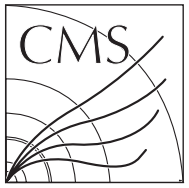
Available on CMS information server

CMS CR -2016/287



The Compact Muon Solenoid Experiment

Conference Report

Mailing address: CMS CERN, CH-1211 GENEVA 23, Switzerland

27 October 2016 (v3, 15 November 2016)

Software and Firmware co-development using High-level Synthesis

Nikhil Pratap Ghanathe for the CMS Collaboration

Abstract

Accelerating trigger applications on FPGAs (using VHDL/Verilog) at the CMS experiment at CERN's Large Hadron Collider at CERN warrants consistency between each trigger firmware and its corresponding C++ model. This tedious and time consuming process of convergence is exacerbated during each upgrade study. High-level synthesis, with its promise of increased productivity and C++ design entry bridges this gap exceptionally well. This paper explores the single source code approach using Vivado-HLS tool for redeveloping the upgraded CMS Endcap Muon Level-1 Track finder (EMTF). Guidelines for tight latency control, optimal resource usage and compatibility with CMS software framework are outlined in this paper.

Presented at *TWEPP-16 Topical Workshop on Electronics for Particle Physics*

Software and Firmware Co-Development Using High-level Synthesis

Nikhil Pratap Ghanathe[a*], Alexander Madorsky[a], Herman Lam[a], Darin E. Acosta[a], and Alan D. George[a]

[a] *University of Florida,*
Gainesville, Florida -32611, United States of America
E-mail: nikhilghanathe@gmail.com

ABSTRACT: Accelerating trigger applications on FPGAs (using VHDL/Verilog) at the CMS experiment at CERN's Large Hadron Collider at CERN warrants consistency between each trigger firmware and its corresponding C++ model. This tedious and time consuming process of convergence is exacerbated during each upgrade study. High-level synthesis, with its promise of increased productivity and C++ design entry bridges this gap exceptionally well. This paper explores the "single source code" approach using Vivado-HLS tool for redeveloping the upgraded CMS Endcap Muon Level-1 Track finder (EMTF). Guidelines for tight latency control, optimal resource usage and compatibility with CMS software framework are outlined in this paper.

KEYWORDS: High-level Synthesis; Vivado-HLS; Latency control; Productivity;

*Corresponding author.

Contents

1. Introduction

Acceleration of trigger applications for the CMS experiment in the Large Hadron Collider at CERN has been traditionally performed on FPGAs using hardware description languages (HDLs) such as VHDL and Verilog. Specifics of large-scale high-energy physics experiments require that each trigger firmware design must be accompanied by a software model that can be used for analyzing its performance, verification of hardware functionality, and other tasks. These software models must be designed in C++ and be compatible with the CMS software framework, CMSSW.

Since CMS upgrades firmware algorithms and trigger hardware at regular intervals, the software models must be constantly kept synchronized with firmware algorithms. The typical approach is to write these models independently. Since the trigger firmware designs are substantially complex, creating and maintaining the software models that exactly match firmware behavior is a major challenge. The final convergence between the firmware and software models is especially tedious; it sometimes takes months to find and fix small mismatches. Since 2002, the CSC track finder system of the CMS Endcap Muon Level-1 trigger has maintained the C++/RTL (register-transfer level) model consistency by using a homemade VPP [1] library which automatically generates C++ and Verilog files from a single source code. This approach worked extremely well for the less complex firmware design of the legacy system, providing full consistency between firmware and software model automatically. However, the

recent hardware upgrade brought much larger FPGAs and much more complex firmware algorithms, and VPP has become inadequate for this task.

Recent advancements in high-level synthesis (HLS) tools hold the promise of high productivity through the use of design entry in C++ that reduces the difficulty for developing and managing code complexity at the HDL level. However, the major challenge in using HLS is to be able to use C++ constructs to perform fine-grained control of the generated firmware in such a way that it satisfies the constraints of CMS trigger applications:

- Stringent latency requirements
- Limited FPGA resources
- Compatibility with CMSSW

In this paper, we present our exploration into using a "single-source code" approach in which we perform software and firmware co-development using Vivado HLS, a C++-based high-level synthesis tool used for Xilinx FPGAs. Vivado HLS enables us to have a single-source code, which can be used as the C++ model for verification by physicists and to generate the RTL model to synthesize firmware for the FPGA.

2. High-level synthesis languages and tools

High-level synthesis holds considerable promise in mitigating the cost of firmware development. It allows the designer to orchestrate the synthesis of hardware from a higher level of abstraction. A typical HLS tool consists of a special type of compiler which allows the designer to implement their designs using a high-level language and then translates the high-level language description into a RTL description such as VHDL/Verilog. Due to the possibility of using mainstream software languages such as C/C++ as the design entry, it enables developers to speed up design space exploration while increasing flexibility and reducing development time.

There are many HLS tools available today and the choice of any HLS tool depends on a broad set of criteria such as source language, ease of implementation, tool complexity, support for data types, verification, latency, and resource usage after synthesis. Some of the HLS tools that were evaluated for our research were Altera OpenCL for FPGAs [2], BlueSpec [3], and Vivado HLS [4]. The key evaluation criteria for this project are good latency control, resource usage after synthesis, compatibility of the source code in CMSSW, and ease of verification. For our development, Vivado HLS perfectly aligned with the stated requirements.

Vivado HLS is an architecture-aware, directives-driven compiler from Xilinx. The design flow starts with an algorithmic description in C/C++. The design is then functionally verified using a C testbench. The Vivado HLS compiler synthesizes the RTL design by extracting the control and datapaths from the HLS code, and maps it to hardware by using scheduling and binding processes while considering the directives supplied by the user. Vivado HLS supports bit-accurate validation and provides a useful feature called C/RTL co-simulation where the C and the RTL design can be co-simulated and validated.

A key attribute of Vivado HLS critical for this study is that the user can override the defaults of the HLS compilation process by adding directives (pragmas) to the developed code to satisfy performance and other requirements and thereby has excellent control over the synthesized RTL. Just as importantly, the HLS code is compatible with CMSSW [5].

3. CMS Level-1 Endcap Muon Track Finder

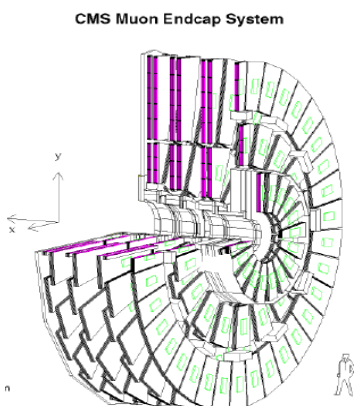


Fig1: CMS Endcap Muon system

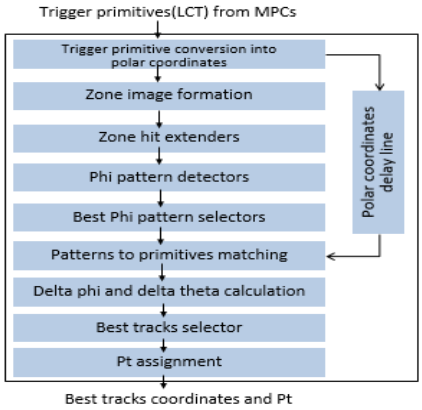


Fig2: CSC track finder logic

The objective of the CMS [6] Level-1 Muon Trigger is to detect and efficiently retain muons with the lowest possible transverse momentum (P_T) that meets the rate reduction requirement of less than 100 kHz out of a 40MHz input rate.

The CMS endcap region consists of 4 stations (Fig 1) and each station is comprised of cathode strip chambers (CSC). For ease of processing, the endcap region is divided into six 60 degree azimuthal sectors. The track finder logic [7] in each sector, as shown in Fig 2, has 10 modules which reconstruct tracks from the track segments a muon registers when it traverses all 4 muon stations. The track finder logic looks for track segments in predefined patterns across all 4 stations.

The CMS Endcap Muon Level-1 track finder (EMTF [8]) module is a part of the Level-1 trigger architecture with a stringent latency requirement. As the design and algorithmic complexity of trigger algorithms have steadily increased with each upgrade study, the prototyping and validation processes have also grown increasingly complex. Also, a major challenge for the trigger algorithms is the uncompromising latency constraint. The permitted latency of the EMTF is 15 clock cycles at 25ns per cycle. To satisfy the stringent latency requirement, it is necessary to control the low-level RTL constructs from the higher abstraction layer. Our study aims to develop the EMTF firmware using Vivado HLS such that the firmware complies with the uncompromising latency and compatibility requirements while increasing the design productivity and flexibility considerably.

4. Increased productivity and flexibility using HLS

High-level synthesis with its automation and refinement of the algorithmic description to the RTL level reduces the burden of code development and verification drastically. The amount of code reduces dramatically, saving time and minimizing the probability of mistakes. The higher level of abstraction eases the handling of increased design complexity and opens avenues for extensive design space exploration with less effort. The following lessons learned from our study illustrate the productivity features of HLS.

4.1 Increasing throughput using HLS

Performance throughput of a design can be limited through several reasons such as memory bottleneck, limited pipelined design, false dependencies, and so on. Attaining maximum throughput for any design requires the developer to systematically analyze the code and optimize it to obtain the best possible design throughput. The following example describes how Vivado HLS features can be used to achieve a throughput constraint of 1 output per clock cycle in the Trigger primitive conversion module (module-1).

Consider the for-loop in Fig 3 from the Trigger primitive conversion module. The for-loop contains 9 iterations, with each iteration being independent from each other and each iteration accesses a look-up table (LUT) called "params". The synthesized design, without any optimization directives (pragmas), resulted in a design with a throughput of 1 output in 9 clock cycles. Also, the default behaviour of the Vivado HLS compiler is to implement the LUT (declared as a 1-D array in the HLS code) as a Block-RAM (BRAM), thus resulting in a memory bottleneck which schedules each of the 9 iterations sequentially. To improve the design throughput, we had to resolve the bottlenecks and coerce the compiler to synthesize a parallelized design. This was accomplished using the following approach.

```
#pragma HLS PIPELINE II=1
for (i=0;i<9;i++){
   #pragma HLS UNROLL
       switch(clctpat[i]){
        case 0 : aclct_pat_corr= 0x0;
                 fph[i] = params[ph_init_ix] + ph_tmp;
```

Fig 3: Example HLS code illustrating loop unrolling

1. The operation of the for-loop was parallelized using the **HLS UNROLL** directive, which conveys to the compiler that the user wants to have multiple copies of the for-loop body in the synthesized RTL.
2. The **ARRAY_PARTITION** directive (Fig 4) partitions and resolves the N-element array (LUT) into N individual registers. The result is a design that contains N individual registers, which effectively removes the memory bottleneck.
3. The **HLS PIPELINE** directive pipelines the entire design to obtain a throughput of 1 output per clock cycle.


```
#pragma HLS ARRAY_PARTITION variable=params complete dim=1
```

Fig 4: Array partitioning

Using a simple set of directives, multiple iterations are unrolled, each having its own loop body without the designer needing to worry about synchronization issues as required in an HDL design. The throughput of the design has now improved from 1 output in 9 clock cycles to 1 output per clock cycle. Furthermore, the corresponding HDL design would require the designer to manually pipeline the design and individually create and track register assignments.

4.2 Flexibility - Instantiation of multiple identical modules

The nature of trigger algorithms is such that they demand massive parallelization. Handling of extremely large problem instances and processing immense data sizes while guaranteeing throughput establishes the need for parallelization. Such a design in the Phi pattern detector module (module-4) of the EMTF algorithm requires 488 separate instances of a function with an additional need that all variables inside each function instance be persistent. This seemingly onerous task (in HDL) was accomplished effortlessly by adopting an object-oriented approach (OOPS) using Vivado HLS.

```
static test inst[5]; //create array of 5 objects
// partition the array completely
#pragma HLS ARRAY_PARTITION variable=inst complete dim=1
// unroll loop
for(int i=0;i<5;i++){
#pragma HLS UNROLL
            inst[i].test_func(a[i],b[i],index[i],&c[i]);
      }
```

Fig 5: Example HLS code to achieve massive parallelism

With some careful investigation and experimentation, a design technique (Fig 5) based on OOPS was developed to successfully satisfy the specified requirements. The technique consists of the following 3 steps:

1. An array of objects is defined with the keyword "**static**".
2. The array of objects is partitioned completely using the **ARRAY_PARTITION** directive.
3. The for-loop that schedules the multiple instances of the function is completely unrolled using the **HLS UNROLL** directive.

The static keyword on the object makes all the member variables persistent over different function invocations. The ARRAY_PARTITION and the HLS UNROLL directive helps create multiple independent instances. Thus, by adopting our developed method, we were able to attain massive parallelism in the Phi pattern detectors module without worrying about synchronizations or keeping track and updating of innumerable variables to have persistence as required in a VHDL/Verilog implementation.

5. Fine-grained control using HLS constructs

A major challenge in using HLS for firmware development is to be able to use high-level HLS programming constructs to perform fine-grained control of the generated firmware to satisfy stringent constraints. For example, in the Level-1 trigger, exercising complete control over the latency of the design is of paramount importance. While it is indeed challenging to control the lower level RTL implementations from the HLS level, the task is not as strenuous as one might imagine. To exercise strong control over the latency of the generated design, some novel coding guidelines and design techniques were devised in the process of our study. The following examples show how we were successful in exacting control from the Vivado HLS compiler and emulate HDL-like control employing these schemes.

5.1 Latency control - Scheduling of functions and operations

The Sorter module of the EMTF algorithm outputs the 3 best-quality tracks from each of the 4 zones in an azimuthal sector. In simpler words, the module operation consists of selecting the 3 highest numbers from an unsorted array. The baseline Verilog code implements the module by constructing a comparison tree, retrieving the highest number in the first clock cycle, the second highest in the second clock cycle, and so on. The latency of the baseline Verilog implementation is hence 3 clock cycles. This corresponds to 3 separate function calls in HLS. In addition, it is desirable to reduce the latency of the new design from 3 to 2 clock cycles. The following two approaches were investigated:

1. A synthesis of the design with the directives (like PIPELINE, UNROLL, etc.) for maximum throughput still resulted in a latency of 3 clock cycles, with each invocation of the "sort" function needing 1 clock cycle.
2. Our next approach to optimize the latency was to allow the compiler to optimize the design in any way it saw fit (using the INLINE directive) but with a hard-line latency upper bound of 2 clock cycles (specified using the LATENCY directive). It was observed that after synthesis, the compiler tries to fit the entire design in 1 clock cycle, resulting in the critical path delay exceeding the clock period. Further analysis revealed that the undesired behaviour was the result of the INLINE directive, which performed optimization at its own discretion.

It was noted in our investigation that neither having full control nor allowing HLS to take full control of the synthesis process was instrumental in improving the latency. Hence, a method was devised that partially gave the control to the compiler and partially to the user:

1. A duplicate version of the sort function was created and renamed "sort_1". This is essentially the same "sort" function albeit with a different name.
2. The first function call to "sort" was not in-lined, but the next two function calls were in-lined.

```
sort  (inp_array, winner0, &winid[0]);   // INLINE OFF
sort_1(inp_array, winner1, &winid[1]);   // INLINE
sort_1(inp_array, winner2, &winid[2]);   // INLINE
```

Fig 6: Example HLS code illustrating scheduling of functions

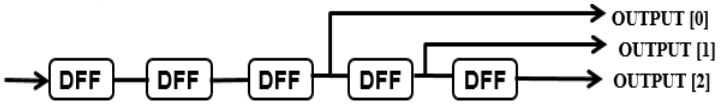
By not in-lining the first call, we restrict the HLS compiler from optimizing it in any undesired manner and hence forcing the compiler to schedule it in a separate clock cycle; i.e., the 1st clock cycle. The next two function calls that are in-lined allow the HLS to optimize it and schedule them both in the 2nd clock cycle. Thus, by striking the right balance between the amount of control given to the HLS compiler and the amount of control retained by the user, we were able to control the scheduling of the operations. As a result, the latency of the sorter module was improved to 2 clock cycles, saving a valuable cycle in the EMTF algorithm, as opposed to 3 clock cycles in the corresponding Verilog implementation.

5.2 Construction of a delay line

A shift register is one of the most widely used digital components in a data processing system. Designing a shift register involves controlling operations and scheduling on each clock edge thus requiring very fine-grained control. The Polar angle coordinate delay module (module-6) employs a similar design to implement a delay line for the outputs generated from the Trigger

primitive conversion module for use in the Patterns to primitive matching module (module-7). Realizing a shift register at the HLS level of abstraction is not trivial, as it requires tight latency control.

Our first attempt at synthesizing a shift register yielded a design where all the intermediate registers between the input and the outputs were eliminated (i.e., synthesized away). Careful investigation revealed that the Vivado HLS compiler thinks that the intermediate registers are just redundant assignments and hence removes them.




```
void test (ap_uint <4> in, ap_uint<4> out[3]) {
volatile ap_uint<4> temp[5];
temp[4]=in;
for(i=4;i>0;i--){
  #pragma HLS unroll
    temp[i-1] = temp[i];
}
out[2]=temp[0];
out[1]=temp[1];
out[0]=temp[2];
}
```

Fig 7: Example HLS code illustrating synthesis of a shift register

A simple coding technique was developed to force the HLS compiler to consider/preserve every intermediate assignment. This is accomplished by adopting the approach shown in Fig 7.

1. A shift register is explicitly created as shown in the for-loop by assigning the previous element of the array to the next element and so on.
2. The array which is used to create the shift register is declared as "**volatile**". This important keyword coerces Vivado HLS into synthesizing a shift register by preserving each operation/assignment involving the "volatile" array.

6. Resource usage comparison

The design flow of a high-level synthesis process provides us with an extra level of enhancement of our implementation by refining the algorithmic description using the HLS compiler. As a result, our design now undergoes two levels of optimization, one during synthesis from HLS code to RTL and the other from RTL to bitstream. With a mature HLS compiler like Vivado HLS and the efficient coding practices and techniques developed in our study, it was observed that our HLS design of the EMTF algorithm occupies less area as compared to the baseline RTL implementation in the majority of the cases, as shown in Table 1.

- The HLS versions of the Phi-pattern detectors, Patterns to primitive matching, and the Polar Angle Co-ordinate delay modules were observed to exhibit decrease in the resource usage compared to their corresponding Verilog implementations.

- The rest of the modules were observed to use the same amount of logic resources as their corresponding Verilog implementations, with only the Trigger primitive conversion module being an exception.

Module	HLS (% LUTs)	Verilog baseline (% LUTs)
Trigger primitive conversion	12 %	6%
Zone image formation	1%	1%
Zone hit extender	1%	1%
Phi-pattern detector	11%	16%
Sorter	3%	3%
Polar Co-ordinate delay	0% (uses FFs)	2%
Patterns to primitive matching	10%	16%
Delta phi and delta theta calculation	2%	2%

Table 1: Resource statistics of EMTF modules

7. HLS code compatibility and performance comparison on CMSSW

CMSSW is the collection of all the software developed for CMS built around a framework, an event data model (EDM) and other services for data analytics. It has an extensive toolkit, which is used to carry out analyzes of data. The primary objective of CMSSW is to facilitate the development of software for reconstruction and analyzes. Any software developed for CMS must be compatible in the CMSSW environment. In simpler words, the HLS code developed must be compatible with a gcc/g++ compiler.

As stated in Section 2, one main reason that Vivado HLS was chosen for this project is that it is a C/C++-based language, which allows for a simpler path to CMSSW compatibility. The main challenge to achieve this compatibility is the presence of special data-types in the HLS code called arbitrary precision data-types, which allow the designer to define input and output ports with an arbitrary number of bits. A C-based design compiled with a gcc compiler does not recognize these data-types and hence fails to reflect bit-accurate behavior. However, after some investigation, it was observed that a C++ based design supports the use of arbitrary precision data-types defined in the SystemC standard. Thus, by using a C++ based design and headers, the HLS code can be compiled, without change, using a g++ compiler hence making it compatible with the CMSSW environment.

The performance of the HLS code was also compared to that of the previous manually written C++ code (emulator code) on CMSSW. It was observed that the emulator code was faster by only a factor of 2, a tolerable factor. The HLS and emulator code for the Trigger primitive conversion module (module-1) were compiled in CMSSW and the execution measured times are shown in Table 2.

	HLS execution time (s)	Emulator code execution time (s)
Trigger primitive conversion module	**1.197 e-07**	**5.836 e-08**

Table 2: Performance comparison of HLS and emulator code

8. Summary and Conclusions

FPGAs remain the indisputable choice today in accelerating trigger applications in CMS and it is highly unlikely to change anytime soon. The desire for more processing capabilities for CMS trigger applications with each upgrade study results in an increased design complexity, lengthy code development time and higher verification effort. High-level synthesis languages and tools provide us with an excellent opportunity to lower these barriers considerably.

This paper presents the EMTF algorithm as a case study in illustrating the convenience and advantages of using HLS for firmware development. The paper also outlines several coding methods and devises techniques to exercise control over the synthesized RTL. Example guidelines for CMSSW compatibility were also established.

It was observed that all the modules of the EMTF satisfied the stringent performance constraints. The resource usage statistics were better than the Verilog implementation in the majority of the cases. At the time of writing this paper, all 10 modules of the EMTF algorithm (Fig 2) have been verified via simulation. Four out of the 10 modules have been successfully tested in firmware on the Xilinx **Virtex-7 XC7VX690T** FPGA for 1000 events containing complete tracks. The hardware output of the Vivado HLS generated RTL code matches perfectly the output of the baseline Verilog implementation.

Acknowledgments

This work was done in collaboration with The Institute for High Energy Physics and Astrophysics (IHEPA) and the NSF Center for High Performance Reconfigurable Computing (CHREC) at University of Florida, supported in part by the I/UCRC Program of the National Science Foundation under Grant Nos. EEC-0642422 and IIP-1161022.

References

[1] A. Madorsky and D. E. Acosta, "VPP - a Verilog HDL simulation and generation library for C++," *2007 IEEE Nuclear Science Symposium Conference Record*, Honolulu, HI, 2007, pp. 1927-1933. doi: 10.1109/NSSMIC.2007.4436533

[2] https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/wp/wp-01173-opencl.pdf

[3] http://www.bluespec.com/high-level-synthesis-tools.html

[4] http://www.xilinx.com/support/documentation/sw_manuals/xilinx2015_4/ug902-vivado-high-level-synthesis.pdf

[5] G.L. Bayatian et al. (CMS Collaboration), "CMS Physics Technical Design Report, Volume 1:Detector Performance and Software," CERN/LHCC 2006-001 (2006)

[6] CMS Collaboration, JINST 3 S08004 (2008)

[7] https://twiki.cern.ch/twiki/bin/view/CMS/L1CSCTrackFinder

[8] Tapper, A *et al*, "CMS Technical Design Report for the Level-1 Trigger Upgrade" - CERN-LHCC-2013-011