Summer Student Programme 2024

Implementation of calorimeter data processing module on TELL40 board using intel oneAPI SYCL

Stefan Delić

Supervisor: Alberto Perro

List of images

List of tables

1. Introduction

The objective of this project is to reimplement the calorimeter data processing module using Intel oneAPI, specifically focusing on the High-Level Synthesis (HLS) IP flow. In the current LHCb setup, this module is implemented using Hardware Description Language (HDL). By reimplementing it with oneAPI, the project aims to evaluate the performance and effectiveness of Intel oneAPI in comparison to the custom HDL implementation.

The implementation is based on the LHCb upgrade technical note, along with the FE data format note and insight into the internals of the HDL implementation. [1] [2] The code of the whole project can be found on the Gitlab link: https://gitlab.cern.ch/sdelic/calo_oneapi.

1.1 LHCb Data Acquisition chain, TELL40 Boards

The readout board is a generic component that has been designed for the data acquisition of all detectors, the distribution of the timing and fast commands, and the slow control. The core of LHCb's trigger system is the data acquisition system, consisting of approximately 160 computer servers equipped with 480 custom electronic cards that receive and aggregate information from nearly 1 million

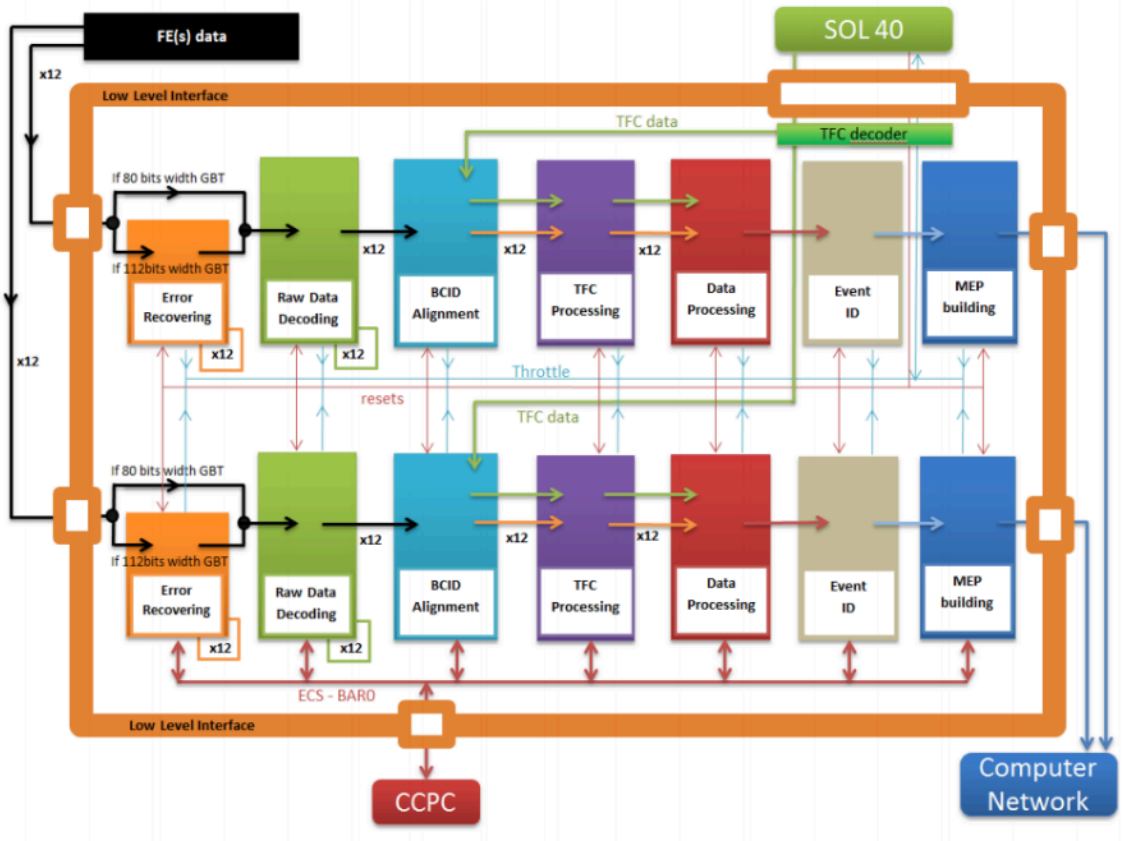

Figure 1. TELL40 firmware architecture [1]

individual electronic channels across the entire detector. This data, corresponding to a data rate of 5 terabytes per second, is then passed on to the first stage of the trigger (HLT1). [3]

The PCIe40 firmware defines the device's functions, assigning it a different moniker depending on whether it is used for slow control, data acquisition, or timing and fast control. For this discussion, only the data acquisition variant (known as “TELL40”) will be described, and that term will be used to refer to both the firmware and the card itself. [4]

The architecture of the TELL40 board common for all subdetectors can be seen in Figure 1. This architecture contains two data streams linked to the two PCIe interfaces of the board. During this project, only the data processing submodule of TELL40 is implemented using oneAPI HLS. For the rest of the paper, this module will be the focus.

In Figure 2, the internals of the data processing submodule can be seen. This block diagram resembles the internal view of the current HDL implementation. The general structure is preserved as much as possible in the implementation of this project. For further information on data formats and internal functionality consult the following documentation. [1] [2]

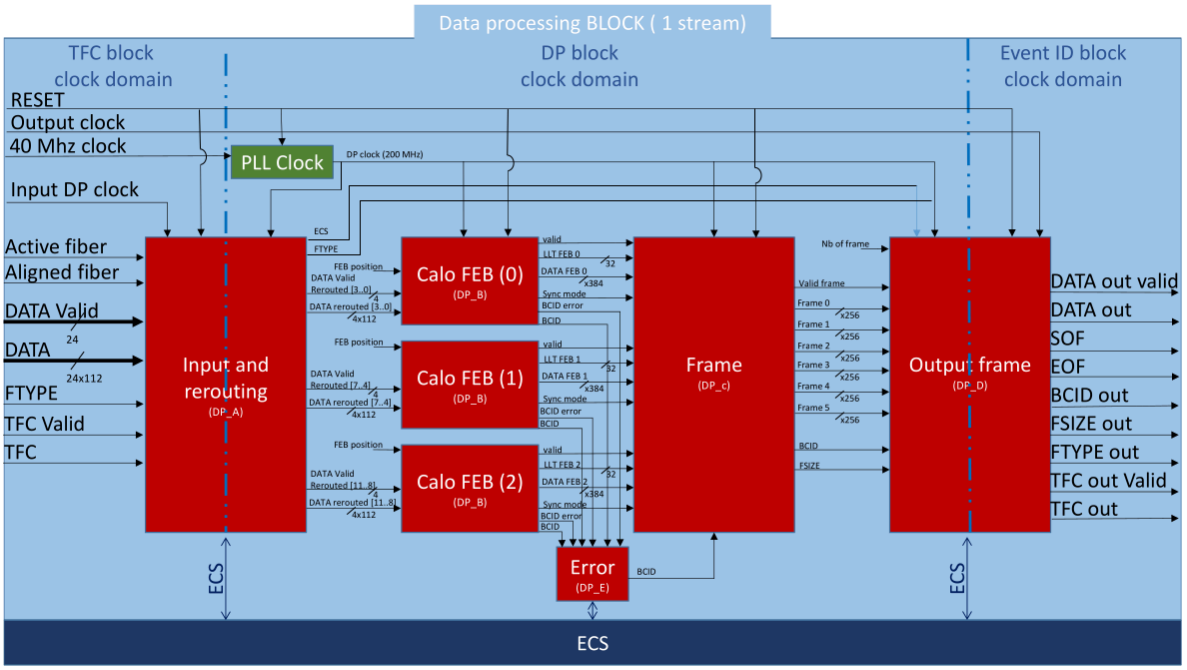

Figure 2. Data processing architecture [1]

2. Intel oneAPI

Intel oneAPI is a comprehensive, open, cross-industry programming model designed to simplify the development of high-performance applications across diverse computing architectures. It provides a unified environment for developers to create code that can run efficiently on a range of hardware platforms, including CPUs, GPUs, FPGAs, and other accelerators. This approach helps streamline the development process by allowing a single codebase to be used across different types of processors, reducing the need for hardware-specific optimizations and enabling greater portability and flexibility. [5]

Key Features of Intel oneAPI: [5]

1. **Unified Programming Model:** OneAPI integrates various programming models and libraries into a single framework, offering a consistent set of APIs and tools. This unified model supports diverse hardware architectures, including Intel's CPU, GPU, and FPGA products.

2. **Data Parallel C++ (DPC++):** At the core of oneAPI is Data Parallel C++, an extension of C++ designed to simplify the development of parallel and heterogeneous computing applications. DPC++ allows developers to write code that can be executed on different types of processors with minimal modifications, enabling efficient data parallelism and task parallelism.

3. **Optimized Libraries and Toolkits:** OneAPI includes a suite of optimized libraries and toolkits for various domains such as math, data analytics, and machine learning. These libraries provide pre-built functions and routines that are optimized for Intel hardware, helping developers accelerate their applications without needing to implement these functions from scratch.

4. **Intel Hardware Acceleration:** OneAPI is designed to fully leverage Intel's hardware capabilities, including advanced features of CPUs, GPUs, and FPGAs. This enables applications to achieve higher performance and efficiency by utilizing the specific strengths of different types of Intel processors.

5. **Open Standards and Cross-Vendor Compatibility:** OneAPI is based on open standards and aims to be cross-vendor compatible, allowing developers to write portable code that can run on various hardware platforms beyond Intel's ecosystem. This approach promotes flexibility and reduces the risk of vendor lock-in.

The Intel oneAPI DPC++/C++ Compiler is a robust tool that enables developers to write and compile code that can be executed across diverse computing architectures. This flexibility is central to optimizing application performance by

leveraging various processing units effectively. The compiler offers two primary modes of operation:

1. **Hybrid Compilation for Host and Accelerators:** The DPC++/C++ Compiler allows for the partitioning of code into sections that can be compiled and executed on different architectures. This capability supports the offloading of specific code segments to hardware accelerators such as GPUs or FPGAs, while keeping the remaining parts of the application running on CPUs. The compiler's ability to handle such heterogeneous computing environments means that developers can optimize performance by leveraging the strengths of each processing unit. For instance, computationally intensive tasks can be offloaded to GPUs for parallel processing, while more control-oriented or sequential tasks remain on the CPU.

2. **High-Level Synthesis (HLS) Flow for FPGA IP Generation:** Beyond the hybrid compilation, the Intel oneAPI toolchain includes a High-Level Synthesis (HLS) flow that enables the generation of standalone Intellectual Property (IP) cores for FPGAs. This flow allows developers to write high-level code, which the compiler then transforms into optimized FPGA IPs. These custom IP cores can be integrated into FPGA designs to achieve specialized acceleration for targeted applications. This approach simplifies the development of FPGA solutions by abstracting away much of the low-level hardware design work, making it accessible to developers who are more familiar with high-level programming languages. [6]

Overall, the Intel oneAPI DPC++/C++ Compiler offers a flexible and powerful approach to programming across different architectures. Its ability to compile code for both CPU and hardware accelerators, as well as its support for HLS to generate FPGA IPs, provides developers with multiple pathways to achieve optimal performance and efficiency for their applications.

2.1 Intel oneAPI interfaces

Intel oneAPI provides a variety of interfaces that allow efficient communication and data management between the host system and hardware accelerators like FPGAs and GPUs. These interfaces are crucial for enabling high-performance computing in heterogeneous environments. The primary types of interfaces in Intel oneAPI include Invocation Interfaces, Streaming Data Interfaces, and Memory-Mapped Host Interfaces. In the following section, only invocation interfaces and streaming data interfaces will be explained since they are crucial for this project. [7]

Invocation Interfaces

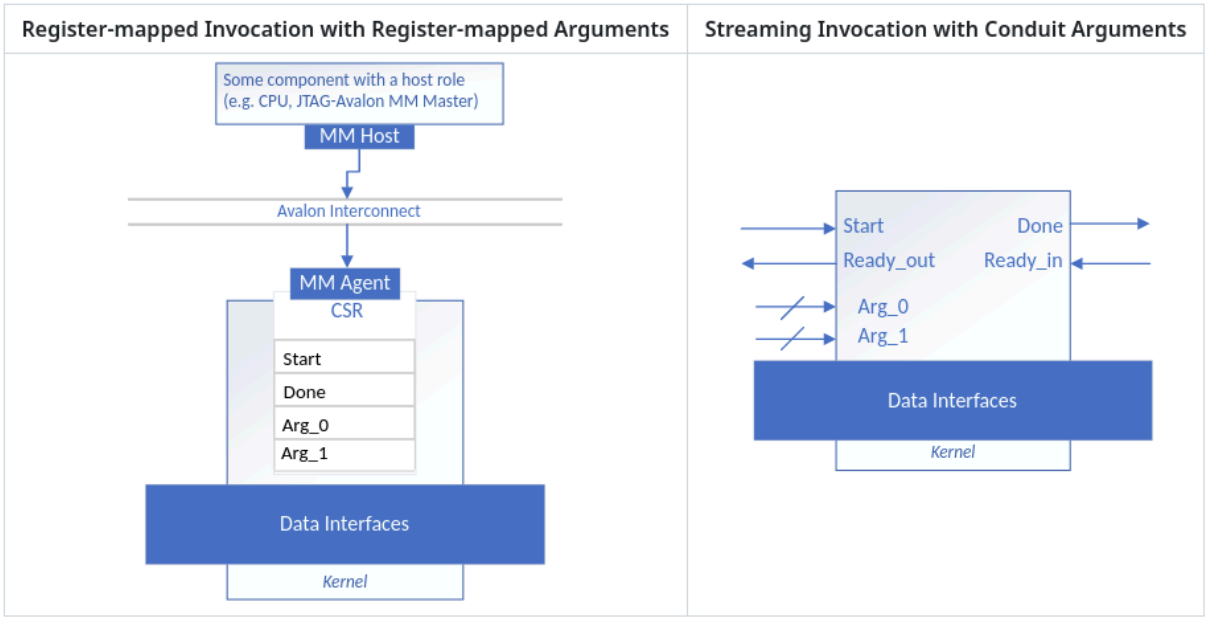

Figure 3. Invocation interfaces in Intel oneAPI [8]

In Intel oneAPI, invocation interfaces are key components that manage how kernels (functions executed on hardware accelerators like FPGAs or GPUs) are initiated and controlled. These interfaces define how and when a kernel starts executing, as well as how it communicates its completion back to the host. There are different ways to implement these interfaces, depending on the specific needs of the application. [8]

Register-Mapped Invocation with Register-Mapped Arguments

Register-mapped invocation method, where both the kernel invocation (start and done signals) and the kernel arguments are managed through control and status registers (CSRs). This method is straightforward and uses standard memory-mapped I/O techniques to pass control signals and data between the host and the kernel. In this scenario, the host writes to specific registers to start the kernel, and the kernel updates other registers to indicate when it has completed its task.

Streaming Invocation with Conduit Arguments

In the **streaming invocation** method, the invocation interface uses a ready/valid handshake mechanism. Here, the kernel starts execution when both the host and the kernel are ready, which is indicated through ready and valid signals. This method is more dynamic and allows for continuous data processing, as the kernel can begin processing data as soon as it becomes available. In this setup, kernel arguments can be passed through dedicated conduits instead of CSRs.

Streaming Data Interfaces

Streaming Data Interfaces in Intel oneAPI enable continuous data transfer between different components in a system, such as between kernels or between the host and a kernel.

Pipes are a powerful and intuitive construct in Intel oneAPI that serve as first-in, first-out (FIFO) buffers. They are used to create data links between different elements within a hardware design, enabling efficient streaming of data. Pipes are accessed through simple read and write operations, without the need for memory addresses or pointers, making them ideal for continuous data flow scenarios. [9]

Pipes can be **blocking** and **non-blocking**. **Blocking pipes** block the execution of the rest of the kernel until the pipes are read. On the other hand, **non-blocking pipes** try to read the input, but they do not block the rest of the kernel. A useful feature of non-blocking pipes is that it returns a bool value of read success.

3. HLS Implementation

For the implementation of the Data Processing Unit on the TELL40 board, Intel oneAPI's High-Level Synthesis (HLS) flow is used. The HLS flow allows for the translation of high-level C++ or DPC++ code into hardware description languages (HDLs) like VHDL or Verilog, which can then be synthesized into FPGA logic.

Input of the kernel

Interfaces used in the implementation are the streaming invocation interface that uses a ready/valid mechanism for control of the kernel and the streaming data interface implemented using pipes for passing the collision data. This setup allows for the continuous processing of the data which is crucial for this part of the data acquisition system which is not allowed to generate any back-pressure on the input.

The input and rerouting module that has 24 fibers on the input is implemented in HDL since in that module the clock domain crossing is handled along with the rerouting of the fibers. Clock domain crossing is not impossible to implement using oneAPI HLS.

On the input side, the kernel has 12 fibers that are coming from the input and rerouting module written in HDL. For each of the input fibers, one non-blocking pipe that uses the Avalon streaming interface is used. The width of each pipe is 116 bits.

In addition to the main collision data, non-blocking pipes are also used for the continuous transmission of TFC and FTYPE data, with widths of 64 bits and 32 bits, respectively.

Output of the kernel

The output logic is implemented with the function from Figure 4. This module outputs 5 frames of data for each read of the input pipes using the Avalon ST interface. The attribute **[[intel::speculated_iterations(0)]]** is used in front of the for loop which allowed the decrease of II (Iteration Interval) to 1. Without this attribute, the II was not 1 and there was a delay of 6 cycles between the packets on the output. [10]

```
// Outputing data from the kernel
[[intel::speculated_iterations(0)]]
for (int i = 0; i < output_frame_cnt; i++) {

  if (i == 0) {
    start_of_packet = 1;

    output_packet.set_slc(0, high);                          // TFC valid
    output_packet.set_slc(1, TFC_data_i);                    // TFC data
    output_packet.set_slc(65, BCID_from_DP_frame);           // BXID
    output_packet.set_slc(77, FTYPE_out_from_DP_frame);      // FTYPE
    output_packet.set_slc(85, Fsize_from_DP_frame);          // FSIZE
    output_packet.set_slc(101, output_frame[0]);             // Frame
  }
  else {
    start_of_packet = 0;

    output_packet.set_slc(0, low);                           // TFC valid
    output_packet.set_slc(101, output_frame[i]);             // Frame
  }

  if (i == (output_frame_cnt-1)) {
    end_of_packet = 1;
  }
  else {
    end_of_packet = 0;
  }

  OutStreamingBeatT out_beat(output_packet, start_of_packet, end_of_packet);
  OutPipe::write(out_beat);
}
```

Figure 4. Logic that outputs PCIe packet in the oneAPI kernel

Details of kernel implementation

The Data Processing unit is separated into the following modules:

- Front-End Board (**FEB**)
- Frame Assembly
- Error Management Unit
- Outputting logic

The function of **FEB** is to take the data from 4 fibers, separate the Low-Level Trigger (LLT) data and collision data from each fiber, and connect them into packets. The inputs and outputs of FEB can be seen in Figure 5.

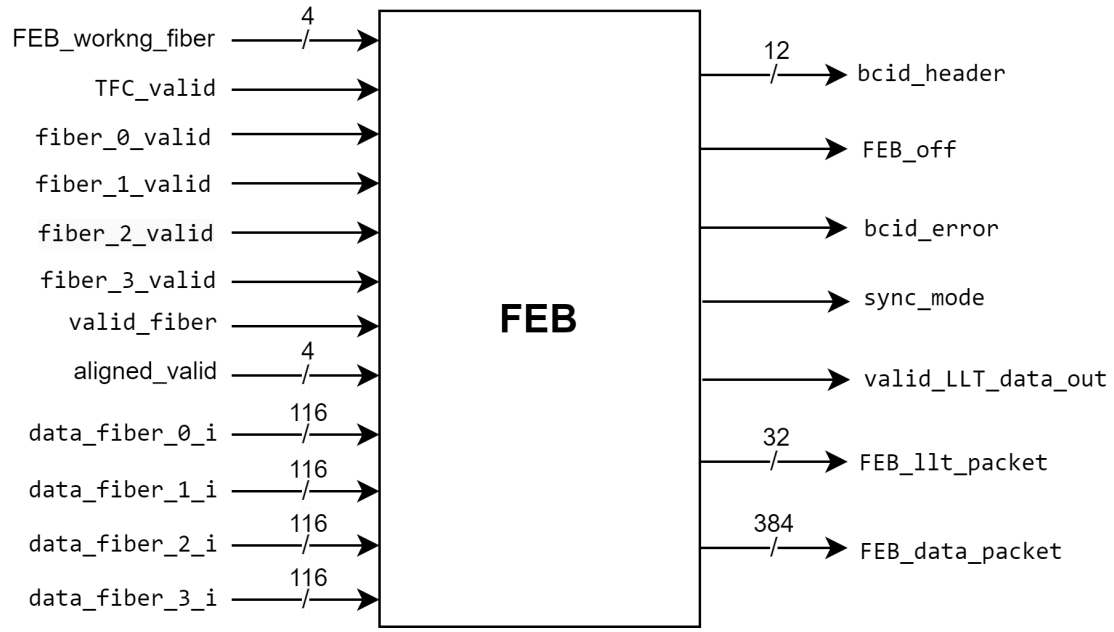

Figure 5. Block diagram of the Front-End Board function in the kernel

Frame assembly submodule takes the output of 3 FEB modules and assembles frames of data according to the outputs of FEBs. The inputs and outputs of the frame assembly submodule can be seen in Figure 6.

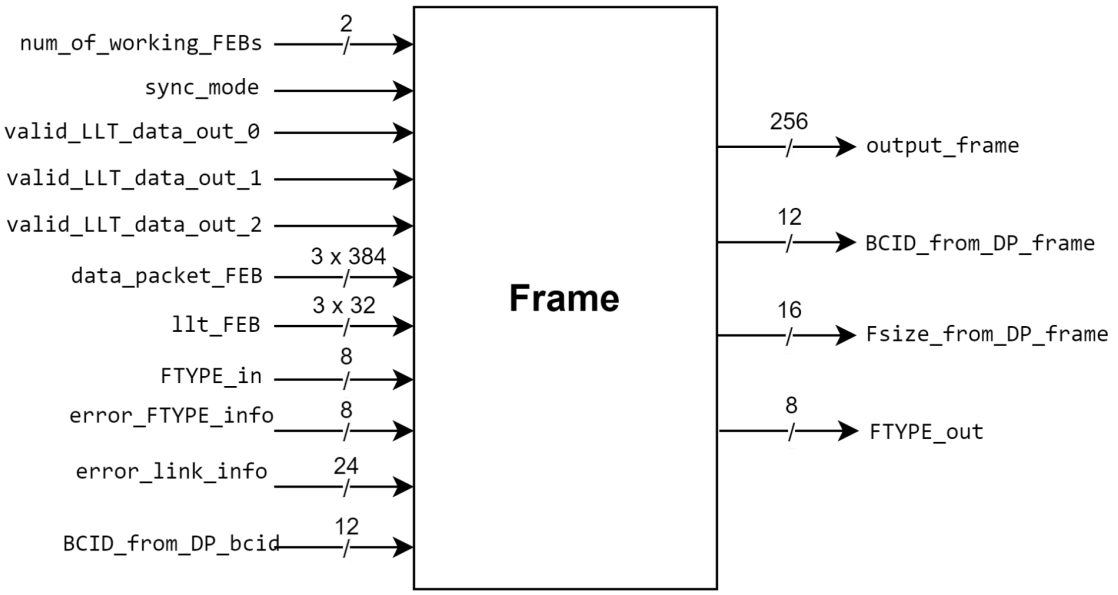

Figure 6. Block diagram of the Frame Concatenate function in the kernel

The error manager module handles all the errors inside the unit and does the FTYPE processing. The block diagram of the error manager can be seen in Figure 7.

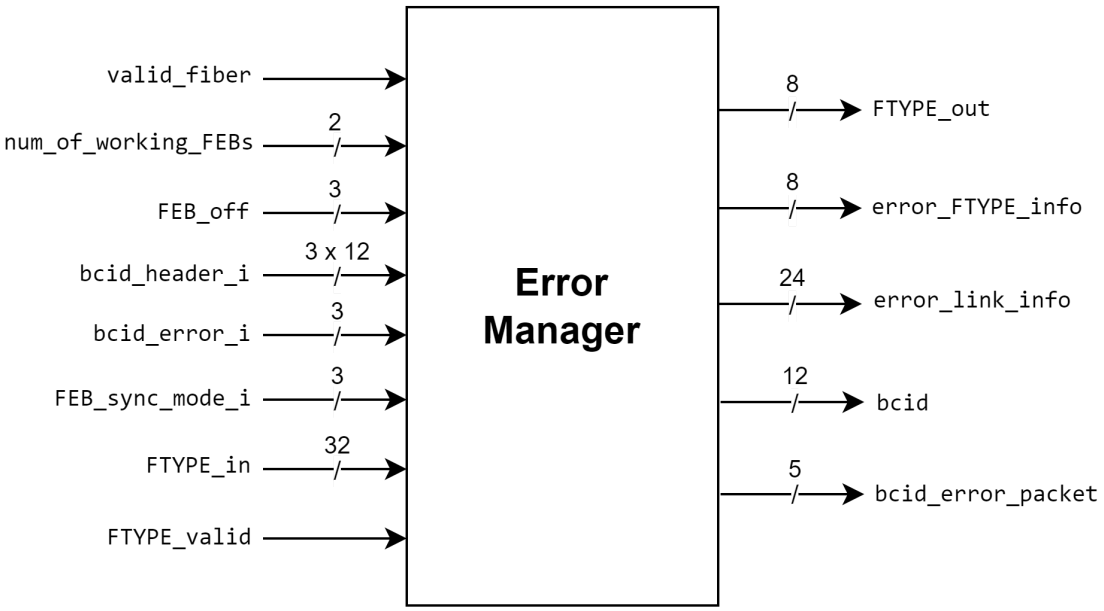

Figure 7. Block diagram of the Error Manager function in the kernel

4. Problems encountered in implementation

Through the course of the project, different problems were encountered. Some of them were solved and some of them remain. In this section, some of the common problems of the IP flow will be explained.

Clock domain crossing

Intel oneAPI does not natively support clock domain crossing, so this aspect of the design must still be implemented using traditional HDL methods. One kernel can be clocked with only one clock signal. If there is a need to use multiple clock signals, one kernel for each of them must be designed.

Custom interface

Some problems with custom interfaces of the kernel were encountered. When using pipes, oneAPI will implement the Avalon ST interface. Some simple custom signals can be added using conduits as annotated arguments of the kernel. A developer has to be careful when using annotated arguments for continuous reading throughout the execution because the readout of the annotated arguments is connected to busy signals of the invocation interface.

Reset signals

One kernel can have only one reset signal. If there is a need to have multiple reset signals for different submodules, each submodule with a different reset signal has to be a separate kernel. Communication between kernels can be implemented using pipes.

Another way would be to use only one kernel and pass the reset signals using conduits and annotated arguments. The potential problem would arise with synchronization between the reset signals passed through conduits and the default reset signal of the whole kernel.

Reading valid signals of pipes

The valid signals of blocking pipes are not natively accessible. If the valid signal of the input pipe has to be read non-blocking pipe has to be used. When using the non-blocking pipe, the success value of the read is returned.

Loop exit condition

The function that outputs PCIe frames had a problem with the exit condition. The II was not 1 and the delay of 6 cycles was shown between packets. This was fixed using the **[[intel::speculated_iterations(0)]]**.

Stall-free logic (exit FIFOs)

OneAPI by default inserts exit FIFOs that make the kernel stall-free. These FIFOs consume distributed RAM (MLAB) and if MLABs are scarce there is an attribute that tells the DPC++ compiler to exclude the exit FIFOs - **[[intel::use_stall_enable_clusters]]**. This will increase the utilization of LUTs and FFs because of stall-enabled logic. [11]

FPGA Emulator

The compilation for FPGA emulator is found to be not very useful for debugging, but useful for fast checking of functionality of the code. The drivers of the emulator input stimulus are not cycle-accurate. There is little control over when the inputs are written when using the emulator.
Another problem that was encountered was that the emulator complained about a property of pipes called bits_per_symbol. The width of the pipe has to be a multiple of bits_per_symbol. That condition was met, but the emulator still reported an error about a mismatch. Compilation for the simulator and FPGA implementation didn’t have that problem.

FPGA Simulator

The compilation for the FPGA simulator uses the same input stimulus drivers as the emulator so the problem with input control is still present. The simulator allows the view of waveforms which is useful for checking of the iteration interval and kernel latencies (and functionality of course). After compilation for the simulator, the tool outputs qsys file which can be used to create a custom test bench. When using the custom test bench (for example in SystemVerilog) the problem of the input stimulus control is not present.

5. Results

The kernel was compiled for FPGA emulator and FPGA simulator. Besides the default oneAPI-generated simulation, a custom test bench was written. Data resembling the calorimeter input is generated on the input side of the test bench. The output was checked from the waveforms and debug prints. The waveforms of the simulation can be seen in Figure 8.

Besides the standalone kernel simulation, the simulation in the whole PCIe40 chain was done. The HLS generated module was placed in the PCIe40 simulation environment and the data was seen to be produced on the output.

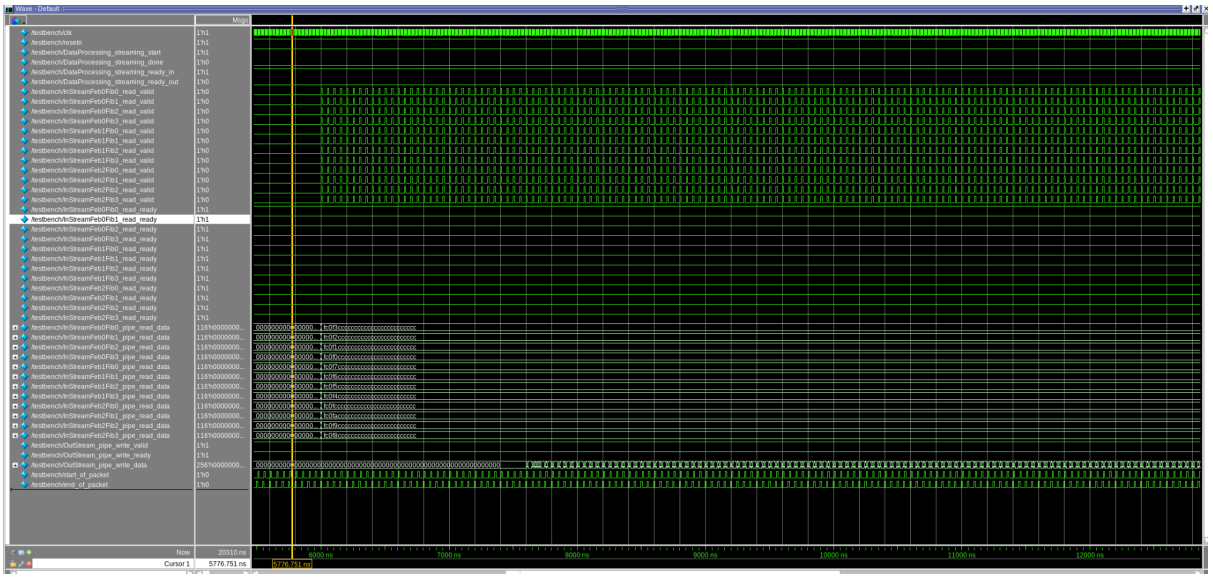

Figure 8. Waveforms of the standalone kernel simulation using a custom testbench

The comparison between the FPGA resource utilization of the HDL implementation and the HLS implementation can be seen in Table 1. The column *Other DP units* is the equivalent of the oneAPI implementation that is written in HDL so the oneAPI implementation is going to be compared to that column. The table shows the result of the compilation for the FPGA board Intel Arria 10.

When looking at the utilization of the DSPs, the oneAPI implementation doesn't use any of the DSPs while the HDL uses 2. The block RAM (BRAM) is not utilized in oneAPI implementation while some of the bits are stored in BRAMs in HDL implementation.

The first iteration of oneAPI implementation contains the exit FIFOs which are inserted by default. The presence of the exit FIFOs is the reason for the increased number of distributed RAM bits (MLAB). The second iteration does not contain the exit FIFOs and we can see that the number of distributed RAM bits is halved. For the first iteration of oneAPI implementation, the number of utilized FFs is comparable to the HDL implementation, while the second is increased due to the stall enable logic.

Entity	LUTs	FFs	Distributed RAM (bits)	BRAM (bits)	DSPs
Data Processing	22774	31665	67704	33852	2
DP/A input and rerouting	15306	18686	48120	24060	0
Other DP units	7468	12979	19584	9792	2
OneAPI implementation	**6348**	**13084**	**106752**	**0**	**0**
OneAPI without exit FIFOs	**13569**	**41804**	**53888**	**0**	**0**

Table 1. FPGA resources utilization when compiling for Intel Arria 10

The most scarce resource are LUTs. The first iteration has a lower utilization of LUTs than the HDL implementation by approximately 15%. The HDL implementation contains the ECS interface while the oneAPI implementation does not which is probably the reason for the difference. In the second iteration, the stall enable logic increased the utilization of the LUTs.

Overall, the oneAPI implementation gave satisfying results in comparison to the traditional HDL implementation. The results of oneAPI implementation are completely comparable to the HDL implementation, still leaving room for further optimization and fine-tuning. When taking into account the time that was needed to develop the custom HDL version versus the time that was spent during this project, oneAPI turned out to increase the productivity of a developer and to allow for rapid development and testing of the kernels.

6. Future developments

While the simulation within the entire PCIe40 chain has been completed, the waveforms have yet to be fully debugged. This is a crucial task that needs to be revisited in the future. Along with thorough waveform analysis, checking all corner cases of the kernel will provide a more precise assessment of the quality of the oneAPI implementation.

Two modules still need to be integrated into the oneAPI implementation: the input and rerouting submodule, and the ECS interface. The input and rerouting submodule involves clock domain crossing, which must still be implemented using traditional HDL methods. However, the rerouting portion of this submodule can be handled by oneAPI. The ECS interface, which is memory-mapped, can be implemented more efficiently using the Avalon MM interface provided by oneAPI.

The FPGA resource utilization, as shown in the results, can be further optimized by applying various attributes and fine-tuning the kernel. The oneAPI compiler offers opportunities for user-specific tuning, which can be valuable for achieving an optimized design.

Once these tasks are completed, the kernel must be tested on the actual FPGA hardware. This final step will allow for a more precise evaluation and verification of the oneAPI IP flow.

7. References

[1] *LHCb Upgrade Calorimeter* [TELL40 Data Processing]. (2021, February). https://edms.cern.ch/ui/file/1889215/4/LHCb_Upgrade_Calorimeter_TELL40_Data_Processing.pdf

[2] *LHCb Upgrade 'Sub-detector CALO' Front-End Data Format*. (2021, March).

[3] *LHCb's unique approach to real-time data processing*. (2023, March 1). LHCb Outreach. https://lhcb-outreach.web.cern.ch/2023/03/01/lhcbs-unique-approach-to-real-time-data-processing/

[4] K. Hennessy et al., "Readout Firmware of the Vertex Locator for LHCb Run 3 and Beyond," in IEEE Transactions on Nuclear Science, vol. 68, no. 10, pp. 2472-2479, Oct. 2021, doi: 10.1109/TNS.2021.3085018.

[5] *Intel oneAPI tool overview*. https://www.intel.com/content/www/us/en/developer/articles/technical/oneapi-what-is-it.html

[6] *Intel oneAPI Programming Guide.* https://cdrdv2-public.intel.com/785315/oneapi_programming-guide_2024.0-771723-785315.pdf

[7] *Intel oneAPI Github link for HLS flow interfaces.* https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL_FPGA/Tutorials/Features/hls_flow_interfaces

[8] *Intel oneAPI Github link for invocation interfaces.* https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL_FPGA/Tutorials/Features/hls_flow_interfaces/invocation_interfaces

[9] *Intel oneAPI Github link for streaming data interfaces.* https://github.com/oneapi-src/oneAPI-samples/tree/master/DirectProgramming/C%2B%2BSYCL_FPGA/Tutorials/Features/hls_flow_interfaces/streaming_data_interfaces

[10] *Intel oneAPI documentation for Speculated interactions attribute.* https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/optimization-guide/2023-1/speculated-iterations-attribute.html

[11] *Intel oneAPI Optimization guide, stall enable attribute.* https://www.intel.com/content/www/us/en/docs/oneapi-fpga-add-on/optimization-guide/2023-1/reduce-kernel-area-and-latency.html