Implementation of OMTF trigger algorithm with High-Level Synthesis

Wojciech M. Zabołotny[a,b]

[a]Warsaw University of Technology, Institute of Electronic Systems, ul. Nowowiejska 15/19, 00-665 Warszawa, Poland
[b]University of Warsaw, Institute of Experimental Physics, ul. Pasteura 5, 02-093 Warszawa, Poland

ABSTRACT

The paper presents the implementation of the Overlap Muon Track Finder algorithm using the High-Level Synthesis (HLS). That algorithm has been previously implemented in a highly optimized VHDL code. Therefore, comparison of the implementations based on VHDL code, and on HLS provides an assessment of the suitability of HLS for implementation of complex algorithms. Additionally, analysis of the correlation between the HLS code and parameters of the produced solution gives suggestions regarding the correct implementation of typical data-processing algorithms in HLS.

Keywords: HLS, FPGA, OMTF, Trigger algorithm, Logic synthesis, VHDL

1. INTRODUCTION

High-Level Synthesis[1] draws the attention of many system designers and FPGA developers. It has created a hope that C programmers will be able to efficiently implement algorithms in FPGA. The previous experiments with HLS applied to the stream heap-sorting system[2] have shown, that for such simple algorithms usage of HLS in certain conditions may result in better performance, than thoroughly handcrafted HDL code. Even though HLS was not able to achieve the shortest possible Initialization Interval (II) of the pipelined design, for more relaxed requirements, it provided the solution with significantly smaller resource consumption. Those results encouraged testing the suitability of the HLS approach to more complex algorithms. The good candidate is the Overlap Muon Track Finder algorithm,[3] that was carefully optimized both regarding the resource usage and achievable speed of processing.

2. STRUCTURE OF THE ALGORITHM

The Overlap Muon Track Finder (OMTF) algorithm has been developed for the upgrade of the muon trigger in the CMS experiment, and has been used during the LHC Run 2.[4] The OMTF algorithm is described in detail in the articles Ref. 3 and Ref. 5, and the theory behind it is shortly described in Ref. 6. In the CMS detector, the bunches of accelerated particles collide roughly every 25 ns. Those collisions are called bunch crossings (BX). The muons produced during the collision are passing through the CMS detector and generating the electrical signals, called hits in the 18 layers of the detector. The task of the algorithm is to find the transversal momentum p_T of those muons. The algorithm uses averaged tracks (patterns) generated in physical simulations for different values of p_T and finds the track that is best matched to the observed set of hits. The main concept of the algorithm is shortly presented in Figure 1. The PDF values are based on the logarithm of the probability that the muon with particular p_T generates hit with the particular azimuthal angle. The precalculated PDF values are stored in lookup tables.

The block diagram of the hardware that performs the OMTF algorithm is shown in Figure 2. The following is a short and simplified description of the algorithm explaining the structure of the hardware implementation. The full description may be found in the referenced articles.[3,5]

- In the first stage, the hits recorded in specially selected “reference layers” are analyzed. They are ordered by priority. From each BX up to 4 hits from those layers, so-called “reference hits”, with highest priorities are taken and used as a basis for track reconstruction.

Further author information: (Send correspondence to W.M.Z.)
W.M.Z.: E-mail: wzab@ise.pw.edu.pl, Telephone: +48 22 234 7717

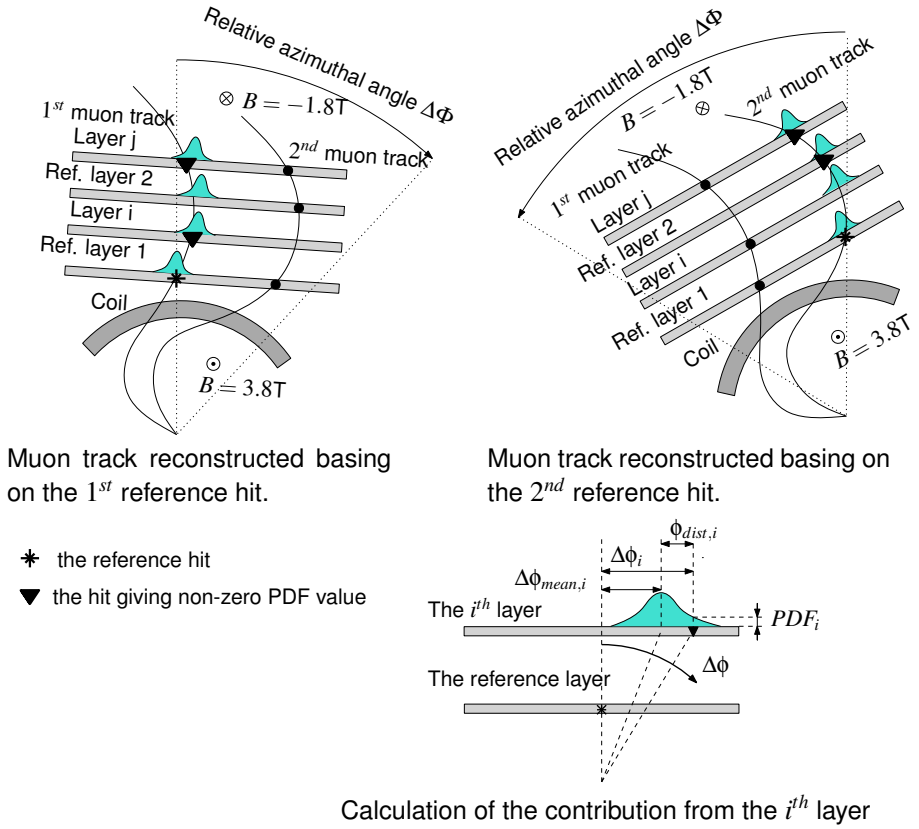

Figure 1: Reconstruction of two muon tracks with OMTF algorithm. Two reference hits from the first reference layer are used. Each of them is associated with another muon track. The lower figure shows how the PDF contribution from the particular detector layer is calculated. The figure is reproduced with minimal modifications from the Open Access publication Ref. 5 according to the CC-BY 4.0 license.[7]

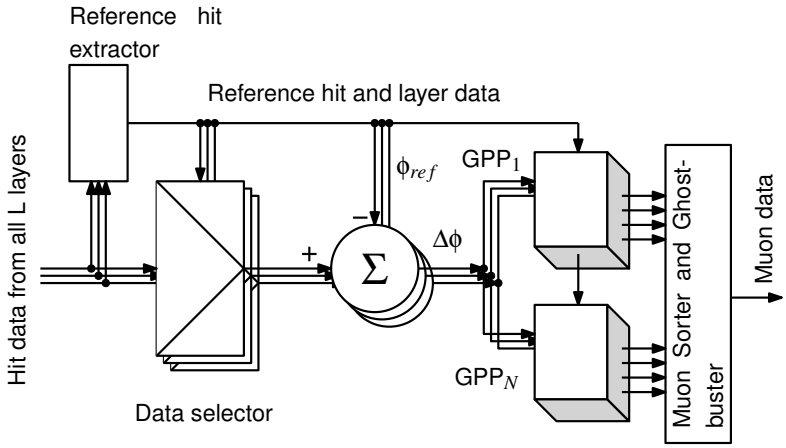

Figure 2: Block diagram of the OMTF algorithm implementation. The GPP (Golden Pattern Processor) units are matching the delivered hits with the individual sets of track patterns. The figure is reproduced with minimal modifications from the Open Access publication Ref 5 according to the CC-BY 4.0 license.[7]

- Next, the detector links, that are likely to provide the hits belonging to the muon track associated with the selected "reference hit" are selected. The azimuthal angle ϕ_{ref} of the reference hit is subtracted from the azimuthal angle ϕ of those hits, forming the $\Delta\phi$ values.
- Further processing is done in parallel for multiple patterns corresponding to different values of the p_{T} and different sign of the charge of the muon[*]
  - For each layer, the average bending of the muon track $\Delta\phi_{mean}$ is taken, and the hit nearest to the pattern track is selected.
  - From the lookup table, the PDF contribution from each layer is found (as shown in Figure 1).
  - The sum of PDF contributions from all layers and the number of contributing layers is calculated[†].
- Finally, the pattern with the highest number of contributing layers and the highest total PDF is selected.

The internal structure of the block calculating the total PDF for the single track pattern is shown in Figure 3.

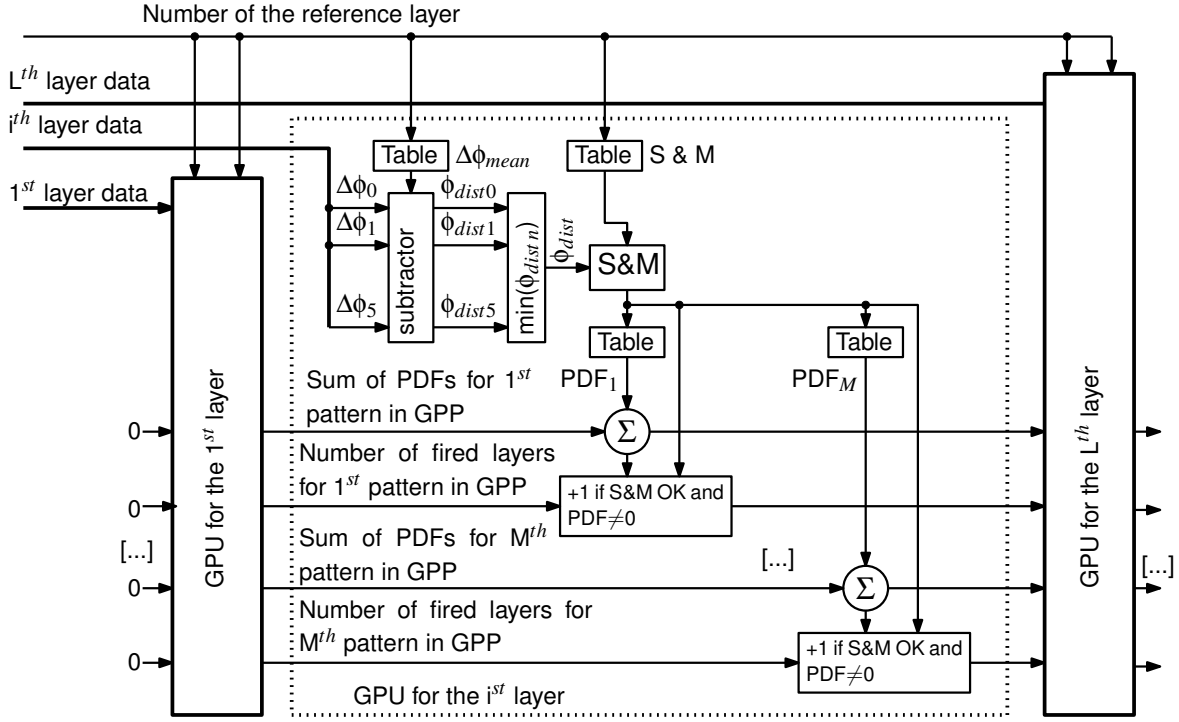

Figure 3: Implementation of the single GPP handling M patterns. The Golden Pattern Unit (GPU) is the block which analyses hits in a single layer. L is the number of layers, M is the number of patterns in the set of patterns handled in the GPP. "S&M" (shift & mask) are the blocks used to scale $\Delta\phi$ depending on the PDF distribution width. The figure is reproduced with minimal modifications from the Open Access publication Ref. 5 according to the CC-BY 4.0 license.[7]

The OMTF algorithm was created and initially tested as a C++ program. Then it was carefully translated into the parameterized high-level HDL description,[3,8] tested and optimized. The whole iterative process requiring multiple iterations is described in Ref. 5. Certain modifications required repeated retuning of delays in parallel branches of the algorithm. That process was especially time-consuming and error-prone, and therefore for future similar systems, an automated tool has been created.[9]

[*]To minimize resource consumption, the patterns are grouped in sets containing 2 or 4 patterns, that are processed simultaneously.

[†]Layers belonging to one type of detectors - the DT, are analyzed as pairs in a special way. The details are described in Ref. 3

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
static ap_uint<1> sdin[128];
static int first = 0;
int prior_enc(ap_uint<1> load,
              ap_uint<1> ref_hit_bits[GP_N_OF_REF_HITS])
{
#pragma HLS PIPELINE II=1
#pragma HLS UNROLL
#pragma HLS ARRAY_PARTITION variable=sdin complete
#pragma HLS ARRAY_PARTITION variable=ref_hit_bits complete
    int i;
    int found = -1;
    if (load) {
        first = 0;
        for (i = 0; i < GP_N_OF_REF_HITS; i++)
            sdin[i] = ref_hit_bits[i];
    }
    for (i = 0; i < GP_N_OF_REF_HITS; i++) {
        if ((i >= first) && (found == -1)) {
            if (sdin[i]) {
                first = i + 1;
                found = i;
            }
        }
    }
    return found;
}
```

Figure 4: The simplest and "naive" HLS implementation of the priority encoder used to select the "reference hit". That code resulted in unacceptable Initialization Interval.

3. PREPARATION OF THE HLS IMPLEMENTATION

The features described in the previous section, make the OMTF algorithm an ideal object for HLS implementation. If such implementation is reliable and successful, the development cycle could be significantly reduced, because any modification of the algorithm done in C could be quickly and automatically translated into the FPGA code.

All experiments have been performed using the Vivado HLS 2018.3 environment. The computer used for tests was equipped with Intel Xeon CPU E5-2623 v4 working at 2.60 GHz clock frequency. The pattern definition used for tests consisted of 20 pattern sets, containing 52 track patterns in total.

The HLS source code was the C/C++ reimplementation of the OMTF algorithm. The design was tested in simulations using the same test data sets that were used for testing of the HDL implementation. The testbench for C simulations was created basing on the original testbench written for the HDL implementation. That testbench also has been modified to allow testing of the HDL code generated by HLS synthesis.

3.1 Initial HLS implementation

To achieve similar functionality, as in the original, the design was supposed to be implemented in pipeline architecture with Initialization Interval equal to 1 and the clock frequency equal to 160 MHz (4 clock periods per BX to test up to 4 reference hits). Correct operation of the code in C simulations was achieved without problems. However, ensuring proper synthesis and implementation was more complicated.

The first version of the code, written without considering specific features of the FPGA hardware, resulted in very high Initialization Interval values - up to 128. The analysis of the problem has shown that the priority encoder used to select the reference hit with the highest priority requires reimplementation. The original "naive" code is shown in Figure 4. To improve the Initialization Interval, the encoder has been decomposed into a two-level hierarchical structure described with the code shown in Figure 5. Similarly, it was necessary to implement a hierarchical sorter finding the best-matched pattern on the output of the algorithm.

Unfortunately, the implementation shown in Figure 5 has exposed a serious problem. The C simulations of the whole algorithm have produced correct results for the test data. Synthesis and the implementation didn't generate any critical warnings or errors. However, simulations of the generated HDL code resulted in corrupted output. Not all reference hits were processed. Resolving the problem was complicated by the fact that the code generated by HLS is huge and almost illegible. Additionally, each synthesis and implementation of the complete algorithm took even up to 12 hours.

Fortunately, the problem was again associated with the priority encoder, that could be easily extracted as an individual block and tested separately. However, to correctly integrate the extracted priority encoder with its testbench, it was necessary to add the interface definitions (*#pragma HLS INTERFACE*). It appeared that forgetting to remove them before the compilation of the whole design resulted in an increase of Initialization Interval from 1 to 4[‡]. The final version of the priority encoder that works correctly with II=1 both in C simulations and as the HLS-produced HDL code is shown in Figure 6.

[‡]That problem may be solved by creating the separate wrapper for the block and specifying interfaces in the wrapper.

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <ap_utils.h>
static ap_uint<8> code = 0;
ap_uint<1> sdin[128];

void prior_sub_enc(ap_uint<1> rhb[8], ap_uint<4> *code)
{
#pragma HLS LATENCY max=0
#pragma HLS INLINE
#pragma HLS UNROLL
    *code = 8;
    for (int i = 0; i < 8; i++) {
        if (rhb[i] == 1) {
            *code = i;
            break;
        }
    }
}

int prior_enc(ap_uint<1> load,
              ap_uint<1> ref_hit_bits[128])
{
#pragma HLS PIPELINE II=1
#pragma HLS UNROLL
#pragma HLS ARRAY_PARTITION variable=ref_hit_bits complete
#pragma HLS ARRAY_PARTITION variable=sdin complete
    ap_uint<4> sub_code[16];
#pragma HLS ARRAY_PARTITION variable=sub_code complete
    if (load == 1) {
        for (int i = 0; i < 128; i++) {
            sdin[i] = ref_hit_bits[i];
        }
    } else {
        if (code < 128)
            sdin[code] = 0;
    }
    code = 128;
    for (int i = 0; i < 16; i++) {
        prior_sub_enc(&sdin[8 * i], &sub_code[i]);
        if (sub_code[i] < 8) {
            code = i * 8 + sub_code[i];
            break;
        }
    }
    return code;
}
```

Figure 5: The initial hierarchical HLS implementation of the priority encoder used to select the “reference hit”. That code has correct II=1 and behaves correctly in C simulation, but produces the HDL code that does not work correctly.

```
#include <stdint.h>
#include <stdio.h>
#include <string.h>
#include <ap_int.h>
#include <ap_utils.h>
static ap_uint<1> sdin[128];
void prior_sub_enc(ap_uint<1> rhb[8], ap_uint<4> *code) {
#pragma HLS LATENCY max=0
#pragma HLS INLINE
#pragma HLS UNROLL
        *code = 8;
        for (int i = 0; i < 8; i++) {
                if (rhb[i] == 1) {
                        *code = i;
                        break;
                }
        }
}

int prior_enc(ap_uint<1> load, ap_uint<1> ref_hit_bits[128]) {
#pragma HLS PIPELINE II=1
#pragma HLS UNROLL
#pragma HLS ARRAY_PARTITION variable=ref_hit_bits complete
#pragma HLS ARRAY_PARTITION variable=sdin complete
        ap_uint<4> sub_code[16];
        int code = 0;
#pragma HLS ARRAY_PARTITION variable=sub_code complete
        code = 128;
        for (int i = 0; i < 16; i++) {
                prior_sub_enc(&sdin[8 * i], &sub_code[i]);
                if (sub_code[i] < 8) {
                        code = i * 8 + sub_code[i];
                        break;
                }
        }
        if (load == 1) {
                for (int i = 0; i < 128; i++) {
                        sdin[i] = ref_hit_bits[i];
                }
        } else {
                if (code < 128)
                        sdin[code] = 0;
        }
        return code;
}
```

Figure 6: The final hierarchical HLS implementation of the priority encoder used to select the “reference hit”. That code has correct II=1, behaves correctly in C simulation, and produces correct HDL code.

Table 1: Comparison of the source code complexity for HDL and HLS approaches. It can be seen that the HLS reduced the number of files and lines of the code roughly by a factor of 3. The analysis excludes the automatically generated files with definitions of patterns and similar constant values and structures.

Form of description]	Files	Code	Comment	Comment %	Blank	Total
HDL (VHDL)	21	1724	787	31.3%	302	2813
HLS	7	564	260	31.6%	78	902

In the whole process of correcting the HLS implementation, it was important that the reference HDL implementation existed. Obtaining Initialization Interval equal to 1 (II=1) was really difficult and required significant effort. Without the knowledge that the optimized HDL implementation is able to work at II=1, it was easy to come to the conclusion that the problem can't be solved with a pipeline with so short II.

4. RESULTS AND DISCUSSION

It was possible to obtain the correct working implementation of the OMTF algorithm using the HLS synthesis. The implementation is based on a pipelined architecture with parallel branches, but the whole task of equalization of the delays in parallel branches was correctly handled by the HLS tool. It is a significant advantage, that seriously reduces the risk of errors and effort to maintain the design.

4.1 Code complexity

Use of HLS also allows significant complexity of the source code. Table 1 compares the amount of code needed to implement the HDL and HLS versions of the algorithm. It is also worth to emphasize that the C/C++ description is better legible and easier to maintain by a software engineer not skilled in FPGA programming. However, in case of problems with the generated HDL code, the FPGA related skills may be important. They may be also useful to prepare the C/C++ code best suited for FPGA implementation.

4.2 Time of compilation

The HLS and HDL approaches differ significantly regarding the time of compilation. For huge and complex designs like OMTF, the HLS synthesis and implementation may be significantly increased. The values obtained on Xeon-equipped computer used for tests were as follows

- **For HDL (VHDL) implementation**
  - Synthesis: 53 minutes
  - Implementation: 20 minutes
  - **Total: 73 minutes**
- **for HLS implementation**
  - Synthesis: 430 minutes
  - Implementation: 248 minutes
  - **Total: 678 minutes**

Due to the long compilation, it may be important to provide the possibility to decompose a large design into smaller blocks and prepare testbenches for all of them. Without a possibility to test both in C and in produced HDL the "priority encoder" it would be impossible to isolate and fix the problems described in section 3.1. It is important, that for smaller blocks the HLS synthesis and implementation times are much shorter. Therefore, it is very important to test parts of the design with smaller blocks and perform the final HLS compilation after resolving all problems exposed during testing of smaller blocks.

Table 2: Resource usage and minimal clock period in FPGA (predicted values after HLS synthesis and final values after implementation).

	LUT	FF	BRAM	Min. clock period
Available	433200 (100%)	866400 (100%)	2940 (100%)	6.125 ns
Synthesis	12174356 (2810%)	1348296 (155%)	720 (24%)	7.480 ns
Implementation	123964 (29%)	112240 (13%)	720 (24%)	5.925 ns

Table 3: Comparison of the results of synthesis based on HLS and HDL description of the OMTF algorithm.

Design method	Slice LUTs	Slice Registers	F7 Muxes	F8 Muxes	Slices	LUT as Logic	LUT as Memory	Block RAM Tile
HLS	123964 (28.6%)	112240 (12.9%)	437 (0.2%)	33 (0.03%)	39312 (36.3%)	106900 (24.7%)	17064 (9.8%)	360 (24.49%)
HDL	114791 (26.5%)	92845 (10.7%)	272 (0.13%)	39 (0.04%)	40607 (37.5%)	108905 (25.1%)	5886 (3.4%)	360 (24.49%)

The obtained results of HLS synthesis suggested that the design can't be implemented both regarding the required clock frequency and FPGA occupancy. Fortunately, the optimizations performed during the implementation significantly reduced the critical path and resource occupancy. The results of HLS synthesis and implementation are presented in Table 2.

4.3 Resource consumption and performance

Both HDL and HLS-based designs are capable of working at clock 160 MHz with Initialization Interval equal to 1. However, they differ in latency:

- The latency of HDL-based design: 38 clock periods
- The latency of HLS-based design: 54 clock periods (significantly higher, but acceptable)

The resource consumption of both designs is shown in Table 3. Both methods give similar FPGA occupancy.

5. CONCLUSIONS

The results obtained during the development and testing of the HLS-based implementation of the OMTF algorithm have exposed certain advantages and disadvantages of such an approach. They are listed below.

Advantages of HLS implementations:

- It is possible to translate the algorithm implemented in C/C++ into FPGA implementation.
- The source code is smaller than in HDL-based design, easier to understand and maintain.
- It is easier to test the algorithm as the C/C++ program than in HDL simulation.
- The HLS ensures correct synchronization of data in parallel paths of the pipelined design.

Possible problems with HLS implementations:

- Not all algorithms correctly working in C/C++ are correctly translated into synthesizable HDL code.

- Preparation of C/C++ code for the HLS implementation may require knowledge about the operation of FPGAs.
- The correct results of the C simulation do not warrant the correct operation of the final HDL implementation.
- For complex designs, the synthesis and implementation times of the design may be significantly increased.
- The generated HDL code is huge and illegible. It makes almost impossible debugging that code in case of erroneous operation.

It may be stated that indeed the HLS allows implementing even complex algorithms in FPGA, based on the C or C++ description of the algorithm. It seems, that the basic implementation may be produced even by a software engineer with minimal FPGA skills. However, when really high performance is needed, the deep knowledge of FPGA capabilities and limitations is very important. The organization of C code and selection of options (HLS pragmas) may significantly affect the results. Automated generation of HDL code by HLS tools may reduce the amount of human work needed to maintain complex designs (e.g., eliminating the need to manually adjust latencies in parallel branches). In theory testing of algorithms in C simulations should increase the speed of development based on iterative testing and corrections. Unfortunately, the very annoying finding that results of C simulation may sometimes disagree with the behavior of the generated HDL code, imposes frequent verification in HDL simulations. In connection with long times of HLS compilation, that may significantly slow down the development process. Hopefully, the HLS is a quickly developing technology, and probably the problem may be eliminated in future versions.

HLS seems to be a promising technology, however, its usage for complex designs must be carefully evaluated regarding benefits and losses.

ACKNOWLEDGMENTS

The author thanks all the members of the Warsaw CMS group who participated in the development of the original OMTF algorithm.

Work partially supported by statutory funds of Institute of Electronic Systems. OMTF project is supported by National Science Centre, Poland, grant UMO-2015/19/B/ST2/0286

REFERENCES

[1] Xilinx, "Vivado design suite user guide, high-level synthesis." https://www.xilinx.com/products/design-tools/vivado/integration/esl-design.html [Online; accessed 29-April-2018].

[2] Zabołotny, W. M., "Implementation of heapsort in programmable logic with high-level synthesis," *Proc.SPIE* **10808**, 108084C (Oct. 2018).

[3] Zabolotny, W. and Byszuk, A., "Algorithm and implementation of muon trigger and data transmission system for barrel-endcap overlap region of the CMS detector," *Journal of Instrumentation* **11**, C03004–C03004 (Mar. 2016).

[4] Konecki, M. A., "The CMS Level-1 muon triggers for the LHC Run II," Tech. Rep. CMS-CR-2018-227, CERN, Geneva (Sep 2018).

[5] Bluj, M., Buńkowski, K., Byszuk, A., Doroba, K., Drabik, P., Górski, M., Kalinowski, A., Kierzkowski, K., Konecki, M., Miętki, P., Okliński, W., Olszewski, M., Poźniak, K., Zabołotny, W., Zawistowski, K., and Żarnecki, G., "From the Physical Model to the Electronic System — OMTF Trigger for CMS," *Acta Physica Polonica B Proceedings Supplement* **9**(2), 181 (2016).

[6] Bunkowski, K., "The algorithm of the CMS Level-1 Overlap Muon Track Finder trigger," *Nuclear Instruments and Methods in Physics Research Section A: Accelerators, Spectrometers, Detectors and Associated Equipment* (Oct. 2018).

[7] "Attribution-sharealike 4.0 international." https://creativecommons.org/licenses/by-sa/4.0/ [Online; accessed 29-April-2019].

[8] Zabolotny, W. M. et al., "FPGA implementation of overlap MTF trigger: preliminary study," *Proc. SPIE* **9290**, 929025–929025–11 (2014).

[9] Zabołotny, W. M., "Automatic latency equalization in VHDL-implemented complex pipelined systems," *Proc.SPIE* **10031**, 10031 – 10031 – 12 (2016).