Automatic latency equalization in VHDL-implemented complex pipelined systems

Wojciech M. Zabołotny[a,b]

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

ABSTRACT

In the pipelined data processing systems it is very important to ensure that parallel paths delay data by the same number of clock cycles. If that condition is not met, the processing blocks receive data not properly aligned in time and produce incorrect results. Manual equalization of latencies is a tedious and error-prone work. This paper presents an automatic method of latency equalization in systems described in VHDL. The proposed method uses simulation to measure latencies and verify introduced correction. The solution is portable between different simulation and synthesis tools. The method does not increase the complexity of the synthesized design comparing to the solution based on manual latency adjustment. The example implementation of the proposed methodology together with a simple design demonstrating its use is available as an open source project under BSD license [*].

Keywords: FPGA, pipeline, latency balancing, delay equalization, VHDL

1. INTRODUCTION

The pipeline architecture is known for a very long time and used to increase the throughput of digital blocks.[2] This concept has also been early adopted to signal or data processing systems implemented in FPGA.[3] The pipeline architecture allows to increase the clock frequency, because the complex operations, that would result in long critical paths in FPGA are divided into multiple significantly simpler operations. Those operations may be performed in a single clock cycle even at the much higher clock frequency. The time needed to process the set of data will be the same or even slightly longer, due to the introduction of additional registers. However, the overall throughput of such system will increase because, in each clock cycle, the system can accept a new set of data, and results of processing of certain previous data set are produced on the output. Of course, such a system will introduce a latency of a certain number of clock cycles, between the moment of delivery of the data set to the input and the moment when results of its processing are available on the output.

Implementation of algorithms in pipelined architecture is more complicated when the processing consists of different operations performed in parallel, and each of them requires a different number of elementary single-cycle operations. The latency of each operation is different, and if we want to produce correct results, it is necessary to add certain delay blocks (typically consisting of shift registers) in shorter-latency paths (see Figure 1). In real applications, the system may contain multiple paths with different latencies, which must be equalized to ensure the proper operation.

1.1 Overlap Muon Track Finder as an example of the complex pipelined system

The Overlap Muon Track Finder is a new trigger system developed for the overlap region of the CMS detector at CERN.[4] The OMTF measures the transversal momentum (p_T) of muons generating electrical signals (so called "hits") in three different detectors (CSC, RPC, and DT), organized in 18 "layers" in the CMS detector. The OMTF algorithm has been described in details in previous publications [5–7]. Therefore, below only the simplified description is provided, to explain the idea of its pipelined implementation and latency-equalization related problems. The OMTF processes hits generated during the bunch crossings (BX) in the CMS detector, which occur at a frequency of ca. 40 MHz. The OMTF implements the pipeline, that performs the following operations.

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

[*]This is the extented and modified version of the arXiv preprint [1].

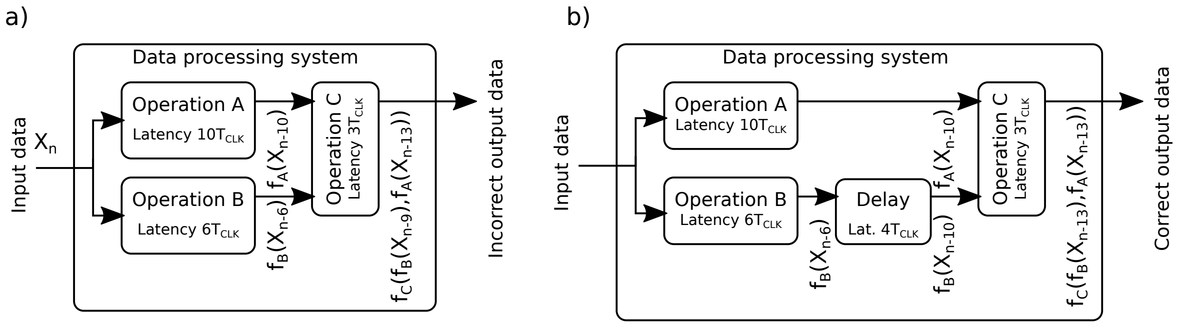

Figure 1. An example of the data processing system where results of two operations calculated with different latencies are used as arguments for a third operation. a) Without additional delays, the output data are incorrect, as arguments for operation C were calculated from different data sets. b) To assure correct output data, it was necessary to add the 4 T_{CLK} delay after the operation B block. Now both arguments of operation C are derived from the same data set.

- Finds a hit in one of so called "reference layers", which will be the basis of the muon track reconstruction (the "reference hit"). It is desirable, that multiple "reference hits" may be analyzed in each BX, to allow correct analysis of tracks generated by multiple muons. This requires that the OMTF should operate at clock frequency of $N \cdot 40$ MHz, where N is the number of processed "reference hits".
- Basing on the selected "reference hit" OMTF selects the detector channels, which may be used for muon track reconstruction.
- The p_T estimation is based on comparison of the received hits with generated averaged tracks of the muons with certain p_T (patterns). There are 52 such patterns generated. Therefore, the following operations are performed in parallel for each pattern:
  - In each detector layer find the "best matched" hit, which is closest to the center of the pattern
  - In the lookup table find the fit measure (so-called probability density function - PDF) for that hit.
  - If the PDF is non-zero, add it to the total fit value derived from previous layers, and increase the number of matching hits.
- The pattern with the highest number of matching hits, and the highest total fit value is selected as the best one, and its p_T value is assumed to be the p_T of the measured muon.

The above implementation is also shown in Figure 2.

Implementation of the OMTF was subjected to different contradictory requirements. On the one hand, it should operate at the highest possible frequency, to maximize the number of reference hits processed in a single BX. On the other hand, the OMTF blocks should consume as few resources as possible to increase the number of patterns. Basing on these constraints some OMTF blocks may be optimized. The examples may be blocks finding the maximum[8] or the minimum value from the values delivered to its inputs. Such blocks are used to find the best matching hit in each layer and the best matching pattern. The typical architecture of such blocks, together with explanation how their optimization affects their latency is shown in Figure 3. Another reason for changing the path latency in the OMTF design was the addition of pipeline registers required to shorten critical paths and achieve timing closure .[9–11]

The above optimizations had to be performed many times during the development, and the manual calculation and adjustment of latency in different parallel paths appeared to be a difficult task. The mistakes in latency adjustment resulted in serious, and difficult to diagnose errors in OMTF operation. That has shown, that for such complex pipelined systems it is necessary to develop an automated method for latency balancing.

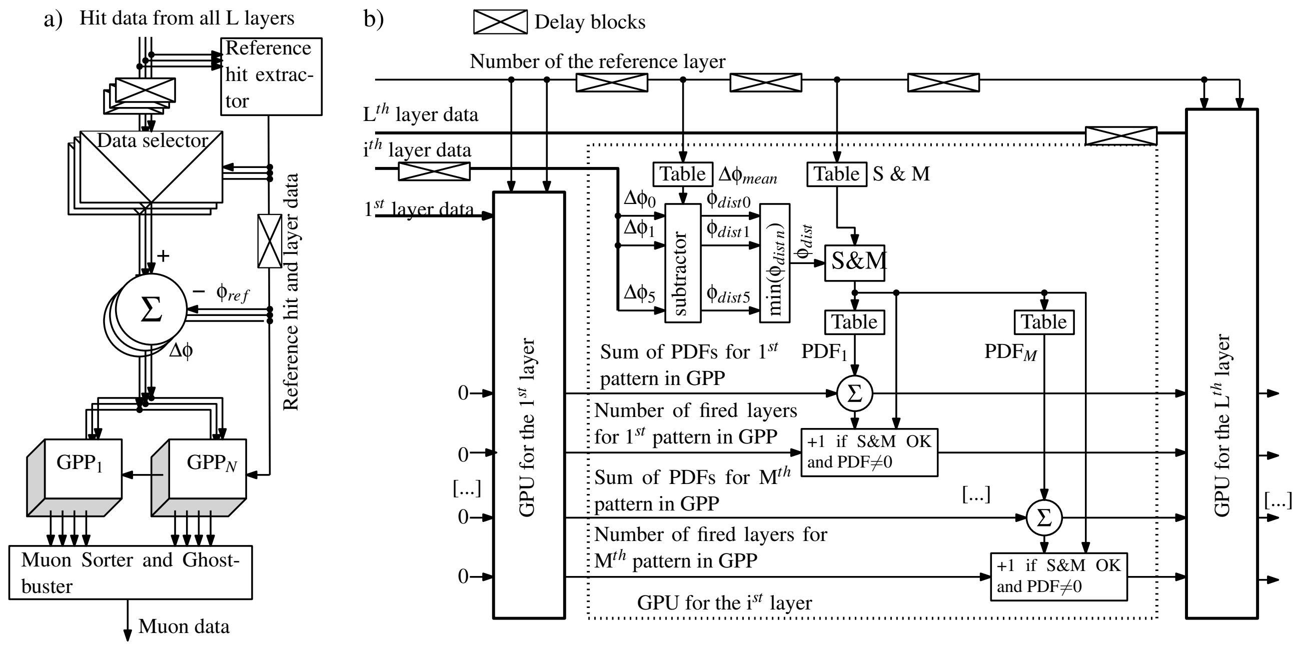

Figure 2. The block diagram of the OMTF. Delay blocks necessary to ensure path latency equalization are shown as crossed rectangles. a) The whole OMTF algorithm, b) Internal structure of the single GPP block. Figure based on [7].

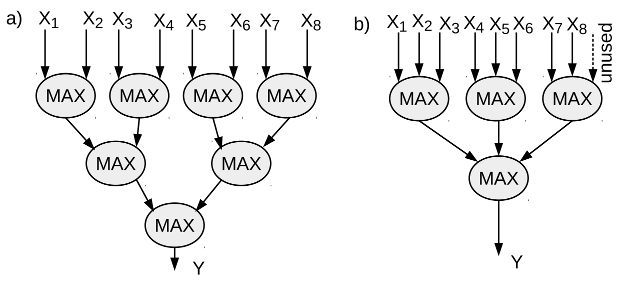

Figure 3. The figure showing two implementations of the block finding the maximum of 8 input values. The "MAX" block compares its input values and produces on the output the maximum value together with its original input number. The "MAX" block introduces the latency of one clock pulse. a) Each "MAX" block has two inputs, so finding the maximum value requires 3 stages, and introduces the latency of 3 clock pulses. b) Each "MAX" block has three inputs, so finding the maximum requires 2 stages, and introduces the latency of 2 clock pulses. However, the blocks with 3 inputs are more complex so that the critical path may be longer and the maximum clock frequency may be lower than in case "a".

2. AVAILABLE SOLUTIONS

Of course, the problem of latency equalization between paths in pipelined designs is not new. The graphical tools, allowing to build data or signal processing systems from predefined blocks implementing basic operations addressed that problem more than 15 years ago. Old versions of Xilinx System Generator for Simulink provided the "sync" block, which operation is described as follows: *"The Xilinx Sync Block synchronizes two to four channels of data so that their first valid data samples appear aligned in time with the outputs. The input of each channel is passed through a delay line and then presented at the output port for that channel. The lengths of the delay lines embedded in this block, however, are adaptively chosen at the start of simulation so that the first valid input samples are aligned. Thus, no data appears on any channel until a first valid sample has been received into each channel."* ([12], page 47). This sync block was later synthesized using the hardware shift registers ([13], slide 22).

The modern block-based tools also provide similar functionality. For example, the Altera DSP Builder can automatically add delays in paths with lower latency "to ensure that all the input data reaches each functional unit in the same cycle".[14,15] No detailed information about this methodology, revealing the implementation details is disclosed, though.

The article [16] describes the system level retiming, automatic pipelining and delay balancing (including the multi-rate pipelining) implemented in the MathWorks HDL Coder .[17] The delay balancing algorithm used by the authors depends on the transformation of the design into the Parallel Implementation Representation (PIR), and further analysis of the PIR graph. There are no known tools able to convert the generic HDL code into the PIR form, and, therefore, this solution is not suitable for designs implemented in pure VHDL.

Finally, the existing solutions have significant disadvantages:

- They are available only for systems built in graphical environments from predefined blocks (however the user may also add his or her own block with needed functionality).
- They are closed solutions offered for the particular proprietary environment. Therefore, they are not portable.
- Due to their closed source nature, it is not clear how the latency balancing is implemented and if it can be reused in designed entirely based on HDL description. The only exceptions are:
  - The old "Xilinx Sync Block" which uses the approach based on simulation, where the main concept is described in the accompanying documentation. The interesting thing, however, is that this block has been removed from newer versions of Xilinx System Generator (see [18], page 6);
  - The algorithm implemented in the MathWorks HDL Coder that unfortunately utilizes a special intermediate representation of the design. This representation may be created from the Simulink model, but not from the arbitrary VHDL source.

As we can see from the above review. If we want an open and portable solution applicable on the level of VHDL source, a new approach is necessary.

3. LATENCY ANALYSIS AND EQUALIZATION IN VHDL BASED DESIGNS

As the VHDL source fully describes the behavior of the system, it should be possible to find the data path latencies via analysis of the source code. Unfortunately, calculation of latency introduced in the VHDL code may be extremely difficult. The correct handling of different factors affecting the final latency (e.g. the "if generate" and "for generate" constructs), would require duplicating significant part of a functionality of the VHDL compiler. Maybe in the case of an open source compiler like GHDL,[19] the necessary modifications may be added, but such a solution would not be portable. Therefore another approach is needed.

3.1 Simulation-based analysis and equalization of latencies - simplified approach

The simulation is anyway an important part of the development of IP cores for FPGA. The design should be accompanied with testbenches allowing to verify its correct operation in a simulation. Therefore, a method allowing to check and equalize data path latencies in a simulation using a dedicated testbench may be a viable solution.

Such approach was employed by the "Xilinx Sync Block", mentioned in section 2. However, it seems that the method based only on the time of arrival of the first valid data may be not fully reliable. It is desirable that the latency of different paths is verified during the whole simulation.

The general idea of the proposed method is to supplement (in simulation only) each data set with the time marker (TM) describing the moment (the clock period number), in which these data were delivered to the analyzed system. Therefore in simulation the system must be equipped with an additional block, generating the current TM value. In the simplest implementation, the TM may be just an integer signal, starting from a certain value (e.g. -1) and increased every clock pulse (more detailed description of time markers implementation is available in Section 4.1).

When the system is working, the time markers propagate through the system together with the processed data and results of their processing.

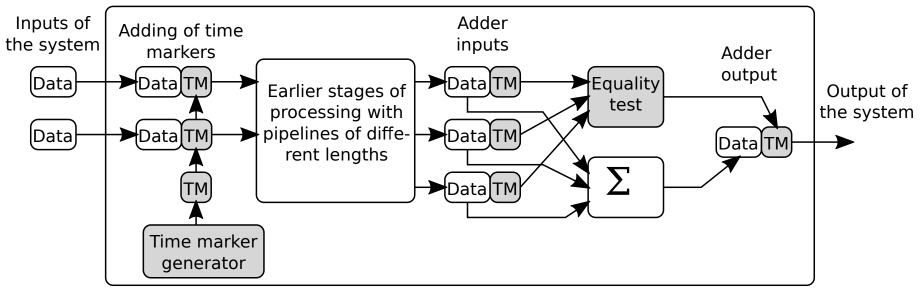

Figure 4. An adder as an example of the processing block implementing the described method.
The data entering the system in the simulation are labeled with the time markers. The preprocessing of data involves pipelines with different latency. Finally, the sum of the three values resulting from the preprocessing is calculated. Equality of time markers on the adder inputs is verified, and the same time marker is produced on the output.
Grayed objects are used only for simulation. They are excluded from synthesis using the "`--pragma translate_off`" and "`--pragma translate_on`" metacomments.

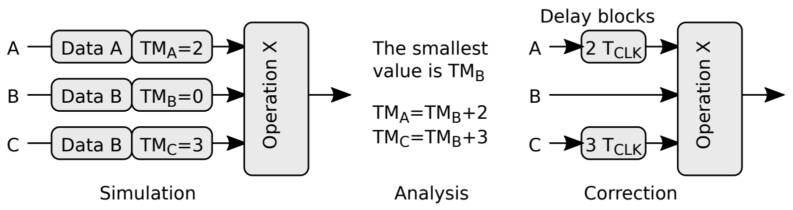

Figure 5. The idea of simulation based latency correction. During the simulation, in a certain place of the design three data sets: A, B and C are used to perform operation X. The time markers (TM) associated with the data sets are compared and found to be different. Therefore in the data paths with the highest values of TM the delay blocks are added. The latency of added delay is equal to the difference between the minimal TM and the TM in the particular path.

Of course, the time markers, and all logic associated with their processing must be included only in simulation, so that they do not increase the complexity of the synthesized design. Fortunately, most synthesis tools offer the "`--pragma translate_off`" and "`--pragma translate_on`" metacomments allowing to exclude certain fragments of the VHDL code from synthesis. Using them, we can ensure that only the delay blocks, needed to balance latencies in parallel paths will be added to the synthesized design.

An example of an adder block implementing the described method is shown in Figure 4.

Whenever an operation on two or more subsets of data is performed, the time markers should be checked, and in case if they are different, it is a symptom of unequal data path latencies. In such case, the simulation should be aborted, and latency-equalization error should be reported. The difference between the time markers should be written to the file, and used to correct the design. The shorter data path (or data paths if more than two data subsets were used as operands) should be supplemented with additional delay block with latency equal to the detected difference (see Figure 5). The above procedure should be repeated until the design is found to work properly. Unfortunately, the number of repetitions may be high because each simulation-analysis-correction cycle allows correcting latencies on the input of one block only.

The preferable method should provide equalization of all latencies in a single simulation-analysis-correction cycle.

3.2 Improved simulation-based latency analysis and correction

In the first approach, the simulation was stopped, when the first inconsistency of time markers was detected. However in most pipelined systems the misaligned data produce incorrect results, but otherwise do not affect the operation of the system. Despite incorrect results, the system may be further simulated, and time markers differences between input data in other blocks may be analyzed.

There is, however, one problem. If the input time markers are equal, the time marker of the result is simply copied from them. If the input time markers are not equal, it is not clear what should be the time marker of the result. The solution is to imitate the appropriate latency equalization so that latencies in the rest of the system may be still analyzed. The latency

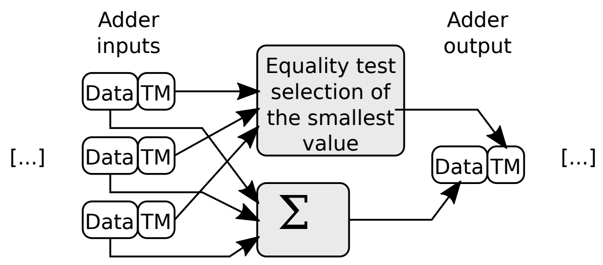

Figure 6. An adder as an example of the processing block implementing the improved method. A sum of the three input values is calculated. Equality of time markers on the input is verified. If they are equal, the same time marker is produced on the output. In the case of inequality, we pretend that latencies are properly equalized. As we can only increase the delay by adding delay blocks, the minimal time marker is produced on the output. In that way, from the point of view of time markers, we can imitate the operation of the system after proper latency balancing in this block (of course, the data are still misaligned, and results are incorrect).

equalization is achieved by introducing the delay blocks, which results in the decrease of the time marker. To imitate the proper equalization, the output time marker should be set to the lowest one from the input time markers. Of course, such situation must be reported, as the processing results will be incorrect due to incorrect time alignment of processed data. Additionally, also the values of input time markers must be reported, so that in the correction phase they can be used to find the required additional latency.

Using the described method we can test latencies of all paths in the system and calculate delays of all necessary delay blocks in a single simulation-analysis-correction cycle.

Certainly, the testbench should also allow testing the properly latency-balanced design at the end. Therefore it must offer two modes of operation:

- **The analysis mode**, in which the time marker inequalities do not cause the simulation failure and in each block the output time marker is set to the smallest one from the input time markers
- **The final test mode**, in which any difference between time markers causes the simulation error

4. IMPLEMENTATION OF THE PROPOSED METHOD IN VHDL

To allow inclusion of the time marker in the processed data, those data should be encapsulated in a record type, with optional (used only in simulation) time marker field. An example of code implementing such a record type and the adder using this type is shown in Figure 7. If the user had to modify all his or her processing blocks to include the TM handling (as in Figures 4 and 6), the proposed method would be very inconvenient. To simplify its adoption, the checking and equalization of latencies is implemented outside the processing blocks in the dedicated "Latency Checking and Equalizing" blocks (LCEQ). The LCEQ block should offer configurable number of signal paths and should behave in the following way:

- In the analysis mode:
  - Checks the time markers on its input, reporting all detected inequalities. Additionally, the time markers values should be recorded for further analysis.
  - Verifies the time markers on its output (after delay blocks) and in the case of their inequality, copies the smallest time marker to all outputs (to allow single-cycle analysis, as described previously).
- In the final test mode:
  - Checks the time markers on its outputs and abort the simulation in the case of any inequality.

Additionally, it should be possible to configure the latency value introduced by the LCEQ in each path. The block diagram of the proposed LCEQ block is shown in Figure 8.

Presented implementation of the LCEQ block may be used only when all paths carry data of the same type. It can be acceptable in some applications, but to assure maximal flexibility it should be possible to define the type of data in

```
[...]
-- pragma translate_off
subtype T_TIME_MARKER is integer;
-- pragma translate_on

-- User data without time marker
subtype T_USER_DATA is unsigned(17 downto 0);

type T_USER_DATA_MRK is record
    data : T_USER_DATA;
    -- pragma translate_off
    tm  : T_TIME_MARKER;
  -- pragma translate_on
end record T_POS_INT_MRK;
[...]
entity adder_tm is
  port (
    in1  : in  T_USER_DATA_MRK;
    in2  : in  T_USER_DATA_MRK;
    in3  : in  T_USER_DATA_MRK;
    out1 : out T_USER_DATA_MRK)
end entity adder_tm;

architecture example of adder_tm is

begin
out1 <= in1 + in2 + in3 ;
-- pragma translate_off
-- We check delays only on active edge
-- of the clock! (otherwise there will be
-- a lot of false alarms due do delta cycles)
tm1 : process(clk) is
  variable out_tm : T_TIME_MARKER;
begin
  if not ( in1.tm = in2.tm and
           in2.tm = in3.tm) then
     -- We assume, that we have a special function
     -- for reporting unequal time markers
     report_inequality(in1.tm, in2.tm, in3.tm);
  end if
  out_tm := in1.tm;
  if in2.tm < out_tm then
    out_tm := in2.tm;
  end if;
  if in3.tm < out_tm then
    out_tm := in3.tm;
  end if;
  out1.tm <= out_tm;
end process tm1;
-- pragma translate_on

end architecture example;
```

Figure 7. An example the adder using the type encapsulating the user data and the time marker.

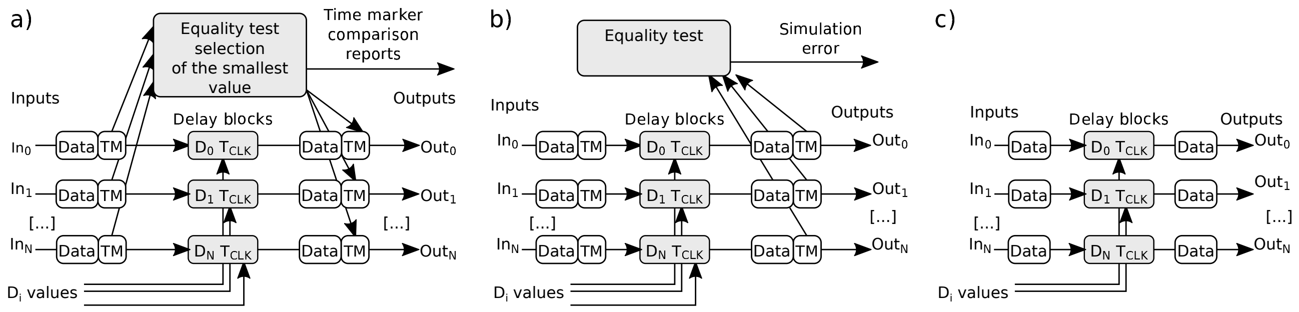

Figure 8. The block diagram of the proposed latency checking and equalizing block. a) The block works in the analysis mode. b) The block works in the final test mode. c) The synthesized block - time markers and related functionalities are removed.

each path independently. Unfortunately, the VHDL language supported by most simulation and synthesis tools does not allow to implement a port that is an array of records of different types. The VHDL-2008[20] introduces generic types, but even with that it still does not provide the necessary functionality. In fact, the VHDL-2008 is still not fully supported by most simulation and synthesis tools. Therefore for such more general case with different types of data, another solution is necessary. The proposed method instead of providing the fully versatile block in VHDL offers a tool that generates the dedicated LCEQ block for given number of paths and given types of data. More details are provided in section 4.4.

To create a working implementation of the LCEQ block, it is necessary to solve a few practical problems, described in the next subsections.

4.1 Generation of time markers

As it was already mentioned in section 3.1, in the simplest implementation, the time markers may be just the integer numbers. For example, the -1 value may be set as the initial value for all time markers. During the simulation, the time markers of input data should be increased by 1 after each clock pulse. That allows special handling of uninitialized data, which otherwise could impair measurement of latencies. To allow long simulations (above 2^{31} clock pulses) it is necessary to slightly modify the procedure. After reaching $2^{31}-1$, the time marker should be set to zero[†].

[†]That still allows detecting uninitialized data with TM=-1. Of course, the differences between time markers must be calculated in signed 31-bit arithmetics to handle the “wrapped” values properly.

4.2 Reporting of time markers and calculation of additional latencies in the LCEQ blocks

The essential part of the proposed methodology is reporting of time markers from different inputs in LCEQ blocks and delivering them to the program that calculates additional latencies in different paths in the LCEQ blocks. In the tested implementation, the time markers are simply written to the file. In each clock pulse the value from each input of each LCEQ block is written to the file in a line containing: the unique identifier (LEQ_ID) of the particular LCEQ block, the number of the input, and the value of the time marker.

After the markers from each input in that clock cycle are reported, yet another line containing only the LEQ_ID and the word "end" is written to the file. That allows the analysis tool to check if the latency difference remains constant during the whole simulation[‡].

The additional latencies in all LCEQ blocks must be known at the elaboration time. To ensure that, after the simulation in analysis mode, the analysis tool is executed. It generates the VHDL package with the function, which accepts two parameters: the unique ID of the LCEQ block (LEQ_ID) and the number of the path in this block. This function returns required latency as an integer value[§] . The analysis tool (latreadgen.py) is written in Python. Its calling syntax is as follows:

```
latreadgen.py /file/with_time_markers package_file package_name function_name
```

The additional delay is calculated basing on the difference between TM values, as shown in Figure 5.

With this approach, the sources of the system prepared for synthesis contain only the standard VHDL files.

4.3 Unique identifiers of LCEQ blocks

The latency analysis and correction relies on unique identifiers assigned to all LCEQ blocks. Each identifier must be the same during the simulation and during the synthesis.

Theoretically, VHDL offers the INSTANCE_NAME attribute, which should univocally identify each instance of each component used in the design. Unfortunately, it appears, that various simulation or synthesis tools use slightly different formats of the generated identifier. Additionally, during the simulation, the analyzed system is instantiated in the testbench, while, during the synthesis, it may be either a top entity or may be a component of a bigger system. That also leads to different INSTANCE_NAME values during the simulation and during the synthesis.

The proposed method works around these problems using the generic LEQ_ID of string type in each LCEQ block. This generic should be set to the unique LCEQ identifier during each instantiation of the block. If the block is instantiated inside of another block, then this “container” block should also be equipped with its unique ID. In such case, the internal LCEQ block should be instantiated with its LEQ_ID set to the value:

```
"ID_of_container_block:ID_of_LCEQ_block"
```

If the block is instantiated in the for-generate loop, the loop variable should be converted to the string (using the `integer'image` function), and concatenated to the ID of the instantiated block. That approach allows to create unique identifiers, identical for simulation and synthesis and portable between different tools.

4.4 Generation of the dedicated LCEQ blocks for different types of data

As it was mentioned in section 4, if the paths analyzed and equalized by the LCEQ blocks do not use the same type of data, it is necessary to generate the dedicated VHDL source code of the specialized LCEQ block for each combination of data types. The demonstration project provides such a tool, named “lateqgen.py”, which should be called with the following arguments: the entity name of the generated block, the path to the file in which the sources of the block are to be generated, the list of the types of data used in consecutive data paths of the created block. The length of the list of types defines the number of data paths in the block.

[‡] In the future implementations, it may be possible to connect the analysis tool directly to the VHDL simulator via named sockets or VHPI interface. That will eliminate writing a huge amount of data to the disk. Additionally, the parallel operation of the simulator and analysis tool may reduce the execution time of the simulation-analysis-correction cycle on a multiprocessor machine.

[§]If no time markers report is available (e.g. before the first simulation), the generated function always returns 0.

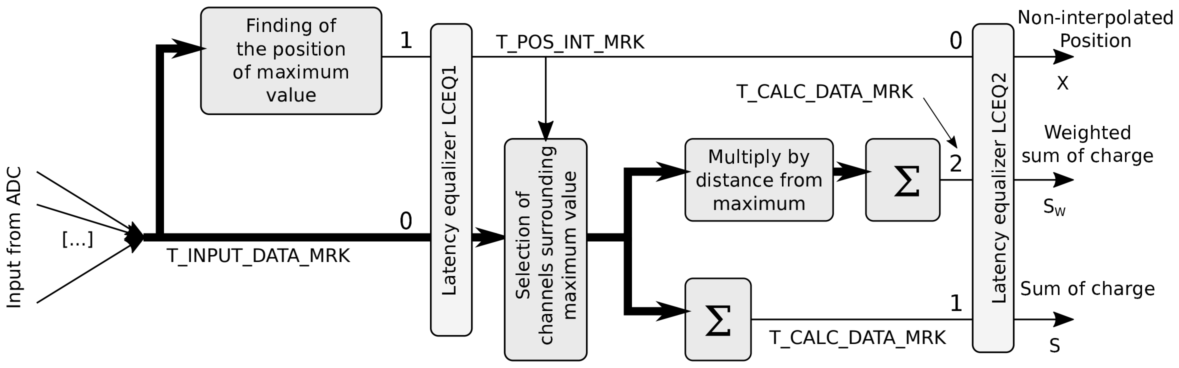

Figure 9. Block diagram of the example system using various types in different data paths. The most important type names are written in the figure. The path numbers are also shown at the inputs of the LCEQ blocks (they are referred to in the section describing the results).

For example to generate the lceq1.vhd file with sources of the entity lceq1 implementing the LCEQ block with four paths where the first two of them handle data of type `T_VOLTAGE`, the third one uses data of type `T_WIDTH` and the fourth - `T_POSITION`, the user should call that tool as:

```
lateqgen.py lceq1 lceq1.vhd T_VOLTAGE T_VOLTAGE T_WIDTH T_POSITION
```

Due to the way how the code is generated, there are some limitations on the names of the data types handled by the generated LCEQ blocks. The name of each type should start with `T_`. Additionally, for each such type the user should define the constant providing the initial value of signals of that type. The name of that constant must be derived from the name of the type by replacing the initial `T_` with `C_` and by adding `_INIT` at the end.

Similarly to the delay definition function, described in section 4.2, the generated LCEQ source is the pure synthesizable and portable VHDL.

5. TESTS AND RESULTS

Even though the OMTF system was the inspiration to create the decscribed methodology, it could not be used as an example in the demonstration project. The OMTF is not Open Source and is too complex to be used as a test case included in the reference implementation. To verify the proposed method, a simple "proof of the concept" implementation has been created and published under open source BSD license on the OpenCores website.[21]

5.1 Test data processing system

The system receives data from ADC converters connected to M readout channels of a particle detector. The voltage level in each channel is proportional to the amount of charge received by that channel in the previous clock period. The particle passing through the detector generates a certain charge that is distributed between neighboring channels. The amount of this charge is proportional to the particle's energy, and the center of gravity of the collected charge defines the position of the hit. The system is aimed at finding the energy and the position of the particle with the highest energy.

In each clock cycle, the system finds the number of the channel with the highest level of the signal N_{max}. This value is treated as a non-interpolated position of the hit $X = N_{max}$. Then the system selects signals from this channel and K neighbouring channels at each side: V_i for $N_{max} - K < i < N_max + K$. Next, the system calculates the sum of charges (basing on the proportionality between the charge and the voltage): $S = \sum_{i=N_{max}-K}^{N_{max}+K} V_i$, and the weighted sum of charges: $S_W = \sum_{i=N_{max}-K}^{N_{max}+K} i \cdot V_i$. Calculated values are transmitted to the external system (in the simulation to the testbench), which calculates the center of gravity of the charge and finally, the interpolated position of the particle hit: $X = N_{max} + \frac{S_W}{S}$.

The block diagram of the example system is shown in Figure 9 [¶].

[¶]Please note, that this block is not of production quality. E.g., it may incorrectly handle the situation, where the maximum signal is too near to the edge of the detector (i.e. $N_{max} < K$ or $N_{max} > M - 1 - K$).

Table 1. The results of latency adjustments for different values of parameters of the test system. The upper part of the table shows the parameters values for different test cases, the lower part - the values of additional latencies calculated by the method. The path numbers are defined in sources and shown in Figure 9. In all cases, the correct operation of the system after latency balancing was confirmed. Only results for implementation with various data types is shown, as the number of paths in the LCEQ1 block for single type implementation is very high.

Parameter name		Test case						
		1	2	3	4	5	6	7
C_N_CHANNELS		64	64	32	32	64	64	64
C_N_SIDE_CHANS		3	3	3	3	5	5	5
EX1_NOF_INS_IN_CMP		3	3	2	2	2	3	3
EX1_NOF_INS_IN_ADD		3	2	3	2	2	2	3
LCEQ block	Path	Calculated additional latency						
LCEQ1	0	4	4	5	5	6	4	4
	1	0	0	0	0	0	0	0
LCEQ2	0	4	5	4	5	6	6	5
	1	1	1	1	1	1	1	1
	2	0	0	0	0	0	0	0

The latency of different paths in the example system may be modified by adjustment of certain parameters in the `ex1_pkg.vhd` and `ex1_trees_pkg.vhd` files.

Finding the maximum value is performed in a multi-level tree based comparator consisting of a certain number of basic comparators. The number of inputs of each basic comparator should be chosen depending on the hardware features of the particular FPGA device and required speed of operation. Each level of the comparator is also equipped with a pipeline register. Therefore, the total latency of the whole "Maximum Value Finder" block depends on the number of inputs in the entire system (parameter `C_N_CHANNELS` in `ex1_pkg.vhd`) and also on the number of inputs in a single basic comparator (parameter `EX1_NOF_INS_IN_CMP` in `ex1_trees_pkg.vhd`).

Similarly, the adders calculating the sum of charge and the weighted sum of charge have a multilevel tree-based structure, and again their latency depends on the number of channels selected for those calculations (parameter `C_N_SIDE_CHANS` in `ex1_pkg.vhd`) and the number of inputs in a basic adder (parameter `EX1_NOF_INS_IN_ADD` in `ex1_trees.pkg.vhd`).

There are two implementations of the demonstration system. The first one, located in the `hdl_single_type` directory uses one type `T_USER_DATA_MRK` in all paths in the system. That allows to avoid using generated LCEQ blocks but requires additional effort to find a common representation for different data (the input signal, the sum of charges, the position of maximum, etc.). The second implementation, located in the `hdl_various_types` directory, shows how to use the proposed methodology with different types, individually suited for different kinds of information processed in the system. Therefore the LCEQ blocks are generated as follows:

```
lateqgen.py ex1_eq_mf ex1_eq_mf.vhd T_INPUT_DATA_MRK T_POS_INT_MRK
```

```
lateqgen.py ex1_eq_calc ex1_eq_calc.vhd T_POS_INT_MRK T_CALC_DATA_MRK T_CALC_DATA_MRK
```

Provided sample implementation is licensed under the BSD license, so it may be used not only to verify and investigate proposed methodology but also as a starting point for its adoption in user's own projects.

5.2 Tests of the proposed method

In the described parameterized implementation of the test system, each change of its parameters may result in a change of latency of corresponding paths. Without the described method, these latencies should be afterward manually balanced by the user. Thanks to the proposed method, the user may perform automatic equalization of latencies. During the tests, the parameters described in the previous subsection were changed, and the additional latencies calculated by the proposed method were checked. Correct operation of the system was also verified, using the simulated hit data in the testbench. Obtained results are presented in Table 1. In all cases, the correct operation of the system after latency balancing was confirmed.

To allow the user to verify the presented results, and to allow to perform experiments with the modified or own design, the dedicated makefile is prepared. To run the provided demonstration, the user must have installed on his or her computer Python version 3,[22] GHDL simulator[19] and GTKWave viewer.[23] The test makefile defines a few targets:

- **make clean** - removes the compiled files and simulation results.
- **make initial** - generates the initial version of latency configuration function, which sets latency to 0 in all paths of all LCEQ blocks.
- **make final** - performs simulation in the "final test" mode. If latencies are not properly balanced, one should expect error messages about unequal latencies
  (e.g.: `EQ1 inequal latencies: out0=0, out1=-1`).
- **make synchro** - performs the simulation-analysis-correction cycle. After this command, the latencies should be properly equalized, and further running of "make final" should not report any errors. In fact, the testbench should also report two correctly analyzed particle hits like "Hit with charge: 2.5e2 at 1.476e1, Hit with charge: 2.65e2 at 2.549e1".
- **make reader** - allows to start the GTKWave viewer and see values of the signals in the demonstration system during the last simulation. This target may be used to analyze the internals of the system.

5.3 Tests of synthesizability of the generated sources

The sources generated by the test makefile with "synchro" target have been successfully synthesized with the Xilinx Vivado[24] tools. The blocks related to time markers generation and checking have been correctly removed from the synthesized design, and only the additional delay blocks have been inserted. Due to high number of pins, the xc7vx690tffg1930 Virtex 7 chip was selected for implementation.

6. CONCLUSIONS

The method presented in this paper is a solution of an important problem of equalization of latencies between parallel paths in complex pipelined data processing systems implemented in FPGA. The method extends the concept of simulation based pipeline delay balancing method offered by the "sync" block in the early versions of Xilinx System Generator for Simulink environment. The solution described in the paper is suitable for systems implemented entirely in the VHDL, and should be compatible with all recent simulation and synthesis tools. The simulation of the designed subsystem allows to calculate latencies of necessary additional delay blocks in a single simulation-analysis-correction cycle. The method allows equalizing latency between paths with data of different types which is crucial in complex systems. To achieve that, dedicated tools have been written in Python 3 to overcome limitations of the VHDL language and to generate source code of necessary blocks. The results of latency equalization are implemented in a standard VHDL package with function defining latencies of all added delay blocks.

The sources of the first "proof of the concept" implementation of the proposed methodology are published on the Open Cores website,[21] under the BSD license. The correctness of the method has been verified with the complete example data processing system included in the sources. Further improvements of the proposed method should be focused on optimization of communication between the simulator and the latency analysis tool. Probably using the named sockets or VHPI interface may significantly improve the simulation and analysis speed. Anyway even in the current state, the proposed method may be a useful tool for designing and maintenance of complex pipelined IP cores implemented in VHDL.

ACKNOWLEDGMENTS

The work is partially supported by Polish NCN grant 2013/11/B/ST2/04202

REFERENCES

[1] Zabolotny, W. M., "Automatic latency balancing in VHDL-implemented complex pipelined systems," (2015). http://arxiv.org/abs/1509.08111.

[2] Hallin, T. G. and Flynn, M. J., "Pipelining of arithmetic functions.," *IEEE Transactions on Computers* **C-21**(8), 880–886 (1972).

[3] Ho, H., Szwarc, V., and Kwasniewski, T., "Pipelined digital design in SRAM FPGAs," in [*Electrical and Computer Engineering, 1997. Engineering Innovation: Voyage of Discovery. IEEE 1997 Canadian Conference on*], **1**, 23–26 vol.1 (May 1997).

[4] CMS Collaboration, [*CMS Technical Design Report for the Level-1 Trigger Upgrade*], no. CERN-LHCC-2013-011. CMS-TDR-12 in Technical Design Report CMS (Jun 2013). https://cds.cern.ch/record/1556311.

[5] Zabołotny, W. M., Bartkiewicz, D., Bluj, M., et al., " FPGA implementation of overlap MTF trigger: preliminary study ," *Proc. SPIE* **9290**, 929025–929025–11 (2014). http://dx.doi.org/10.1117/12.2073380.

[6] 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**(03), C03004 (2016).

[7] Bluj, M., Buńkowski, K., Byszuk, A., et al., "From the physical model to the electronic system - OMTF trigger for CMS." Presented on Nica Days 2015, Warsaw, Poland, accepted to Acta Physica Polonica B, Proceedings Supplement (2016).

[8] Zabolotny, W., "VHDL sources of the parametrized pipelined block for finding the maximum value." Online post to alt.sources, also available at http://ftp.funet.fi/pub/archive/alt.sources/2851.gz (June 2014). [Online; accessed 30-June-2016].

[9] Xilinx, "Ultrafast design methodology guide for the Vivado design suite." http://www.xilinx.com/support/documentation/sw_manuals/ug949-vivado-design-methodology.pdf (June 2015). [Online; accessed 30-June-2016].

[10] Xilinx, "Timing closure user guide." http://www.xilinx.com/support/documentation/sw_manuals/xilinx14_7/ug612.pdf (October 2012). [Online; accessed 30-June-2016].

[11] Altera, "AN 584: Timing closure methodology for advanced FPGA designs." https://www.altera.com/content/dam/altera-www/global/en_US/pdfs/literature/an/an584.pdf (December 2014). [Online; accessed 30-June-2016].

[12] Xilinx, "Xilinx System Generator v2.1 for Simulink." https://safe.nrao.edu/wiki/pub/CICADA/WebHome/xilinx_ref_guide.pdf (May 2002). [Online; accessed 30-June-2016].

[13] Gallagher, S., "Accelerating DSP algorithms using FPGAs." http://klabs.org/mapld04/presentations/session_p/p188_gallagher_s.ppt (June 2004). [Online; accessed 30-June-2016].

[14] Altera, "DSP Builder advanced blockset." https://www.altera.com/en_US/pdfs/literature/hb/dspb/hb_dspb_adv.pdf. [Online; accessed 30-June-2016].

[15] Michael, P. and Jervis, M., "The most under-rated FPGA design tool ever." http://www.eetimes.com/author.asp?section_id=36&doc_id=1327664 (September 2015). [Online; accessed 30-June-2016].

[16] Venkataramani, G. and Gu, Y., "System-level retiming and pipelining," in [*Field-Programmable Custom Computing Machines (FCCM), 2014 IEEE 22nd Annual International Symposium on*], 80–87 (May 2014).

[17] MathWorks, "Delay balancing and validation model workflow in HDL coder." http://www.mathworks.com/help/hdlcoder/examples/delay-balancing-and-validation-model-workflow-in-hdl-coder.html. [Online; accessed 30-June-2016].

[18] Xilinx, "Xilinx System Generator for DSP Version 9.1.01." http://www.xilinx.com/support/sw_manuals/sysgen_ug.pdf (March 2007). [Online; accessed 3-October-2015].

[19] "GHDL Where VHDL meets gcc." http://ghdl.free.fr/. [Online; accessed 30-June-2016].

[20] Ashenden, P. J. and Lewis, J., [*VHDL-2008: just the new stuff*], The Morgan Kaufmann Series in Systems on Silicon, Morgan and Kaufmann, Burlington, MA (2008).

[21] "Automatic latency equalizer for pipelined designs implemented in VHDL." http://opencores.org/project,lateq (September 2015). [Online; accessed 30-June-2016].

[22] "Python." https://www.python.org/. [Online; accessed 30-June-2016].

[23] "Welcome to GTKWave." http://gtkwave.sourceforge.net/. [Online; accessed 30-June-2016].

[24] "Vivado Design Suite." http://www.xilinx.com/products/design-tools/vivado.html. [Online; accessed 30-June-2016].