Digital Down-conversion Chain Implementation on AI Engine

XAPP1351 (v1.0) February 15, 2021

Summary

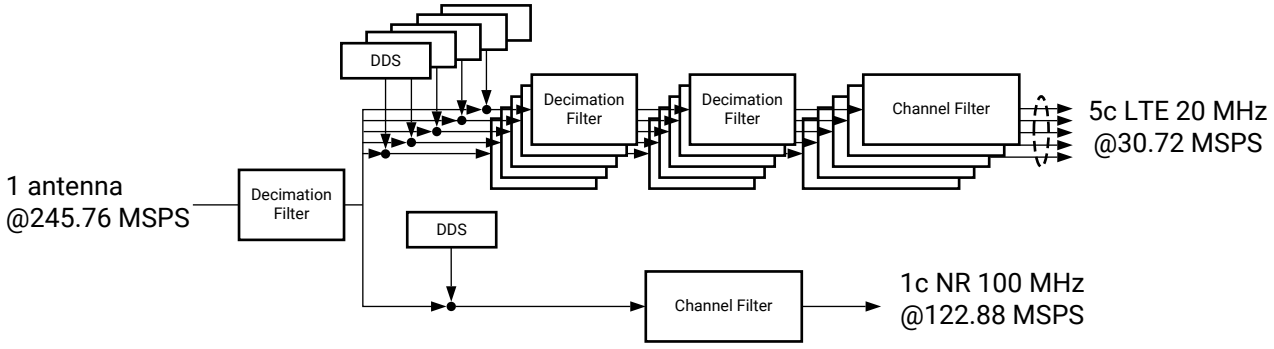
This application note focuses on the design of a Digital Down-Converter (DDC) chain using AI Engine technology in Xilinx® Versal™ AI Core devices. The AI Engine is designed for high-density multiply-and-accumulate (MAC) computation that is typically seen in high-performance signal processing applications. The DDC chain is responsible for the extraction of carriers from a composite signal at a high sample rate, which requires intense computation as well as high flexibility to support various carrier configurations. This application note shows an innovative method of mapping the DDC functions to the AI Engine array by leveraging the unique architecture of Versal devices to deliver high performance and efficiency with a small memory footprint. The design methodology illustrated by this application note is applicable to a wide range of use cases including but not limited to wireless signal processing.

Download the reference design files for this application note from the Xilinx website. For detailed information about the design files, see Reference Design.

Introduction

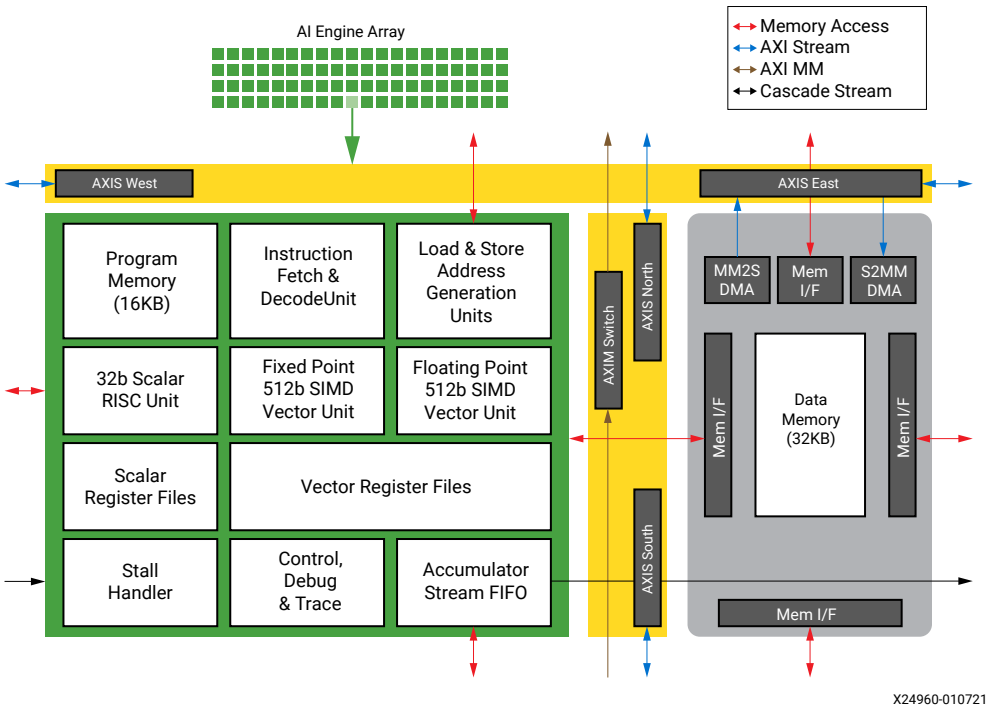
The DDC chain is a key component in wireless communication systems. It is part of the receive path that links the baseband processing and radio front end. A DDC performs down-conversion on the input signal to the baseband sample rate. For instance, in a 100 MHz 5G New Radio (NR) system the sample rate of the radio front end is 245.76 Mega samples per second (MSPS), whereas the nominal sample rate of the baseband signal is 122.88 MSPS. For these cases, sample rate conversion from 245.76 MSPS to 122.88 MSPS must be performed in the DDC. Furthermore, 5G NR signals have a narrow transition band which calls for a long channel filter to offer good passband flatness and steep stop band attenuation. The following figure shows a typical DDC implementation for a 100 MHz 5G NR system which can support 5G NR or Long-Term-Evolution (LTE) carriers. Because the nominal sample rates of 20 MHz LTE and 100 MHz 5G NR carriers differ by a factor of four, two filter chains have to be instantiated as shown in the following figure.

Figure 1: **DDC Implementation for 5G NR and 4G LTE**



The Versal AI Core series has an array of AI Engines that are optimized for wireless radio applications supporting multiple numerologies and carrier configurations. The array consists of a number of AI Engines, each comprising a 32-bit scalar RISC processor, fixed and floating-point vector units, data memory, and interconnect. In each AI Engine the vector unit is capable of 32 real-by-real 16-bit MAC operations in one 1 GHz+ clock cycle, and memory load and store units that can read 512 bits data from and write 256 bits into local memory every clock cycle. There are hundreds of such AI Engines in one single chip that are suitable for compute-intensive applications such as wireless radio.

Figure 2: **Block Diagram of One AI Engine Tile**



This application note provides a method to design a flexible, scalable, and resource-efficient filter chain that runs on AI Engine in Versal AI Core devices. It also shows advanced techniques for mapping complex functions to the AI Engine array by leveraging the unique architecture of Versal devices for high performance and efficiency. This application note uses digital down-conversion as an example for illustration purpose, but this methodology is applicable to a wide range of applications including but not limited to wireless signal processing.

DDC Chain Architecture

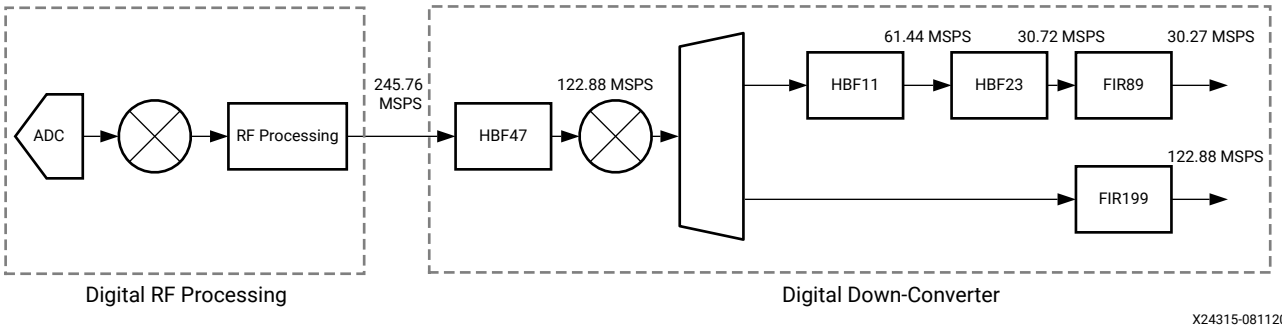
This section explains the architecture of the DDC chain and how it works. The input to the DDC chain is a composite signal comprising of one or more radio or intermediate frequency (RF or IF) carriers and the output is one or more carriers at the baseband sample rate for further processing. The following procedures are performed in the DDC chain.

1. Mixing to shift the signal spectrum from the selected carrier frequencies to the baseband frequency.
2. Decimation to reduce the sample rate.
3. Filtering to remove adjacent channels, minimize aliasing, and maximize the received signal-to-noise ratio (SNR).

An example architecture for a DDC chain is shown in the following figure. An analog-to-digital converter (ADC) samples the analog signal and feeds it into the DDC processing chain. Optionally, there is an initial frequency translation (to shift the center frequency from passband to baseband), RF processing, and additional filters (decimation) that can be performed prior to the DDC function, shown in the following figure as the Digital RF Processing block.

This application note covers the functions of the main mixer and filters shown in the Digital Down Converter block in the following figure. The half band filter (HBF47) decimates the input signal by two. If it is a 4G five carrier (5c) LTE 20 MHz signal, then five channels are extracted by mixers with selected carrier frequencies and followed by a filter chain (HBF11/HBF23/FIR89) to reduce the sample rate to 30.72 MSPS. If it is a 5G NR one carrier (1c) 100 MHz signal, then the signal coming from the mixer goes to FIR199 directly. Because 5G NR has a much narrower transition band than that of LTE, the 5G NR channel filter has longer taps. In this case study, a channel filter with 199 taps is employed for the 5G NR 100 MHz carrier.

Figure 3: **DDC Block Diagram Example**



AI Engine Utilization Estimation

The AI Engine processes data block by block and uses a data structure called window to describe one block of input or output data. The window size, which is the number of samples in each block of input or output data, represents a tradeoff between efficiency and processing latency. Long windows lead to high efficiency, but the latency increases proportionally to the window size. Sometimes a short latency is preferable at a loss of a 5-10% AI Engine processing efficiency.

For example, in this application note, the input window size is set to 512 samples to limit the latency to within 2.1 µs. The window sizes and sample rates of DDC filters are listed in the following table.

Table 1: **Window Size and Sample Rate of DDC Filters**

Filter	Input Sample Rate (MSPS)	Output Sample Rate (MSPS)	Input Window	Output Window
HB47	245.76	122.88	512	256
HB11	122.88	61.44	256	128
FIR199	122.88	122.88	256	256
HB23	61.44	30.72	128	64
FIR89	30.72	30.72	64	64
Mixer	122.88	122.88	256	1280/5x carriers

A cycle budget is the number of instruction cycles a function can take to compute a block of output data, given by:

$$\text{Cycle budget} = (\text{Input window} \times \text{AI Engine system clock}) / (\text{Input sample rate}) \qquad \textit{Equation 1}$$

At a 1 GHz AI Engine clock in the lowest speed-grade device, the processing of 512 samples at 245.76 MSPS has a cycle budget of 2083 cycles.

Suppose every output needs P 16-bit-real by 16-bit-real multiplications. The AI Engine can compute 32 such real-by-real multiplications every cycle. For an ideal implementation, the utilization lower boundary is given by:

$$\text{Utilization}_{(\text{lower boundary})} = (P \times \text{Output\_window\_size}) / (32 \times \text{Cycle\_budget}) \qquad \textit{Equation 2}$$

Take FIR199 as an example. FIR199 has 199 real symmetric filter taps and it takes 100 16 bit-complex by 16 bit-real multiplications to compute each output. Therefore, every output of FIR199 needs 200 16-bit-real by 16-bit-real multiplications at 122.88 MSPS, and the utilization lower boundary is given by 200 cycles × 256 samples / (32 × 2083 cycle budget) = 76.8%. Similarly, the utilization lower bounds of other DDC filters are calculated and listed in the following table.

Table 2: **AI Engine Utilization Lower Bound Analysis**

Filter	Input Window Size	Output Window Size	Number of Taps	Number of MACs/ Output	Utilization / Instance	Number of Inst	Utilization Lower Bound
FIR199	256	256	199	200	76.8%	1	76.8%
FIR89	64	64	89	96	9.3%	5	46.5%
HB47	512	256	47	32	12.3%	1	12.3%
HB11	256	128	11	8	1.6%	5	8%
HB23	128	64	23	16	1.6%	5	8%
Mixer[1]	256	1280	-	8	23%	1	23%
						Total	174.6%

Notes:

1. To support configurable carrier frequency at run time, an on-line DDS calculation consumes extra 180 cycles in each mixer kernel execution.

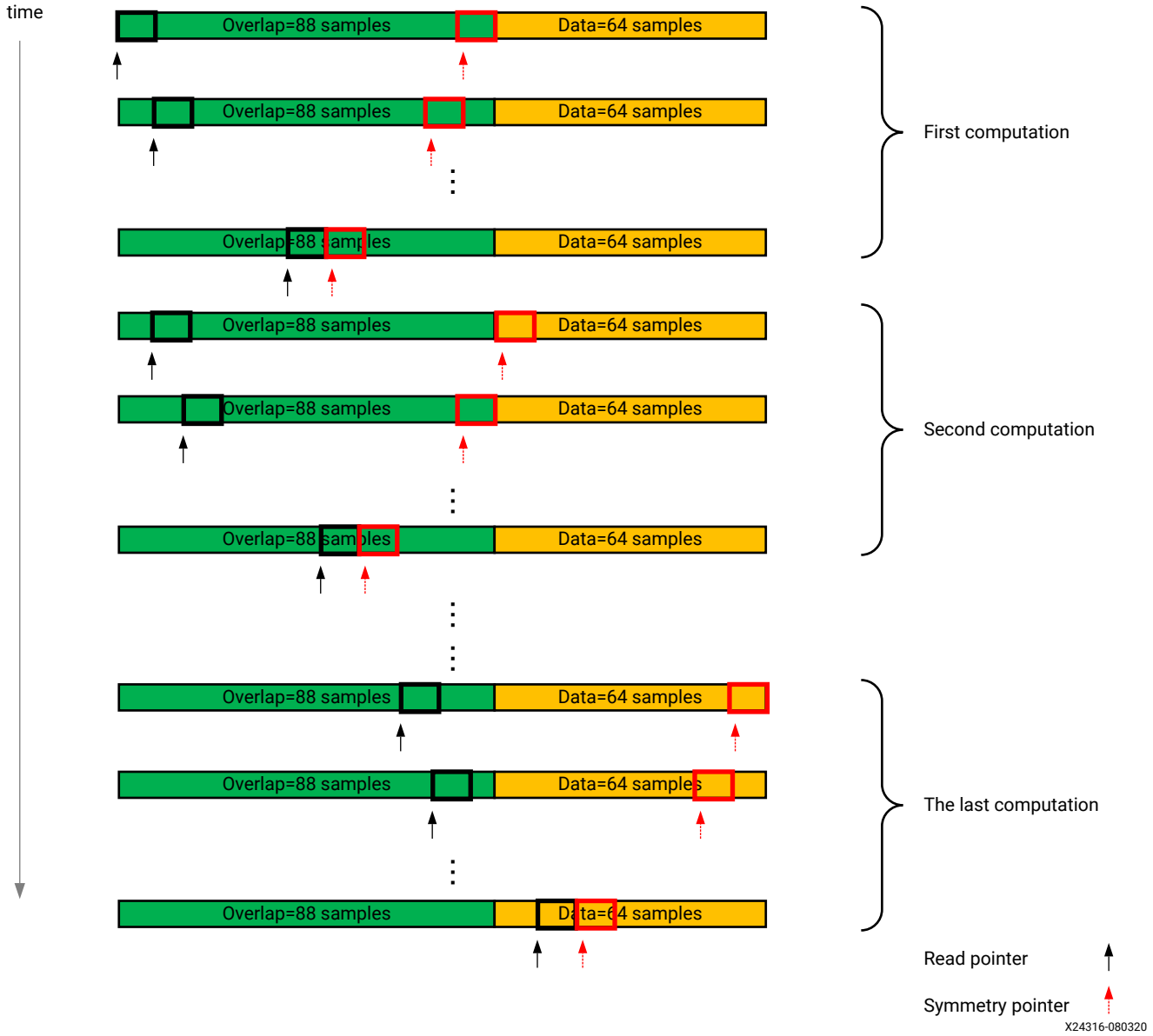
Although in theory this DDC can be implemented on two AI Engines with 87.3% utilization each, such high utilization requires very long windows and undesirable latency. One method to reduce the utilization is to take advantage of the fact that 5G NR and 4G LTE carriers do not co-exist in this case and the filters for unused carriers can be disabled during run time, depending on the carrier configuration. Detailed analysis and explanation is provided in the following sections.

Design Challenges

In this section, a traditional method of filter design on the AI Engine is reviewed and the challenges to the DDC chain implementation are analyzed.

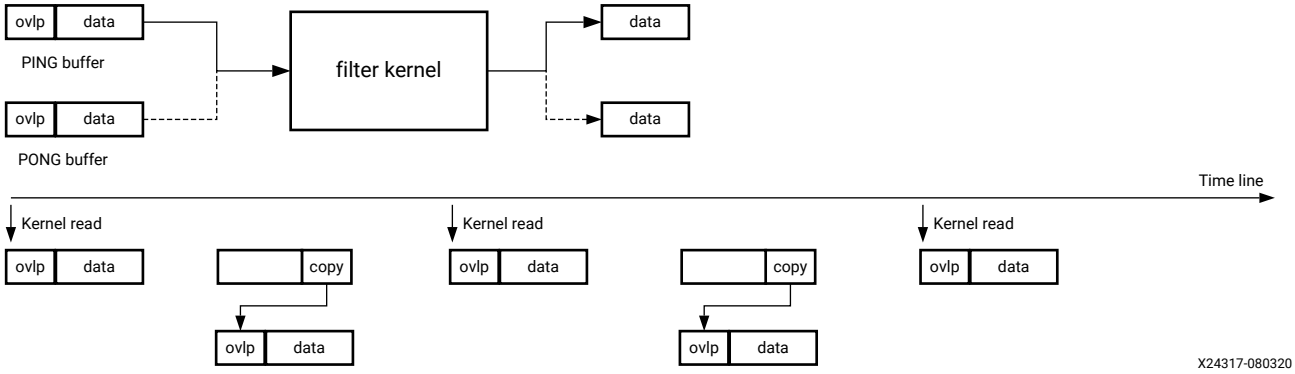
In a traditional AI Engine design, the input window contains the incoming data block and the Vitis™ unified software platform automatically puts an overlap in front of it. For example, a filter with 89 taps might need an overlap of 88 samples as shown in the following figure. In this case, the physical buffer size is assigned to 88 + 64 samples. The following figure shows the pointer movement during the processing of the data. Note that a prerequisite for the traditional filter to work is that the overlap has to be placed directly in front of the data in a circular buffer.

Figure 4: **Traditional FIR89 Pointers Behavior**



When the data comes from another AI Engine or the PL, a ping-pong buffer is implemented for the input window of the kernel, and the overlap is automatically copied to the front of the data before each run of the filter function. This memory copy operation is illustrated in the following figure. When a filter has many taps that are more than the size of data window, copying the overlap for every execution of the filter function can consume a considerable amount of time.

Figure 5: **Conventional Filter Kernel Behavior**

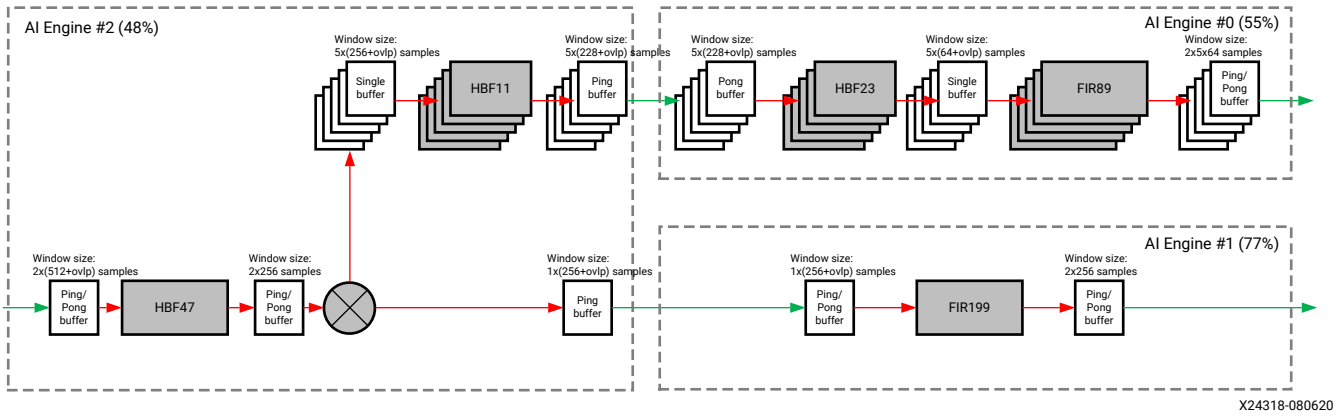


Note that the overlap is determined by the filter taps and its size must be a multiple of 256 bits for maximum memory access efficiency. The size of overlap is subject to the following equation:

$$OVLP_{size} = ceil(((Taps + 7)/8) \times 8) \quad \textit{Equation 3}$$

The size of overlap and window must be specified at compilation time, which means that filters of different taps cannot share data buffers. Using traditional filter architecture, it is necessary to partition the DDC into three AI Engines for each antenna as shown in the following figure.

Figure 6: **DDC System Partition (Three AI Engines)**



Though it is possible to bypass some DDC filters depending on the carrier configuration, the traditional filter design approach under consideration has the following challenges:

1. The memory footprint is very high because every filter of every channel needs a buffer for input samples + overlap.
2. The number of output windows is high because they cannot be shared by multiple filters.
3. The copying overlap of long channel filters, that is, FIR89 and FIR199, leads to considerable efficiency loss.
4. The AI Engine efficiency of the long half-band decimation filter is low because of the difficulty in achieving a perfect inner loop.

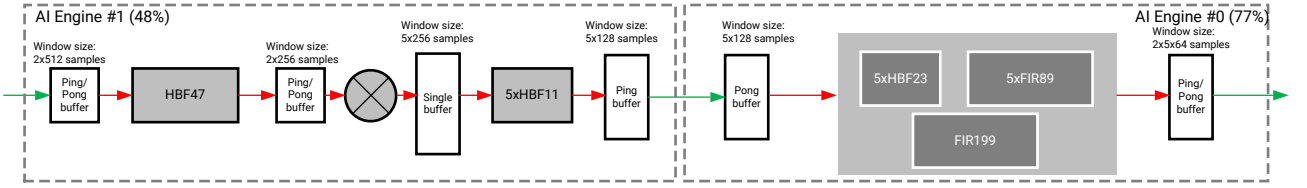
Though it is possible to design the filter chain using the traditional method, the design will have a large foot-print and require at least three AI Engines instead of two for each DDC. This 50% increase in AI Engines is significant when the number of antennas is high.

This application note proposes an innovative design approach for the stated carrier configuration to reduce AI Engine resource utilization by 30% by leveraging adaptable and scalable compute capabilities designed into the Versal AI Engine. The savings will be even higher in 5G NR wireless radio systems supporting a larger number of carrier configurations. The proposed approach is also applicable to digital up-conversion chain (DUC) and other modules in the system.

Novel Filter Design on AI Engine

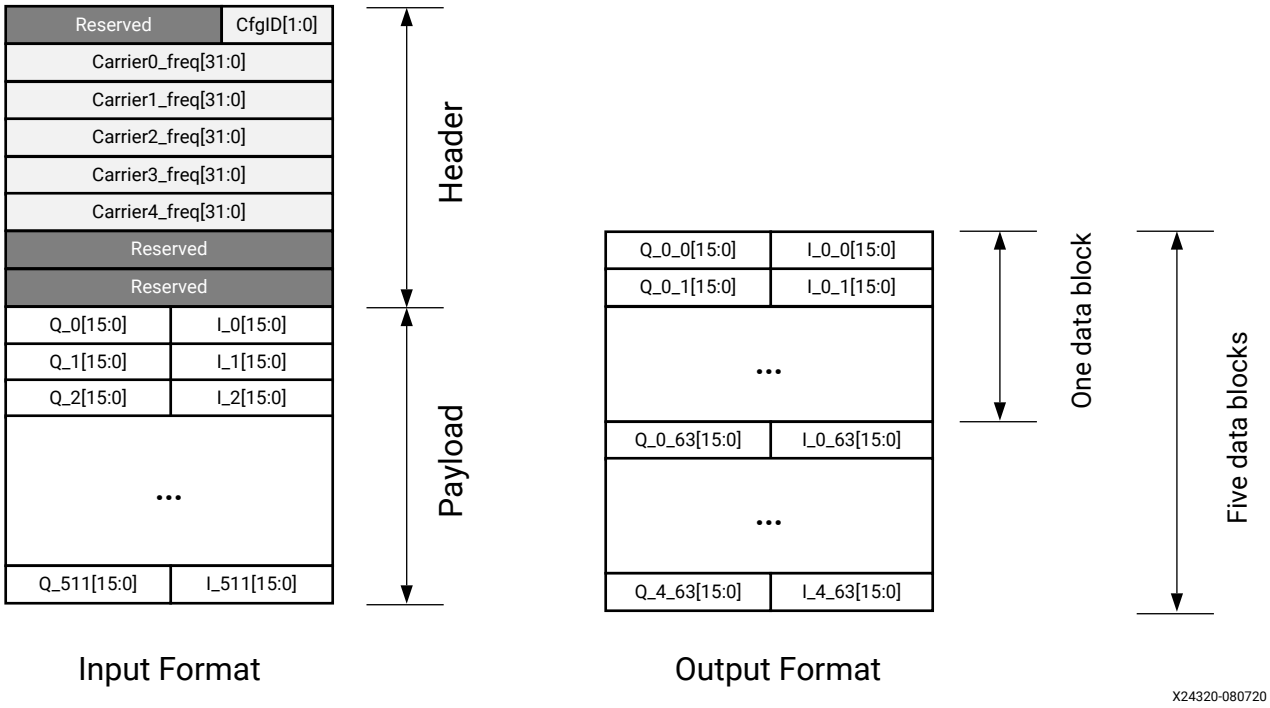
It is observed that the 5G NR and LTE carriers do not co-exist in the system. Also, the maximum utilization between the 5G NR channel filter and the 5c LTE channel filter is less than 80% of the AI Engine capacity. (As shown in the previous figure, the utilization of AI Engine #0 is about 55% and the utilization of AI Engine #1 is about 77%). This allows packing the FIR89 and FIR199 filter chain on a single AI Engine, resulting in one antenna design using two AI Engines instead of three. The proposed design partitioning is illustrated in the following figure.

Figure 7: **DDC System Partition (Two AI Engines)**



A header at the front of the input window, which can change on a block-by-block basis, indicates the carrier configuration. The output window has a fixed size but can contain the data for one 5G NR 100 MHz carrier or five LTE carriers. A diagram of input/output data format is shown in the following figure.

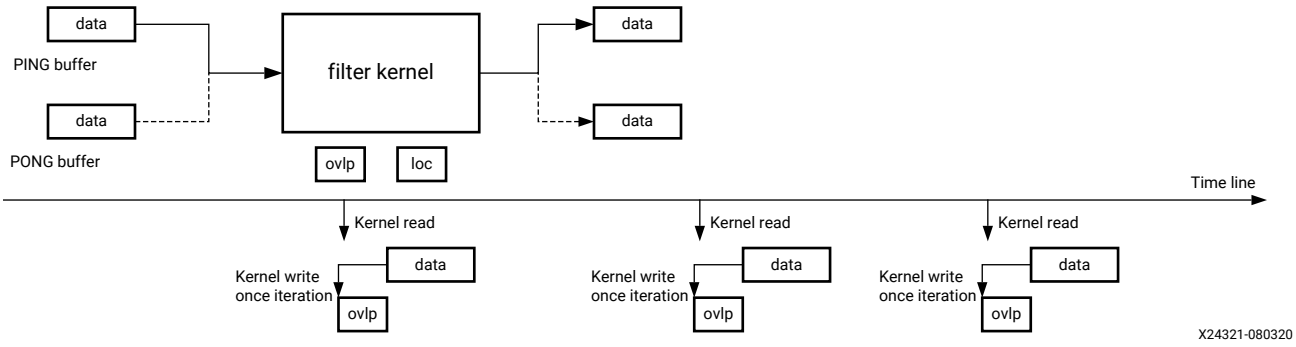
Figure 8: **DDC Interface Format**



The optimized design implementation is explained step by step in the following.

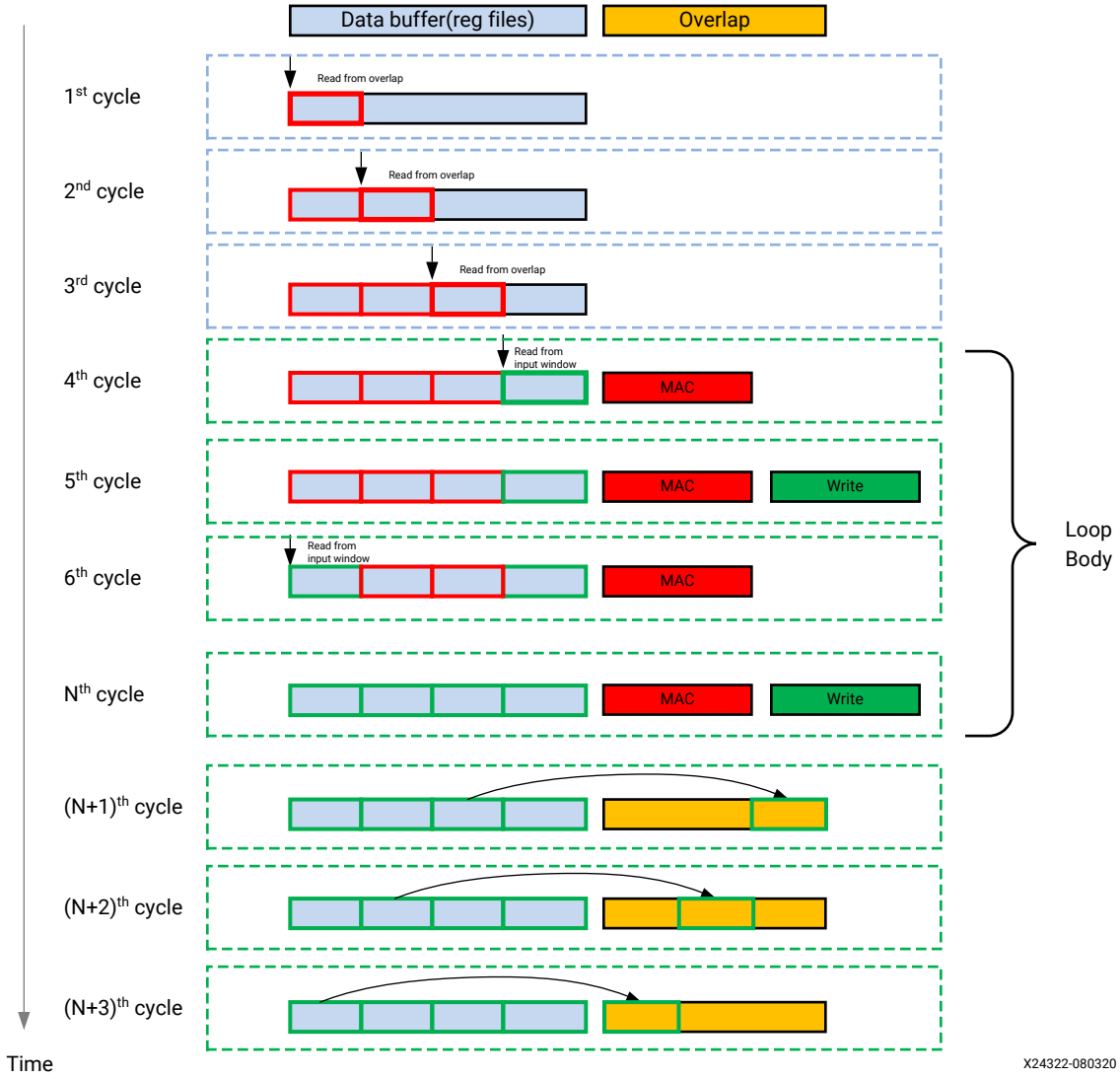
First, the memory footprint has to be reduced by manually managing filter overlaps. Consider the buffer allocated to one filter kernel; only the overlap portion of the buffer is unique to the filter and the data portion can be shared by multiple filters. Especially for short filters, the saving in memory is large. The following figure shows the new approach where each kernel is assigned a pair of ping-pong buffers without overlap, and a separate memory is allocated for overlap only. The separation of data and overlap makes it possible for filters of different taps and sample rates to share a single data buffer.

Figure 9: **Novel Filter Kernel Behavior**



The following figure illustrates how a half band filter (HBF23) works with such an overlap buffer scheme, which differs from the traditional method in the removal of the overlap buffer in the input window to eliminate the automatic overlap copy. During the first three cycles, the kernel initializes the register by loading data from the overlap buffer rather than the input window. The main loop starts from the fourth to the N^{th} cycle, during which the kernel reads data from the input window, shuffles them, and performs MAC operations. Before the end of the function, the data from the last few cycles (three in this case) are stored back into the overlap buffer.

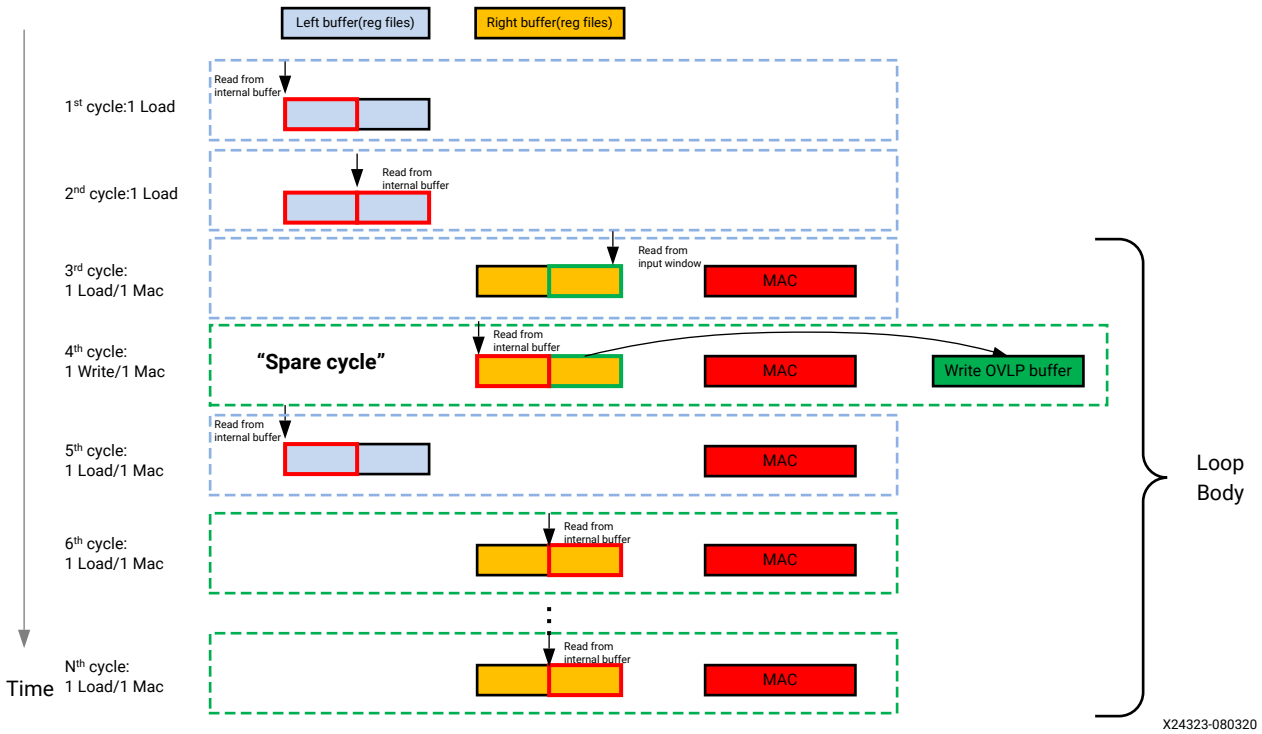
Figure 10: **Concept of Manually Managed Overlap**



The separation of overlap and data buffers leads to an implementation challenge. If the overlap is short enough to be loaded into registers without the need for reloading, which is the case for HBF11 and HBF23, then it is easy to handle, in that the overlap can be loaded into register space at the beginning as shown in the preceding figure. However, when the filter is long, a memory copy is required to merge the memory space. FIR89 and FIR199 are such cases. This application note proposes to use the overlap memory buffer for filtering and parallelize the data copy process with computation to maximize throughput.

The Versal ACAP AI Engine is a very long instruction word (VLIW) vector engine. The VLIW based instruction level parallelism implemented in the AI Engine allows execution of up to seven different operations in one cycle. The AI Engine can support two loads, one store, one vector MAC, one scalar ALU operation, and two data move instructions in a single cycle. This application note proposes a method that can use the AI Engine VLIW instruction bundling to find spare cycles to write data into an internal overlap from the input window in parallel with other operations. Refer to the following figure which uses a 64-sample input window. The first two cycles are used to load data into register files (left and right buffer, 16 samples each). The third to the N^{th} cycle is the main body of the for loop which is the key part of the filter design. The idea is to find the spare cycle to write the internal buffer with new data from the input window. Here the fourth or fifth cycle can be the spare cycle. From the sixth cycle the new data is overwritten by another load operation. In this way the costly overlap copy is perfectly merged with the MAC operation without using any extra cycles.

Figure 11: **Spare Cycle**



Implementation Details

Interfaces

A diagram of the input interface is shown in Figure 8. The input window size to the DDC of one antenna is set to 512 samples long with eight extra words reserved for the configuration which specifies the carrier configuration ID and central frequencies of the carriers. The 512 input sample is stored in natural order to minimize buffering on the programmable logic side; however, interleaving is performed by the AI Engine to improve the efficiency of the HBF47 decimation filter.

Internal Overlap Buffer

The overlap buffer behaves like a shift-register chain of a traditional filter. The key difference is that the data in the overlap buffer does not shift but the pointers do after every iteration. The initial value of the overlap buffer should be all zeros. To simplify the manipulation of the overlap buffer, the depth of the overlap buffer is set to an integer power of two. The depth of overlap buffer is given by the following equation.

$$OVLP\_BUF_{depth} = 16 \times 2^{(ceil(\log_2((Taps-1)/16)))} \quad \textit{Equation 4}$$

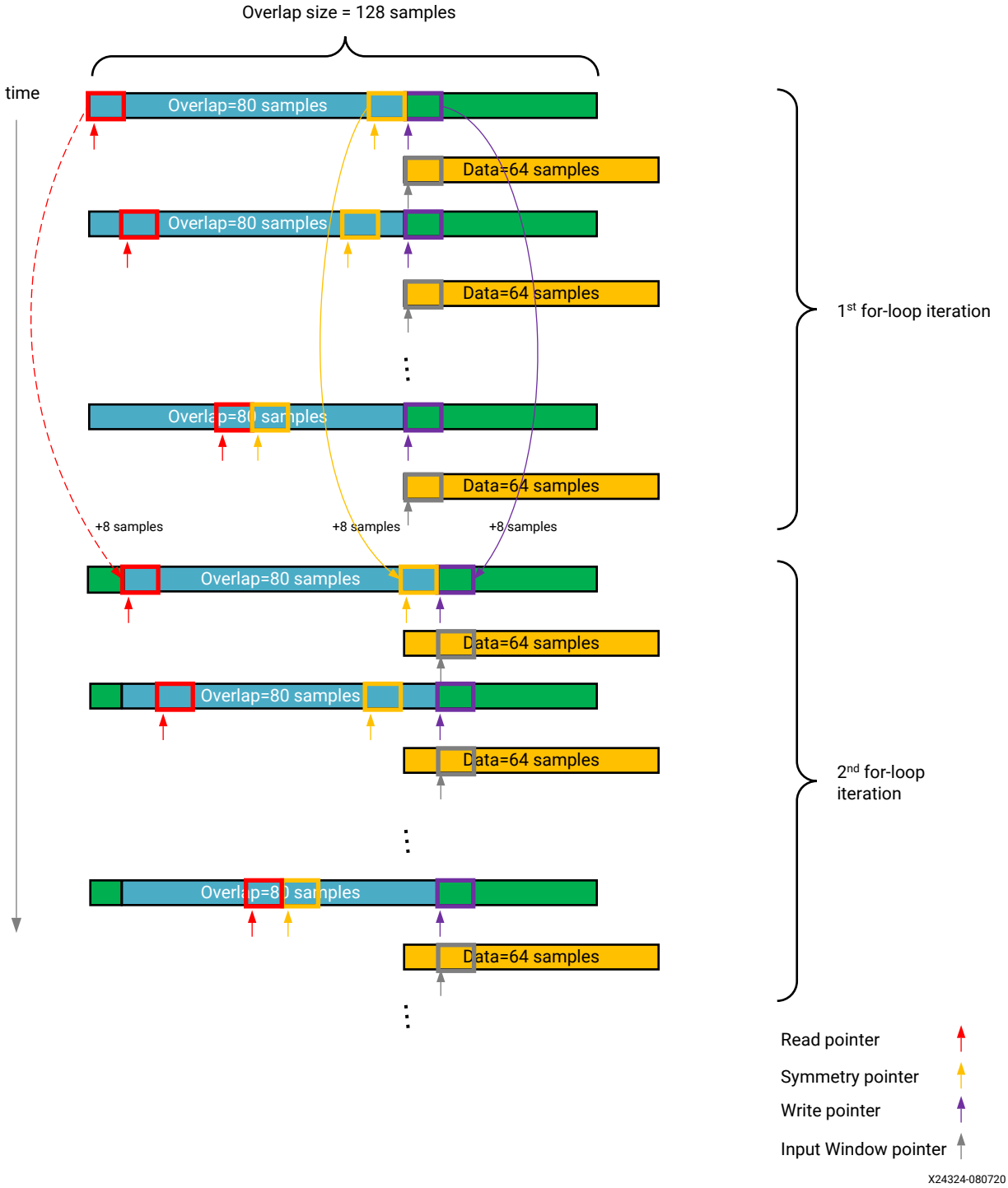
For example, an 89-tap filter has an overlap buffer of 128 samples.

Pointer Locations

One major difference between the novel filter design and the traditional method is that samples for MAC operations are read from the overlap buffer instead of the data buffer. For every eight output results, it only takes one read operation in the input window, and all the other data are from the overlap buffer. During the MAC operations, the newly read eight input data are written to the overlap memory for the next iteration. Every overlap buffer has three pointers, a read pointer, a symmetry pointer, and a write pointer. The starting locations of the overlap buffer pointers can be different in each iteration depending on the size of input window.

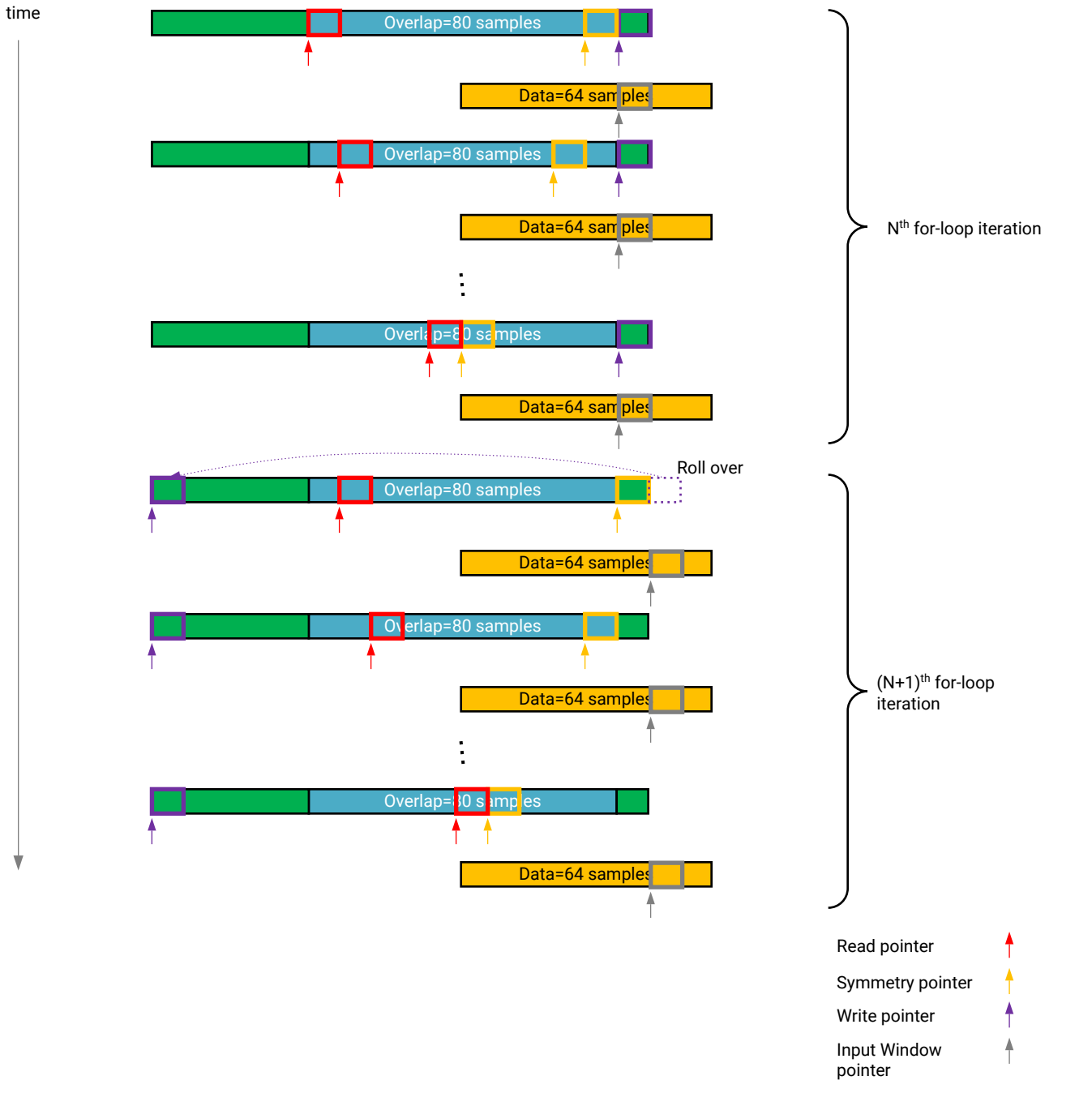
In the case of FIR89, an overlap of 80 samples depth is needed. The following figure illustrates the behavior of each pointer. At first the read pointer points to address 0, the symmetry pointer points to the address, `overlap-depth` - 8, the write pointer points to address, `overlap-size`, and the input window pointer points to the beginning of the input widow. In each iteration, the read pointer and symmetry pointer move against each other in step sizes of eight samples until all the data in the delay line is processed. At the beginning of the next iteration all the pointers are reset to their initial locations with an offset of eight samples relative to the initial location of the previous iteration.

Figure 12: **Novel FIR89 Pointers**



When an overlap pointer reaches the bottom of the overlap, roll-over occurs. As illustrated in the following figure, the write pointer reaches the bottom and it rolls over to the beginning of the overlap buffer. This is implemented by the `cyclic_add()` function for the pointer update.

Figure 13: **Write Pointer Rolls Over**



The following figure is an example of pointer movement from the kernel execution point of view for FIR89. Each kernel execution contains several inner loop (for loop) iterations. Assuming the size of the input window is 64 samples, each inner loop consumes eight samples, and then one kernel execution has 64/8=8 inner loops. At first, the read pointer points to #0 (*v8cint16), the symmetry pointer points to #10 (*v8cint16) and the write pointer points to #11 (*v8cint16). Each pointer increases by a step of 8 samples (8 × 32 bits = 256 bits for maximum memory access efficiency). As shown in the following figure, at the beginning of the second inner loop iteration, the read pointer points to #1 (*v8cint16), the symmetry pointer points to #11 (*v8cint16) and the write pointer points to #12 (*v8cint16) respectively. If any of the pointers reaches the bottom of the overlap, it will roll over to the beginning of the overlap.

Figure 14: **Overlap Buffer Pointer Movement**

At the beginning of the second kernel execution, the pointer locations should be initialized to 8/2/3 respectively and then the locations will be 0/10/11 again at the beginning of the third execution. This pattern keeps repeating as the data processing continues.

The Versal AI Engine software tools support a function called `cyclic_add` in `cardano.h`. It can be used to implement the cyclic roll-over of the pointers for when the pointers reach the end of the buffer. For example, the following code defines an inline function of cyclic increase to construct a buffer of depth, 16 × v8cint16.

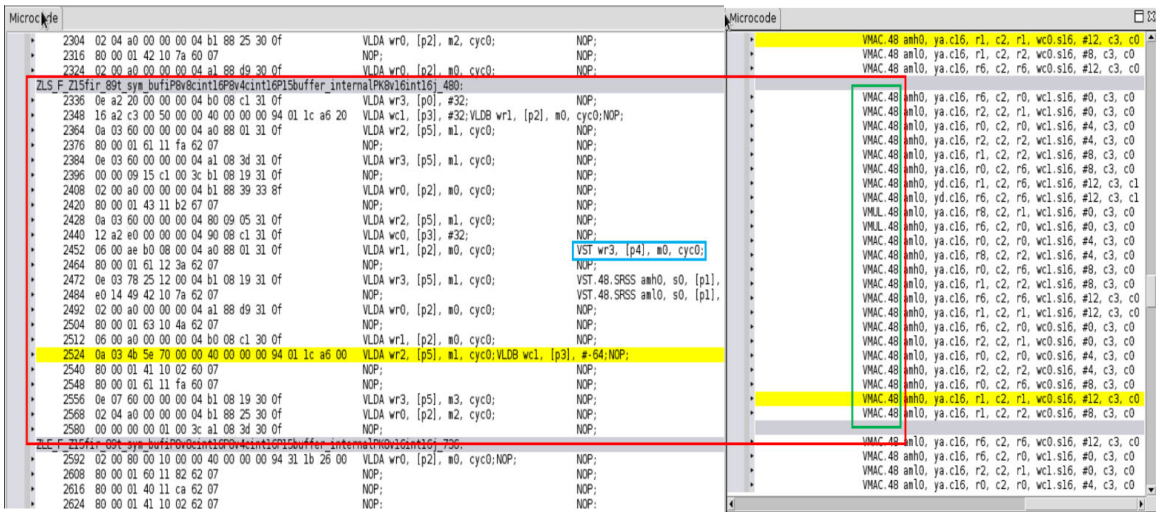
```
struct buffer_internal
{
  buffer_datatype * restrict head;
  buffer_datatype * restrict ptr
}
inline __attribute__((always_inline)) void
buffer128_incr_v8(buffer_internal * w, int count) {
  w->ptr=cyclic_add(w->ptr, count, w->head, 16);
 }
```

where

- `w` represents the overlap structure instance.
- `w->ptr` is the current pointer to the overlap.
- `w->head` refers to the starting address of the overlap.
- `count` means how many steps(v8cint16) to increase.
- The constant, 16, means it is an overlap with a fixed 128 sample depth (16 × v8cint16).

The following figure shows the microcode of FIR89. It is observed that the inner loop is perfect, and every cycle of the inner loop contains a MAC operation as indicated in the green box. As indicated in the blue box, the overlap buffer update operation (VST in microcode) is absorbed by the cycle that also performs the MAC operation.

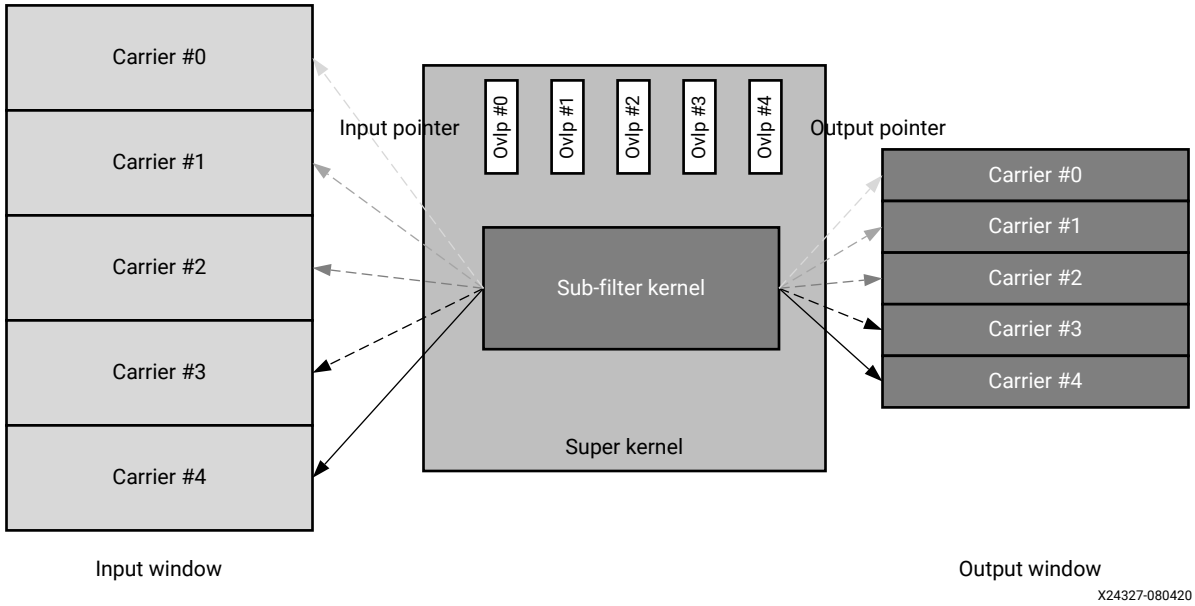
Figure 15: **FIR89 Kernel Compile Result**



Shared Window and Data Buffer

Manually managed overlap allows multiple carriers and filters to share the same input and output windows as shown in the following figure. When there are five carriers of 20 MHz LTE, the window can be logically divided into five buffers, one for each carrier. When there is only one carrier of 100 MHz 5G NR, the first four fifths of memory space will be allocated to the 100 MHz carrier alone and leaving the remaining one fifth of memory space unused. One single kernel with manually managed overlap can serve all the carriers of the same type with correctly specified input, overlap, and output pointer locations.

Figure 16: **5x HBF Sharing the Windows**



AI Engine kernels with manually managed overlap can share the intermediate results memory among the channels, because samples are executed sequentially in the AI Engine. The following two figures illustrate the method to save AI Engine memory by sharing the intermediate result memory between HBF23 and FIR89/FIR199. The size of the intermediate result memory, shown in the orange block in the following figure, is only one fifth of the original non-sharing method (see Figure 18).

Figure 17: **DDC Data Buffers With Data Memory Sharing (Proposed)**

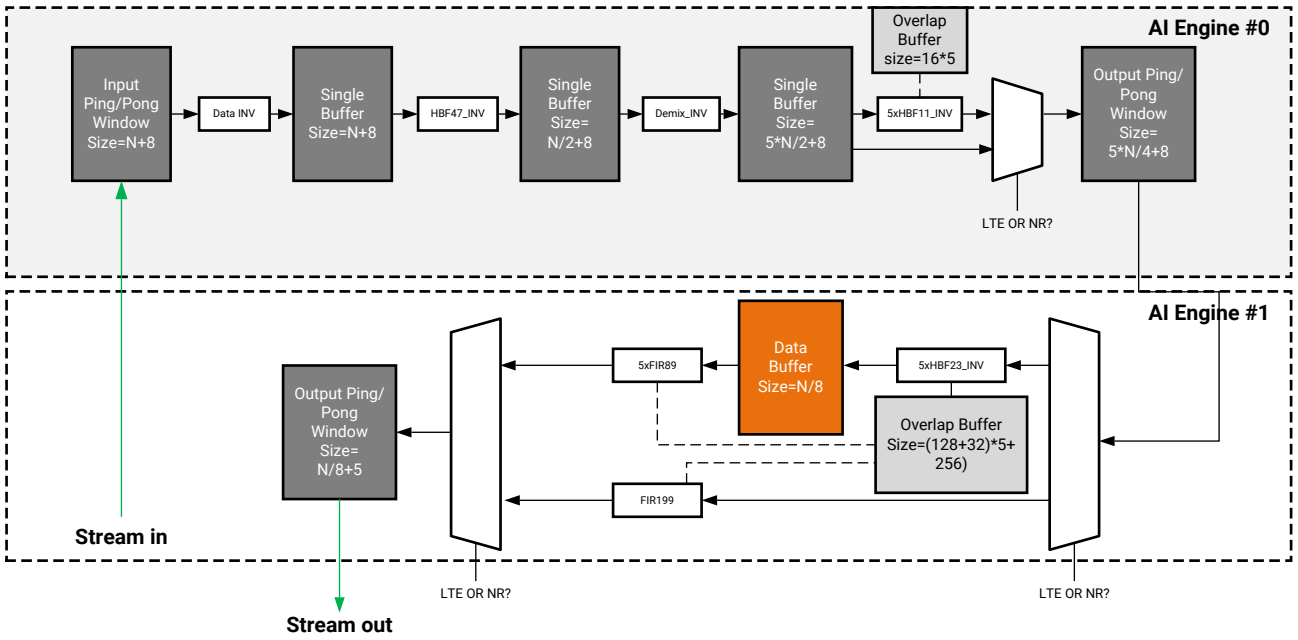
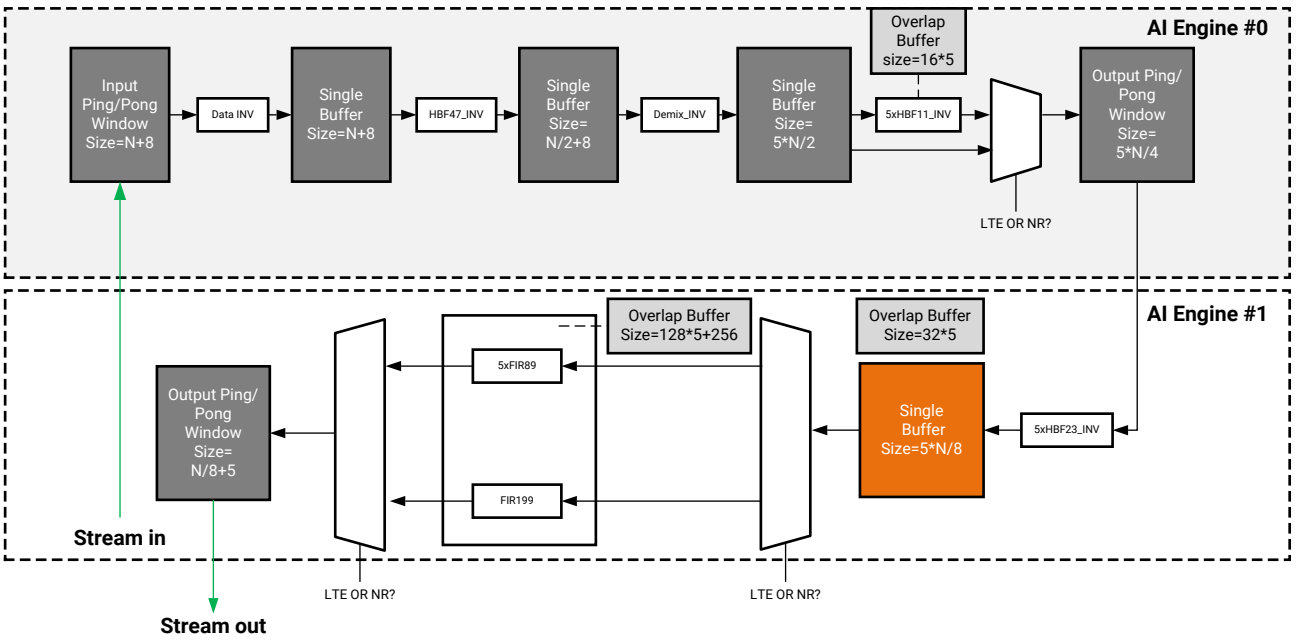


Figure 18: **DDC Data Buffers Without Data Memory Sharing (Original)**

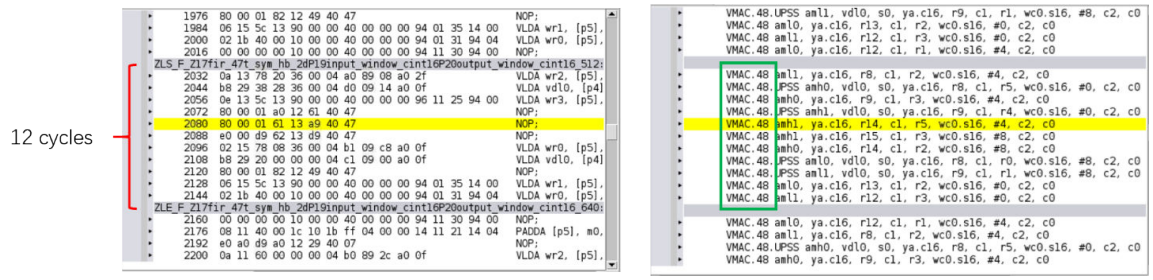


X24329-080720

Data Interleaving

The half-band decimation filter is symmetric and needs two input samples to compute each output. This turns out to be difficult for the kernel design to achieve a perfect loop when the filter has many taps. A solution to the problem is to put an even/odd interleaver in front of the half-band filter to avoid the odd samples being loaded into the registers multiple times. The following figure is an example of an HBF47 implementation result that achieves a perfect inner loop.

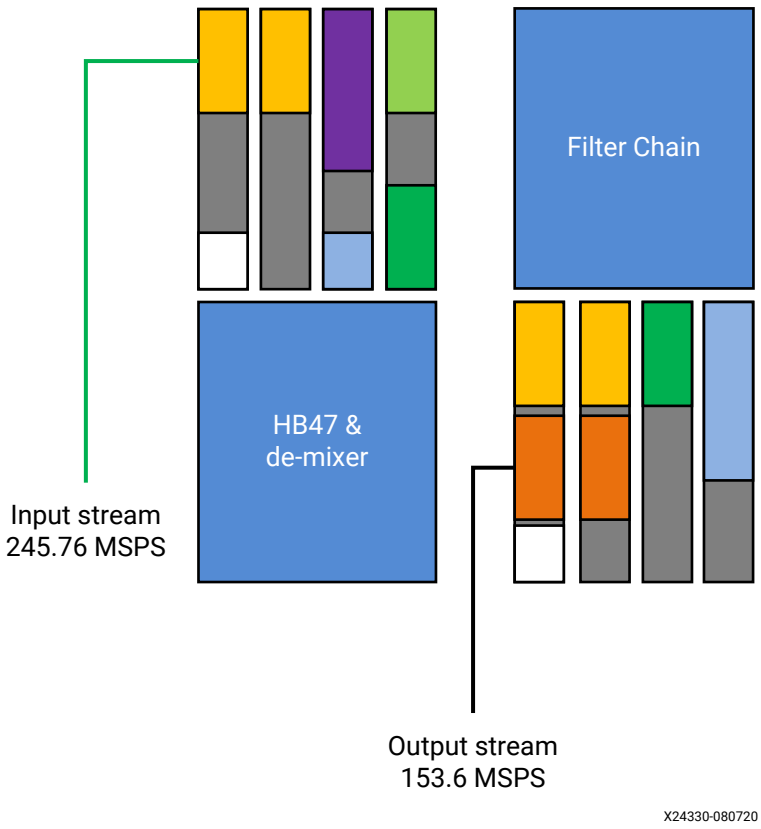
Figure 19: **HBF47 Compiling Result**



Kernel And Memory Constraints

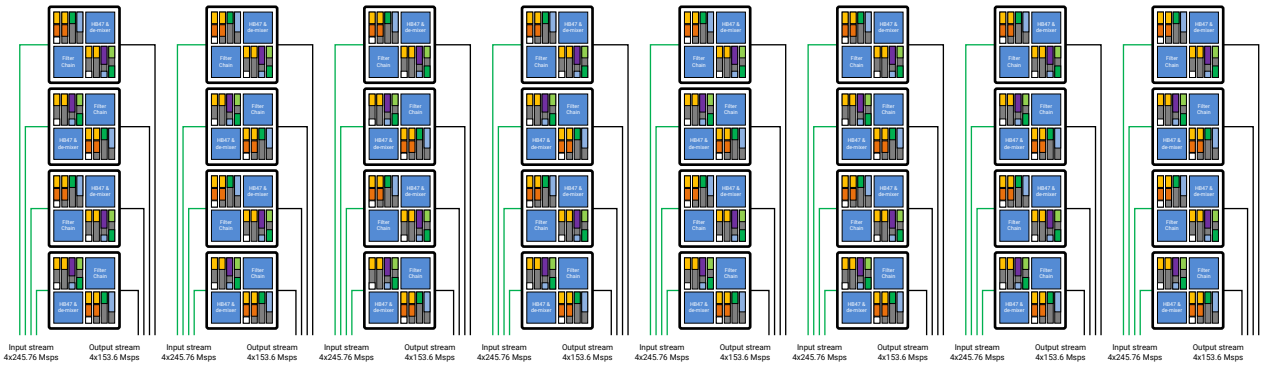
To build a scalable and compact design, it is a good practice to place the kernels, buffers, and windows carefully within a set of AI Engine tiles. The following figure shows a possible placement of two AI Engine tiles for the DDC of one antenna.

Figure 20: **Two AI Engine Tiles**



32 such tile pairs can be stacked in the AI Engine array of eight columns by eight rows to support 32 antennas. As shown in the following figure, each column has four input and four output 32-bit wide AXI4 streams.

Figure 21: **AI Engine Array of 8x8=64 AI Engine Tiles**



Implementation Result

The DDC kernels of this application note are profiled and the results are listed in the following table. The actual utilization is measured by:

$$\text{Utilization}_{(\text{actual})} = (\text{Function counts})/(\text{Cycle\_budget}) \qquad \textit{Equation 5}$$

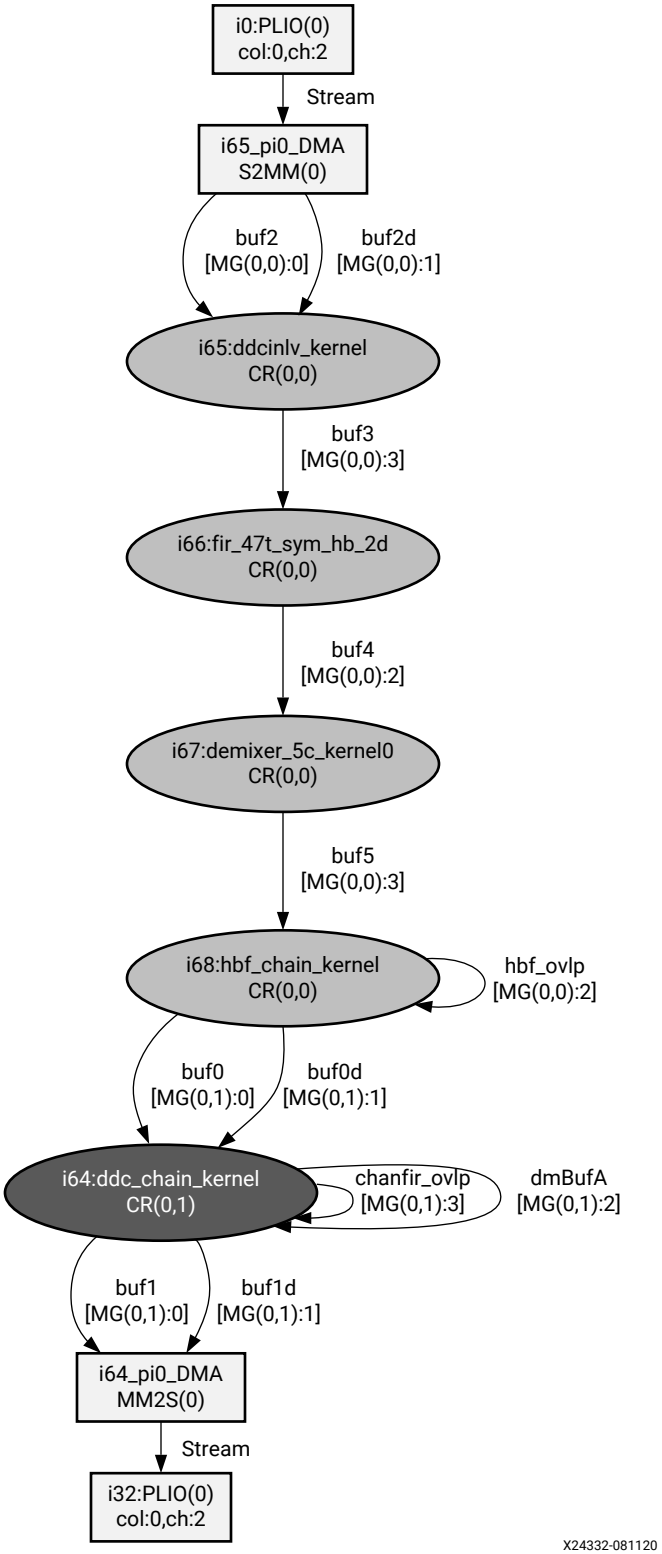
Table 3: **DDC Kernel Profile**

Kernel Name	Interface Format	Function Counts	Utilization Lower Boundary	Actual Utilization	Implementation Loss
FIR199	Nature in Nature out	1705 cycles	76.9%	81.9%	5%
FIR89	Nature in Nature out	238 cycles	9.3%	11.5%	2.2%
HBF47_2D	Interleaved in Nature out	266 cycles	12.3%	12.8%	0.5%
HBF23_2D	Nature in Nature out	65 cycles	1.6%	3.2%	1.6%
HBF11_2D	Interleaved in Nature out	54 cycles	1.6%	2.6%	1%
Mixer	Nature in Interleaved out	571 cycles	23%	27.5%	4.5%

The profiling results show a small implementation loss compared to the utilization lower bound.

The data flow and interconnects among the AI Engines are described in a graph file and the compilation result of a 32-antenna DDC design is shown in the following figure. Bubbles with same color are mapped and executed in the same AI Engine. The light gray boxes are DMAs and buffers automatically generated by the Xilinx tools depending on the data flow specified by the user.

Figure 22: **Compilation Result of a 1 out of 32 Antenna DDC Design**



Design Validation

MATLAB® scripts are used to generate random test vectors as inputs and corresponding reference outputs. The completed AI Engine design is validated against the MATLAB reference model to confirm the correct behavior. For the DDC design, there are 32 output AXI streams for 32 antennas. A make file script is included in the reference design to compare all the outputs of the AI Engine SystemC simulation results with the golden test vectors generated by MATLAB. Figure 23 (a) and Figure 24 (a) show the bit-accurate comparison of the SystemC simulation and the reference MATLAB model of the example design, where the outputs bit-true match those of the reference model.

Figure 23: **1c 100 MHz 5G NR Result**

```
:/DDC $ make check_op
CFG NR100M:  DDC 00 - diff=0 -
CFG NR100M:  DDC 01 - diff=0 -
CFG NR100M:  DDC 02 - diff=0 -
CFG NR100M:  DDC 03 - diff=0 -
CFG NR100M:  DDC 04 - diff=0 -
CFG NR100M:  DDC 05 - diff=0 -
CFG NR100M:  DDC 06 - diff=0 -
CFG NR100M:  DDC 07 - diff=0 -
CFG NR100M:  DDC 10 - diff=0 -
CFG NR100M:  DDC 11 - diff=0 -
CFG NR100M:  DDC 12 - diff=0 -
CFG NR100M:  DDC 13 - diff=0 -
CFG NR100M:  DDC 14 - diff=0 -
CFG NR100M:  DDC 15 - diff=0 -
CFG NR100M:  DDC 16 - diff=0 -
CFG NR100M:  DDC 17 - diff=0 -
CFG NR100M:  DDC 20 - diff=0 -
CFG NR100M:  DDC 21 - diff=0 -
CFG NR100M:  DDC 22 - diff=0 -
CFG NR100M:  DDC 23 - diff=0 -
CFG NR100M:  DDC 24 - diff=0 -
CFG NR100M:  DDC 25 - diff=0 -
CFG NR100M:  DDC 26 - diff=0 -
CFG NR100M:  DDC 27 - diff=0 -
CFG NR100M:  DDC 30 - diff=0 -
CFG NR100M:  DDC 31 - diff=0 -
CFG NR100M:  DDC 32 - diff=0 -
CFG NR100M:  DDC 33 - diff=0 -
CFG NR100M:  DDC 34 - diff=0 -
CFG NR100M:  DDC 35 - diff=0 -
CFG NR100M:  DDC 36 - diff=0 -
CFG NR100M:  DDC 37 - diff=0 -
```

(a)100M NR validation result

```
:/DDC $ make get_tp
CFG 1 DDC 00 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 01 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 02 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 03 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 04 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 05 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 06 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 07 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 10 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 11 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 12 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 13 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 14 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 15 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 16 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 17 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 20 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 21 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 22 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 23 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 24 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 25 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 26 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 27 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 30 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 31 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 32 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 33 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 34 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 35 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 36 Throughput= 178.85 Msps (>153.6)
CFG 1 DDC 37 Throughput= 178.85 Msps (>153.6)
```

(b)100M NR throughput

Figure 24: **5c LTE 20 MHz Result**

```
:/DDC $ make check_op
CFG 5xLTE20M:  DDC 00 - diff=0 -
CFG 5xLTE20M:  DDC 01 - diff=0 -
CFG 5xLTE20M:  DDC 02 - diff=0 -
CFG 5xLTE20M:  DDC 03 - diff=0 -
CFG 5xLTE20M:  DDC 04 - diff=0 -
CFG 5xLTE20M:  DDC 05 - diff=0 -
CFG 5xLTE20M:  DDC 06 - diff=0 -
CFG 5xLTE20M:  DDC 07 - diff=0 -
CFG 5xLTE20M:  DDC 10 - diff=0 -
CFG 5xLTE20M:  DDC 11 - diff=0 -
CFG 5xLTE20M:  DDC 12 - diff=0 -
CFG 5xLTE20M:  DDC 13 - diff=0 -
CFG 5xLTE20M:  DDC 14 - diff=0 -
CFG 5xLTE20M:  DDC 15 - diff=0 -
CFG 5xLTE20M:  DDC 16 - diff=0 -
CFG 5xLTE20M:  DDC 17 - diff=0 -
CFG 5xLTE20M:  DDC 20 - diff=0 -
CFG 5xLTE20M:  DDC 21 - diff=0 -
CFG 5xLTE20M:  DDC 22 - diff=0 -
CFG 5xLTE20M:  DDC 23 - diff=0 -
CFG 5xLTE20M:  DDC 24 - diff=0 -
CFG 5xLTE20M:  DDC 25 - diff=0 -
CFG 5xLTE20M:  DDC 26 - diff=0 -
CFG 5xLTE20M:  DDC 27 - diff=0 -
CFG 5xLTE20M:  DDC 30 - diff=0 -
CFG 5xLTE20M:  DDC 31 - diff=0 -
CFG 5xLTE20M:  DDC 32 - diff=0 -
CFG 5xLTE20M:  DDC 33 - diff=0 -
CFG 5xLTE20M:  DDC 34 - diff=0 -
CFG 5xLTE20M:  DDC 35 - diff=0 -
CFG 5xLTE20M:  DDC 36 - diff=0 -
CFG 5xLTE20M:  DDC 37 - diff=0 -
```

(a)5cLTE20M validation result

```
:/DDC $ make get_tp
CFG 0 DDC 00 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 01 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 02 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 03 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 04 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 05 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 06 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 07 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 10 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 11 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 12 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 13 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 14 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 15 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 16 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 17 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 20 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 21 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 22 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 23 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 24 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 25 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 26 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 27 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 30 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 31 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 32 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 33 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 34 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 35 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 36 Throughput= 207.189 Msps (>153.6)
CFG 0 DDC 37 Throughput= 207.189 Msps (>153.6)
```

(b)5cLTE20M throughput

The time stamps in the Versal SystemC Simulation (ESS) output can be used to measure the time duration from the first output sample to the last. Also because the number of output samples can be counted in the output file too, an estimate of throughput can be computed; that is, the throughput can be calculated by dividing the total number of outputs by the time interval between first sample and the last sample. The reference design comes with a Makefile that measures the throughput. As shown in Figure 23 (b) and Figure 24 (b), the ESS simulation of the AI Engine DDC design indicates that the DDC design throughput is about 207 MSPS for the 5c LTE case, much higher than the nominal sample rate of five LTE carriers at 153.6 MSPS. The throughput for the 5G NR configuration is about 180 MSPS, much higher than the nominal sample rate of 122.88 MSPS. There is sufficient margin in the DDC design based on the computed throughput.

When the AI Engine DDC module passes ESS simulation validation, it is ready for integration in a complete system that consists of programmable logic, processor sub-system, memory controller, and other blocks in Versal AI Core devices.

Reference Design

Download the reference design files for this application note from the Xilinx® website.

Reference Design Matrix

The following checklist indicates the procedures used for the provided reference design.

Table 4: **Reference Design Matrix**

Parameter	Description
General	
Developer name	Xilinx
Target devices	Versal ACAP
Source code provided?	Yes
Source code format (if provided)	MATLAB script, AI Engine C code, and Makefile script
Design uses code or IP from existing reference design, application note, 3rd party or Vivado software? If yes, list.	No
Simulation	
Functional simulation performed	Yes
Timing simulation performed?	No
Test bench provided for functional and timing simulation?	No
Test bench format	C code
Simulator software and version	AI Engine Simulator in Vitis 2020.2
SPICE/IBIS simulations	No
Hardware Verification	
Hardware verified?	Yes
Platform used for verification	VCK190

Conclusion

This application note demonstrates an innovative method of designing adaptable, scalable, and resource-efficient DDCs on Versal AI Engine technology. The kernels and graphs are designed in C/C++, which is easy to maintain and reuse on various Versal ACAPs. The example design provided as part of the application note serves as a template for quick generation of customized DDC filter chains. The methodology introduced by this application note has a wide application to filter designs and is applicable to many use cases including but not limited to wireless signal processing.

References

These documents provide supplemental material useful with this guide:

1. *Versal ACAP AI Engine Programming Environment User Guide* (UG1076)
2. *Xilinx AI Engine and Their Applications* (WP506)

Revision History

The following table shows the revision history for this document.

Section	Revision Summary
02/15/2021 Version 1.0	
Initial release.	N/A

Please Read: Important Legal Notices

The information disclosed to you hereunder (the "Materials") is provided solely for the selection and use of Xilinx products. To the maximum extent permitted by applicable law: (1) Materials are made available "AS IS" and with all faults, Xilinx hereby DISCLAIMS ALL WARRANTIES AND CONDITIONS, EXPRESS, IMPLIED, OR STATUTORY, INCLUDING BUT NOT LIMITED TO WARRANTIES OF MERCHANTABILITY, NON-INFRINGEMENT, OR FITNESS FOR ANY PARTICULAR PURPOSE; and (2) Xilinx shall not be liable (whether in contract or tort, including negligence, or under any other theory of liability) for any loss or damage of any kind or nature related to, arising under, or in connection with, the Materials (including your use of the Materials), including for any direct, indirect, special, incidental, or consequential loss or damage (including loss of data, profits, goodwill, or any type of loss or damage suffered as a result of any action brought by a third party) even if such damage or loss was reasonably foreseeable or Xilinx had been advised of the possibility of the same. Xilinx assumes no obligation to correct any errors contained in the Materials or to notify you of updates to the Materials or to product specifications. You may not reproduce, modify, distribute, or publicly display the Materials without prior written consent. Certain products are subject to the terms and conditions of Xilinx's limited warranty, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos; IP cores may be subject to warranty and support terms contained in a license issued to you by Xilinx. Xilinx products are not designed or intended to be fail-safe or for use in any application requiring fail-safe performance; you assume sole risk and liability for use of Xilinx products in such critical applications, please refer to Xilinx's Terms of Sale which can be viewed at https://www.xilinx.com/legal.htm#tos.

This document contains preliminary information and is subject to change without notice. Information provided herein relates to products and/or services not yet available for sale, and provided solely for information purposes and are not intended, or to be construed, as an offer for sale or an attempted commercialization of the products and/or services referred to herein.

AUTOMOTIVE APPLICATIONS DISCLAIMER

AUTOMOTIVE PRODUCTS (IDENTIFIED AS "XA" IN THE PART NUMBER) ARE NOT WARRANTED FOR USE IN THE DEPLOYMENT OF AIRBAGS OR FOR USE IN APPLICATIONS THAT AFFECT CONTROL OF A VEHICLE ("SAFETY APPLICATION") UNLESS THERE IS A SAFETY CONCEPT OR REDUNDANCY FEATURE CONSISTENT WITH THE ISO 26262 AUTOMOTIVE SAFETY STANDARD ("SAFETY DESIGN"). CUSTOMER SHALL, PRIOR TO USING

OR DISTRIBUTING ANY SYSTEMS THAT INCORPORATE PRODUCTS, THOROUGHLY TEST SUCH SYSTEMS FOR SAFETY PURPOSES. USE OF PRODUCTS IN A SAFETY APPLICATION WITHOUT A SAFETY DESIGN IS FULLY AT THE RISK OF CUSTOMER, SUBJECT ONLY TO APPLICABLE LAWS AND REGULATIONS GOVERNING LIMITATIONS ON PRODUCT LIABILITY.

Copyright