widths and explains the result of the MAC intrinsic on these data widths.

MAC on 32x32 bits

The following figure shows how `start`, `offsets`, and `step` work on the cint16 data type.

Figure 13: **MAC4 on cint16 x cint16 Type**



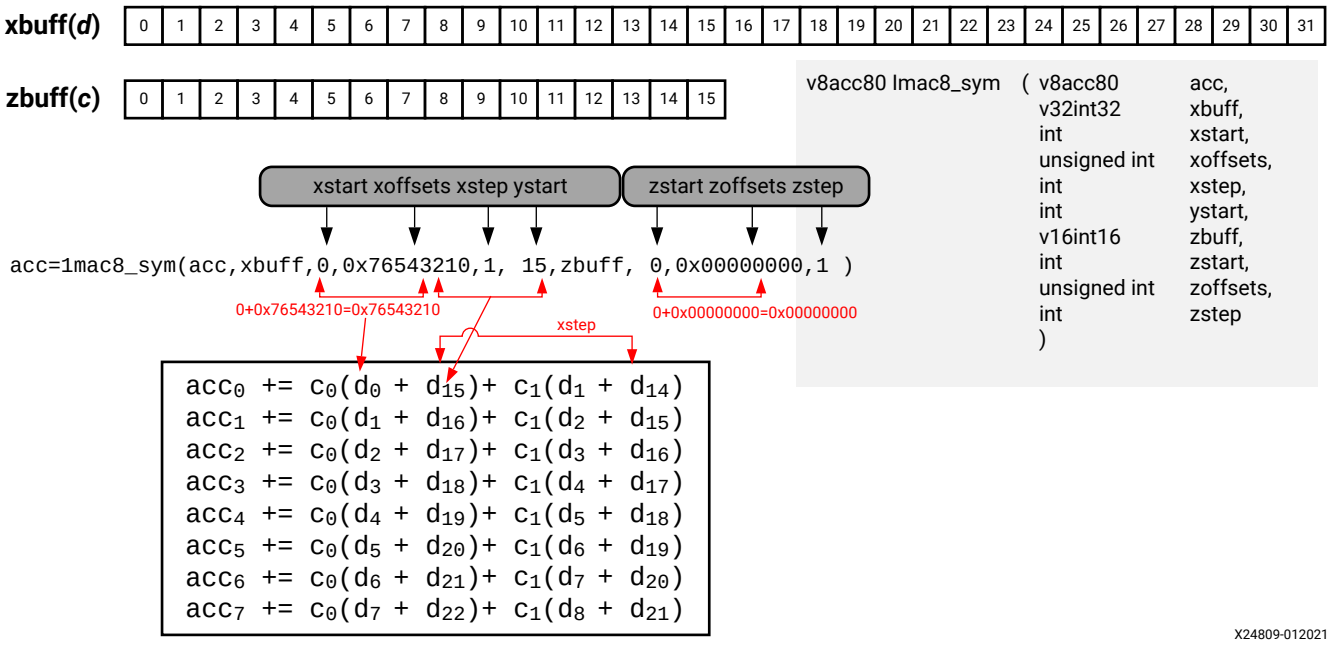
`mac4` has four output lanes. The first column of data is selected by adding `xstart` to every 4 bits of `xoffsets`. The subsequent column of data is selected by adding `xstep` to its previous column. In Table 2, it is seen that there are eight MACs per cycle for the cint16 * cint16 operation. This means that `mac4` has two columns of multiplication.

The coefficients of `mac4` are chosen similarly by `zstart`, `zoffset`, and `zstep`.

MAC on 32x16 bits

An example of MAC with pre-adding is as follows. With pre-adding, the data from `X` buffer can be added by itself, or the data from `X` buffer and `Y` buffer can be added. The `start`, `offsets`, and `step` parameters work similar as previous example. There is a `ystart` parameter for `Y` buffer or another data from `X` buffer. The `step` parameter works reversely for `Y` or another data from `X` buffer.

Figure 14: **LMAC8_SYM on int32 x int16 Type**

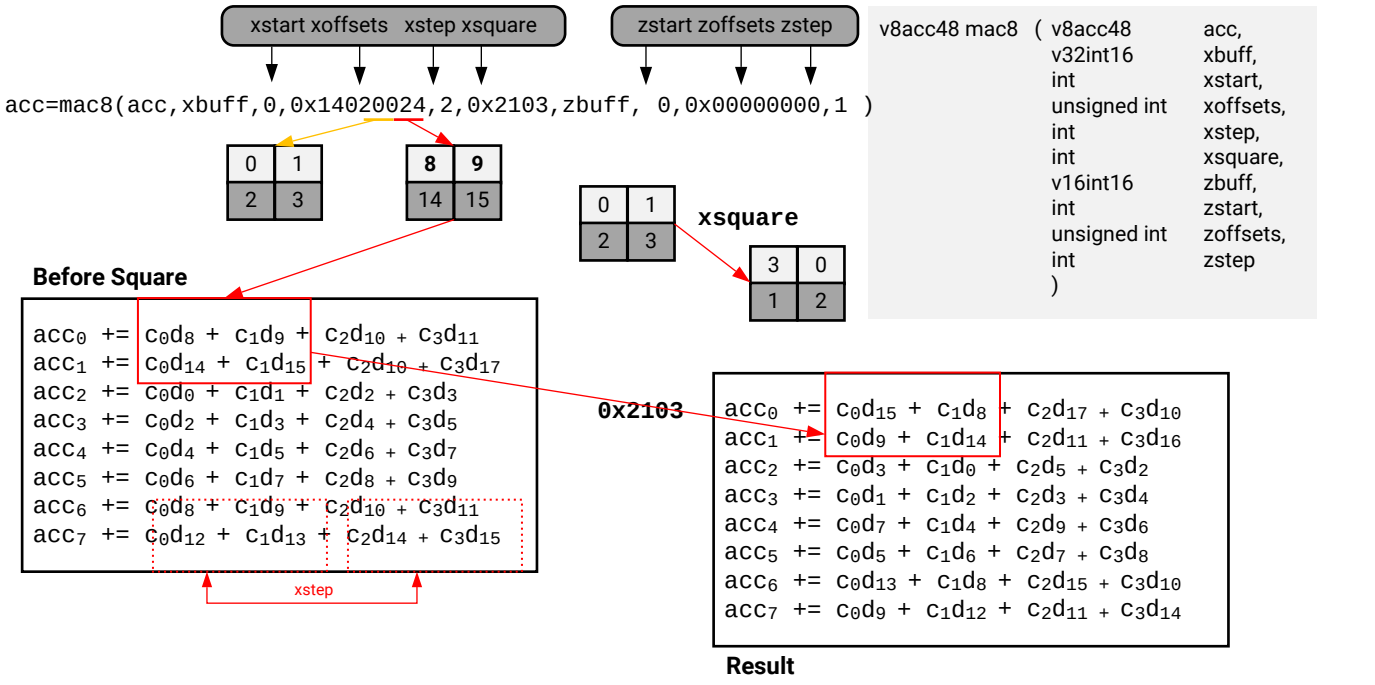


MAC on 16x16 bits

An example of MAC with int16 `X` buffer and int16 `Z` buffer is as follows. Note that the permute granularity for `X` buffer is 32 bits. The `start` and `step` parameters are always in terms of data type granularity. Therefore, a value of 2 for 16 bits data will choose 2 * 16 bits away. The `xoffsets` parameter comes as a pair. The first hex value is an absolute 32 bits offset and picks up 2 x 16 bits values (index, index+1) in the even row. The second hex value is offset from first value + 1 (32 bits offset) and picks up 2 x 16 bits values in the odd row. So the hex value `0x24` in `xoffsets` selects index 8, 9 for even row and index 14, 15 for odd row from `xbuff` and the hex value `0x00` in `xoffsets` selects index 0, 1 for even row and index 2, 3 for odd row from `xbuff`.

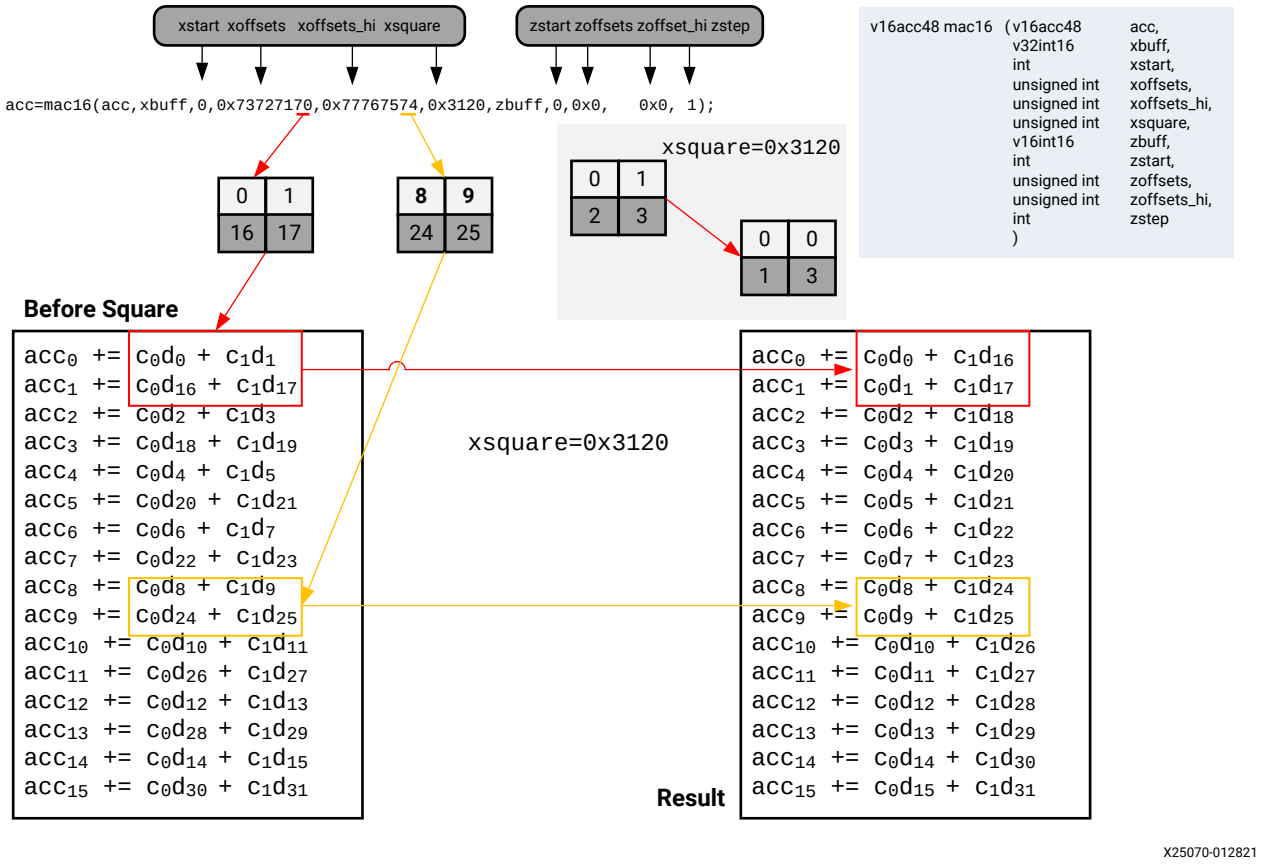
There is another `xsquare` parameter to perform 16 bits granularity twiddling after the main permute. For example, `xsquare` value `0x2103` (see from lower hex value to higher hex value) puts index 3, 0 in the even row and index 1, 2 in the odd row. How the `xsquare` parameter works can be seen in the center of the following figure.

Figure 15: **MAC8 on int16 x int16 Type**



The following figure is an example of `mac16` intrinsic of int16 and int16. It is used in the matrix vector multiplication and matrix multiplication example designs in Single Kernel Coding Examples.

Figure 16: **MAC16 on int16 x int16 Type**



MAC on 8x8 bits

The following figures show MAC with int8 `X` buffer and int8 `Z` buffer. The first figure shows how data is permuted and the second figure shows how coefficients are permuted. Note that the permute granularity for `X` buffer and `Z` buffer are 32 bits and 16 bits, respectively. The `xoffsets` parameter comes in pair. The first hex value is an absolute 32 bits offset and pick up 4 x 8 bits values (index, index+1, index+2, index+3). The second hex value is offset from the first value + 1 (32 bits offset) and picks up 4 x 8 bits values. For example, `0x00` selects index 0, 1, 2, 3 as well as 4, 5, 6, 7, and `0x24` selects index 16, 17, 18, 19 as well as 28, 29, 30, 31.

There is another `xsquare` parameter to do 8 bits granularity twiddling after main permute. How `xsquare` parameter works in this example can be seen in the center of the following figure.

The `start` (`xstart`, `zstart`) and `step` (`xstep`, `zstep`) parameters are always in terms of data type granularity. So, a value of 2 for 16 bits are 2 * 16 bits away, while a value of 2 for 8 bits are 2 * 8 bits away. The `step` parameter applies to the next block of selected data. So, if a pair of `offset` parameters select a 2 * 2 block, the step applies to the next 2 * 2 block. The step added to the index value must be aligned to the permute granularity (32 bits for data, 16 bits for coefficient). For example, when working with 8-bit data, `xstep` needs to be multiples of four. When working with 8-bit coefficient, `zstep` needs to be multiples of two. The following two figures show how `step` works for data and coefficients.

Note that for the coefficient in int8 * int8 types, the 2 * 2 index block is duplicated to construct a 4 * 2 block. See how index 0, 1, 2, and 3 are duplicated in Figure 18.

Figure 17: **MAC8 on int8 x int8 type (X part)**

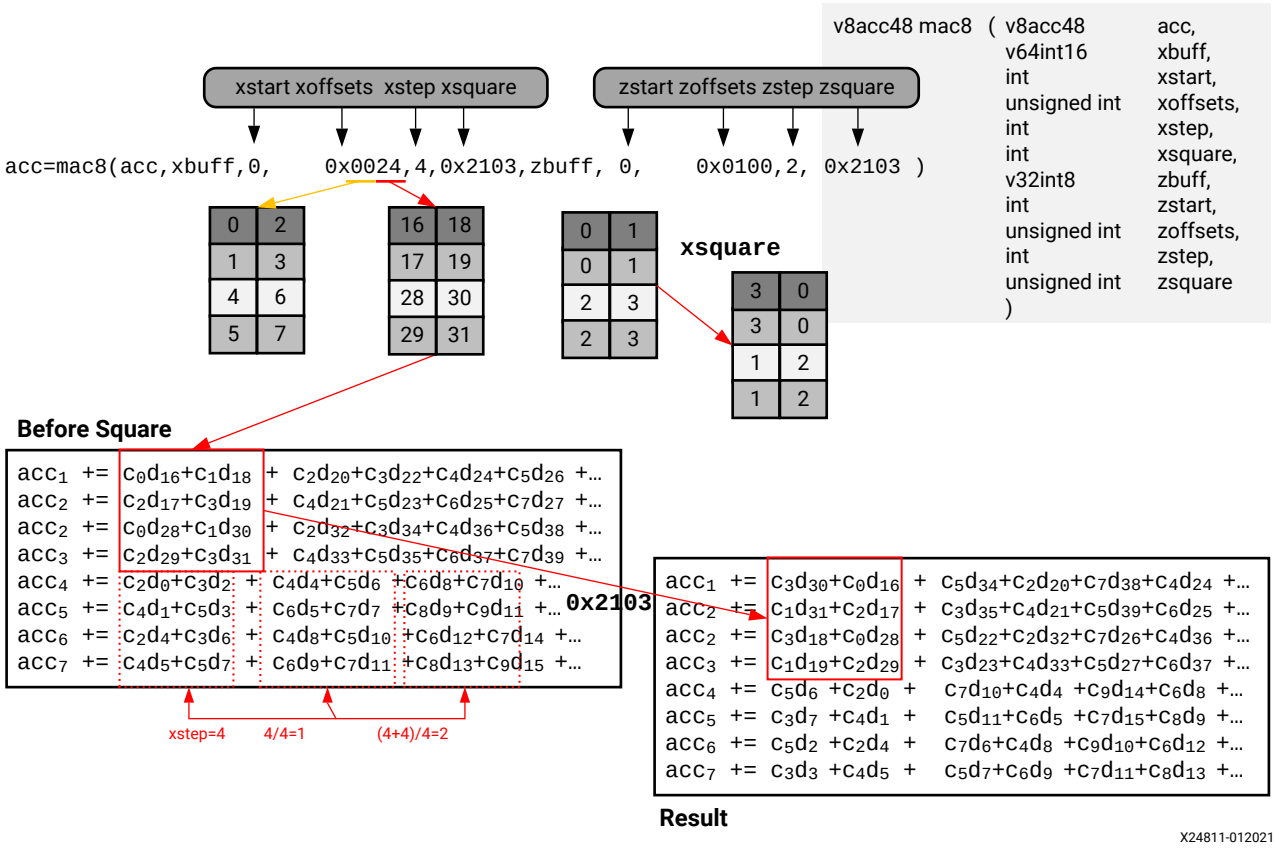
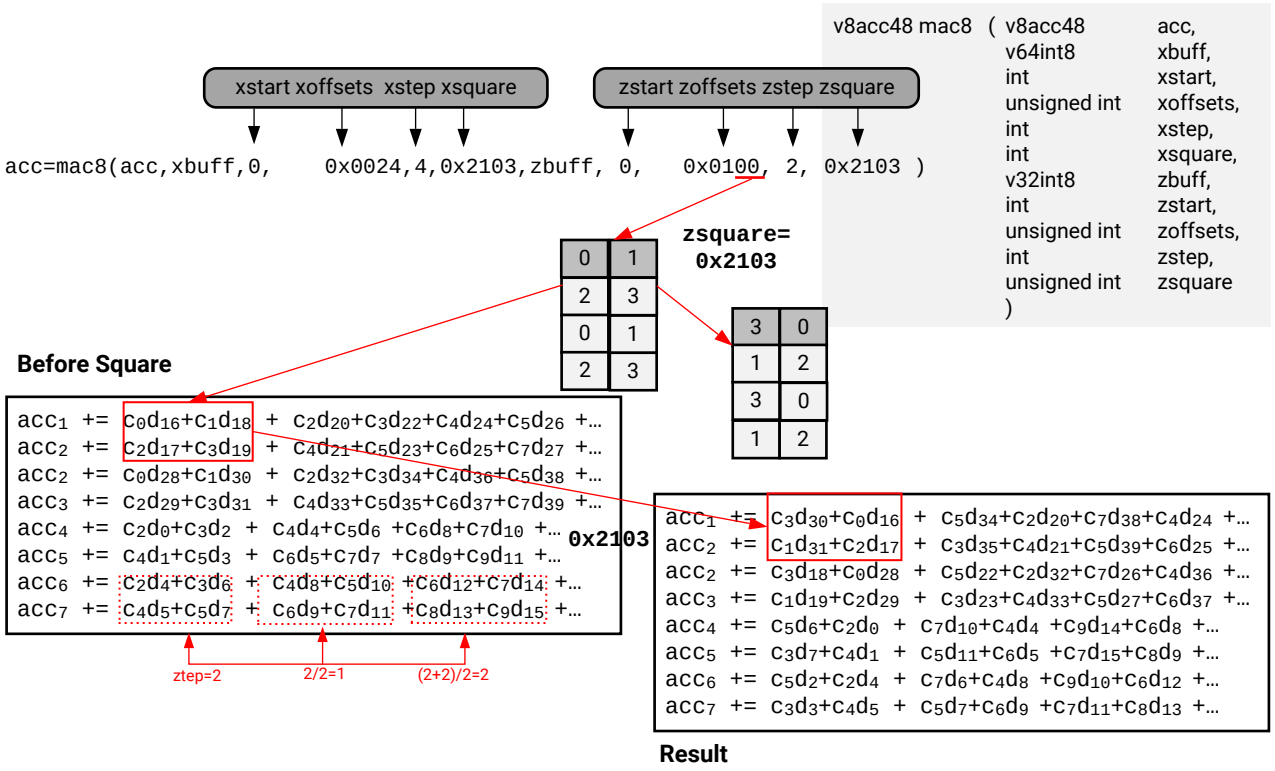


Figure 18: **MAC8 on int8 x int8 type (Z part)**



Options

There are rich sets of MAC intrinsic with additional operations like pre-adding, pre-subtraction, and conjugation. The naming convention for the vector MAC intrinsics is as follows. Optional characteristics are shown in [] and mandatory ones in {}.

```
[l]{mac|msc|mul|negmul}{2|4|8|16}[_abs|_max|_min|_maxdiff][_conj][{_sym|
_antisym}[_ct|_uct]][_c|_cc|_cn|_nc]
```

Every operation will either be a multiplication, initializing an accumulator, or a MAC operation which accumulates to a running accumulator of 2, 4, 8, or 16 lanes.

- `l`: Denotes that an accumulator with 80-bit lanes is used for the operation.
- `sym` and `antisym`: Indicates the use of pre-adding and pre-subtraction respectively.
- `max`, `min`, and `maxdiff`: Indicates the pre-selection of lanes in the `xbuff` based on the maximum, minimum, or maximum difference value.
- `abs`: Indicates the pre-computation of the absolute value in the `xbuff`.
- `ct`: Used for partial pre-adding and pre-subtraction (separate selection for the data input from X for the final column).

- `uct`: Used for unit center optimization for certain types of FIR filters. Refer to the *Versal ACAP AI Engine Intrinsics Documentation* (UG1078) for more information.
- `n` **and** `c`: Used to indicate that the complex conjugate will be used for one of the input buffers with complex values:
  - `c`: The only complex input buffer will be conjugated.
  - `cn`: Complex conjugate of X (or XY if pre-adding is used) buffer.
  - `nc`: Complex conjugate of Z buffer.
  - `cc`: Complex conjugate of both X (or XY if pre-adding is used) and Z buffers.
- `conj`: Indicates that the complex conjugate of Z will be used when multiplying the data input from Y.

Data Permute and MAC Examples

The following example takes two vectors with reals in `rva` and imaginary in `rvb` (with type `v8int32`) and creates a new complex vector, using the offsets to interleave the values as required.

```
v8cint32 cv = as_v8cint32(select16(0xaaaa, concat(rva, rvb),
        0, 0x03020100, 0x07060504,  8, 0x30201000, 0x70605040));
```

The following example shows how to extract real and imaginary portion of a vector `cv` with type `v8cint32`.

```
v16int32 re_im  = shuffle16(as_v16int32(cv), 0, 0xECA86420, 0xFDB97531);
v8int32 re = ext_w(re_im, 0);
v8int32 im = ext_w(re_im, 1);
```

Shuffle intrinsic functions can be used to reorder the elements in a vector or set all elements to the same value. Some intrinsic functions operate only on larger registers but it is easy to use them for smaller registers. The following example shows how to implement a function to set all four elements in a vector to a constant value.

```
v4int32 v2 = ext_v(shuffle16(xset_v(0, v1), 0 ,0, 0), 0);
```

The following example shows how to multiply each element in `rva` by the first element in `rvb`. This is efficient for a vector multiplied by constant value.

```
v8acc80 acc = lmul8(concat(rva,undef_v8int32()),0,0x76543210,rvb,0,0x00);
```

The following examples show how to multiply each element in `rva` by its corresponding element in `rvb`.

```
acc = lmul8(concat(rva, undef_v8int32()),0,0x76543210,rvb,0,0x76543210);
acc = lmul8(upd_w(undef_v16int32(),0,rva),0,0x76543210,rvb,0,0x76543210);
```

The following examples show how to do matrix multiplication for int8 x int8 data types with `mul` intrinsic, assuming that data storage is row based.

```
//Z_{2x8} * X_{8x8} = A_{2x8}
mul16(Xbuff, 0, 0x11101110, 16, 0x3120, Zbuff, 0, 0x44440000, 2, 0x3210);
//Z_{4x8} * X_{8x4} = A_{4x4}
mul16(Xbuff, 0, 0x00000000, 8, 0x3120, Zbuff, 0, 0xCC884400, 2, 0x3210);
```

If the kernel has multiple `mul` or `mac` intrinsics, try to keep the `xoffsets` and `zoffsets` parameters constant across uses and vary the `xtsart` and `zstart` parameters. This will help prevent configuration register spills on stack.

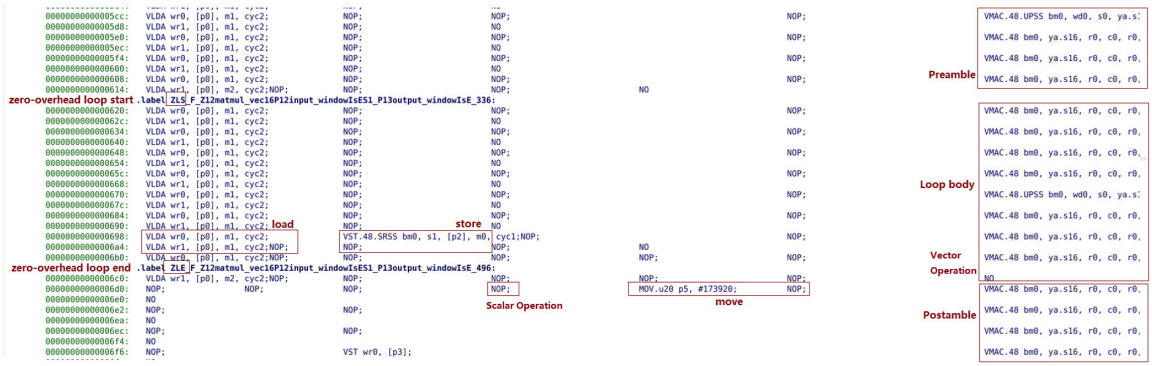
For more information about vector lane permutations, refer to the *Versal ACAP AI Engine Intrinsics Documentation* (UG1078).

Loops

The AI Engine has a zero-overhead loop structure that does not incur any branch control overhead for comparison and branching thus reducing the inner loop cycle count. Pipelining allows the compiler to add pre-amble and post-amble so that the instruction pipeline is always full during loop execution. With a pipelined loop, a new iteration can be started before the previous one ends to achieve higher instruction level parallelism.

The following figure shows the assembly code of a zero-overhead loop. Note that two vector loads, one vector store, one scalar instruction, two data moves, and one vector instruction are shown in order in different slots.

Figure 19: **Assembly Code of Zero-Overhead Loop**



The following pragmas work together to direct the compiler to pipeline the loop and let it know that the loop will always be executed at least three times.

```
for (int i=0; i<N; i+=2)
   chess_prepare_for_pipelining
   chess_loop_range(3,)
```

The `chess_loop_range(<minimum>, <maximum>)` tells the compiler that the corresponding loop is executed at least `<minimum>` times, and at most `<maximum>` times, where `<minimum>` and `<maximum>` are non-negative constant expressions, or can be omitted. When omitted, `<minimum>` defaults to 0, and `<maximum>` defaults to the maximum preset in the compiler. While `<maximum>` is not relevant for the pipeline implementation, `<minimum>` guides the pipeline implementation.

The `<minimum>` number defines how many loop iterations are executed at a minimum each time the loop is executed. The software pipeline is then tuned to allow at least that many iterations to execute in parallel if possible. It also determines that checking the boundaries for the loop is not necessary before the `<minimum>` number of iterations are executed.

The loop range pragma is not needed if the loop range is a compile time constant. In general, the AI Engine compiler reports the theoretical number best suited for optimum pipelining of an algorithm. If the range specification is not optimal, the compiler would issue a warning and suggest the optimal range. Towards that end, it is okay to initially set the `<minimum>` to one `[chess_loop_range(1,)]` and observe the theoretical best suited `<minimum>` being reported by the compiler.

```
Warning in "matmul_vec16.cc", line 10: (loop #39)
further loop software pipelining (to 4 cycles) is feasible with
`chess_prepare_for_pipelining'
but requires a minimum loop count of 3
... consider annotating the loop with `chess_loop_range(3,)' if applicable,
... or remove the current `chess_loop_range(1,)` annotation
```

At this point, you can choose to update the `<minimum>` number to the reported optimum.

This second part of the pipeline implementation can be a reason for potential deadlocks in the AI Engine kernels if the actual `<minimum>` number of iterations is not reached. For this reason, you must ensure that the number of iterations is always at least the number specified in the `chess_loop_range` directive.

Loop carried dependencies impact the vectorization of code. If an inner loop dependency cannot be removed, a strategy to step out a level and manually unroll where there are (effectively) multiple copies of the inner loop running in parallel.

When looping though data, to increment or decrement by a specific offset, use the `cyclic_add` intrinsic function for circular buffers. The `fft_data_incr` intrinsic function enables the iteration of the pointer that is the current target of the butterfly operation. Using these functions can save multiple clock cycles over coding the equivalent functionality in standard C. Depending on the data types, you might need to cast parameters and return types.

The following example uses `fft_data_incr` intrinsic when operating on a matrix of real numbers.

```
pC = (v8float*)fft_data_incr( (v4cfloat*)pC, colB_tiled, pTarget);
```

Try to avoid sequential load operations to fill a vector register completely before use. It is best to interleave loads with MAC intrinsic functions, where the current MAC and next load can be done in the same cycle.

```
acc = mul4_sym(lbuff, 4, 0x3210, 1, rbuff, 11, coeff, 0, 0x0000, 1);
lbuff = upd_w(lbuff, 0, *left);
acc = mac4_sym(acc, lbuff, 8, 0x3210, 1, rbuff, 7, coeff, 4, 0x0000, 1);
```

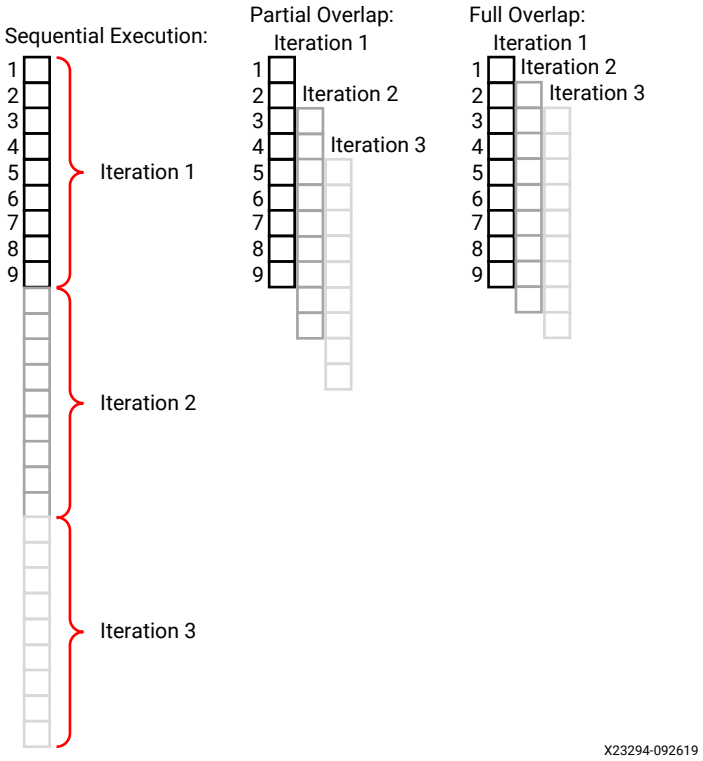
In certain use cases loop rotation, which rotates the instructions inside the loop, can be beneficial. Instead of loading data into a vector at the start of a loop, consider loading a block of data for the first iteration before the loop, and then for the next iteration near the end of the loop. This will add additional instructions but shorten the dependency length of the loop which helps to achieve an ideal loop with a potentially lower loop range.

```
// Load starting data for first iteration
sbuff = upd_w(sbuff, 0, window_readincr_v8(cb_input)); // 0..7

for ( int l=0; l<LSIZE; ++l )
chess_loop_range(5,)
chess_prepare_for_pipelining
{
   sbuff = upd_w(sbuff, 1, window_readincr_v8(cb_input)); // 8..15
   acc0 = mul4_sym(     sbuff,5 ,0x3210,1 ,12 ,coe,4,0x0000,1);

   sbuff = upd_w(sbuff, 2, window_readdecr_v8(cb_input)); // 16..23
   acc0 = mac4_sym(acc0,sbuff,1 ,0x3210,1 ,16,coe,0,0x0000,1);
   acc1 = mul4_sym(     sbuff,5 ,0x3210,1 ,20,coe,0,0x0000,1);
   window_writeincr(cb_output, srs(acc0, shift));
   // Load data for next iteration
   sbuff = upd_w(sbuff, 0, window_readincr_v8(cb_input)); // 0..7
   acc1 = mac4_sym(acc1,sbuff,9,0x3210,1,16,coe,4,0x0000,1);
   window_writeincr(cb_output, srs(acc1, shift));
}
```

Software Pipelining of Loops

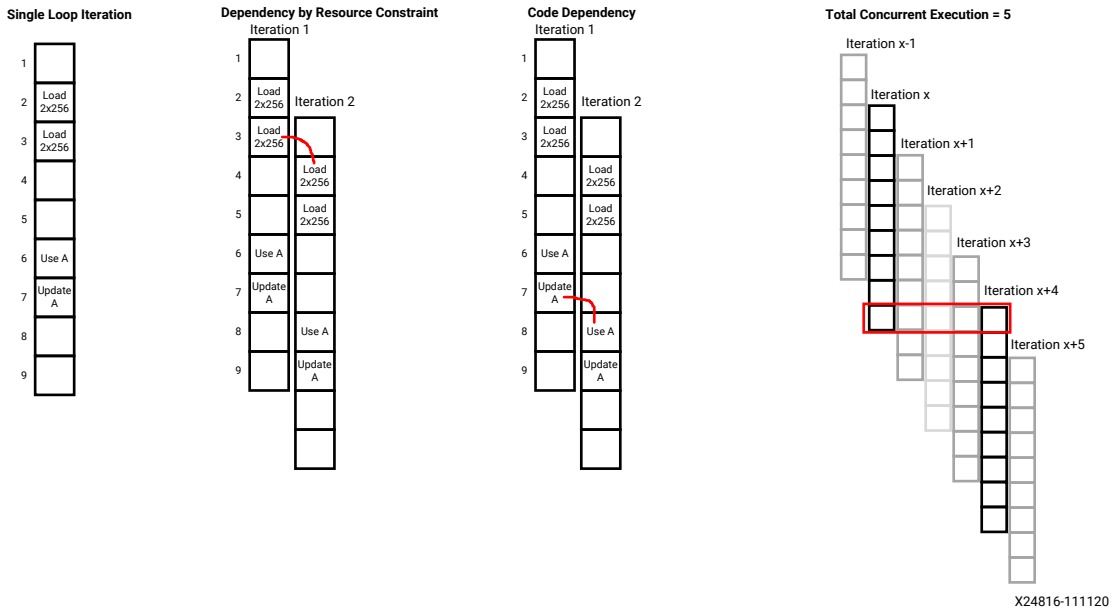
This section dives into software pipelining of loops. This is an important concept that enables the AI Engine to concurrently execute different parts of a program. For example, a loop that requires a total of nine cycles to execute through one iteration is shown in the following figure, where sequential execution all the way to a full overlap pipelining is illustrated.

Figure 20: **Pipelining Example**



Counting the cycles through each of these examples, it is clear that the sequential execution requires 27 cycles to fully execute the three loop iterations, while the partially overlapped pipeline requires 13 cycles, and the fully pipelined loop requires only 11 cycles. From a performance perspective, it is therefore desirable to have a fully overlapping pipeline. However, this is not always possible, because resource constraints, as well as inter-iteration loop dependencies can prevent a full overlap (see the following figure).

Figure 21: **Dependencies in Pipelining**



In this example, the program performs load A (2 x 256-bit) in cycle 2, load B (2 x 256-bit) in cycle 3, and in cycle 6 and 7 it is executes operations on loop variable A. The remaining instructions of this iteration are of no importance with respect to the loop performance analysis.

Cycles 2 and 3 of this loop iteration execute 4 x 256-bit load operations. The required four loads are executed in two cycles because the AI Engines can only execute two loads per cycle. This is called a resource constraint. If the loop containing this iteration is supposed to be pipelined, this constraint limits the overlap to no less than two cycles. Similarly, code dependencies between iterations shown in cycle 6 and 7 can prevent additional overlap. In this case, the next iteration of the loop requires the value of A to be updated before it can be used by the loop, thus, limiting the overlap.

The AI Engine compiler reports on each loop in the following form.

***Note*:** The core compilation report can be found in `Work/aie/core_ID/core_ID.log` and the `-v` option is needed to generate the verbose report.

```
HW do-loop #397 in "testbench.cc", line 132: (loop #16) :
Critical cycle of length 2 : b67 -> b68 -> b67
Minimum length due to resources: 2
Scheduling HW do-loop #397
(algo 1a)        -> # cycles: 9
(modulo)         -> # cycles: 2 ok (required budget ratio: 1)
(resume algo)      -> after folding:  2 (folded over 4 iterations)
  -> HW do-loop #397 in "testbench.cc", line 132: (loop #16) : 2 cycles
NOTICE: loop #397 contains folded negative edges
NOTICE: postamble created
Removing chess_separator blocks (all)
```

In the AI Engine compiler report shown previously, the section `Critical cycle of length` provides feedback on code dependencies, while the `Minimum length due to resources` indicates minimum overlap requirement due to resource constraints. The `algo 1a` line states the total amount of cycles for a single iteration. Given these numbers, there are a maximum of five iterations active at a time creating the pipeline.

The AI Engine compiler reports these five overlapping iterations (the current iteration plus four folded iterations) in the `resume algo` line. In addition, it states the initiation interval (ii), the number of cycles a single iteration has to execute before the following iteration is started, which is two in this example.

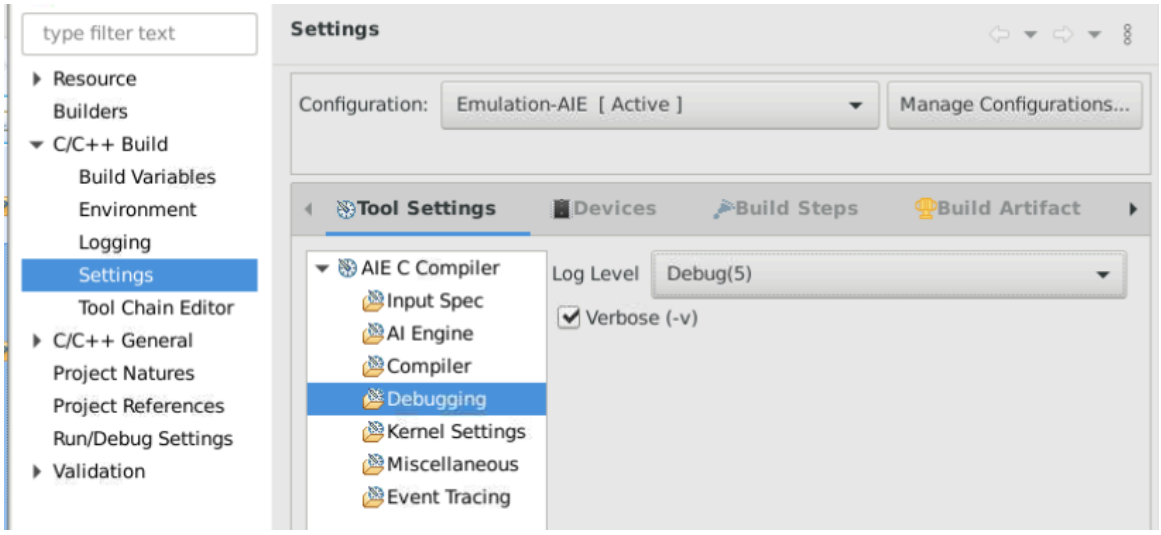
In general, it is sufficient to provide the directive `chess_prepare_for_pipelining` to instruct the compiler to attempt software pipelining. When the number of loop iterations is a compile time constant, the chess compiler creates the optimum software pipeline.

In the case of a dynamic loop range (defined by a variable start/end), the compiler requires additional information to create an effective pipeline loop structure. This is performed through the directive `chess_loop_range(<minimum>, <maximum>)`. Details about the `chess_loop_range(<minimum>, <maximum>)` directive can be found in Loops.

Note: If the number of cycles in the loop exceeds 64 cycles, pipelining can be disabled by the compiler for that loop. In this case, the AI Engine compiler reports the following message.

```
(algo 1a)       -> # cycles: 167 (exceeds -k 64) -> no folding: 167
  -> HW do-loop #511 in "xxxx", line 794: (loop #8): 167 cycles
```

IMPORTANT! *To generate the detailed report, it is important to enable verbose mode in Vitis IDE as shown in the following image or enter the `-v` option for the AI Engine compiler in the command line.*



Module scheduling report can be generated for module scheduled loops by specifying the option `-Xchess=main:backend.mist2.xargs=-ggraph` for the AI Engine compiler. Module scheduling report will be available for software pipelined loop with the name `*_modulo.rpt in Work/aie/core_ID/Release/chesswork/<mangled_function_name>/*.rpt`, where * is the block name. The module scheduling report also contains the information about register live ranges for register files, which may be useful to find inefficiencies in register assignment and can be improved by using `chess_storage`.

After compilation and linking is complete, Vitis Analyzer can be used to open the compile log for an individual kernel. For more information about the Vitis Analyzer, see the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Restrict Keyword

The C standard provides a specific pointer qualifier `__restrict`, intended to allow more aggressive compiler optimization, by explicitly stating independence between data. The compiler, by default, does not distinguish between different access of the same array. Thus, if an array is accessed in the pipeline, it can hinder the pipeline from achieving higher interval between loops with conservative assumption. This makes it is essential in some situations to use a `__restrict` keyword to help guide the tool to achieve better performance. Care must be taken in using the `__restrict` keyword because if the `__restrict` keyword is assigned to pointers in the same scope, undefined behavior may be observed when the pointers are used. For detailed information about the concept of the `__restrict` keyword, see Using the Restrict Keyword in AI Engine Kernels in the AI Engine Documentation flow of the *Vitis Unified Software Platform Documentation* (UG1416).

Floating-Point Operations

The scalar unit floating-point hardware support includes square root, inverse square root, inverse, absolute value, minimum, and maximum. It supports other floating-point operations through emulation. The `softfloat` library must be linked in for test benches and kernel code using emulation. For math library functions, the single precision float version must be used (for example, use `expf()` instead of `exp()`).

The AI Engine vector unit provides eight lanes of single-precision floating-point multiplication and accumulation. The unit reuses the vector register files and permute network of the fixed-point data path. In general, only one vector instruction per cycle can be performed in fixed-point or floating-point.

Floating-point MACs have a latency of two-cycles, thus, using two accumulators in a ping-pong manner helps performance by allowing the compiler to schedule a MAC on each clock cycle.

```
acc0 = fpmac( acc0, abuff, 1, 0x0, bbuff, 0, 0x76543210 );
acc1 = fpmac( acc1, abuff, 9, 0x0, bbuff, 0, 0x76543210 );
```

There are no divide scalar or vector intrinsic functions at this time. However, vector division can be implemented via an inverse and multiply as shown in the following example.

```
invpi = upd_elem(invpi, 0, inv(pi));
acc = fpmul(concat(acc, undef_v8float()), 0, 0x76543210, invpi, 0, 0);
```

A similar implementation can be done for the vectors `sqrt`,`invsqrt`, and `sincos`.

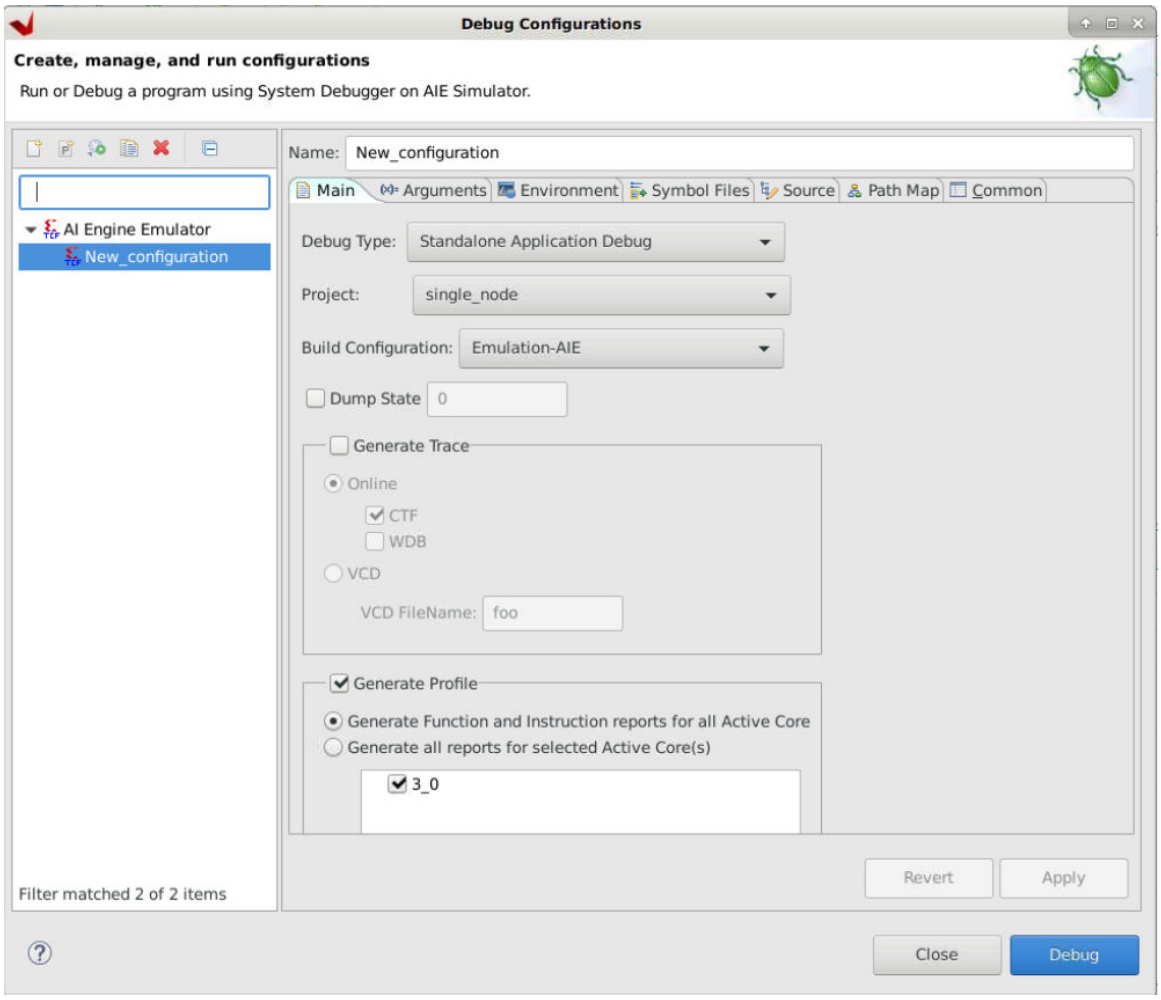
Using Vitis IDE and Reports

The Vitis IDE manages the system project with an AI Engine graph, PL kernels, and PS application. Vitis IDE provides visual views for AI Engine kernel development and it is essential in kernel, as well as PS application debugging.

The Vitis IDE provides a single node graph example that can be used as a starting point for single kernel development. The Vitis IDE has a debug view which displays registers, variables, available breakpoints, variables to register/memory mapping, internal/external memory contents, disassembly view for instruction, and an instruction pipeline (pipeline view) for single AI Engine kernel.

When launching the debug perspective, if Generate Profile is selected in debug configurations, it will show the `printf` output in console and the Runtime Statistics window will show real-time cycle count when stepping through instructions. The Generate Trace check box in debug configurations is for generating event trace data which helps better understand when and how events such as memory stall and stream stall have occurred. Event trace is helpful in performance tuning.

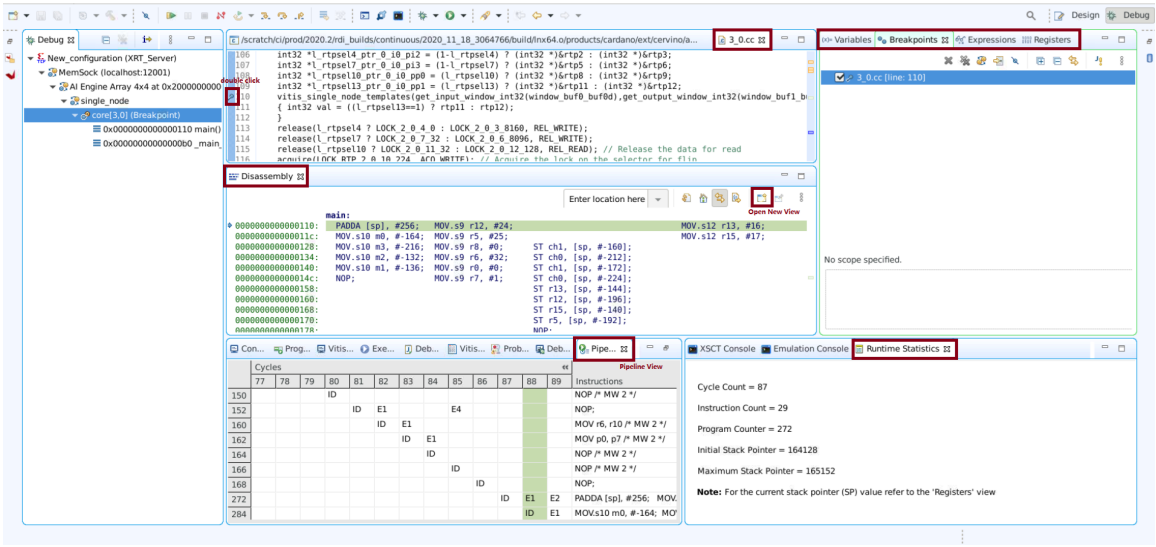
Figure 22: **Debug Configurations**



In debug perspective, debug commands resume, and step into, as well as step over are available. The AI Engine source code is shown and it is possible to set breakpoint by double-clicking lines. The windows Variables, Breakpoints, and Registers are available to look into data memory or register status. The disassembly view is helpful in understanding how intrinsics are used, especially how they are scheduled in pipeline. In the disassembly view, the button **Open New View** can be used to open a new active window. Pipeline view allows you to correlate instructions executed in a specific clock cycle with the labels in the microcode/disassembler view.

***Note*:** The pipeline view is only available for single AI Engine design and it is only enabled when the **Generate Profile** check box is checked when debug is started.

Figure 23: **Debug Code**

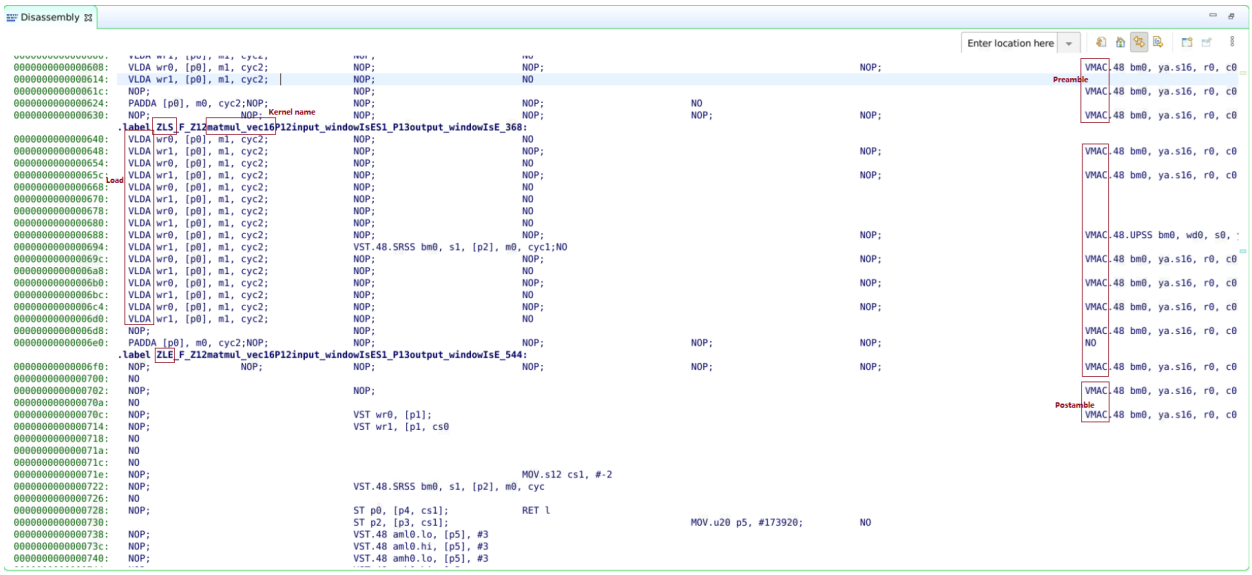


The generated code for an AI Engine (`Col_Row.cc`) includes the AI Engine kernels in the core and wrapper code. From the AI Engine wrapper code, you can step into the AI Engine kernel code by clicking multiple step-in buttons. Alternatively, you can also open the AI Engine kernel source file from the design perspective and set breakpoints in the file. Multiple views, such as the disassembly view, pipeline view, memory view, register view, and variables view can be used for debug, as well as performance tuning.

***Note*:** The number of breakpoints are limited to four for each tile. To set new breakpoints, beyond the number allowed, you must clear existing breakpoints. The tool will issue error messages if you try and set breakpoints beyond the number allowed.

The disassembly view displays the compiler generated microcode target to the hardware. C/C++ source code can also be embedded between the lines for source code referencing. The microcode helps understand the compiled result, especially the loop pipelining result. The following figure shows the microcode generated for a pipelined loop. By scrolling or stepping in the disassembly view, the loop in the kernel can be found. The loop iterates from zero-overhead loop start (ZLS) to zero-overhead loop end (ZLE). It can be seen how load instructions and MAC instructions are placed to be pipelined. The preamble and postamble instructions are placed before and after the zero-overhead loop body to fill and flush the pipeline stages.

Figure 24: **Disassembly View for Loop Pipelining**



Linker memory map reports for AI Engines cores can be found in `Work/aie/core_ID/Release/core_id.map`. It lists the locations of the program and data memories by functions, static variables, and the software stack. From these reports, the stack size, program memory size, and global buffers, as well as their sizes can be extracted. An xml version of the linker report (`core_id.map.xml`) can be generated by specifying the option `-Xchess=\"main:bridge.xargs=-fB\"` to the AI Engine compiler.

`Work/<name>.aiecompile_summary` is the compilation summary that can be opened by the Vitis Analyzer. Work/reports contains multiple reports for the graph compilation result such as, kernels and buffers mapping result. Refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076) for additional information about AI Engine compiler outputs.

Interface Considerations

While single kernel programming focuses on vectorization of algorithm in a single AI Engine, multiple kernel programming considers several AI Engine kernels with data flowing between them.

The ADF graph can contain a single kernel or multiple kernels interacting with PS, PL, and global memory. Each AI Engine kernel has a runtime ratio. This number is computed as a ratio of the number of cycles taken by one invocation of a kernel (processing one block of data) to the cycle budget. The cycle budget for an application is typically fixed according to the expected data throughput and the block size being processed. The runtime ratio is specified as a constraint for every AI Engine kernel in the ADF graph.

The AI Engine compiler allocates multiple kernels into a single AI Engine if their combined total runtime ratio is less than one and multiple kernels fit in the AI Engine program memory. Alternatively, the compiler can allocate them into multiple AI Engine.

To optimally use hardware resources, it is critical to understand the different methods available to transfer data between the ADF graph and PS, PL, and global memory, transfer data between kernels, balance the data movement, and minimize memory or stream stalls as much as possible which are covered in the following sections.

Data Movement Between AI Engines

Generally, there are two methods to transfer data between kernels–window or stream. When using the window, data transfers can be realized as ping-pong buffers and optionally, using a single buffer. AI Engine tools will take care of buffer synchronization between the kernels. Designers need to decide the window size and buffer location between kernels through their partition of the application. If an overlap is needed between different windows of the data, AI Engine tools provide options for setting a margin for the window, that is, to copy the overlap of data by AI Engine tools automatically.

When using the stream, the data movement involves two input as well as two output stream ports, along with one dedicated cascade stream input port and output port. Stream ports can provide 32-bit per cycle or, 128-bit per four cycles on each port. Stream interfaces are bidirectional and can read or write neighboring or non-neighboring AI Engines by stream ports. However, cascade stream ports are unidirectional and only provide a one-way access between the neighboring AI Engines.

Data Communication via Shared Memory

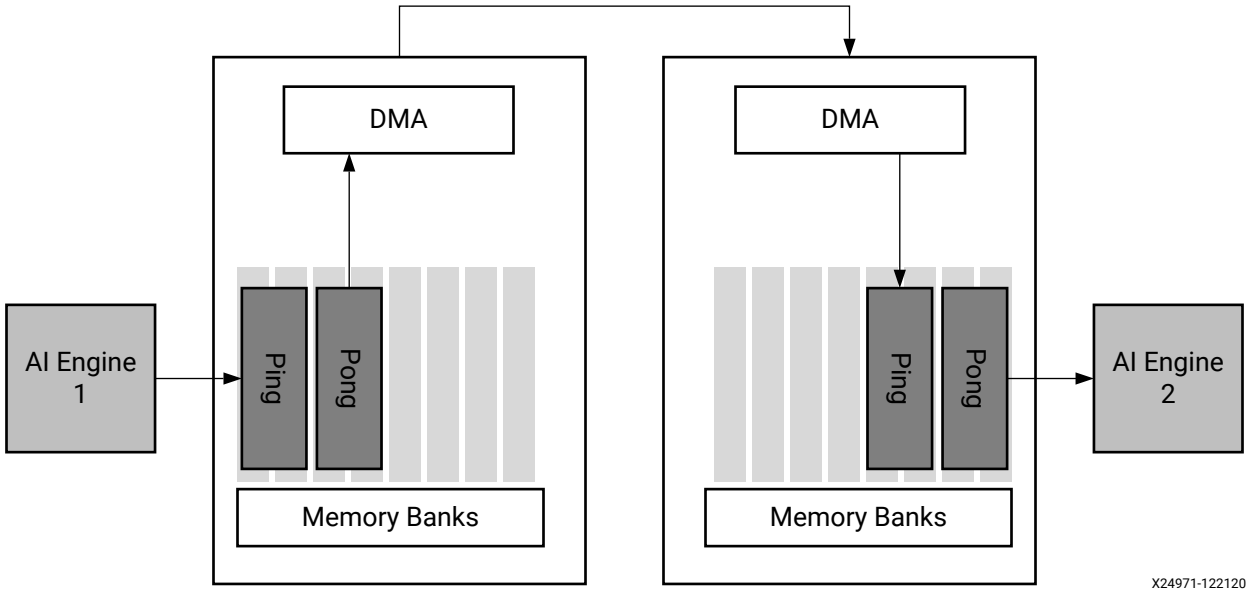
In the case where multiple kernels fit in a single AI Engine, communication between two or more consecutive kernels can be established using a common buffer in the shared memory. In this case, only a single buffer is needed because the kernels are time-multiplexed.

For cases where the kernels are in separate but neighboring AI Engines, the communication can be carried out through the shared memory module that use ping-pong buffers. These buffers are on separate memory banks so access conflicts are avoided. The synchronization is done through locks. The input and output buffers for the AI Engine kernel are ensured to be ready by the locks associated with the buffers. In this type of communication, routing resources are saved and data transferring latency is eliminated because DMA and AXI4-Stream interconnect are not needed.

Data Communication via Memory and DMA

For non-neighboring AI Engines, similar communication can be established using the DMA in the memory module associated with each AI Engine. Ping-pong buffers in each memory module are used and synchronization is carried out with locks. There is increased communication latency as well as memory resources in comparison to shared memory communication.

Figure 25: **Data Communication via Memory and DMA**



Data Communication via AXI4-Stream Interconnect

AI Engines can directly communicate through the AXI4-Stream interconnect without any DMA and memory interaction. Data can be sent from one AI Engine to another or broadcast through the streaming interface. The data bandwidth of a streaming connection is 32-bit per cycle and built-in handshake and backpressure mechanisms are available.

For streaming input and output interfaces, when the performance is limited by the stream number, the AI Engine is able to use two streaming inputs or two streaming outputs in parallel, instead of one streaming input or output. To use two parallel streams, it is recommended to use the following pairs of macros, where `idx1` and `idx2` are the two streams. Add the `__restrict` keyword to stream ports to ensure they are optimized for parallel processing.

```
READINCR(SS_rsrc1, idx1) and READINCR(SS_rsrc2, idx2)
READINCRW(WSS_rsrc1, idx1) and READINCRW(WSS_rsrc2, idx2)
WRITEINCR(MS_rsrc1, idx1, val) and WRITEINCR(MS_rsrc2, idx2, val)
WRITEINCRW(WMS_rsrc1, idx1, val) and WRITEINCRW(WMS_rsrc2, idx2, val)
```

Following is a sample code to use two parallel input streams to achieve pipelining with interval 1. Interval 1 means that two read, one write, and one add are in every cycle.

```
void simple(   input_stream_int32 * __restrict data0,
            input_stream_int32 * __restrict data1,
            output_stream_int32 * __restrict out) {
    for(int i=0; i<1024; i++)
    chess_prepare_for_pipelining
    {
```

```
        int32_t d = READINCR(SS_rsrc1, data0) ;
        int32_t e = READINCR(SS_rsrc2, data1) ;
        WRITEINCR(MS_rsrc1,out,d+e);
    }
}
```

Intrinsics can be used to perform stream operations directly, but it is important to not swap the two streams based on the mapping AI Engine tools has found.

```
v16float off = *(v16float*)offset;
v8float v8in = undef_v8float();
v8float v8out = undef_v8float();
for(int i=0;i<128/4;i++)
chess_prepare_for_pipelining
{
    v8in = concat(getf_wss(0),getf_wss(1)); // reads 8 float values
    v8out = fpadd(v8in,off,0,0);
    put_wms(0,ext_v(v8out,0));
    put_wms(1,ext_v(v8out,1));
}
```

The stream connection can be unicast or multicast. Note that in the case of multicast communication, the data is sent to all the destination ports at the same time and only when all destinations are ready to receive data.

Window vs. Stream in Data Communication

AI Engine kernels in the data flow graph operate on data streams that are infinitely long sequences of typed values. These data streams can be broken into separate blocks called windows and processed by a kernel. Kernels consume input blocks of data and produce output blocks of data. An initialization function can be specified to run before the kernel starts processing input data. The kernel can read scalars or vectors from the memory, however, the valid vector length for each read and write operation must be either 128 or 256 bits. Windows of input data and output buffer are locked for kernels before they are executed. Because the input data window needs to be filled with input data before kernel start, it increases latency compared to stream interface. The kernel can perform random access within a window of data and there is the ability to specify a window margin for algorithms that require some number of bytes from the previous sample.

Kernels can also access the data streams in a sample-by-sample fashion. Streams are used for continuous data and using blocking or non-blocking calls to read and write. Cascade stream only supports blocking access. The AI Engine supports two 32-bit stream input ports and two 32-bit stream output ports. Valid vector length for reading or writing data streams must be either 32 or 128 bits. Packet streams are useful when the number of independent data streams in the program exceeds the number of hardware stream channels or ports available. The AI Engine interconnect to and from the PL is through streams.

The following table summarizes the differences in window and stream connections between kernels.

Table 5: **Window vs. Stream Connections**

Connection	Margin	Packet Switching	Back Pressure	Lock	Max throughput by VLIW (per cycle)	Multicast as a Source
Window	Yes	Yes[1]	No	Yes	2*256-bit load + 1*256-bit store	No
Stream	No	Yes[1]	Yes	No	2*32-bit read + 1*32-bit write, or 1*32-bit read + 2*32-bit write	Yes

Notes:
1. Packet switching is only supported between AI Engine kernels and PL kernels.

Graph code is C++ and available in a separate file from kernel source files. The compiler places the AI Engine kernels into the AI Engine array, taking care of the memory requirements and making all the necessary connections for data flow. Multiple kernels with low core usage can be placed into a single tile.

For a complete overview of graph programming with AI Engine tools, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Free Running AI Engine Kernel

The AI Engine kernel can always be running using `graph::run(-1)`. This way the kernel will restart automatically after the last iteration is complete.

Note: `graph::run()` without an argument runs the AI Engine kernels for a previously specified number of iterations (which is infinity by default if the graph is run without any arguments). If the graph is run with a finite number of iterations, for example, `mygraph.run(3); mygraph.run();` the second run call will also run for three iterations.

However, it requires input buffers and output buffers to be ready before it can start. Thus, it has a small overhead between kernel execution iterations. This section describes a method to construct a type of kernel that has zero overhead and runs forever. It is called the free running AI Engine kernel.

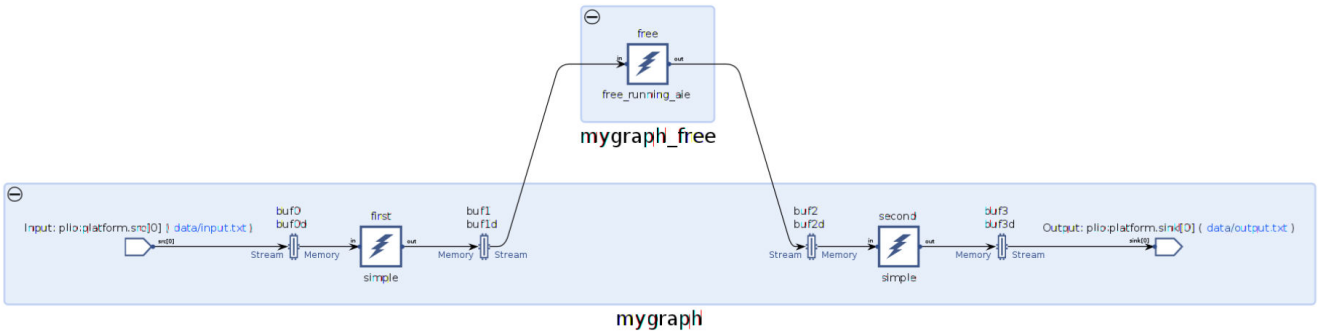
The free running kernels can only have streaming interfaces. Loops with infinite iterations can be inside the kernel. For example:

```
void free_running_aie(input_stream_cint16 * in,
            output_stream_cint16 * out) {
    while(true){ //This can be syntax supported by C++, for example: for(;;)
        writeincr(out, readincr(in));
    }
}
```

The free running kernel must have its own graph defined. This graph must not have any other non-free running kernels, because the graph never stops and non-free running kernels will lose control after being started. The graph containing the free running kernel must be a top-level graph that can be connected to other graphs, or it can be connected to PLIO or GMIO. A sample connection between the free running graph and other graphs is shown as follows.

```
simpleGraph mygraph; //normal graph
freeGraph mygraph_free; //graph with free running kernel
simulation::platform<1,1> platform("data/input.txt", "data/output.txt");
connect<> net0(platform.src[0], mygraph.in);
connect<> net1(mygraph.sout,mygraph_free.in);
connect<> net2(mygraph_free.out,mygraph.sin);
connect<> net3(mygraph.out, platform.sink[0]);
```

Figure 26: **Free Running Graph Connection**



The free running graph can be started using `mygraph_free.run(-1)` or automatically started after loading.

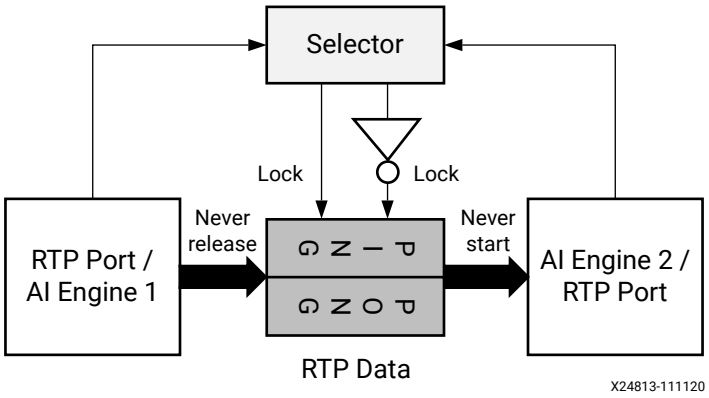
Run-Time Parameter Specification

Run-time parameters (RTP) is another way to pass data to the kernels. Two types of execution model for run-time parameters are supported.

1. Asynchronous parameters can be changed at any time by either a controlling processor such as the Arm®, or by another AI Engine. They are read each time a kernel is invoked. This means that the update of parameters occurs between different executions of the kernel, but it does not require the update to take place in a specific pattern. For example, these types of parameters are used as filter coefficients that change infrequently.
2. Synchronous parameters (triggering parameters) block a kernel from execution until these parameters are written by a controlling processor such as the Arm, or by another AI Engine. Upon a write, the kernel reads the new updated value and executes once. After completion, it is blocked from executing until the parameter is updated again. This allows a different type of execution model from the normal streaming model, which can be useful for certain updating operations where blocking synchronization is important.

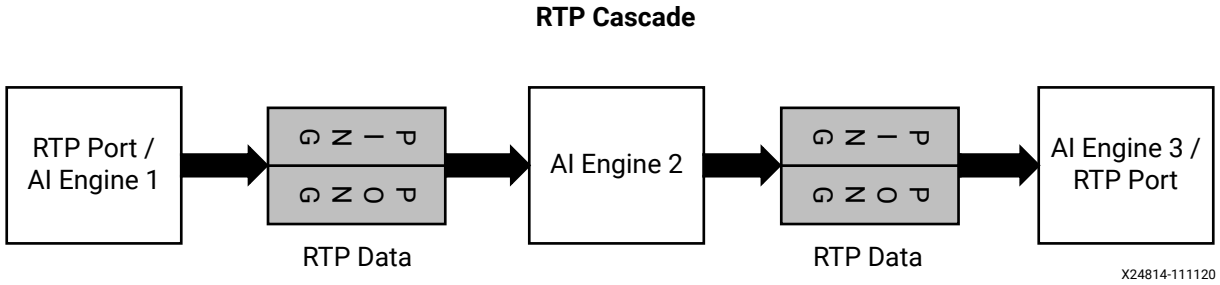
The following figure shows how the AI Engine RTP is realized in hardware. The source of RTP can be a port to be written by the controlling processor, or RTP output by an AI Engine kernel. The destination of RTP can be an output to be read by controlling processor, or RTP input of an AI Engine kernel. Source and destination will use ping-pong buffers for the RTP data transferring. Both source and destination will read a selector value to determine if ping or pong of RTP value must be written or read. Before write or read, source and destination will try to lock the buffers before starting kernel executions.

Figure 27: **AI Engine RTP**



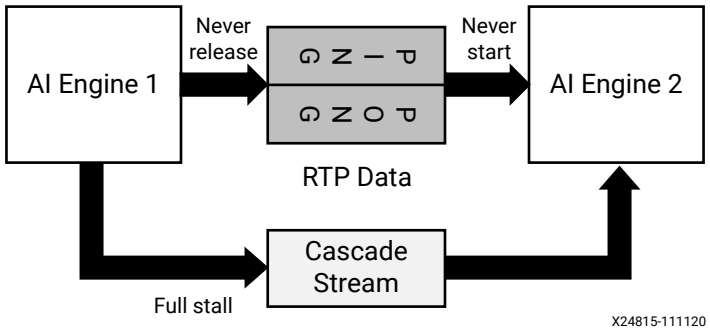
Cascade of RTP is supported as shown in the following figure. Only synchronous-to-synchronous or asynchronous-to-asynchronous modes are allowed for the RTP connection. Async-to-sync or sync-to-async modes are not allowed. RTP port is not allowed to broadcast to multiple destinations.

Figure 28: **RTP Cascade**



It is very important to understand that the RTP interaction between AI Engine kernels only happen in kernel execution boundaries. This means that the RTP output of the source kernel can only be read by destination kernel when the source kernel has completed its current iteration. If the source and destination rely on each other before finishing or starting kernel executions, it may cause deadload. For example, if the source and destination are connected by cascade stream besides the RTP connection, the cascade stream will stall the source AI Engine after it is full. However, because the source kernel has not finished its execution, it will not release the RTP data. Thus, the destination AI Engine will never get RTP data and will never start.

Figure 29: **RTP Cascade Deadlock Example**



***Note*:** RTP ports of AI Engine kernels will need to be acquired lock and released lock before and after kernel execution. This will cause a small overhead for each kernel iteration. When thinking of partitioning the data into frames, the overhead must be taken into consideration according to system level performance requirements.

For more information about run-time parameter usage, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

AI Engine and PL Kernels Data Communication

The AI Engine array interface contains modules to communicate between AI Engines and PL kernels using AXI4-Stream connections. Generally, PL interfaces produce or consume data through stream interfaces. They connect through the AI Engine stream with AI Engine kernels. Based on whether window or stream data is communicated by the AI Engine kernels, DMA and ping-pong buffers could be involved.

Note that PL kernels run at a lower frequency than AI Engine kernels. Data must cross the clock domains (CDC) between the AI Engine clock and PL clock. The Vitis™ environment handles the CDC path automatically. It is recommended to run the PL kernel frequency as an integer factor of the AI Engine frequency if possible. For instance, as ½ or ¼ of the AI Engine clock frequency.

For more information about AI Engine to PL rate matching considerations, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

DDR Memory Access through GMIO

The main data streams from and to the AI Engine are the AI Engine to PL streaming interface and GMIO, which is used to make external memory-mapped connections to or from the global memory. The interface between PS and AI Engine can target low throughput purposes such as, configuration. There are two types of GMIOs, the AI Engine-GMIO and PL-GMIO. The AI Engine-GMIO directly connects to the DDR memory through theAI Engine-NOC master unit (NMU). PL-GMIO uses PL-NOC NMU, but AI Engine connects to DDR memory through PL kernel, which interfaces the AI Engine through the AI Engine-to-PL streaming interface. The PL kernel that uses PL-GMIO is part of the graph, and can be controlled by ADF API together with AI Engine kernels.

The bandwidth of AI Engine GMIO is affected by the number of NMUs and DDR memory controllers used in the platform. PL-GMIO is further affected by PL kernel clock frequency and AI Engine-to-PL interface, besides the number of DDR memory controllers used in the platform and the interface from PL kernel to DDR memory.

The benefits of AI Engine GMIO include that it can directly access DDR memory, and it is not only a virtual platform for AI Engine simulator, but also can work in hardware without PL kernels. For more information about GMIO programming model, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Design Analysis and Programming

AI Engines provide high compute density through large amount of VLIW and SIMD compute units by connecting with each other through innovative memory and AXI4-Stream networks. When targeting an application on AI Engine, it is important to evaluate the compute needs of the AI Engine and data throughput requirements. For example, how the AI Engine interacts with PL kernels and external DDR memory. After the compute and data throughput requirements can be met for AI Engine, the next step involves divide and conquer methods to map the algorithm into the AI Engine array. In the divide and conquer step, it is necessary to understand vector processor architecture, memory structure, AXI4-Stream, and cascade stream interfaces. This step is usually iterated multiple times. At the same time, each single AI Engine kernel is optimized and the graph is constructed and optimized iteratively. AI Engine tools are used to simulate and debug AI Engine kernels and the graph. The graph is then integrated with PL kernels, GMIO, and PS to perform system level verification and performance tuning.

In this chapter, the divide and conquer method to map the algorithm into data flow diagrams (DFD) is briefly introduced. Single kernel programming and multiple kernels programming examples are provided to illustrate how to do kernel partitioning by the compute and memory bound, single kernel vectorization and optimization, and streaming balancing between different kernels.

Mapping Algorithm onto the AI Engine

When starting from a pure software model, one must first identify the application boundary. With the application boundary identified, the input and output for the application can be defined. The application can then be divided into components (processing units) that perform specific operations. Data will flow from input, through one or multiple components, to the outputs. This is called the data flow diagram (DFD).

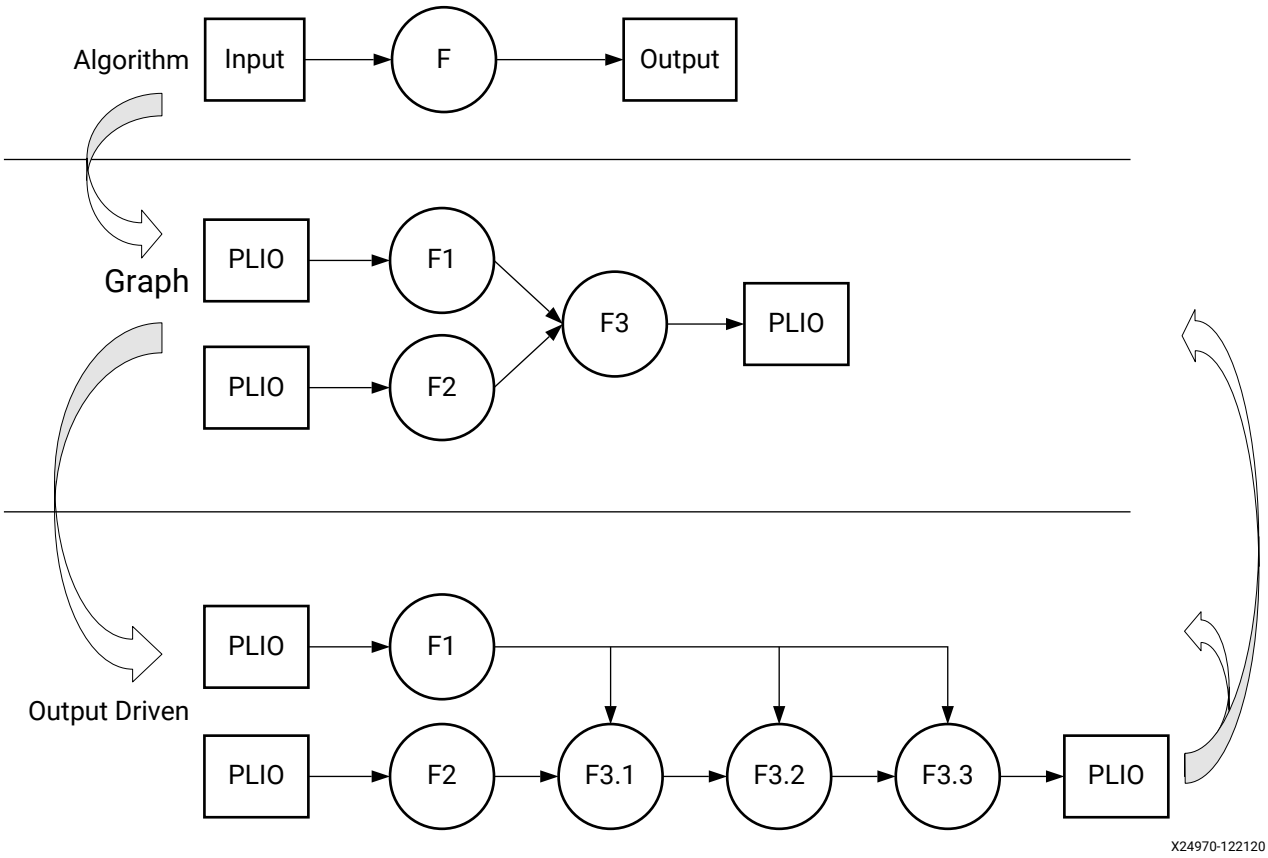
The DFD can be mapped into a graph, which is the actual design running in the AI Engine. When performing the mapping, the input and output sources, as well as their throughputs must be taken into consideration in accordance to the system performance requirements. Ideally, high throughput data must go through PLIO or GMIO. The PS can handle lower bandwidth configuration data through RTP ports of the graph. Depending on the bandwidth the PL kernels can provide, the interfaces between AI-to-PL can be determined.

After the graph interface is defined, elements in the graph can be further divided into different kernels. This kernel partition process is typically throughput driven. The hard limits of AI Engine compute capacity, data memory size, and program memory size must be followed by the single AI Engine kernel. Besides, the data transfer between kernels must be measured and balanced. Multiple data transfer methods between AI Engine kernels are available, like PING-PONG memory, stream, and cascade stream. This requires accounting for stream interface availability, memory availability, and locations of the buffers, because those considerations directly affect data communication between kernels and the data fetch throughput. When splitting a kernel into multiple kernels that are chained by cascade streams, these kernels are tightly integrated and synchronized by producing or consuming data in the cascade stream.

After kernel partitioning, the I/O interfaces and throughput requirement for a kernel should be clear. When the compute, memory, and interfaces can meet the requirements in theory, then it is advisable to think about vectorization of single kernel realization.

Application partitioning and kernel programming can be fast iterated in simulation. These can require engineers to redo partition or restructure the data flow between kernels as the process goes deeper. This iterative method of mapping algorithm into graph, partitioning applications into AI Engine kernels, and refining data flow driven by throughput is shown in the following figure.

Figure 30: **Data Flow Diagram to Graph**



Depending on the run-time ratios of the kernels, one kernel or multiple kernels can be mapped into one AI Engine. Run-time ratio of a kernel can be computed using the following equation:

$$\text{run-time ratio} = (\text{cycles for one run of the kernel})/(\text{cycle budget})$$

The cycle budget is the cycles allowed to run one invocation of the kernel which depends on the system throughput requirement. Cycles for one run of the kernel can be estimated in the initial design stage and profiled in the AI Engine simulator when vectorized code is available.

Run-time ratio setting is required in graph specification and it affects how data is communicated between kernels. Refer to Data Movement Between AI Engines for more information about data communication.

When the graph can be constructed and kernels are achievable in AI Engine tiles, vectorization is done for each individual AI Engine kernel. It is possible to leave some kernels not vectorized for proof of concept initially.

The process of vectorization is based on the vector data type and vector intrinsic functions. This requires accounting for registers and the available data to load for computation. Usually the target of the optimization is to achieve the best initiation interval of the main loop in the kernel. This involves interleaving data load, compute and store, as well as resolving data dependency and compute dependency in the loop. When optimizing hierarchical loops, it is helpful to pipeline the outer loop and unroll the inner loop when the inner loop is not large. The techniques introduced in Chapter 2: Single Kernel Programming are applicable for vectorization and optimization of the single kernel.

Single Kernel Coding Examples

The following sections focus on mapping the algorithm into a single AI Engine. It takes into consideration the compute and memory bounds of the algorithm and capability of the AI Engine. The scalar and vectorized versions of the code are shown to illustrate vectorization.

Matrix Vector Multiplication

The following matrix vector multiplication example focuses on a single AI Engine kernel vectorization. It implements the following matrix vector multiplication equation.

$$C\ (64 \times 1) = A\ (64 \times 16) * B(16 \times 1)$$

The example assumes that the data for the matrices is stored in column based form and data type for the matrices A and B is int16.

```
c0 = a0*b0 + a64*b1 + a128*b2 + a192*b3 + a256*b4 + a320*b5 + a384*b6 +
a448*b7 + …
c1 = a1*b0 + a65*b1 + a129*b2 + a193*b3 + a257*b4 + a321*b5 + a385*b6 +
a449*b7 + …
c2 = a2*b0 + a66*b1 + a130*b2 + a194*b3 + a258*b4 + a322*b5 + a386*b6 +
a450*b7 + …
c3 = a3*b0 + a67*b1 + a131*b2 + a195*b3 + a259*b4 + a323*b5 + a387*b6 +
a451*b7 + …
                                …
c60 = a60*b0 + a124*b1 + a188*b2 + a252*b3 + a316*b4 + a380*b5 + a444*b6 +
a508*b7 + …
c61 = a61*b0 + a125*b1 + a189*b2 + a253*b3 + a317*b4 + a381*b5 + a445*b6 +
a509*b7 + …
c62 = a62*b0 + a126*b1 + a190*b2 + a254*b3 + a318*b4 + a382*b5 + a446*b6 +
a510*b7 + …
c63 = a63*b0 + a127*b1 + a191*b2 + a255*b3 + a319*b4 + a383*b5 + a447*b6 +
a5111*b7 + …
```

Kernel Coding Bounds

In this example, a total of 16 int16 x int16 multiplications are required per output value. As the matrix C consists of 64 values, a total of 16 * 64 = 1024 multiplications is required to complete one matrix multiplication. Given that 32 16-bit multiplications can be performed per cycle in an AI Engine, the minimum number of cycles required for the matrix multiplication is 1024/32 = 32. The summation of the individual terms comes without additional cycle requirements because the addition can be performed together with the multiplication in a MAC operation. Hence the compute bound for the kernel is:

$$\text{Compute bound} = 32 \text{ cycles / invocation}$$

Next, analyze the memory accesses bound for the kernel. If it is going to fully use the vector unit MAC performance, 32 16-bit multiplications are performed per cycle. Vector b can be stored in the vector register because it is only 16*16-bit =256 bits. It does not need to be fetched from the AI Engine data memory or tile interface for each MAC operation. Considering data “a” needed for computation, it needs 32*16-bit = 512 bits data per cycle. The stream interface only supports 2*32 bit per cycle and hence fetching data from memory can be considered. It allows two 256 bits loads per cycle which matches the MAC performance. Thus, if two 256 bits loads are performed each cycle, the memory bound for the kernel is:

$$\text{Memory bound} = 32 \text{ cycles / invocation}$$

Note that compute bound and memory bound are the theoretical limits of the kernel realization. It does not take into account the function overhead outside the main computation loop. When the kernel is only part of the graph, it might be relieved due to bandwidth limitation of other kernels or lower system performance requirements.

Vectorization

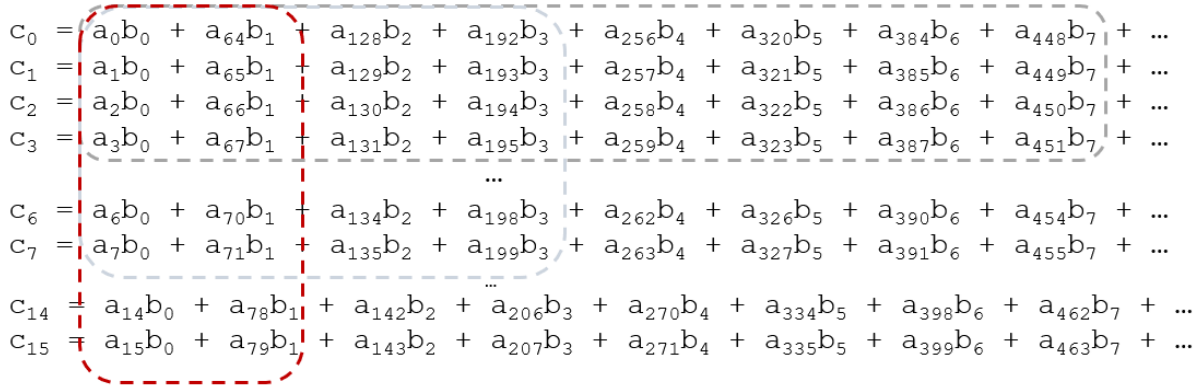
For a complicated vector processing algorithm, starting with a scalar version is recommended because it is also helpful as a golden reference for verifying the accuracy. The scalar version for matrix multiplication is shown as follows.

```
void matmul_scalar(input_window_int16* matA,
      input_window_int16* matB,
      output_window_int16* matC){ //A[M,N], B[N,1], C[M,1]. M=64, N=16
    for(int i=0; i<M; i++){
        int temp = 0 ;
        for(int j=0; j<N; j++){
            temp += window_read(matA)*window_readincr(matB) ;
            window_incr(matA,64); //Jump of 64 elements to access the next
element of the same row
        }
        window_writeincr(matC,(int16_t)(temp>>15)) ;
        window_incr(matA,1); //Jump of one element for moving to the next
row.
    }
}
```

Note that in the previously shown code, `matA` is stored in the column base and `matB` is a circular buffer to the kernel. It can be read continuously by `window_readincr` for computing different rows of output because it will loop back to the start of the buffer.

There are total 64 outputs (M=64), and each output needs 16 (N=16) multiplications. When choosing MAC intrinsics to do vector processing, for the data type int16 * int16, we may select lane 4, 8 ,16 to do the equation. These are illustrated in following figure.

Figure 31: **Lane Selection**

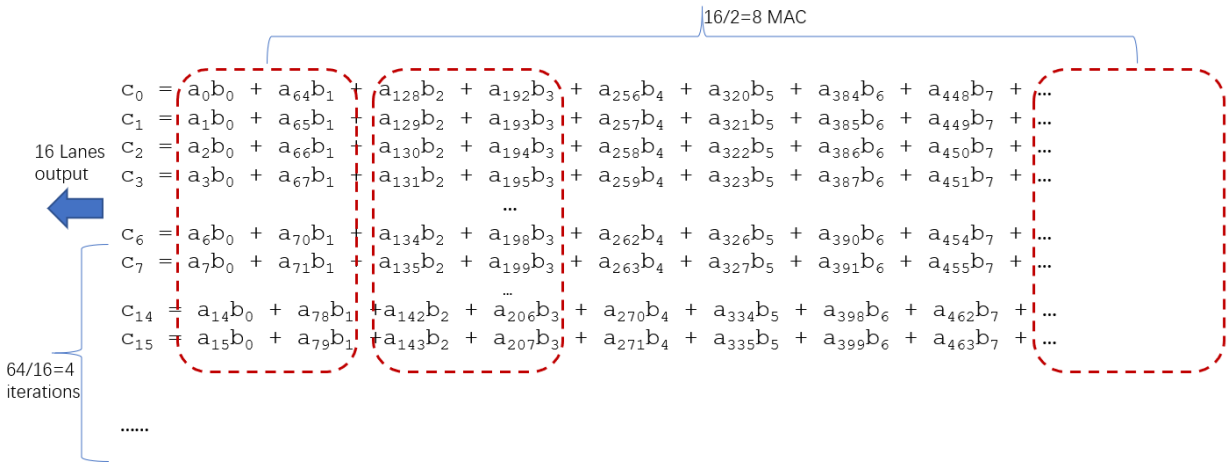


Note that the main difference between 4,8, and 16 lanes MAC is how the data is consumed. If we assume that the data is stored by column, then 16 lanes MAC may be the best choice, because only two parts of continuous data needs to be loaded for the MAC operation, – a_0 to a_{15} and a_{64} to a_{79}. a_0 to a_{15} are 256 bits, which allows one load to load the value into vector register.

To allow two loads to occur at the same cycle, a_0 to a_{15} and a_{64} to a_{79} are required to be in separate data banks. The data needs to be divided column by column into two separate buffers to the kernel. That is to say, a_0 to a_{63} are in the first buffer, a_{64} to a_{127} are in the second buffer, a_{128} to a_{191} are in the first buffer again, and so on.

By vectorization, the matrix multiplication can have a loop with 64/16=4 iterations and each iteration of the loop contains eight MAC operations. Every iteration of the loop produces 16 output data. This is illustrated in the following figure.

Figure 32: **Vectorization**



The mac16() intrinsic function to be used has the following interface.

```
v16acc48 mac16( v16acc48 acc,
    v32int16 xbuff,
    int xstart,
    unsigned int xoffsets,
    unsigned int xoffsets_hi,
    unsigned int xsquare,
    v16int16 zbuff,
    int zstart,
    unsigned int zoffsets,
    unsigned int zoffsets_hi,
    int zstep
)
```

The buffers contain parameters (start, offsets, square, and step) to compute the indexing into the buffers (vector registers). For details about the lane addressing scheme with these parameters, see MAC Intrinsics.

Coding with MAC intrinsics can be seen in the following section.

Coding with Intrinsics

We have analyzed how the function will be mapped into the AI Engine vector processor. Let us now have a look at the first version of the vectorized code.

```
inline void mac16_sub(input_window_int16* matA, v16int16 &buf_matB,
v16acc48 &acc, int i){
    v32int16 buf_matA = undef_v32int16(); // holds 32 elements of matA
    buf_matA=upd_w(buf_matA, 0, window_read_v16(matA));
    window_incr(matA,64);
    buf_matA = upd_w(buf_matA, 1, window_read_v16(matA));
    window_incr(matA,64);
    acc =
mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
}

void matmul_vec16(input_window_int16*  matA,
        input_window_int16* matB,
        output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB); // holds 16 elements of matB
    v16acc48 acc = null_v16acc48(); // holds acc value of Row * column dot
product

    for (unsigned int i=0;i<M/16;i++)  //M=64, Each iteration computes 16
outputs
    {
        acc=null_v16acc48();
        for(int j=0;j<16;j+=2){
            mac16_sub(matA,buf_matB,acc,j);
        }
        window_writeincr(matC,srs(acc,15));
        window_incr(matA,16);
    }
}
```

In the main function `matmul_vec16`, the loop produces 16 output data per iteration. In the outer loop body, there is an inner loop with eight iterations. In each iteration of the inner loop, an inline function `mac16_sub` is called. In the inline function, there is a `mac16` operation, with two loads of data for the MAC operation.

Inside `mac16_sub()`, `buf_matA` is declared as local variable and `buf_matB` and `acc` are declared as local variables in the main function. They are passed between functions by reference (or pointer). This ensures that only one identical vector exists for each variable. The function has one parameter that is used in the `mac16()` intrinsic as follows and this specific intrinsic (i=0) has been introduced in MAC Intrinsics.

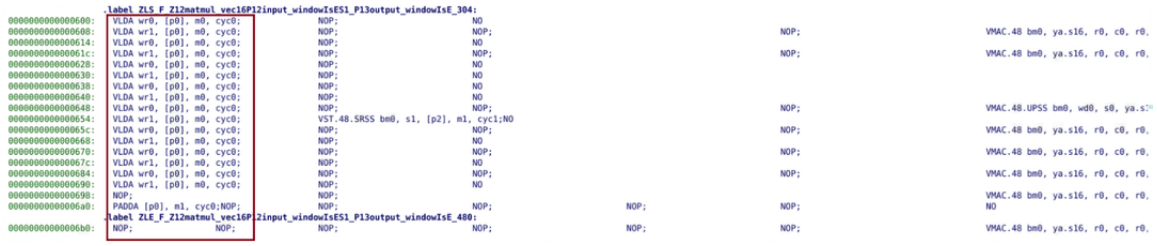
```
acc =
mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
```

At the end of each iteration of the loop, window pointer for the data is incremented by 16 (that is 16 rows for the matrix).

Note: While in the example, `inline` is used to force removal of the boundary of a function, sometimes it is helpful to retain the boundary of a function by `__attribute__ ((noinline)) void func(...)`.

The compiled code for the kernel can be found in the disassembly view in the debug perspective of the Vitis™ IDE. Note that a graph is needed for compiling the kernel with AI Engine tools. For more understanding about the assembly code in disassembly view, refer to Using Vitis IDE and Reports. For additional details on graph coding and Vitis IDE usage, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

Figure 33: **Assembly Code for the Loop**

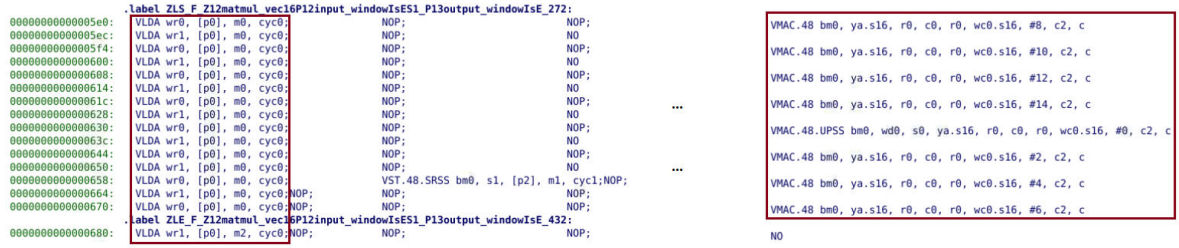


Note that the compiler automatically unrolls the inner loop and pipelines the outer loop. From the previous assembly code for the loop, each iteration requires 19 cycles. However, with one window interface of data (matA), the minimum cycle number required for eight MACs must be 16 (two loads of data per MAC). This degradation of performance is caused by unbalanced window pointer increment at the end of the loop. This can be resolved by pairing the last increment with the last MAC operation. The optimized code is as follows.

```
inline void mac16_sub(input_window_int16* matA, v16int16 &buf_matB,
v16acc48 &acc, int i,int incr_num){
    v32int16 buf_matA = undef_v32int16(); // holds 32 elements of matA
    buf_matA=upd_w(buf_matA, 0, window_read_v16(matA));
    window_incr(matA,64);
    buf_matA = upd_w(buf_matA, 1, window_read_v16(matA));
    window_incr(matA,incr_num);
    acc =
mac16(acc,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,i,0x0,0x0,1);
}

void matmul_vec16(input_window_int16*  matA,
        input_window_int16* matB,
        output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB); // holds 16 elements of matB
    v16acc48 acc = null_v16acc48(); // holds acc value of Row * column dot
product

    for (unsigned int i=0;i<M/16;i++)  //M=64, Each iteration computes 16
outputs
    {
        acc=null_v16acc48();
        for(int j=0;j<16;j+=2){
            int incr_num=(j==14)?80:64;
            mac16_sub(matA,buf_matB,acc,j,incr_num);
        }
        window_writeincr(matC,srs(acc,15));
    }
}
```

Note that the function `mac16_sub` has a new parameter `incr_num`. This parameter is for the pointer increment, which is different for the last function call in the inner loop. This increment number `80` for the last function call is to ensure that data in the next 16 rows is selected in the next iteration of the outer loop. Now the assembled code for the loop is as shown in following figure.

Figure 34: **Optimized Assembly Code for the Loop**



An iteration of the loop requires 16 cycles. This means that the compute bound for this kernel is 16*4=64 cycles per invocation. As seen in the previous section, the theoretical limit is 32 cycles per invocation. That is eight cycles for an iteration of the loop, which means that eight MAC operations must be compacted into eight cycles. Depending on the system performance requirements, this can be achieved by splitting the data input column by column into two window buffers, `matA_0` and `matA_1`. The data of the two windows is first to be read into two v16int16 vectors and concatenated into one v32int16 vector to be used in the `mac16` intrinsic. The code for the kernel is as follows.

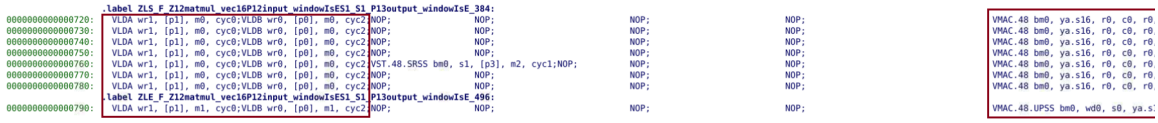
```
inline void mac16_sub_loads(input_window_int16* matA_0, input_window_int16*
matA_1, v16int16 &buf_matB, v16acc48 &acc, int i, int incr_num){
    v16int16 buf_matA0 = window_read_v16(matA_0);
    window_incr(matA_0,incr_num);
    v16int16 buf_matA1 = window_read_v16(matA_1);
    window_incr(matA_1,incr_num);
    acc =
mac16(acc,concat(buf_matA0,buf_matA1),0,0x73727170,0x77767574,0x3120,buf_mat
B,i,0x0,0x0,1);
}

void matmul_vec16(input_window_int16* __restrict matA_0,
        input_window_int16* __restrict matA_1,
        input_window_int16* __restrict matB,
        output_window_int16* __restrict matC){
    v16int16 buf_matB = window_read_v16(matB);
    for (unsigned int i=0;i<M/16;i++)  //M=64, Each iteration computes 16
outputs
    chess_prepare_for_pipelining
    {
        v16acc48 acc=null_v16acc48();
        for(int j=0;j<16;j+=2){
            int incr_num=(j==14)?80:64;
            mac16_sub_loads(matA_0,matA_1,buf_matB,acc,j,incr_num);
        }
        window_writeincr(matC,srs(acc,15));
    }
}
```

Note that two `v16int16` vectors, `buf_matA0` and `buf_matA1`, are defined and concatenated for the `mac16` intrinsic. Also note that `chess_prepare_for_pipelining` is added for the loop and `__restrict` keyword for the window interfaces to ensure that the loop is pipelined and window operations can be well optimized.

IMPORTANT! *The `__restrict` keyword cannot be used freely. Before using it, refer to Using the Restrict Keyword in AI Engine Kernels in the AI Engine Documentation flow of the Vitis Unified Software Platform Documentation (UG1416).*

The assembly code for the version of two window loads in a cycle is as follows.

Figure 35: **Assembly Code for Two Window Loads a Cycle**



Matrix multiplication

The following matrix multiplication example implements the equation:

$$C\ (64 \times 2) = A\ (64 \times 8) * B(8 \times 2)$$

The example assumes that the data for the matrices is stored in column major form and the data type for the matrices A and B is int16.

The first output column is computed as follows.

```
c0 = a0*b0 + a64*b1 + a128*b2 + a192*b3 + a256*b4 + a320*b5 + a384*b6 +
a448*b7
c1 = a1*b0 + a65*b1 + a129*b2 + a193*b3 + a257*b4 + a321*b5 + a385*b6 +
a449*b7
c2 = a2*b0 + a66*b1 + a130*b2 + a194*b3 + a258*b4 + a322*b5 + a386*b6 +
a450*b7
c3 = a3*b0 + a67*b1 + a131*b2 + a195*b3 + a259*b4 + a323*b5 + a387*b6 +
a451*b7
                                    …
c60 = a60*b0 + a124*b1 + a188*b2 + a252*b3 + a316*b4 + a380*b5 + a444*b6 +
a508*b7
c61 = a61*b0 + a125*b1 + a189*b2 + a253*b3 + a317*b4 + a381*b5 + a445*b6 +
a509*b7
c62 = a62*b0 + a126*b1 + a190*b2 + a254*b3 + a318*b4 + a382*b5 + a446*b6 +
a510*b7
c63 = a63*b0 + a127*b1 + a191*b2 + a255*b3 + a319*b4 + a383*b5 + a447*b6 +
a5111*b7
```

The second output column is computed as follows.

```
c64 = a0*b8 + a64*b9 + a128*b10 + a192*b11 + a256*b12 + a320*b13 + a384*b14
+ a448*b15
c65 = a1*b8 + a65*b9 + a129*b10 + a193*b11 + a257*b12 + a321*b13 + a385*b14
+ a449*b15
c66 = a2*b8 + a66*b9 + a130*b10 + a194*b11 + a258*b12 + a322*b13 + a386*b14
+ a450*b15
c67 = a3*b8 + a67*b9 + a131*b10 + a195*b11 + a259*b12 + a323*b13 + a387*b14
+ a451*b15
                                    …
c124 = a60*b8 + a124*b9 + a188*b10 + a252*b11 + a316*b12 + a380*b13 +
a444*b14 + a508*b15
c125 = a61*b8 + a125*b9 + a189*b10 + a253*b11 + a317*b12 + a381*b13 +
a445*b14 + a509*b15
c126 = a62*b8 + a126*b9 + a190*b10 + a254*b11 + a318*b12 + a382*b13 +
a446*b14 + a510*b15
c127 = a63*b8 + a127*b9 + a191*b10 + a255*b11 + a319*b12 + a383*b13 +
a447*b14 + a5111*b15
```

Kernel Coding Bounds

In this example, a total of 1024 int16 x int16 multiplications are required for computing 128 output value. Given that 32 16-bit multiplications can be performed per cycle in an AI Engine, the compute bound for the kernel is as follows.

```
Compute bound = 32 cycles / invocation
```

Matrix B can be stored in the vector register because it is only 16*16-bit =256 bits. It does not need to be fetched from the AI Engine data memory or tile interface for each MAC operation. Considering the data “a” needed for computation, there are total 64*8*2=1024 bytes to be fetched from memory. Given that AI Engine allows two 256 bits (32 bytes) loads per cycle, the memory bound for the kernel is as follows.

```
Memory bound = 1024 / (2*32) = 16 cycles / invocation
```

It is seen that the compute bound is larger than the memory bound. Hence the purpose of vectorization can be to achieve the theoretical limit of MAC operations in the vector processor.

Vectorization

The scalar reference code for this matrix multiplication example is shown as follows. Note that the data is stored in columns.

```
void matmul_mat8_scalar(input_window_int16* matA,
        input_window_int16* matB,
        output_window_int16* matC){

    for(int i=0; i<M; i++){//M=64
        for(int j=0;j<L;j++){//L=2
            int temp = 0 ;
            for(int k=0; k<N; k++){//N=8
                temp += window_read(matA)*window_readincr(matB);//B is
```

```
circular buffer, size N*L
                window_incr(matA,64); //Jump of 64 elements to access the
next element of the same row
            }
            window_write(matC,(int16_t)(temp>>15)) ;
            window_incr(matC,64); //Jump to the next column
        }
        window_incr(matA,1); //Jump of one element for moving to the next
row.
        window_incr(matC,1); //Jump to the next row
    }
}
```

As analyzed in the previous example, Matrix Vector Multiplication, `mac16` intrinsic is the best choice for computing 16 lanes together because 16 int16 from a column can be loaded at once. To compute 16 output data in a column, four `mac16` operations are needed. The same data in vector "a" is used twice to compute the data for two output columns. Thus, two columns of data can be loaded and two `mac16` used for accumulations to the two output columns. These two loads and two MACs are repeated four times to get the results of two output columns. This method is shown in the following pseudo-code.

```
C_[0:15,0] = A_[0:15,0:1]*B_[0:1,0]
C_[0:15,1] = A_[0:15,0:1]*B_[0:1,1]

C_[0:15,0]+= A_[0:15,2:3]*B_[2:3,0]
C_[0:15,1]+= A_[0:15,2:3]*B_[2:3,1]

C_[0:15,0]+= A_[0:15,4:5]*B_[4:5,0]
C_[0:15,1]+= A_[0:15,4:5]*B_[4:5,1]

C_[0:15,0]+= A_[0:15,6:7]*B_[6:7,0]
C_[0:15,1]+= A_[0:15,6:7]*B_[6:7,1]
```

In the previous code, each "*" denotes a MAC operation. `C_[0:15,0]` and `C_[0:15,1]` denote two output columns that are accumulated separately. `A_[0:15,0:1]` denotes the column 0 and 1, and each column has 16 elements. `B_[0:1,0]` denotes column 0 with 2 elements. There will be a loop for the code in the real vectorized code because there are 64 output rows. The `mac16` intrinsic function to be used has following interface.

```
v16acc48 mac16    (    v16acc48     acc,
    v64int16     xbuff,
    int     xstart,
    unsigned int     xoffsets,
    unsigned int     xoffsets_hi,
    unsigned int     xsquare,
    v16int16     zbuff,
    int     zstart,
    unsigned int     zoffsets,
    unsigned int     zoffsets_hi,
    int     zstep
)
```

The buffers contain parameters (start, offsets, square, and step) to compute the indexing into buffers (vector registers). For details about the lane addressing scheme with these parameters, see MAC Intrinsics.

Note that the `mac16` intrinsic function prototype is different with the one introduced in the previous matrix vector multiplication example. The `xbuff` here is `v64int16` which allows two sets of data to be stored and used in an interleaved way.

Coding with MAC intrinsics can be seen in the following section.

Coding with Intrinsics

We have analyzed how the function will be mapped into the AI Engine vector processor. Let us now have a look at the vectorized code.

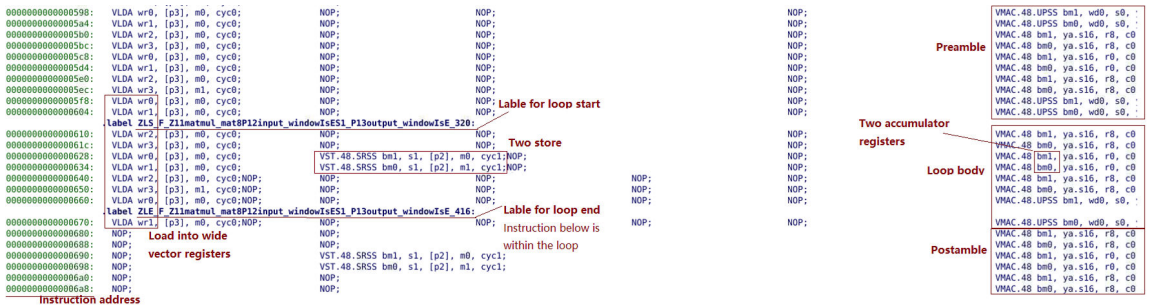
```
void matmul_mat8(input_window_int16* matA,
        input_window_int16* matB,
        output_window_int16* matC){

    v16int16 buf_matB = window_read_v16(matB);

    v64int16 buf_matA = undef_v64int16();
    buf_matA=upd_w(buf_matA,0,window_read_v16(matA));
    window_incr(matA,64);
    buf_matA=upd_w(buf_matA,1,window_read_v16(matA));
    window_incr(matA,64);

    for (unsigned int i=0;i<M/16;i++)  //M=64, Each iteration computes 16
outputs
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        v16acc48 acc0=null_v16acc48();//For first output column
        v16acc48 acc1=null_v16acc48();//For second output column

        acc0 =
mac16(acc0,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,0,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,2,window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,8,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,3,window_read_v16(matA));
        window_incr(matA,64);

        acc0 =
mac16(acc0,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,2,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,0,window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,10,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,1,window_read_v16(matA));
        window_incr(matA,64);

        acc0 =
mac16(acc0,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,4,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,2,window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,0,0x73727170,0x77767574,0x3120,buf_matB,12,0x0,0x0,1);
        buf_matA=upd_w(buf_matA,3,window_read_v16(matA));
        window_incr(matA,80);//point to next 16 rows

        acc0 =
```

```
mac16(acc0,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,6,0x0,0x0,1);
        window_write(matC,srs(acc0,15));
        window_incr(matC,64);
        buf_matA=upd_w(buf_matA,0,window_read_v16(matA));
        window_incr(matA,64);
        acc1 =
mac16(acc1,buf_matA,32,0x73727170,0x77767574,0x3120,buf_matB,14,0x0,0x0,1);
        window_write(matC,srs(acc1,15));
        window_incr(matC,80);//point to next 16 rows
        buf_matA=upd_w(buf_matA,1,window_read_v16(matA));
        window_incr(matA,64);
    }
}
```

In the previous code, `buf_matB` is for matrix B and it is loaded outside the loop. `buf_matA` is for matrix A and two sets of A are stored in lower and higher parts. When `mac16` has the value "0" for `xstart`, the lower part of `buf_matA` is used. When `mac16` has the value "32" for `xstart`, the higher part of `buf_matA` is used. `acc0` and `acc1` are the accumulated values for two output columns.

Note that `buf_matA` is preloaded before the loop. In the loop, the loads with window buffer pointer increment, MAC operations and the stores are interleaved. To understand how the `mac16()` intrinsic works, refer to MAC Intrinsics. The assembled code for the loop is as shown in following figure.

Figure 36: **Assembly Code for the Loop**



From the previously assembled code, it is seen that there is a MAC operation and a load operation in every cycle of the loop. Wide registers `wr0`, `wr1`, `wr2`, and `wr3` are used for `buf_matA`. Accumulator registers `bm0` and `bm1` are used for the two accumulated results.

Keys to make the loop be well pipelined are as follows:

- Preload the data into vector registers before the loop start.
- Interleave data loads, MAC operations, data stores in the loop body.
- Use wide input data vector register (`v64int16` in the example) to make data load and MAC operation perform on different parts of the vector register.
- Use multiple accumulator registers and reuse input data for multiple outputs.

- Data load and buffer pointer increment come in pairs. This applies for data store and buffer pointer increments as well.

Multiple Kernels Coding Example: FIR Filter

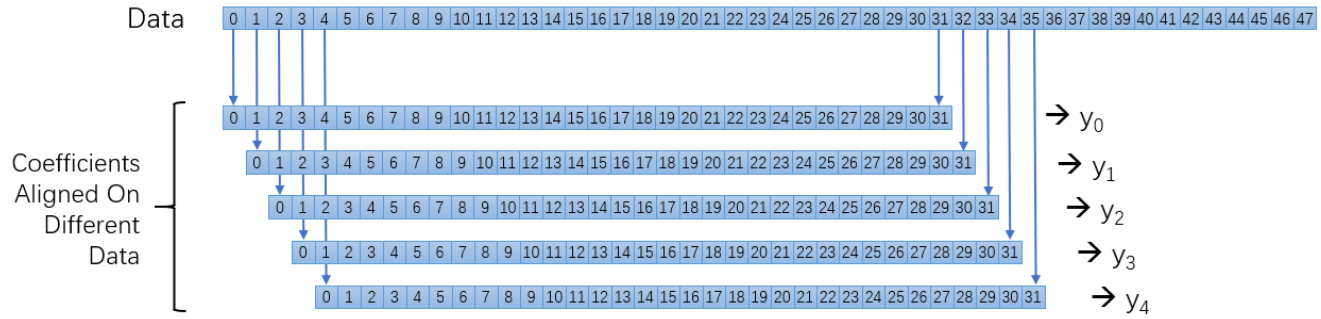
In this section, the filter design is used to demonstrate how to split the application into multiple AI Engines when an application exceeds the computational capacity of a single AI Engine. A finite impulse response (FIR) filter is a filter whose impulse response (or response to any finite length input) is of finite duration.

$$y_n = \sum_{k=0}^{N-1} C_k, x_{n+k}$$

***Note*:** Mathematically, this equation is a correlation instead of a convolution. This is the way computation is organized within the AI Engine. For this to become a convolution (FIR filtering), the coefficients have to be stored in vector C_K in the reverse order. This is not a problem for symmetric filters.

In the previous equation, N denotes the taps to be used to calculate each output. The calculation process when a 32 taps filter is used as an example is shown in the following figure. int16 complex types for data and coefficient are also used as an example.

Figure 37: **FIR Filter**



1 Gsps Implementation with Cascade Stream

The AI Engine vector unit supports 8 MACs per cycle for cint16 multiply-accumulate cint16 types. If a four lane implementation of mul4/mac4 intrinsics is adopted, then there will be two complex operations on each lane.

$$\begin{cases} y_0 = c_0.d_0 + c_1.d_1 \\ y_1 = c_0.d_1 + c_1.d_2 \\ y_2 = c_0.d_2 + c_1.d_3 \\ y_3 = c_0.d_3 + c_1.d_4 \end{cases}$$

16 mac4() are needed for computing four outputs because each output requires 32 complex MACs. This means, to compute four outputs, 16 cycles using an AI Engine are required. So the sample rate of an AI Engine (assuming it runs at 1 GHz) would be as follows.

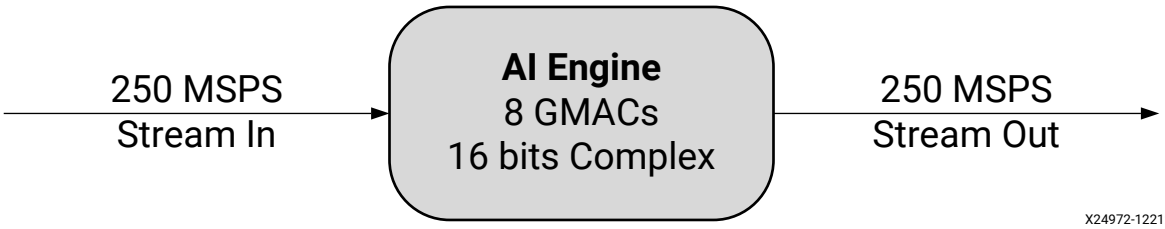
$$4\ \text{Gsps}/16 = 0.25\ \text{Gsps} = 250\ \text{Msps}$$

This calculates the compute bound of an AI Engine. However, the memory bound to see if that sample rate can be met still needs to be considered. Let us assume that only one stream input and one stream output are used for data transfer and the coefficients are stored in the AI Engine internal memory. The stream interface of an AI Engine supports 32 bits per cycle. It is capable of transferring one sample of data every cycle. Thus, the sample rate from the data transferring view is as follows.

$$1\ \text{sample/cycle} * 1\ \text{GHz} = 1\ \text{Gsps}$$

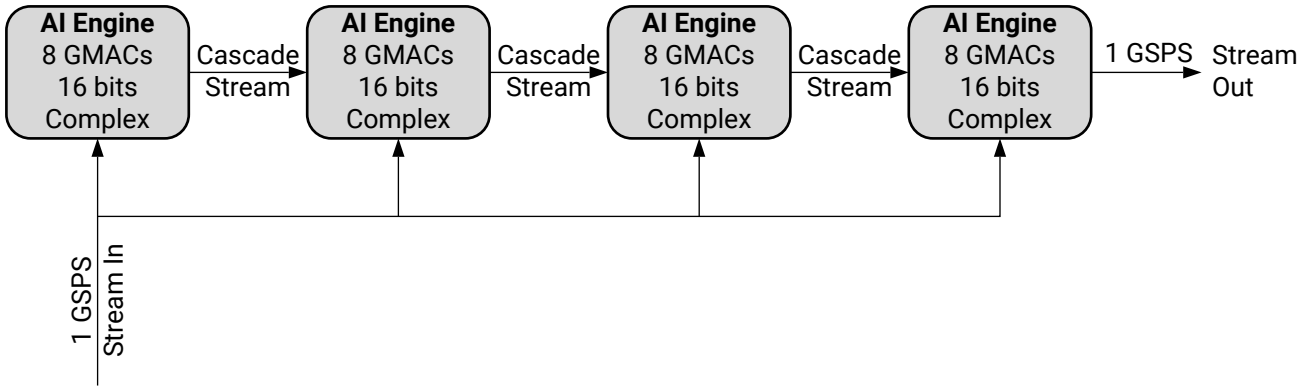
It is larger than compute bound, which is 250 Msps. So an AI Engine implementation will operate at 250 Msps.

Figure 38: **One AI Engine FIR Filter Realization**



Based on the calculations, it is possible to achieve 1 Gsps via a stream input and output stream interface. If the MAC operations of a single kernel implementation are split into four kernels, $4*250\text{Msps} = 1\ \text{Gsps}$, compute throughput can be achieved. Those four kernels are connected through cascade streaming. Therefore, the AI Engine compute bound matches AI Engine interface throughput.

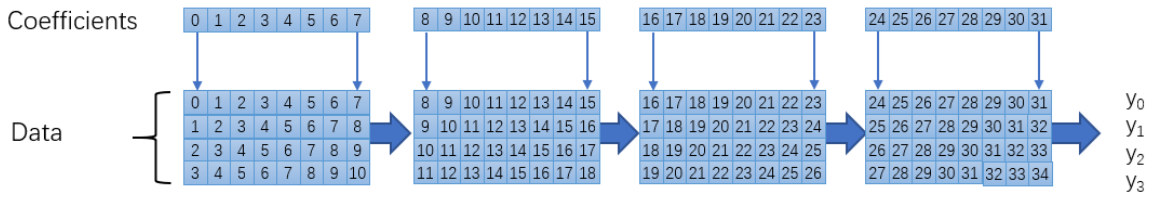
Figure 39: **1 Gsps Implementation with Four Cascaded Kernels**



Coding with Intrinsics

The four kernels in the 1 Gsps implementation can have different sets of coefficients and cascade streams between them. An implementation is shown in the following figure.

Figure 40: **Four Kernels with Split Coefficient and Cascade Stream**



Input data flows from stream to these four kernels. However, the second kernel will discard the first eight input data. The third kernel will discard the first 16 input data. Similarly, the fourth kernel will discard the first 24 input data.

The code for the first kernel is as follows.

```
#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core0(
    input_stream_cint16 * sig_in,
    output_stream_cacc48 * cascadeout)
{
    const cint16_t * __restrict coeff = eq_coef0;
    const v8cint16 *coef_  =  (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;
    v4cacc48 acc;
```

```
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer
power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        acc  = mul4(buff, 0 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 2 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 2, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 4 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 6 , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);

        acc  = mul4(buff, 4 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 6 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 3, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 8 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 10, 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);

        acc  = mul4(buff, 8  , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 10 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 0, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 12 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 14 , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);

        acc  = mul4(buff, 12 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 14 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 1, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 0  , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 2  , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);
    }
    delay_line = buff;
}

void fir_32tap_core0_init()
{
    // Drop samples if not first block
    int const Delay = 0;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }

};
```

Note that the function, `fir_32tap_core0_init`, is going to be the initialization function for the AI Engine kernel, `fir_32tap_core0`, which is only executed once at the kernel start. The purpose of this initialization function is to discard the unnecessary samples to align the input stream.

Similarly, the function, `fir_32tap_core1_init`, is going to be the initialization function for the AI Engine kernel, `fir_32tap_core1`, in the following codes. Same applies for the initialization functions, `fir_32tap_core2_init` and `fir_32tap_core3_init`.

The second kernel code is as follows.

```
#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core1(
    input_stream_cint16 * sig_in,
    input_stream_cacc48 * cascadein,
    output_stream_cacc48 * cascadeout)
{
    const cint16_t * __restrict coeff = eq_coef1;
    const v8cint16 *coef_  =  (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;
    v4cacc48 acc;
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer
power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    chess_loop_range(4,)
    {
        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 0 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 2 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 2, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 4 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 6 , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);

        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 4 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 6 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 3, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 8 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 10, 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);

        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 8  , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 10 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 0, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 12 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 14 , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);

        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 12 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 14 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 1, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 0  , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 2  , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(cascadeout,acc);
    }
    delay_line = buff;
}

void fir_32tap_core1_init()
{
    // Drop samples if not first block
```

```
    int const Delay = 8;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }
};
```

The third kernel is similar to the second one. The last kernel is as follows.

```
#include <adf.h>
#include "fir_32tap.h"
// buffer to keep state
static v16cint16 delay_line;

void fir_32tap_core3(
    input_stream_cint16 * sig_in,
    input_stream_cacc48 * cascadein,
    output_stream_cint16 * data_out)
{
    const cint16_t * __restrict coeff = eq_coef3;
    const v8cint16 *coef_  =  (v8cint16 const*)coeff;
    const v8cint16 coe = *coef_;

    v16cint16 buff = delay_line;

    v4cacc48 acc;

    set_rnd(2);
    set_sat();
    const unsigned LSIZE = (samples/4/4); // assuming samples is integer
power of 2 and greater than 16

    for (unsigned int i = 0; i < LSIZE; ++i)
    chess_prepare_for_pipelining
    chess_loop_range(4,)
        {
        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 0 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 2 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 2, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 4 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 6 , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));

        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 4 , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 6 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 3, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 8 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 10, 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));

        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 8  , 0x3210, 1,  coe, 0, 0x0000, 1);
        acc  = mac4(acc, buff, 10 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 0, readincr_v4(sig_in));
        acc  = mac4(acc, buff, 12 , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc  = mac4(acc, buff, 14 , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));

        acc = readincr_v4(cascadein);
        acc  = mac4(acc, buff, 12 , 0x3210, 1,  coe, 0, 0x0000, 1);
```

```
        acc   = mac4(acc, buff, 14 , 0x3210, 1,  coe, 2, 0x0000, 1);
        buff = upd_v(buff, 1, readincr_v4(sig_in));
        acc   = mac4(acc, buff, 0  , 0x3210, 1,  coe, 4, 0x0000, 1);
        acc   = mac4(acc, buff, 2  , 0x3210, 1,  coe, 6, 0x0000, 1);
        writeincr_v4(data_out,srs(acc,shift));
    }
        delay_line = buff;
}

void fir_32tap_core3_init()
{
    // Drop samples if not first block
    int const Delay = 24;
    for (int i = 0; i < Delay; ++i)
    {
        get_ss(0);
    }
};
```

The graph code is as follows.

```
#include <adf.h>
#include "kernels.h"
using namespace adf;
class firGraph : public graph {
    public:
    kernel k0,k1,k2,k3;
    input_port in0123;
    output_port out;
    firGraph()
    {
        k0 = kernel::create(fir_32tap_core0);
        runtime<ratio>(k0) = 0.9;
        source(k0) = "fir_32tap_core0.cpp";
        connect<stream> n0(in0123,k0.in[0]);

        k1 = kernel::create(fir_32tap_core1);
        runtime<ratio>(k1) = 0.9;
        source(k1) = "fir_32tap_core1.cpp";
        connect<stream> n1(in0123,k1.in[0]);
        connect<cascade> (k0.out[0],k1.in[1]);

        k2 = kernel::create(fir_32tap_core2);
        runtime<ratio>(k2) = 0.9;
        source(k2) = "fir_32tap_core2.cpp";
        connect<stream> n2(in0123,k2.in[0]);
        connect<cascade> (k1.out[0],k2.in[1]);

        k3 = kernel::create(fir_32tap_core3);
        runtime<ratio>(k3) = 0.9;
        source(k3) = "fir_32tap_core3.cpp";
        connect<stream> n3(in0123,k3.in[0]);
        connect<cascade> (k2.out[0],k3.in[1]);
        connect<stream> (k3.out[0],out);

        initialization_function(k0) = "fir_32tap_core0_init";
        initialization_function(k1) = "fir_32tap_core1_init";
        initialization_function(k2) = "fir_32tap_core2_init";
        initialization_function(k3) = "fir_32tap_core3_init";
    };
};
```

The kernels connected through cascade streams are expected to operate synchronously. Conflicts in cascade streams can stall the kernels. Loops in the kernels must have input data available to run smoothly. Hence it is important that the input stream arrives at the appropriate time for each kernel. The input stream stall (if any) can be resolved by adding a large enough FIFO to the net connecting to the AI Engine kernels. For example:

```
fifo_depth(n0)=175;
fifo_depth(n1)=150;
fifo_depth(n2)=125;
fifo_depth(n3)=100;
```

Note that different FIFO depths are specified in the previous example to prevent auto FIFO merge which can occur when a common FIFO depth is used for all nets.

For the purpose of saving FIFO resources, individual FIFO depths can be set by looking at when the event `CORE_INSTREAM_WIDE` occurs for each kernel. The earlier the event occurs, the deeper the FIFO needs to be. For example:

```
fifo_depth(n0)=45;
fifo_depth(n1)=33;
fifo_depth(n2)=23;
fifo_depth(n3)=10;
```

For additional details about coding on graph, refer to the *Versal ACAP AI Engine Programming Environment User Guide* (UG1076).

XILINX

Appendix A

Additional Resources and Legal Notices

Xilinx Resources

For support resources such as Answers, Documentation, Downloads, and Forums, see Xilinx Support.

Documentation Navigator and Design Hubs

Xilinx® Documentation Navigator (DocNav) provides access to Xilinx documents, videos, and support resources, which you can filter and search to find information. To open DocNav:

- From the Vivado® IDE, select **Help → Documentation and Tutorials**.
- On Windows, select **Start → All Programs → Xilinx Design Tools → DocNav**.
- At the Linux command prompt, enter `docnav`.

Xilinx Design Hubs provide links to documentation organized by design tasks and other topics, which you can use to learn key concepts and address frequently asked questions. To access the Design Hubs:

- In DocNav, click the **Design Hubs View** tab.
- On the Xilinx website, see the Design Hubs page.

Note: For more information on DocNav, see the Documentation Navigator page on the Xilinx website.

References

These documents provide supplemental material useful with this guide:

1. *Versal ACAP AI Engine Architecture Manual* (AM009)
2. *Versal ACAP AI Engine Programming Environment User Guide* (UG1076)
3. *Versal ACAP AI Engine Intrinsics Documentation* (UG1078)

Please Read: Important Legal Notices

Copyright