CERN - LHCb 99-006, TRIG
May 3rd, 1999

INTERFACING DETECTORS TO TRIGGER AND DAQ ELECTRONICS

Dario B. Crosetto

900 Hideaway Pl.
DeSoto, TX 75115
Crosetto@vxcern.cern.ch; Crosetto@riph6.rice.edu

Abstract

The complete design of the front-end electronics interfacing LHCb detectors, Level-0 trigger and higher levels of trigger with flexible configuration parameters has been made for a) ASIC implementation, and b) FPGA implementation. The importance of approaching designs in technology-independent form becomes essential with the actual rapid electronics evolution. Being able to constrain the entire design to a few types of replicated components: a) the fully programmable 3D-Flow system, and b) the configurable front-end circuit described in this article, provides even further advantages because only one or two types of components will need to migrate to the newer technologies. To base on today's technology the design of a system such as the LHCb project that is to begin working in 2006 is not cost-effective. The effort required to to migrate to a higher-performance will, in that case, be almost equivalent to completely redesigning the architecture from scratch. The proposed technology-independent design with the current configurable front-end module described in this article and the scalable 3D-Flow fully programmable system described elsewhere, based on the study of the evolution of electronics during the past few years and the forecasted advances in the years to come, aims to provide a technology-independent design which lends itself to any technology at any time. In this case, technology independance is based mainly on generic-HDL reusable code which allows a very rapid realization of the state-of-the-art circuits in terms of gate density, power dissipation, and clock frequency. The design of four trigger towers presently fits into an OR3T30 FPGA. Preliminary test results (provided in this paper) meet the functional requirements of LHCb and provide sufficient flexibility to introduce future changes. The complete system design is also provided along with the integration of the front-end design in the entire system and the cost and dimension of the electronics.

This material is largely based upon work supported by the Department of Energy under Grant No. DE-FG03-95ER81905.

1 INTRODUCTION

Implementation of electronics in an environment where technology is improving rapidly requires special attention. This is particularly true in the case where the design is targeted to be operational a few years from now.

This paper describes a practical example from beginning to end of an electronic design implemented for ASIC and for FPGA that is targeted to meet the requirements of LHCb front-end electronics; however, flexibility is provided to address the needs of future changes of the specifications and even to meet the requirements of other experiments.

The specific design is discussed in Section 5, Figure 9. Details for the modifications of the configuration parameters are provided in Section 4. Methodology and tools used in the design are described in Section 4. Integration of this module in a general scheme for interfacing is described in Section 2, while its integration in the hardware is described in Section 6.

2 INTERFACING DETECTORS TO TRIGGER AND DAQ ELECTRONICS

2.1 General scheme of the interface between detectors, triggers and DAQ electronics

In a High Energy Physics (HEP) experiment, hundred of thousands of electrical signals are generated every few tens of ns (called bunch crossing; in the case of the Large Hadron Collider LHC the bunch crossing is 25 ns) by different types of sensors installed on different sub-detectors, and are sent to the electronics for parallel signal analysis.

Since the sub-detectors may be far from each other (each one thus detecting the hit of the same particle at different times required by the Time Of Flight – TOF – of the particle in reaching the sensors at different locations), and since the cables from the sub-detectors to the electronics may have different lengths, all signals (also called "raw data" after conversion to digital form) belonging to the same bunch crossing time must be synchronized by the electronics. (This function is implemented in the component called Front-End FPGA (Field Programmable Gate Array) shown in Figure 1 and indicated by the number '1' inside a circle).

Since the data rate is very high (hundreds of MHz), trigger decisions must be based on a wisely chosen sub-sample of the signals because it is not possible to analyze all the signals for all events.

This fast processing unit analyzing and correlating many signals at 40 MHz input data rate is called "Trigger Unit." (A fully programmable, general-purpose trigger unit implemented with the 3D-Flow is described elsewhere[2], [1], [2], [3].). The input signals needed by the "Trigger Unit" are extracted from the overall raw data in the front-end chip by the block indicated by 2 inside the circle in Figure 1.

During the time the trigger unit is analyzing the sub-set of data and arriving at a decision whether to accept or reject an event (event is defined as all signals belonging to a certain bunch crossing time), the full granularity (intended as full time and spatial resolution information from all sensors) of all signals from all sub-detectors is stored into a circular pipeline buffer. This functional block is indicated by the number 3 inside a circle in the Front-End chip of Figure 1.

Typically, in most of the current experiment, the time required for this first stage of data rate reduction is of the order of 3 μs. This includes not only the processing time by the trigger unit, but also all cable delay and other electronics.

The entire process is synchronous. Every 25 ns, a new set of data is received from all sub-detectors and at the same time a Yes/No global-level trigger signal (indicated as G_L0 in Figure 1) accepts (by transferring all data into the FIFO) or rejects the data relative to the event that occurred 128 bunch crossings (or cycles) before. (In this specific case, 128 x 25 ns = 3.2 μs).

Since we do not know which event will be accepted, but we do know instead from Monte Carlo simulation that an average acceptance rate at this stage ranges from 100,000 to 1 million events per second, the electronics sustaining the highest expected acceptance rate for a given experiment should be designed and built.

The 3D-Flow trigger system is totally flexible to sustain the entire acceptance range and to serve all types of experiments. The design and implementation of the Front-End chip has followed the same criteria of flexibility, modularity, and commonality as was the case for the 3D-Flow for the fully programmable trigger design. In the Front-End chip design, the depth/width of the FIFO, the bits that form the trigger word to be sent to the trigger processor, the depths of the pipeline buffer and the variable delay applicable to each input bit in order to synchronize the signals from the detector are configurable and can be adapted to the requirements of different experiments or can accommodate future changes for the same experiment.

Finally, the reduced raw data are available in the FIFO to be sent to the Data Acquisition system and to the higher level of the trigger system shown in the right-hand side of Figure 1.

The FIFO is used to derandomize the accepted event by the global level-0 trigger and the input of the level-1 trigger unit. The depth of the FIFO is determined by the maximum number of accepted events within a given time period.

The decision to fetch a new event from the FIFO is taken by the next trigger level that sends a read-FIFO signals when it is ready to read a new event.

The present design also provides the next level trigger with the information on the exact number of events in the FIFO at each given time. This information is useful in case the next level trigger has the capability of increasing its input data read rate, preventing the FIFO from getting full.

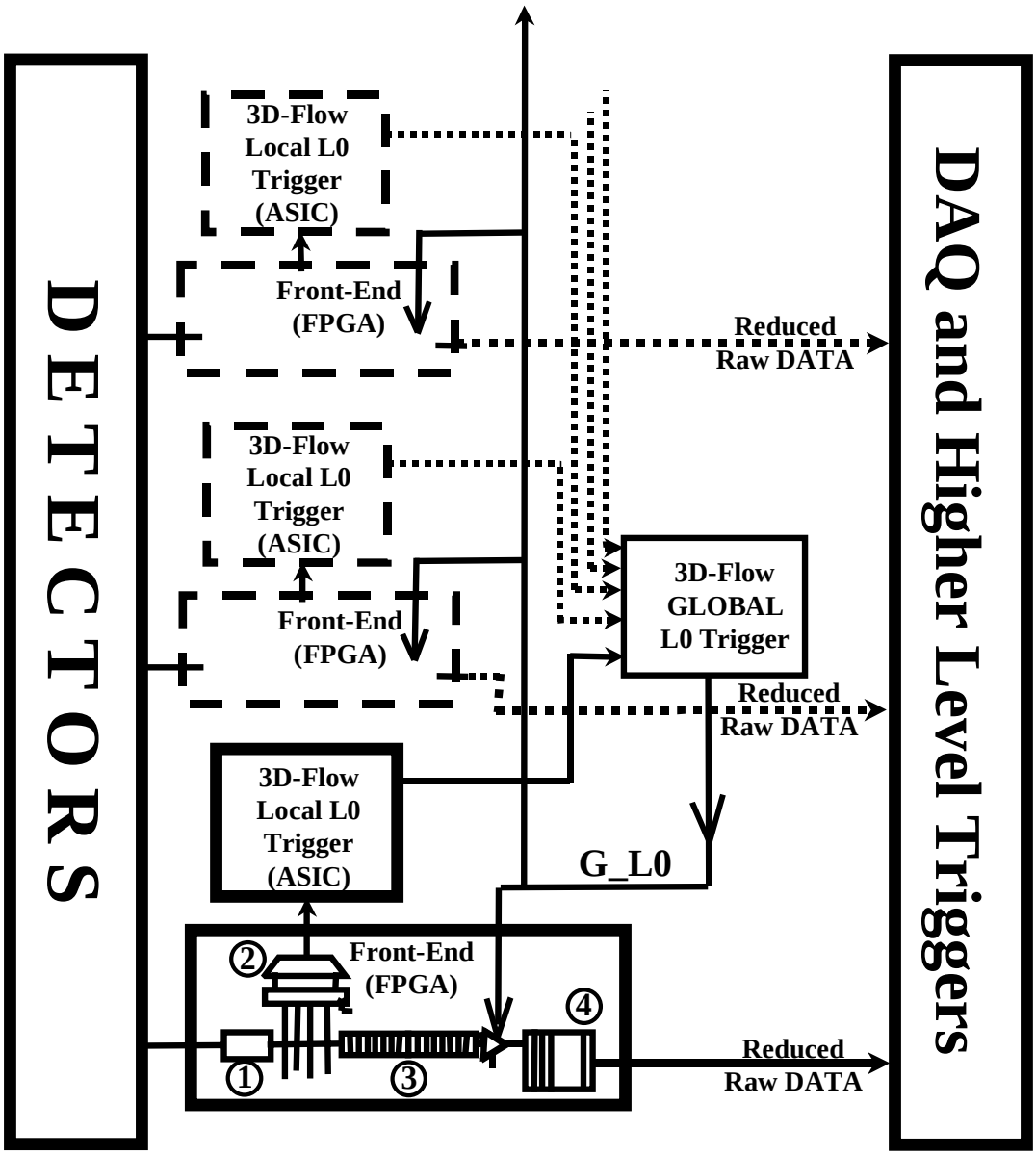


Figure 1. General scheme of the interface between detectors, triggers, and DAQ electronics.

2.2 Overview of the LHCb Level-0 trigger

Figure 2 shows an overview of the entire LHCb level-0 trigger with the path of the signals from the different sub-detectors to the electronics, and the corresponding time delays.

The LHCb detector, consisting of several sub-components (ECAL, HCAL, PreShower, Muon, VDET, TRACK, and RICH) monitors the collisions among proton bunches occurring at a rate of 40 MHz (corresponding to the 25 nsec bunch crossing rate). At every crossing, the whole information from the detector (data-path) is collected (indicated in the figure by the number 4), digitized (indicated by the number 6), synchronized, and temporarily stored (indicated by 7) into digital pipelines (conceptually similar to 128 deep, 40 MHz shift registers), while the Trigger Electronics (indicated by 8 and 9), by examining a subset of the whole event data (trigger path), decides (indicated by 10) whether the event should be kept for further examination or discarded. In the LHCb design, the input rate of 40 Tbytes per sec[11] (see top of the figure) needs to be reduced, in the first level of

triggering to 1 Tbytes/sec, i.e. a 1 MHz rate of accepted events. The selection is performed by two trigger systems (indicated by 8) running in parallel, the Calorimeter Trigger, utilizing mainly the information from the ElectroMagnetic and Hadronic Calorimeters (ECAL and HCAL) to recognize high transverse momentum electrons, hadrons and photons; and the Muon Trigger, utilizing the information from five planes of muon detectors to recognize high transverse momentum muons.

The resulting global level-0 trigger accept signal (indicated by 10 in the figure) enables the data in the data-path to be stored first into a derandomizing FIFO and later to be sent through optical fiber links to the higher level triggers and to the data acquisition system (see in Figure 8 the signal Global L0 distributed to all front-end 128 bunch crossing (bx) pipeline buffers). Real-time monitoring systems (L0 CAL monitor and L0 MUON monitor) supervise and diagnose the programmable level-0 trigger from the distant control room.

	Time [ns]	Δt	t
1	TOF at Chann.	50	50
2	Detector Dly	25	75
3	Cable on-det.	175	250
4	On-det. Electr.	40	290
5	Cable off-det.	150	440
5.1	Cable off-det.	300	--
6	Receiv.+ADC	50	490
7	Form. + Sync.	100	590
8	Tp Data Red.	300	890
9	Tp Chan. Red.	1300	2190
10	Global L-0	100	2290
11	TTC distrib.	500	2790
	TOTAL		2790

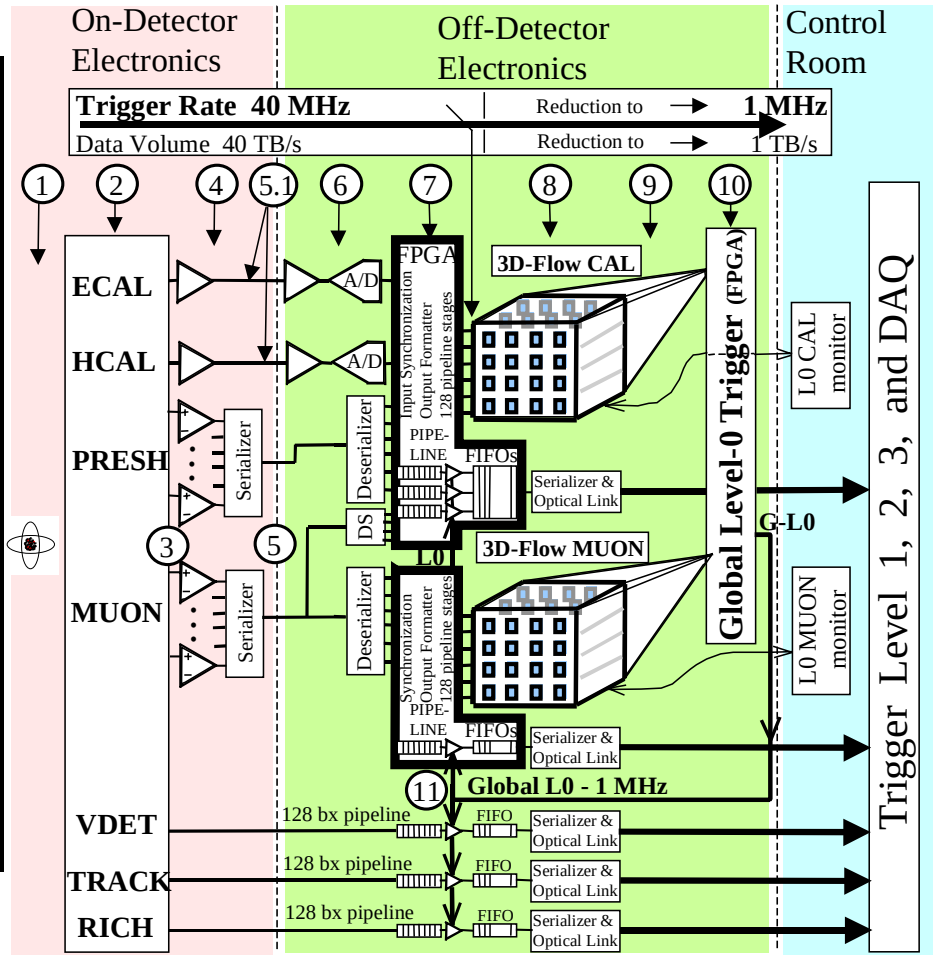


Figure 2. LHCb Level-0 Trigger overview.

The current article describes in detail the design and implementation of the Front-End circuit indicated by the number 7 in Figure 2.

3 CONFIGURABLE FRONT-END (FE) INTERFACING MODULE FOR SEVERAL APPLICATIONS

In the present design, the problem of interfacing detectors, triggers, and DAQ electronics has been approached keeping in mind the general scheme shown in Figure 1 and the specific needs of LHCb shown in Figure 2 (indicated by 7).

Even if the goal was to make a design that meets the requirements of LHCb front-end circuit interfacing specific subdetectors to the electronic with specific functions of the trigger and DAQ (see Figure 3b), the approach followed provides a much more general solution (see Figure 3a). This approach is such that the same front-end module can equally solve the problem of the front-end circuitry of the LHCb muon sub-detector and serve as the front-end of other experiments.

Instead of limiting the design to a circuit that interfaces the signals from 8 PADs of M1 muon station, 4 signals from preshower, 4 signals from the electromagnetic, and 1 signal from the hadronic calorimeter of the LHCb specific geometry to the triggers and DAQ, one can look at the present design as if a circuit with general features were available to the user.

The general features of the circuit are those of providing a certain number of interface channels from any detector type (one or more bits per detector) to the DAQ and higher level triggers.

Each channel has a pipeline buffer to store the information during trigger decision time, and each value received from the sensors has a time-stamp associated to it that will be sent out, together with the sensor value, in case the event that occurred at its specific time stamp is accepted.

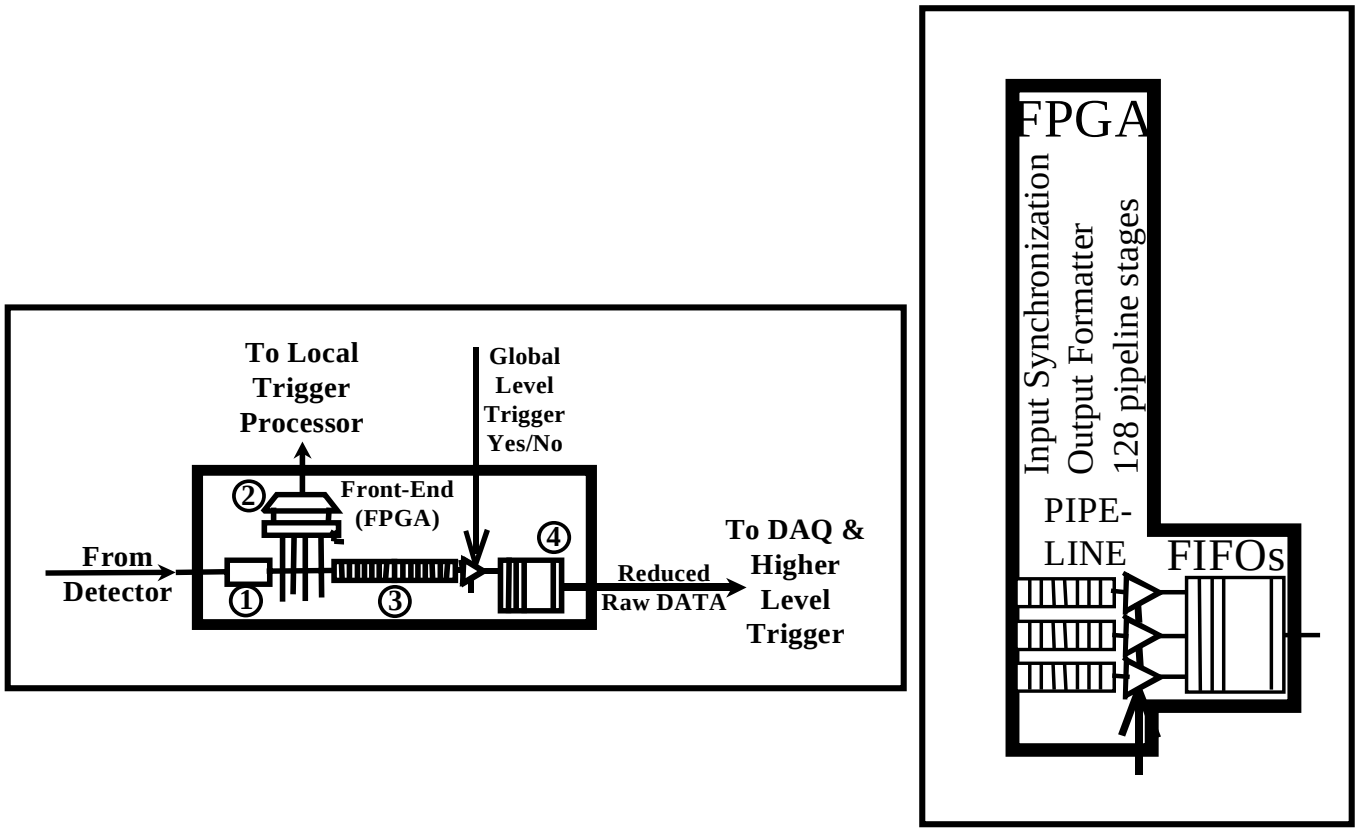


Figure 3. Configurable front-end (fe) interfacing module for several applications

For the specific case described in this document of the LHCb front-end calorimeter trigger, each FPGA component described in Section 5.2, accommodates 72 channels with data associated to a time stamp. In this case, the channels are assigned as follows: 12-bit for the hadronic, 4 x 12-bit for the electromagnetic, 4 x 1-bit for the preshower and 4 x 2-bit for the PADs.

As the circuit is currently conceived, a channel (representing 1-bit of information received from the sensors) can be associated to 1-bit of the 12-bit ADC converter, to 1-bit of the preshower, or to any of the information received from the sub-detectors.

At each channel, a delay can be inserted for the purpose of synchronizing the information belonging to the same event (or bunch crossing time). Each channel stores the information in a circular pipeline buffer to allow the lower level trigger unit to take a decision within a few microseconds. The candidates accepted by the global lower level trigger unit are stored in a derandomizing FIFO, ready to be read out by the DAQ and higher level triggers. Any of the channels can be selected and combined in any order to form the trigger-word to be sent to the trigger processor. The feature of receiving information from neighboring elements such as the PADs for the calorimeter trigger that are to be used in formatting the trigger word is also implemented without needing to duplicate all channels (pipeline, FIFO, etc.).

All the above parameters (FIFO depth/width, input delay, pipeline buffer depth, trigger word extraction and grouping) can be configured differently for each application. The changes need to be introduced only in one file (shown in Table 1), that is kept separate from the other code. Thus, the same front-end circuit can be used for the front-end circuit of the LHCb muon sub-detector, as well as for other experiments.

After the parameters have been changed in the configuration file, the execution of the script file reported in Table 2 recompiles the entire project making it ready to be simulated by software simulation tools such as Model Technologies, and to be synthesized into FPGA(Field Programmable Gate Array) or ASIC (Application Specific Integrated Circuit).

The selection of accommodating 72 channels is a good compromise between several factors such as: a) the number of components that will be required on a board (16), b) the size of each component, c) the number of inputs/outputs per chip, d) a

good partition of a “Trigger Tower,” i.e., a logical group of signals from the LHCb subdetectors, e) the fact that each component can accommodate four of them, and f) the fact that the front-end circuit can be implemented either on a medium-cost FPGA, offering maximum flexibility, or in a small-cost ASIC.

Table 1. Configuration parameters for the front-end chip.

```
--                                                                      --
-- Copyright (c) 1999 by 3D-Computing, Inc.                             --
--                                             All rights reserved.     --
--  Author       : Dario Crosetto                                       --
--                                                                      --
-- This source file is FREE for Universities, National Labs and         --
-- International Labs of non-profit organizations provided that the     --
-- above statements are not removed from the file,                      --
-- that the revision history is updated if changes are introduced, and  --
-- that any derivative work contains the entire above-mentioned notice. --
--                                                                      --
--  Package name : FE_config.vhd
--
--  Project   : Front-End Electronics Logic
--  Purpose   : This file contains the configuration parameters of the
--              chip. A change of a parameter in this file will affect
--              changes in all the modules of the front-end project design.
--              After the changes, the user should recompile the entire
--              project using the script macro.
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
-------------------------------------------------------------------------
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;

PACKAGE FE_config IS
-------------------------------------------------------------------------------
-- declare the constants used in the design.                                 --
-------------------------------------------------------------------------------

       CONSTANT PS_del  : std_logic_vector(1 DOWNTO 0) := "10";     --select delay 2
       CONSTANT HD_del  : std_Logic_vector(1 DOWNTO 0) := "00";     --select delay 0
       CONSTANT EM_del  : std_logic_vector(1 DOWNTO 0) := "00";     --select delay 0
       CONSTANT M1_del  : std_logic_vector(1 DOWNTO 0) := "01";     --select delay 1

    CONSTANT Time_ID_width  : INTEGER:= 8 ; -- # bits of the time_stamp info
    CONSTANT M1_width       : INTEGER:= 2;  -- # bits of M1 data
    CONSTANT adc_width      : INTEGER:= 12; -- # bits of ADC data
    CONSTANT Width_To3DF    : INTEGER:= 8;  -- width of 3D-Flow input data port
    CONSTANT fifo_depth     : INTEGER:= 5;  -- depth of output fifo (power of 2)
    CONSTANT fifo_width     : INTEGER:= 80; -- width of output fifo (# of bits)
    CONSTANT PIPE_depth     : INTEGER:= 128;-- depth of pipeline buffer (# of locations)
    CONSTANT EM_trig_width  : INTEGER:= 8;  -- EM bits used for trigger
    CONSTANT HA_trig_width  : INTEGER:= 8;  -- HAD bits used for trigger
    CONSTANT PS_trig_width  : INTEGER:= 1;  -- PS bits used for trigger
    CONSTANT M1_trig_width  : INTEGER:= 2;  -- M1 bits used for trigger

END FE_config;
```

Table 2. Script file that recompiles the entire front-end chip for simulation.

```
vcom -work work -explicit -93 c:\3d_comp8\ORCA_VHDL_notime_FE\source\FE_config.vhd
vcom -work work -explicit -93 c:\3d_comp8\ORCA_VHDL_notime_FE\source\FE_syncinput.vhd
vcom -work work -explicit -93 c:\3d_comp8\ORCA_VHDL_notime_FE\source\FE_fifo.vhd
vcom -work work -explicit -93 c:\3d_comp8\ORCA_VHDL_notime_FE\source\FE_pipeline.vhd
vcom -work work -explicit -93 c:\3d_comp8\ORCA_VHDL_notime_FE\source\FE_trig_formatter.vhd
vcom -work work -explicit -93 c:\3d_comp8\ORCA_VHDL_notime_FE\source\FE_top.vhd
vcom -work work -explicit -93 c:\3d_comp8\ORCA_VHDL_notime_FE\source\FE_testbench_v2.vhd
```

4 CIRCUIT VERIFICATION: METHODOLOGY BASED ON NEW SOFTWARE TOOLS vs. OLD HARDWARE PROTOTYPING

The world of electronics design is changing rapidly. The construction of a circuit (to prove its feasibility) done in the past on a printed circuit board, using different components purchased at the store, is now replaced by the design or purchase of Virtual Components (VC, in the form of IP blocks, -- Intellectual Property --). Catalogs of VC IPs are replacing catalogs of hardware components such as AND, OR, FLIP-FLOP, etc.

Powerful software tools allowing synthesis into FPGA logic blocks, or ASIC logic gates and simulation down to the gate level with timing information linked to a specific technology, replace the hard work of assembling the old hardware component on a printed circuit board and testing the board with a pulse generator, oscilloscope, or logic state analyzer.

The evaluation of a circuit implemented with certain components in the past is now replaced by the evaluation of a certain style to write the HDL (Hardware Description Language) code, the software testbenches, and simulation tools used. The fundamental concept of IPs (Intellectual Property) Virtual-Components (VC) is redefining the world of electronics. A catalog[4] of 200 IP VCs was introduced at DAC '98.

4.1 The evolution of IC Design

The evolution of ICs is illustrated in Figure 4. Information from specialized electronic magazines *(Electronics Journal, EE Times, ELECTRONIC magazine,* etc.) indicates that the evolution will continue. (e.g.: "... When Samsung Semiconductor Inc., migrated from its 0.35-micron ASIC process early this year to 0.25-micron, it was able to offer up to 230% increase in gate density, along with an 83% reduction in power consumption at 1.8 V..." as reported by *ELECTRONIC magazine in Oct. 98*). Analog design retains its investment for several years, while digital design becomes outdated in about one year.

IC Design Complexity (average component size)

	Current Industry		Forecast	
	1997	1998	1999	2001
Logic Gates	**300K**	**1M**	**5M**	**12M**
Technology	0.5 µm	0.35 µm	0.25 µm	0.18 µm
Density [gates/mm^2]		~ 14000	~ 30000	~ 65000
Power [Gate/MHz]		700 nW	250 nW	23 nW
Speed	1.4 x '96	1.4 x '97	1.4 x '98	1.4 x '2000
Cost/100K gates	$25	$5	~ $2	???

Figure 4. The evolution of IC design.

The progression of figures from 1997 to the year 2001 (see Figure 4), shows that not only is there an advantage in lower prices, but more so that the important advantage is in increased performance. The consequences of using a technology that dissipates only 23 nW per gate/MHz instead of 700 nW facilitate the construction of boards, crates, power supplies, cooling systems, and so forth.

4.2 System design and verification process

The importance of approaching designs in technology-independent form becomes essential with such a rapid electronics evolution. The fact of being able to constrain the entire design to a few types of replicated components: a) the fully programmable 3D-Flow system, and b) the configurable front-end circuit described in this article, provides even further advantages because only one or two types of components need to migrate to the newer technologies.

To base on today's technology the design of a system such as the LHCb project that is to begin working in 2006 is not cost-effective. The effort required to to migrate to a higher-performance will, in that case, be almost equivalent to completely redesigning the architecture from scratch. The proposed technology-independent design with the current configurable front-end module described in this article and the scalable 3D-Flow fully programmable system described elsewhere, based on the study of the evolution of electronics during the past few years and the forecasted advances in the years to come, aims to provide a technology-independent design which lends itself to any technology at any time. In this case, technology independance is based

mainly on generic-HDL reusable code which allows a very rapid realization of the state-of-the-art circuits in terms of gate density, power dissipation, and clock frequency.

One significant advantage is that all progress already made in developing the design and verification tools and the real-time monitor software remains, whereas in a system that is not fully programmable, not scalable, not modular, not technology-independent, and that lacks the commonality to meet the requirements of different experiments, all software development/verification/real-time monitoring tools have to be redesigned from scratch each time.

Even if we acknowledge the existence of a catalog of Virtual Components, the world of the "design for reuse" is still shaping and the process of standardization is still going on. Several proposals for standards are currently in review. Many new terms (macro, subblock, virtual component, hard macro, soft macro, etc.) in this field have a meaning that is not always accepted unanimously, and sometimes the same meaning is conveyed by different terms by different parties.

In our case, the most practical way to demonstrate that the world of electronics is changing and that the exchange/verification of hardware prototype boards that occurred in the past has been replaced with today's exchange/verification of VHDL code and software testbenches, is to exchange files that describe circuits instead of exchanging hardware.

While the standardization is defining the new terms and rules for Virtual Components and for circuit validation, we merely need to define a few basic concepts and the terminology in order to arrive at a common understanding of how the technology-independent design can be achieved and a "design for test" implemented.

There are two main categories of modules that a chip designer can use:

- Hard macro (or core, or block) is usually delivered as a GDSII file. The supplier provides a complete design for this module that includes place and route.
- Soft macro (or core, or block) is a synthesizable RTL code.

Examples of macros are: a) a microprocessor core; b) a memory block; c) a UART interface; d) a PCI interface, etc.

Synthesizable means that the RTL (Register Transfer Logic) code can be converted by the synthesis tools available from several vendors into silicon gates (or cells, PFU for the ORCA FPGA, LC for Altera FPGA, or CLB for Xilinx FPGA, etc.).

A technology-independent design will obviously derive the most benefit from using the "soft macro," however, with its protection as an IP (Intellectual Property), it is also more difficult. The current design reported in this article, "the configurable front-end circuit" is provided as a "soft macro" in a set of five VHDL synthesizable source files and one configuration file.

The industry prefers to provide the files of a "hard macro," which are easier to protect from being copied, because, in most cases, they are the ones that also own also the silicon foundry with the technology process.

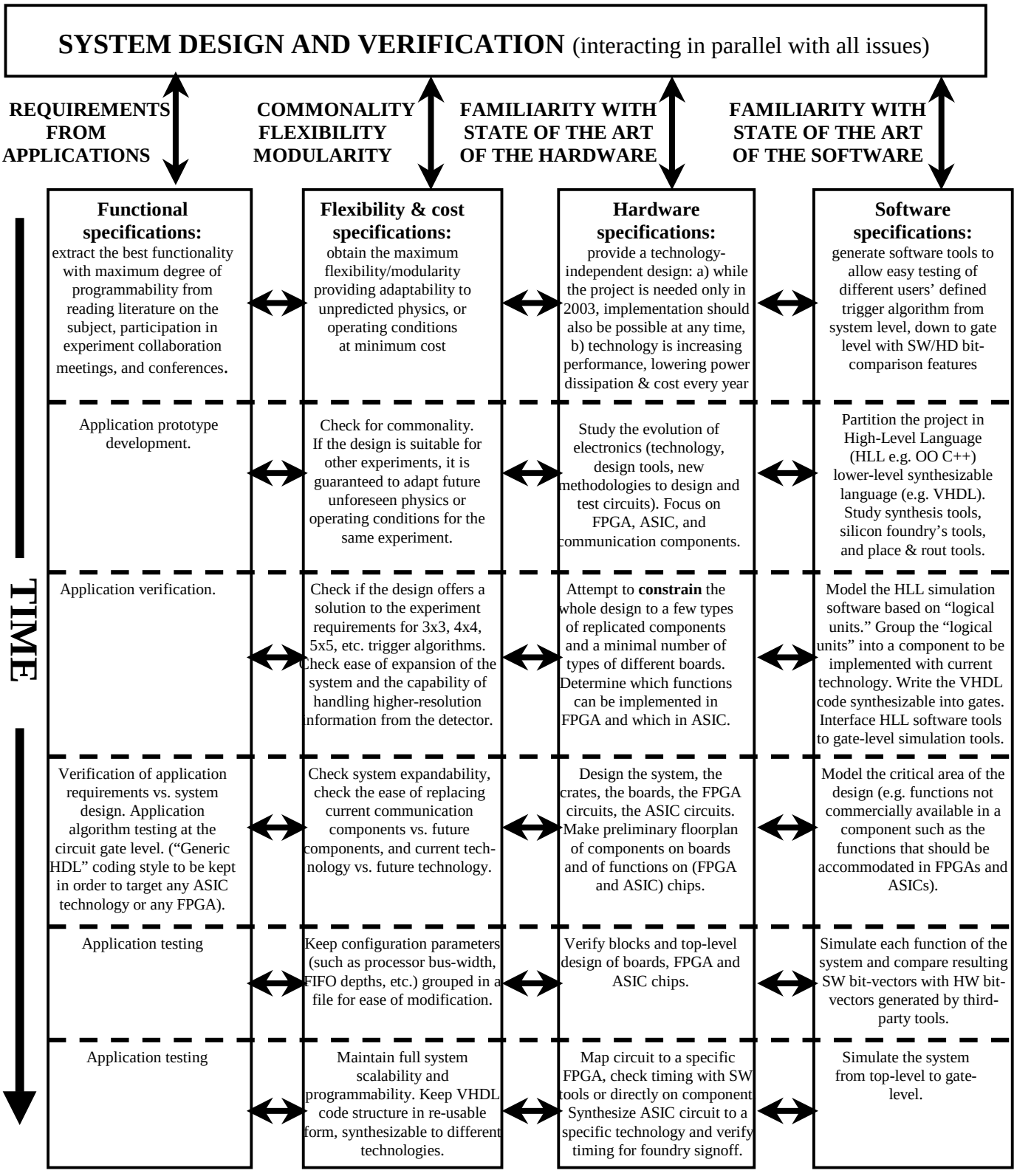
Having established the importance of a VHDL synthesizable code nowadays replacing a hardware circuit, the focus and the great advantage of the approach described in this article and in the previous article[5] is the overall architecture of both designs.

The basis of the approach is the study of the requirements of several applications followed by the attempt to **constrain** the whole design to a few types of replicated components and a minimal number of types of different boards.

The result of this approach leads to one type of front-end circuit, one type of 3D-Flow processor, and one type of board accommodating both components. The resulting system is flexible, programmable, modular, scalable, and has the commonality to meet the requirements of different HEP experiments or applications.

Table 3 shows the design and verification process used in the past four years in achieving these results.

Table 3. System and design verification process.



Besides the theory described in Table 3, some practical examples follow of how this process was conducted:

- The best achievement was obtained by being able to constrain the entire design to a single type of board with two types of replicated components containing special functions, still in a programmable form, however.
- The front-end chip has only one output line to the DAQ and to the higher level triggers. This satisfies all requirements and matches well with the other sections of the board, system, design. For example, a) 1 MHz output of 80-bits raw data is the requirement from the LHCb TP; b) 80-bit @ 1 MHz is equivalent to 1-bit @ 80 MHz from one pin; c) the pin count of the FPGA is reduced by 15, while the 80 MHz is close to the limit of the I/O speed of today's FPGAs; d) the number of FPGA components on one board is 16 as shown in Figures 15 and 16; 16-bit x 80 MHz is a good match with the speed of a low-cost optical serializer/transmitter @ 1.28 Mbps).
- One front-end chip can accommodate the functionality of four trigger tower FEs. The size of the FPGA and the number of channels per chip, allows for only one column of components in front of the 3D-Flow section of the board (see Figure 15 and 16), thus allowing connection with equal-length optimization. Four front-end chips provide the signals to the closest 3D-Flow chip. Output speed, protocol signals, and pins can match the input speed, protocol, and signals of the 3D-Flow chip.
- The 3D-Flow can communicate with neighbors, off-board and off crate, matching the capabilities of the speed and the number of signals that a backplane and connectors can carry (the design foresees only a speed of 400 Mbps, while existing implementations already carry 622 Mbps and there are R&D for 1 Gbps connections on the backplane). The number of communication lines also satisfies the number of layers of 3D-Flow processors that would execute the most complex algorithm simulated and required by the LHCb TP.
- The testability of the design has been kept in mind since the conception of the architecture and since the beginning of the 3D-Flow approach. The choice of real-time monitoring through RS232 was made after analyzing the simplicity, low-cost, and world-wide use of that interface in the industry with respect to VME or other more complicated interfaces. A transparency with the comparison of the use of different interfaces around the world was shown by the author at a collaboration meeting in September 1997.

As the design is structured at the present stage, it is easy to verify all different sections from specification to silicon against existing technology. By having designed each section with the concept of reusable code and methodology in mind, improvements with advances in technology are facilitated.

The monitoring of the system during real time can easily be debugged at present with the software tools developed during the past few years that allow for communication between a System Monitor (SM) and a Virtual Processing System (VPS) through RS232 interfaces.

4.3 FPGA design and verification tools

The generic design flow of an FPGA (Field Programmable Gate Array) can follow two main paths: a) using HDL coding (Hardware Description Language. The two main languages used are: VHDL and Verilog. In this article, all HDL coding refers to VHDL), b) using schematic entry.

The first technique of HDL coding may be further subdivided into three methods, each one providing a different degree of technologically-independent code style.

- Method I: written in "Generic HDL" code with no FPGA specific architecture knowledge and no "hard macro" instantiation.
- Method II: written in "Generic HDL" code with some FPGA specific architecture knowledge and no "hard macro" instantiation
- Method III: written in HDL code with some FPGA specific architecture knowledge and with some "hard macro" instantiation

The result is an HDL technology-independent code for method I with the advantage of being a) ready for simulation, and b) ready to be used in a different FPGA (or ready to be targeted to an ASIC). However, this method has the drawback of a) not taking advantage of all the architectural features of the PFU (Programmable Functional Unit) of the FPGA chip, and b) thus requiring one and half times to even more than twice the number of PFUs with respect to a design best optimized with "hard macro" for a specific FPGA architecture. (This requires a large, thus more costly FPGA chip).

Method III will result instead in the drawbacks of a) taking advantage of the architectural features of the PFU of the FPGA, b) a smaller number of PFUs, and thus requiring a smaller and less costly chip. However, a) this method will not be technology-independent, and b) this method will not have a model ready for simulation, so that the user will have to design a model with behavior approximating the behavior of the "hard macro," c) some additional tools may be required such as a graphic layout editor, etc.

Method II still provides a code ready to be simulated, and the HDL can be kept technology-independent, however, it requires the knowledge of the PFU architecture of the FPGA, the user capability to write HDL code in a different style, more suitable to take advantages of the PFU architecture.

An alternative design entry to the HDL is the schematic entry. In this case, the efficiency in the best utilization of the features of the PFUs of the FPGA may be very high (thus lending itself to smaller FPGAs at a lower cost); however, the design is no longer technologically independent.

Whether HDL entry or schematic entry is used, the sequence of the steps required for an FPGA implementation is:

- Design Entry
- RTL (Register Transfer Level) simulation
- RTL Synthesis (Using third party tools)
- Place and route (Using vendors tools. In the present case ORCA Foundry tools from Lucent Technologies)
- Timing simulation
- Program/Fuse of the FPGA.

The design process is iterative. Feed-back from timing simulation may require corrections at different level of the design (e.g., place and route, RTL synthesis, or HLD source code).

The current project of the front-end chip described in this article has been implemented using HDL method I as well as using schematic entry.

The first method provides a technology-independent solution (that may also be targeted to ASIC). The second method provides the best utilization of the features of ORCA' PFUs resulting in the utilization of a smaller FPGA OR3T30 at a cost of about $75, while the first method (using the best third-party synthesis tools from VHDL to ORCA OR3Txx technology) requires a OR3T80 FPGA at a cost of about $300.

Files on both approaches are provided on the web and the complete source code of Method I is also provided in the appendix of this article.

Design verification and testing was done by a preliminary testbench program written by the author (the code is in VHDL form, the complete listing is provided in the appendix and is made available on the web), and are applied to the behavioral VHDL model and the “routed” version of the front-end chip.

The tools used for the verification and test of the design are Model Technologies simulation tools and Lucent Technology ORCA Foundry tools Version 9_35.57. ORCA Foundry tools generate a back-annotated VHDL file (this file that is called “routed.vhd” is also provided on the web) with timing information that can be used by Model Technologies simulation tools for simulation verifying the timing provided by ORCA FPGA technological process. The user can download all files from the web to perform the same verification.

4.4 ASIC design and verification tools

The same front-end circuit (in units of at least 4 trigger-tower groups of signals per component), as was the case for the 3D-Flow processor, can be targeted to an ASIC in CMOS gate-array or standard-cell technology by following all steps from the “Generic HDL” code.

Different VHDL coding styles are pursued when the VHDL code designer has the capability to think like hardware, anticipating the netlist that the logic synthesis will produce.

Among the different technologies available on the market (GaAs, BiCMOS, ECL, MOS, NMOS), the CMOS (Complementary Metal Oxide Silicon) has established a predominant position in the market, is manufactured in much greater volume than any other technologies, and thus offer the best prices. Around early 1980, the aluminum gates of the transistors were replaced by polysilicon gates; however, the name MOS remained. The use of polysilicon material provided major advantages: a) it was easier to make the two types of transistors, n-channel MOS and p-channel MOS on the same IC (Integrated Circuit), b) there is a major advantage of CMOS over NMOS in lower power consumption, c) the fabrication process could be simplified, allowing devices to be scaled down in size. (The unit currently used to indicate a technology of 0.5 μm refer to the length of the smallest transistor that a specific process can implement. Thus a 0.18 μm process, means that the smallest transistors are 0.18 μm in length.)

The gate-array is made of several gates. One gate is made of four transistors; thus, to convert from gate to transistor one should multiply the number of gates by 4 to obtain the number of transistors.

Like the gate array, the standard-cell also makes use of predefined cells. The difference between the two is that while it is possible to change the transistor size in a standard-cell to optimize the speed and performance, it is not possible to change it in a gate-array which has a fixed device size. Typically a Non Recurring Engineer (NRE) cost is higher for standard-cell (as it will require a longer technological process, consequently taking longer), with respect to the gate-array that comes with predefined elements.

The paths to design an ASIC are the following:

- **Testbench validation** (RTL)
- **Behavioral simulation** (models large pieces of a system as black boxes with inputs and outputs)

- **Functional simulation** (ignores timing and includes unit-delay simulation, which sets delays at a fixed value)
- **Preliminary synthesis** (The design at this stage may still be technology-independent. It provides a link between and HDL code and a netlist. EDIF –Electronic Design Interchange Format-- is widely used)
  - **Logic synthesizer** (The design at this stage may still be technology-independent. It is based on a generic cell library).
  - **Logic optimization** (It attempts to improve the technology-independent network under automatic iteration of several parameter settings, or under the designer's control).
  - **Preliminary floorplan/layout/routing** (The design is mapped to a specified technology-dependent target cell or gate-array library. ASIC vendor's design tool checking for design rule violation is executed. Preliminary place and route is executed in order to obtain suggestions on the placement of hard macro such as the data memories and the program memories of the 3D-Flow processors. The back-annotated files generated by these tools will provide the information on a) timing, b) RAM placement suggestion, c) die size, d) wire load, and e) identification of risk area in the design).
  - **Timing simulation**
    - **Static timing analysis** (Works best with synchronous designs such as the front-end chip and the 3D-Flow, does not require the creation of a set of stimulus vectors. The user sets the top level constraints to be used for timing verification, software tools provide the path with the longest delay that is called "the critical path." In our case, for both projects: the front-end chip and the 3D-Flow processors chip, the constraint was set @ 80 MHz).
    - **Gate level simulation** (A logic gate is treated as a black box model by a function whose variables are the input signals. The function may also model the delay through the logic cell).
- **Resolve discrepancies**. Any discrepancies existing between gate-level simulation, RTL simulation, and C++ system level simulation results are resolved. (See bottom part of Figure 5, the Bit-Vector comparison).
- **Pinout/bonding diagram** (The package is selected, pinout is assigned, rule checking software is executed to verify that the bonding diagram meets the ASIC vendor's requirements).
- **Full synthesis** (Optimization of critical paths)
- **Test insertion** (Insertion of JTAG and full scan for factory and in-circuit test)
- **Final floorplan/layout/routing**. (Back-annotated results will be used to perform a further check on static timing analysis).
- **Back-annotated netlist/synthesis** (Use the back-annotated data to perform a final static analysis, and to perform netlist and synthesis changes until static timing gives satisfactory results)
- **Back annotated layout** (Back annotated layout results will be used to perform a further check on static timing analysis).
- **Static timing verification** (Final verification of static timing analysis)
- **Final functional verification**. Final verification and results comparison between gate-level simulation, RTL simulation, and C++ system level simulation are performed. (See bottom part of Figure 5, the Bit-Vector comparison).
- **Create factory test vector set.**

Figure 5 shows the ASIC verification design process. The user's real-time algorithm is simulated on the SYSTEM TEST-BENCH. Expected results (top right) are checked versus different input data set (top left). Bit-vectors for one or more PEs (for any PE in the system) are saved to a file (center bottom). Test-bench parameters for any PE(s) are generated by the system test-bench for software (center right) and hardware (center left) simulator. All bit-vectors are compared for design validation.

These tools are essential for the design verification from the user's defined real-time system algorithm to the gate-level simulation. The results from the comparison of the Bit-Vectors generated by the above-mentioned tools (at the items: "resolve discrepancies," and "Final functional verification") are verified to assure design integrity at any stage.

The top-level constraints of 80 MHz do not present difficulties to be achieved with current technologies. The design of a 3D-Flow chip with four processing elements has been synthesized into a 0.5 μm technology and a 0.35 μm technology. Simulation has been performed, Bit-Vectors have been compared between the system simulator and a 3D-Flow chip implemented with 0.35 μm CBA technology at 3.3 Volts. The CBA ASIC design tools show dissipation of 884 mW @ 60 MHz and a die size of 63.75 mm2 for a chip with 4 3D-Flow processors.

The difference in ~ 14,000 gate count/mm^2 for 0.35 μm to ~ 65,000 gate/count/mm^2 for 0.18 μm and the difference on power dissipation [gate/MHz] from 700 nW to 23 nW from 0.35 μm technology to 0.18 μm technology respectively (0.25 μm technology does not offer a noticeable power dissipation improvement @ 250 nW), indicates as the 0.18 μm technology @ 1.8 Volts power supply offers considerably more advantages with respect to 0.25 μm technology and that 16 3D-Flow processors in a chip would be the optimal implementation for a few years to come including the time the LHCb experiment needs to be ready for operation (considering that each 3D-Flow processor is approximately 100K gates, the total die size of the chip would be about 25 mm^2).

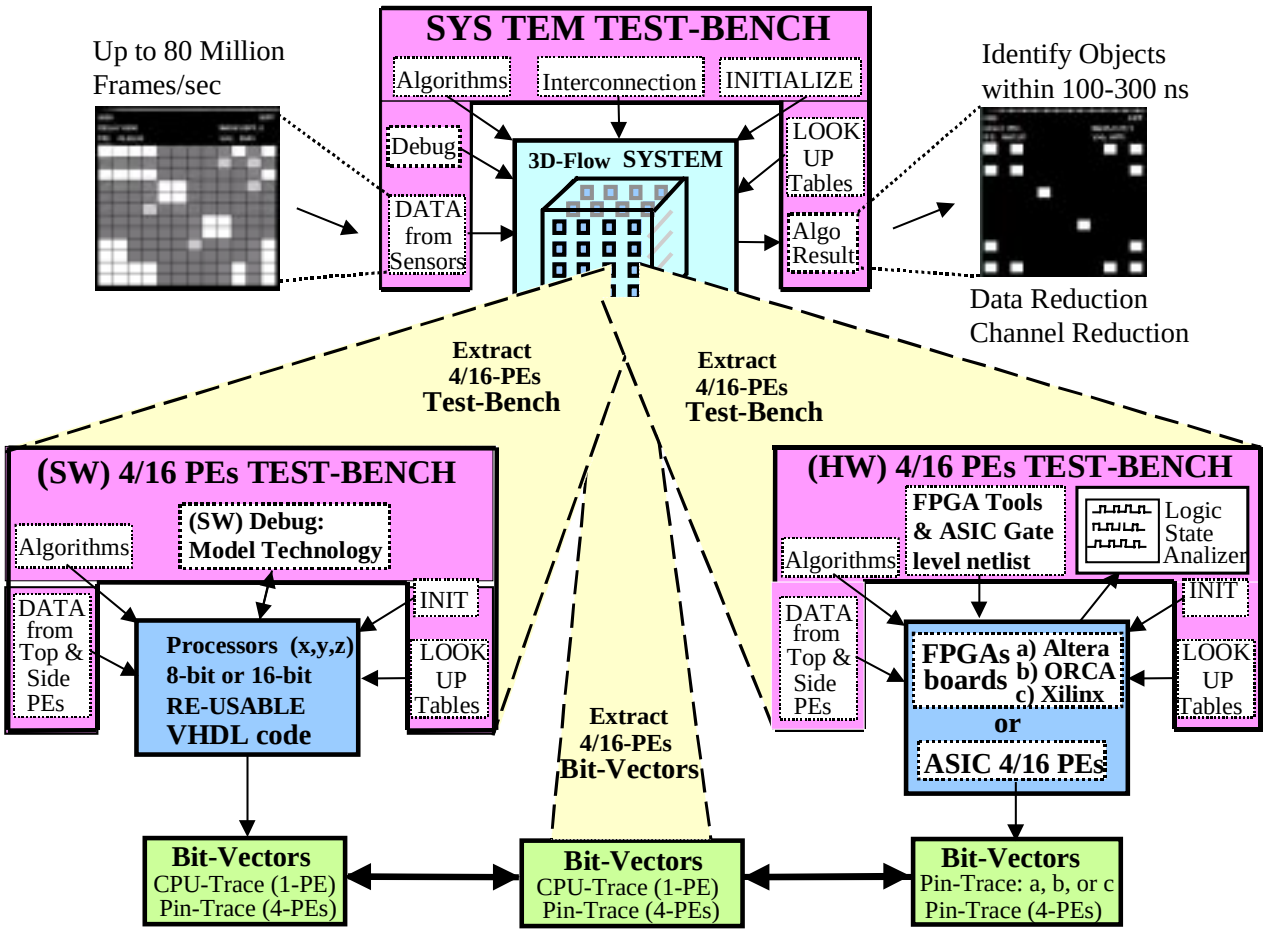


Figure 5. ASIC design verification. From user's system algorithm down to the gate-level circuit

4.5 Design Real-Time: the interface between Application, FPGA, and ASIC for a system design

The "link" between the third party tools described in the previous sections and the requirements of very high-speed real-time applications (with large volume of data to be correlated and processed in parallel), such as the one of the HEP experiments, is provided by the "Design Real-Time 2.0 tools."

The 3D-Flow Design Real-Time is a set of tools that allows the user to:

1. create a new 3D-Flow application (called project) by varying size, throughput, filtering algorithm, and routing algorithm, and by selecting the processor speed, lookup tables, number of input bits and output results for each set of data received for each algorithm execution;
2. simulate a specified parallel-processing system for a given algorithm on different sets of data. The flow of the data can be easily monitored and traced in any single processor of the system and in any stage of the process and system; and
3. monitor a 3D-Flow system in real-time via the RS232 interface, whether the system at the other end of the RS232 cable is real or virtual;
4. create a 3D-Flow chip accommodating several 3D-Flow processors by means of interfacing to the Electronic Design Automation (EDA) tools.

A flow guide helps the user through the above four phases.

A system summary displays for a 3D-Flow system created by the Design Real-Time tools, the following information:

1. characteristics, such as size, maximum input data rate, processor speed, maximum number of bits fetched at each algorithm execution, number of input channels, number of output channels, number of layers filtering the input data, number of layers routing the results from multiple channels to fewer output channels;
2. time required to execute the filtering algorithm and to route the results from multiple channels to fewer output channels.

A log file retains the information of the activity of the system when:

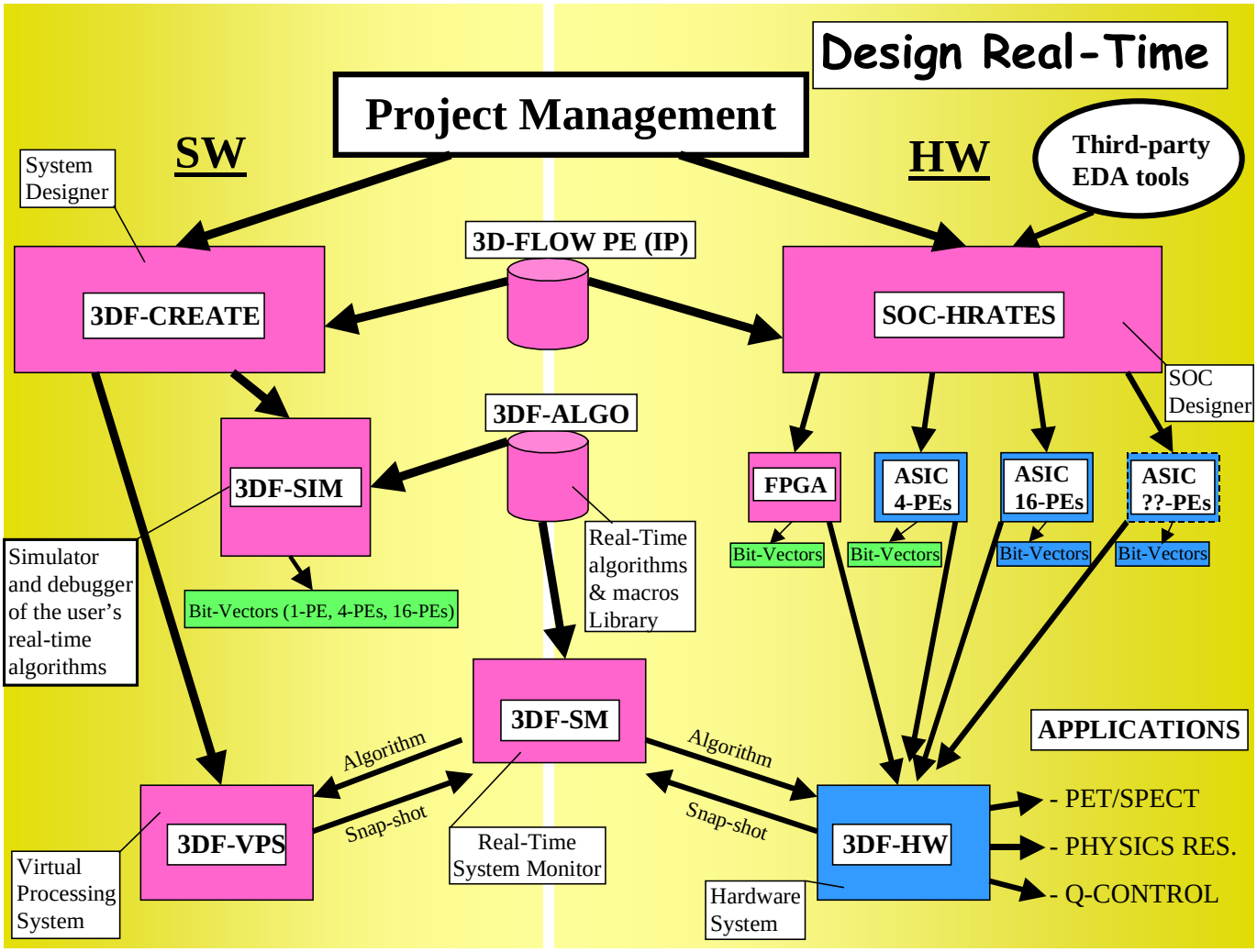
1. loading all modules in all processors;
2. initializing the system;
3. recording all faulty transactions detected in the system (e.g., data lost because the input data rate exceeded the limit of the system or because the occupancy was too high and the funneling of the results through fewer output channels exceeded the bandwidth of the system);
4. recording any malfunction of the system for a broken cable or for a faulty component.

A result window can be open at any time to visualize the results of the filtering or pattern recognition algorithm applied to the input data as they come out at any layer of the system.

The generation of test vectors for any processor of the system can be selected by the user at any time to create the binary files of all I/O's corresponding to the pins of a specific FPGA or ASIC chip. These vectors can then be compared with those generated by the chip itself or by the VHDL simulation.

Figure 6 shows the interrelation between the entities in the Real-Time Design Process.

The figure is separated into two sections. On the left is shown the flow of the software design and simulation process to create and simulate a 3D-Flow system, on the right is shown the System-On- a Chip for High-speed Real-Time Applications and



TESting (SOC-HRATES) process.

The centre of the figure shows the common entities of the system: a) the IP 3D-Flow processing element as the basic circuit to which it has been constrain the functionality required by different applications, b) a set of 3D-Flow real-time algorithms and macro grouped into a library, c) the System Monitor software package that allows, during operation in the target application, to monitor each 3D-Flow processor of the system via RS-232 lines. The System Monitor performs the function of a system-supervising host that loads different real-time algorithms into each processor during initialisation phase, detects malfunctioning

components during run-time and excludes with software repair malfunctioning processors. The System monitor can operate on a hardware 3D-Flow system as well as on a Virtual 3D-Flow system. Booths system communicates with the System Monitor via RS-232 lines. Data are captured at high-speed (at the processor speed of 80 MHz in the hardware system) on the 3D-Flow system at a preset trigger time for 8 consecutive cycles (called snap-shot), and are transferred at low speed (at the RS-232 speed of 230 KBaud) to the system monitor for system debugging, and/or monitoring.

During the usual procedure to create a 3D-Flow system to solve an application problem, typically the user define a size in "x" and "y" of the 3D-Flow system, based on the size of the detector to be interfaced, its number of channels, the number of bits per channels, and the correlation required between signals that is required by the trigger algorithm.

The third dimension "z" of the 3D-Flow system is determined by the complexity of the real-time algorithms (for HEP experiments first levels of trigger algorithms as the ones reported in the TP, they require less than 10 layers).

The software module "3DF-CREATE" shown on the left part of Figure 6, allows to define a 3D-Flow system of any size, and to interconnect processors to build a specific topology with or without the channel reduction stage of the pyramid. It allows also creating sample input data files from a random number generator or from the conversion of some pictures in bitmap of .tif form, to latter test the system.

The module "3DF-SIM" allows to simulate and debug the user's system real-time algorithm and generates the "Bit-Vectors" to be compared later with the ones generated by the third party silicon foundry tools. (shown on the right of Figure 6)

The module "3DF-VPS" is the Virtual Processing System that emulates a 3D-Flow hardware system. At the place of receiving real data from a detector, loads into the 3D-Flow processing system the data from a file that were previously recorded from a detector system (or loads data from Monte Carlo simulation, or from bitmap images, or from the internal random data generator). The VPS is connected to the System Monitor via RS-232 lines, as it would be the hardware system. It allows testing of the overall functionality, the communication protocol and efficiently of monitoring system's malfunctions through RS-232 lines before hardware construction.

The right side of Figure 6 show the hardware flow of the 3D-Flow system implementation in a System-On-a-Chip (SOC). The same common entity, the IP 3D-Flow processing element (PE) shown in the centre of the figure and previously used as behavioral model in the simulation, by using the same code it is now synthesized in a specific technology. In the case when an application consist of a number of 3D-Flow processing elements that can fit into a single piece of silicon (considering that each PE is about 100K gates, and that this year technology offers 30,000 gate/mm^2, and next year will offers 65,000 gates/mm^2, a considerable number PEs can be accommodate into a single chip), the entire 3D-Flow system can fit into a chip. For this reason it is also called SOC 3D-Flow. However, when an application requires to built a 3D-Flow system that cannot be accommodated into a single chip, than several chips, each accommodating several 3D-Flow PEs can be interfaced with glueless logic to build a system of any size to be accommodated on a board, on a crate, or on several crates[5].

The Design Real-Time tools offers the user the possibility to test at the gate-level the same system that was designed previously to solve a specific application and simulated before using a behavioral model.

Currently, the a single 8-bit internal bus 3D-Flow PE version has been synthesized for FPGA, and four PEs with 16-bit internal bus version has been synthesized for 0.5 μm and 0.35 μm technologies. Bit-Vectors generated by third party tools have been compared with the Bit-Vectors generated by the 3D-Flow simulator.

The verification process of an entire 3D-Flow system can be performed completely. It is just a matter of simulation time. The steps to be performed are the those shown in Figure 5.

- The 3D-Flow simulator: a) extracts the input data for the selected 3D-Flow processor (or group of processors) for which it has been created an equivalent hardware chip targeted to a specific technology (at present one PE is target to FPGA and four PEs are targeted to 0.5 and 0.35 μm technologies), and b) generate the Bit-Vectors for the selected processors;
- The same input data and the same real-time algorithm are applied to the hardware 3D-Flow model and are simulation is performed using the third party tools;
- Bit-Vectors generated by the third party tools using the hardware model are compared with the Bit-Vectors obtained by the previous software simulation;
- Discrepancies are eliminated.

Practically, when a 3D-Flow system is made of thousands of 3D-Flow processors, not all the single processors (or the group of four processors) of the entire system are simulated, but only the processors of the system that execute different algorithms.

Figure 7 shows some of the windows available to the user to create, debug, and monitor a 3D-Flow system with different algorithms of different sizes, and to simulate it before construction.

Figure 6. Interrelation between entities in the Real-Time Design Process

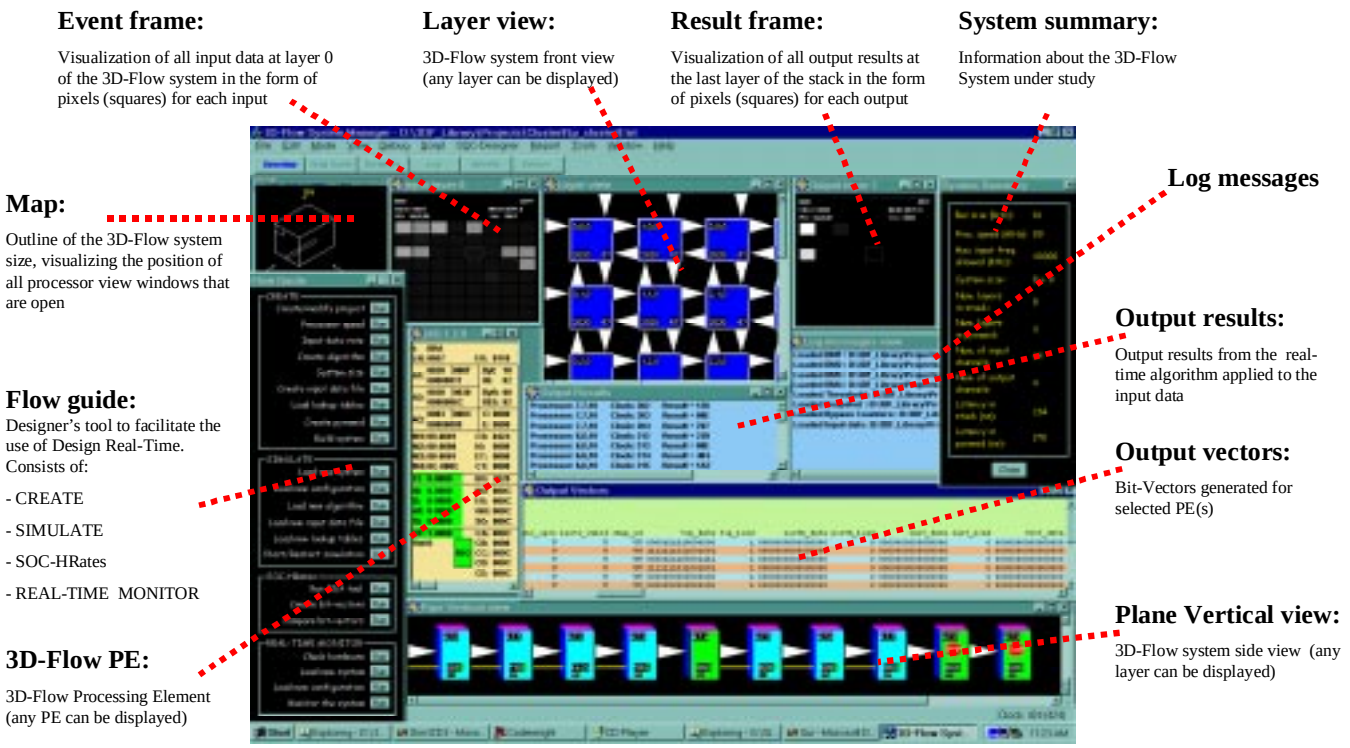


Figure 7. Design Real-Time software tools (Designed for Windows '95, '98 and NT)

5 APPLICATION EXAMPLE: LHCb LEVEL-0 CALORIMETER TRIGGER FE CIRCUIT

5.1 LHCb calorimeter level-0 trigger overview

The front-end chip described in this article was specifically designed to meet the requirements of calorimeter Level-0 front-end electronics of the LHCb[11] experiment; however, it can also be viewed as a more general-purpose design configurable to a) satisfy the requirements of the front-end electronics of other subdetectors of the LHCb experiment, b) meet the requirements of the front-end electronics of other experiments, c) accommodate future changes within the same experiment, and it can also be viewed as a general purpose front-end circuit of the 3D-Flow programmable system for very high-speed real-time applications.

Figure 8 shows the components of the calorimeter Level-0 trigger of the LHCb experiment.

The left column of Figure 8 summarizes the data rates at the different stages of the calorimeter trigger.

Out of 540 GB/s raw data rate from the LHCb subdetectors participating in the calorimeter Level-0 trigger (calculated as the sum of: 12-bit x 6000 EM, 12-bit x 1500 HAD, 1-bit x 6000 preshower, and 2-bit x 6000 PADs, equivalent to 108,000-bit received by the calorimeter Level-0 trigger electronics every 25 ns. Furthermore, 108,000/8-bit x 40 MHz is 540 GB/s), only a subset of raw data is used by the calorimeter Level-0 trigger unit. Thus, the calorimeter Level-0 trigger receives only a data rate of 390 GB/s (calculated as the sum of of: 8-bit x 6000 EM, 8-bit x 1500 HAD, 1-bit x 6000 preshower, and 2-bit x 6000 PADs, equivalent to 108,000-bit received by the calorimeter Level-0 trigger electronics every 25 ns. Furthermore, 78,000/8-bit x 40 MHz is 390 GB/s) from the subdetectors.

The above calculates that 390 GB/s received from the LHCb subdetectors is increased to about 690 GB/s by the front-end electronic circuit because the "Trigger-Tower" is defined as all the signals that are within a view angle taking as a reference point the electromagnetic calorimeter sensor. Consequently, signals from other detectors have to be duplicated by the front-end circuitry. More specifically, the Level-0 trigger algorithm requires that the information of adjacent pads from the M1 detector be considered, and with the hadronic element four times the size of an electromagnetic element, the front-end circuit has to copy the value of one hadronic element into the trigger-tower information of four different electromagnetic elements. (The 690 GB/s is calculated as the sum of 8-bit EM, 8-bit HAD, 1-bit preshower, and 6-bit PADs, that also have the information of the adjacent PADs. The resulting sum of 23-bit is the trigger word received by each 3D-Flow processor in the first layer of the stack as shown in Figure 8. Furthermore, 23-bit x 6000 = 138,000-bit received by the 3D-Flow system every 25 ns, is equivalent to 690 GB/s).

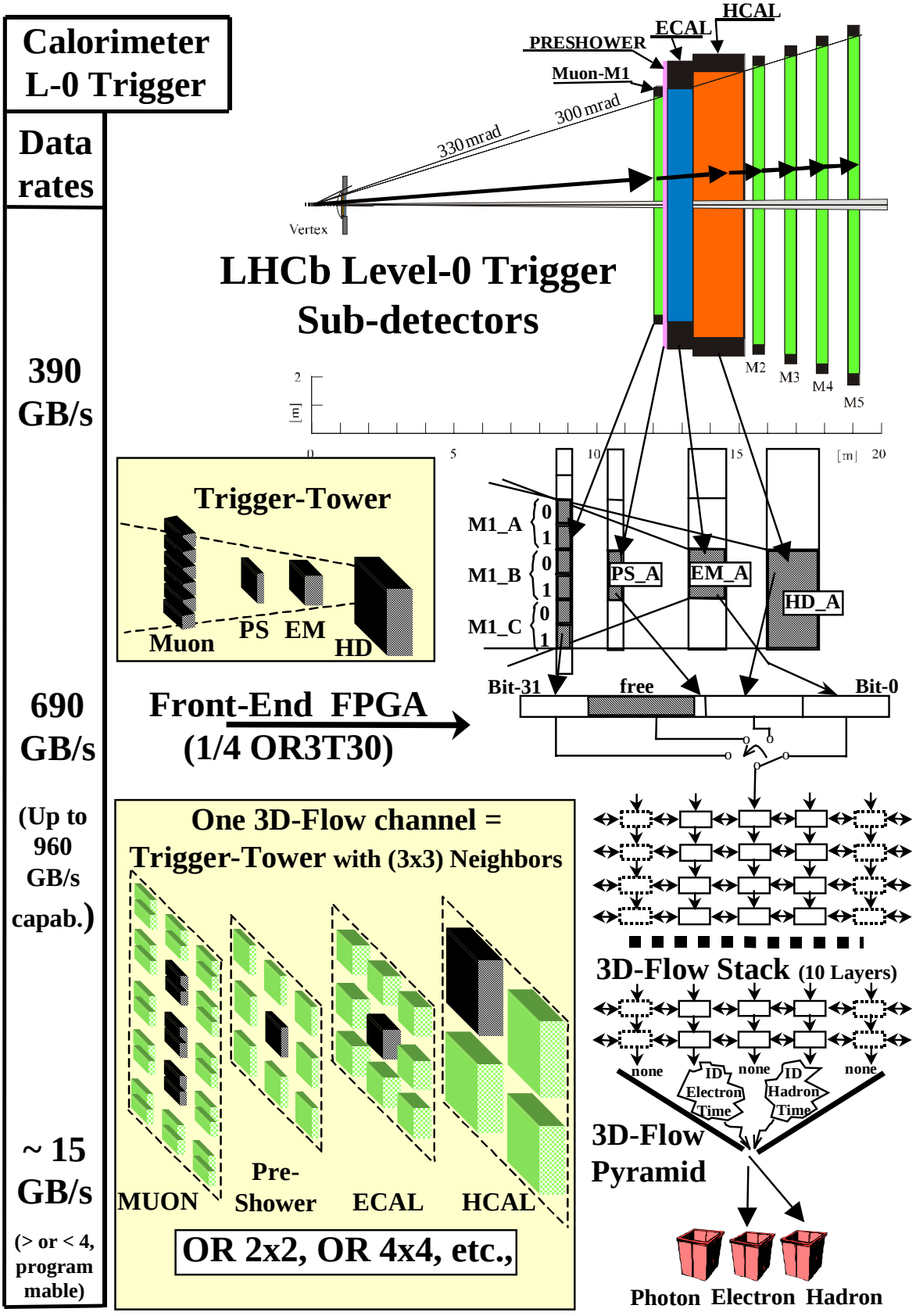


Figure 8. LHCb calorimeter Level-0 trigger layout.

The front-end design, together with the 3D-Flow design, allows the user to change a) the trigger tower segmentation, b) the trigger word definition, and c) the real-time trigger algorithm at any time after construction, provided that the total input data rate for 6000 trigger channels does not exceed 960 GB/s and that the trigger algorithm does not exceed 20 steps (considering that each 3D-Flow processor can execute up to 26 operations per step, inclusive of compare, ranging, finding local maxima, and efficient data exchange with neighboring channels). The 960 GB/s is calculated as 32-bit x 6000 = 192,000-bit received by the 3D-Flow system every 25 ns, which is equivalent to 960 GB/s.

Since the LHCb experiment expects an average of one candidate per 20 bunch crossing with at the most 4 candidates in a single bunch crossing, the provided output bandwidth from the 3D-Flow Level-0 trigger system (as it was described in the previous article[5]), to the global level-0 trigger of 15 GB/s (providing a maximum acceptance rate at this stage of about 2 billion candidates per second with 64-bit of information per candidate) provides a very safe margin to avoid system saturation.

At a rate of 2 billion candidates per second at the output stage of the 3D-Flow system the use is foreseen of only one LVDS cable per digital processing board for a total of 96 cables to the global level-0 trigger. The bandwidth can, however, be reduced by the user by applying a trigger algorithm with a higher filtering function at this stage.

The center of Figure 8 shows the components of a "Trigger-Tower." Starting from the right and moving toward the left we have: 8-bit from the electromagnetic calorimeter, 8-bit from the hadronic calorimeter, 1-bit from the preshower, 9-bit free, and 6-bit from the PADs. Further on the left of the figure there is a 3D representation of the elements of a trigger-tower.

The bottom-left of Figure 8 shows the 3D representation of the elements of a trigger-tower with all the adjacent elements used by the 3x3 level-0 trigger algorithm.

The information of the elements shown in the bottom-left part of Figure 8 will be available on each 3D-Flow processor after acquisition and data exchange with the neighbors. Equally, it is possible to implement the trigger algorithm with 2x2, or 4x4, or 5x5, etc. data exchange and clustering.

The bottom-right section of Figure 8 shows the 3D-Flow system from the first layer of the stack which is connected to the front-end chip that receives the data from the detector, down to the last layer connected to the pyramid that is exploiting the function of channel reduction.

5.2 A single type of front-end circuit for the entire LHCb system

The complete detailed study for the overall level-0 front-end electronics has been performed. Detailed circuits in VHDL source code are provided in the appendix and, together with the files for the ORCA OR3T30 FPGA, the testbenches for easy verification of the correlation between signals and their timing performance are provided on the website[6]:

For the mixed-signal processing board, after the task of amplification and conversion of analog signals to digital by means of an ADC such as Analog Devices AD9042[15] converting to 12-bit at 40 MHz, all digital information is sent to 16 FPGAs. Each FPGA can implement all functions described below for four channels out of 64 channels in a board. The study has been made referring to using the component from Lucent Technologies ORCA OR3T30[31] with 256-pin BGA with a package dimension of 27 mm x 27 mm.

The digital information relative to four trigger towers is sent to the input of one FPGA (see Figure 9). If a PAD from the muon station is used by more than one trigger tower, it will be sent to all the appropriate FPGA units.

All data are strobed into a register inside the FPGA at the same time; however, the present design allows for the possibility that data from different detectors (e.g. muon Pad vs. ECAL) be out of phase by one or two bunch crossings.

Next, a delay can be inserted from 0 to 2 clock counts at each bit received at the input of the FPGA. This function, called "variable delay," is shown in Figure 9, while the VHDL code is described in Section 5.2.1.

For each channel we have: 12-bit information from the electromagnetic calorimeter, 12-bit information from the hadronic calorimeter, 1-bit information from the preshower, and 2-bit information from the muon pad chamber, for a total of 27-bits per input-channel.

The above 27-bits input channels are stored into a level-0 pipeline buffer of 128 clocks (or bunch crossings. The depth of the pipeline buffer can be changed as shown in Table 1), while the trigger electronics verifies whether the event should be retained or rejected. This function is called "128 pipeline." (See Section 5.2.3 for the description of the VHDL code).

When an event is accepted, the global level-0 trigger decision unit sends a signal to all the pipeline buffers to move the accepted bit (corresponding to an accepted event) to a derandomizing FIFO buffer (See Section 5.2.4 for VHDL code description). This function is called "FIFO." For each channel, the full information relative to the accepted event will be stored into a FIFO. Even though the entire process is synchronous, it is safer to extend the width of the FIFO in each FPGA. At present, 8-bit has been reserved for the time-stamp bunch-crossing counter; however, it is defined in the configuration file shown in Table 1 and can be changed at any time.

Each FPGA handles the information of four trigger-tower channels, memorizes the information for 128 clock cycles, stores the information relative to the accepted events (at an average of 1 MHz) into a 32-bit deep (this parameter can be changed at any time as shown in Table 1), 80-bit wide FIFO. The width of the output FIFO in each FPGA is calculated as follows: 4 x 12-bit electromagnetic, 12-bit hadronic, 4 x 1-bit preshower, 4 x 2-bit pads of muon stations, and 8-bit time-stamp from a bunch crossing counter that will allow verification of partial event information at different stages of the data transmission (optical

fibers, deserializer, etc.). Thus for each accepted event, each FPGA will send 80-bit through the serializer and the optical fiber to the upper level trigger and DAQ.

A strobe signal received from the upper level decision units and DAQ (called EnOutData in Figure 9 and Table 4) will read all output FIFOs from the FPGAs at an estimated rate of 1 MHz.

Besides the synchronization, 128 pipeline storage, and derandomization of the full data path, it is also necessary to generate the trigger word to be sent to the 3D-Flow trigger processor (see Section 5.2.2 for VHDL code description). In order to save some 3D-Flow bit-manipulation instruction, the function of formatting the input trigger word can also be implemented in the FPGA.

At present, the input trigger word is defined as:

- 8-bit electromagnetic calorimeter
- 8-bit hadronic calorimeter
- 1-bit preshower
- 6-bit Pad from muon station M1.

The format could be redefined at any time by reprogramming the FPGA through the RS-422 link.

In order to provide the reader with the degree of the limits of the RTL synthesis tools for FPGA with respect to an optimized "hard macro," both approaches were followed, and results compared.

- A set of synthesizable VHDL source files are provided in the web and in the appendix. The user can verify, by using the best synthesis tools, that the Front-end circuit described in this article requires an FPGA of the size of OR3T80 with 484 PFUs (Programmable Functional Unit) in order to be able to accommodate the design.
- A "hard macro" implementation, instead, of the same design optimized for the ORCA PFU basic elements can be accommodated into a OR3T30 with 196 PFUs, which costs about five times less than an OR3T80.

The great difference in the result of the two approaches indicates that there is still room for great improvement for synthesis tools for FPGA. Instead, the synthesis tools for gate array or standard cell are very well optimized since the basic element of a gate or a cell is very small compared to the PFU of a FPGA.

Following are the details of the functions listed above written in "generic VHDL" code suitable to several FPGAs or ASICs.

Furthermore, a breakdown of the above functions mapped to the ORCA[31] Programmable Function Units (PFUs) is also provided.

The bottom of Figure 9 shows the detail schematic of the front-end circuit receiving signals from 4 trigger-tower, mapped to the OR3T30 FPGA., The top of Figure 9 shows in 3D the detector elements that are sending the signals to the front-end FPGA.

Table 4 lists the VHDL code of the same circuit shown in Figure 9. The coding style is technology-independent, synchronous design, generic HDL. The complete VHDL code is attached in the Appendix.

The use of the same signal names in the Figures and the VHDL coding should help to correlate the functions from the detector physical layout, to the schematic, to the VHDL code.

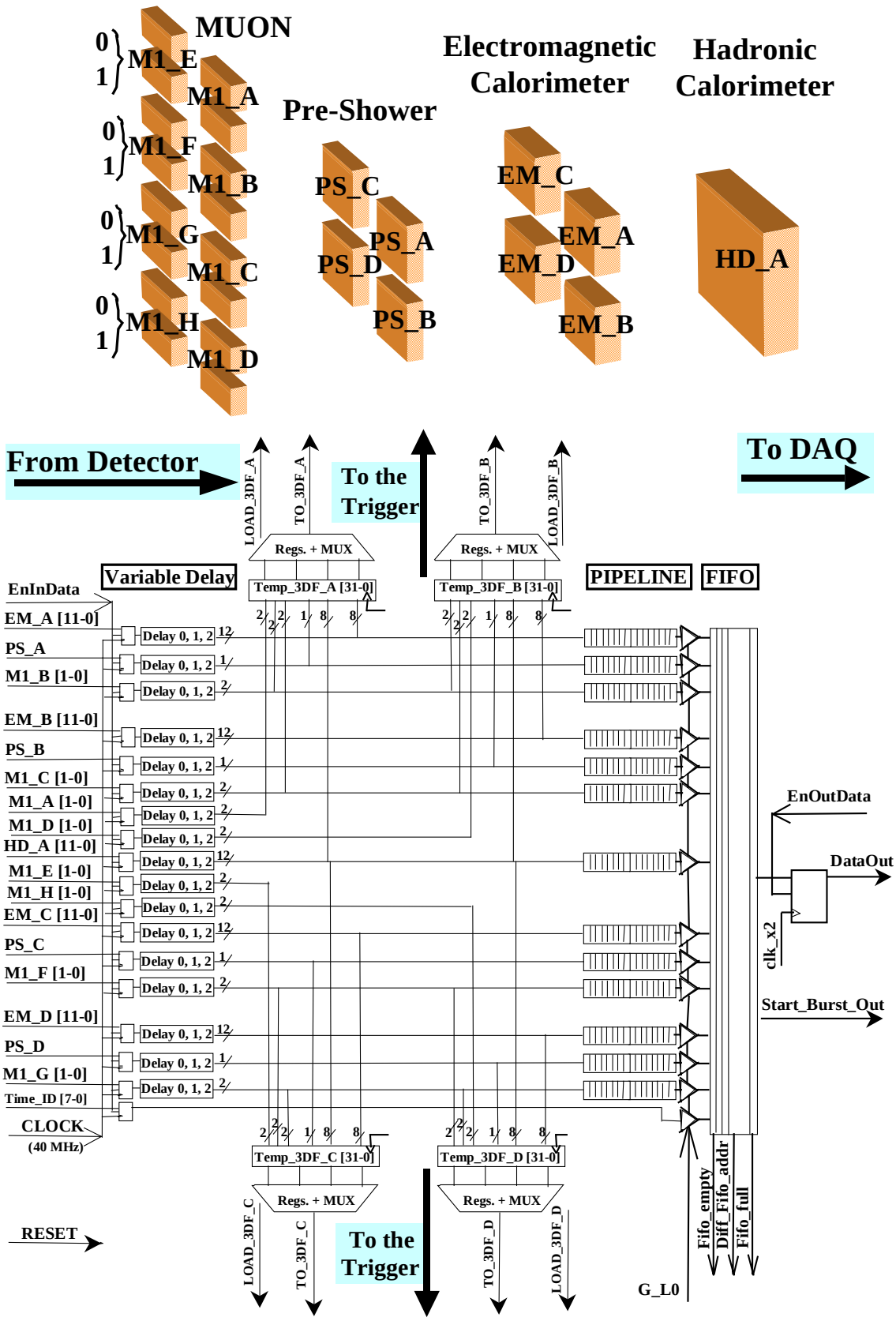

MUON
M1_E
M1_A
M1_F
M1_B
M1_G
M1_C
M1_H
M1_D
Pre-Shower
PS_C
PS_A
PS_D
PS_B
Electromagnetic Calorimeter
EM_C
EM_A
EM_D
EM_B
Hadronic Calorimeter
HD_A
From Detector
To the Trigger
To DAQ
LOAD_3DF_A
TO_3DF_A
TO_3DF_B
LOAD_3DF_B
Regs. + MUX
Temp_3DF_A [31-0]
Temp_3DF_B [31-0]
Variable Delay
PIPELINE
FIFO
EnInData
EM_A [11-0]
PS_A
M1_B [1-0]
EM_B [11-0]
PS_B
M1_C [1-0]
M1_A [1-0]
M1_D [1-0]
HD_A [11-0]
M1_E [1-0]
M1_H [1-0]
EM_C [11-0]
PS_C
M1_F [1-0]
EM_D [11-0]
PS_D
M1_G [1-0]
Time_ID [7-0]
CLOCK
(40 MHz)
Delay 0, 1, 2
EnOutData
DataOut
clk_x2
Start_Burst_Out
Temp_3DF_C [31-0]
Temp_3DF_D [31-0]
RESET
To the Trigger
LOAD_3DF_C
TO_3DF_C
TO_3DF_D
LOAD_3DF_D
Fifo_empty
Diff_Fifo_addr
Fifo_full
G_L0

Figure 9. (Top of Figure) Physical layout of detector elements sending signals to one FPGA front-end chip. (Bottom of Figure) Schematic of the front-end electronics of 4 Trigger-Towers mapped to one FPGA.

Table 4. VHDL code of the inputs/outputs of the front-end chip mapped to one FPGA.

```
--------------------------------------------------------------------------
--                                                                      --
-- Copyright (c) 1999 by 3D-Computing, Inc.                             --
--                                          All rights reserved.        --
--  Author       : Dario Crosetto                                       --
--                                                                      --
-- This source file is FREE for Universities, National Labs and         --
-- International Labs of non-profit organizations provided that the     --
-- above statements are not removed from the file,                      --
-- that the revision history is updated if changes are introduced, and  --
-- that any derivative work contains the entire above-mentioned notice. --
--                                                                      --
--  Package name : FE_top.vhd
--
--  Project   : Front-End Electronics Logic
--  Purpose   : This file implements the front-end signals synchronization,
--              pipelining, derandomizing, trigger word formatter.
--              The code is for four trigger channels
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
--------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
LIBRARY work;
USE work.FE_config.ALL;

-----------------------------------------------------------------
--Entity Definition
-----------------------------------------------------------------

ENTITY FE_top IS
   PORT (      clock, reset   : IN STD_LOGIC;
        EM_A                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_B                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_C                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_D                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        HD_A                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        PS_A                  : IN std_logic;
        PS_B                  : IN std_logic;
        PS_C                  : IN std_logic;
        PS_D                  : IN std_logic;
        M1_A                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_B                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_C                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_D                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_E                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_F                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_G                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_H                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        Time_ID               : IN STD_LOGIC_VECTOR(Time_ID_width -1 DOWNTO 0);
        G_L0                  : IN std_logic;
        EnInData             : IN std_logic;
        EnOutData             : IN std_logic;
        clk_x2                : IN STD_LOGIC; -- Replaced by the internal PLL –
        clk_x4                : IN STD_LOGIC; -- Replaced by the internal PLL –

        fifo_empty            : OUT std_logic;
        fifo_full             : OUT std_logic;
        diff_fifo_addr        : OUT std_logic_vector(fifo_depth - 1 downto 0);
        LOAD_3DF_A            : OUT std_logic;
        TO_3DF_A              : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_B            : OUT std_logic;
        TO_3DF_B              : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_C            : OUT std_logic;
        TO_3DF_C              : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_D            : OUT std_logic;
        TO_3DF_D              : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        DataOut               : OUT std_logic;
        St_Burst              : OUT std_logic
        );
END FE_top;
```

5.2.1 Coding of the Input-Synchronizer module (VHDL)

Insert the header statement of Table 1, or Table 4 in case this code needs to be used or copied

The input synchronizer module registers all inputs and insert at each channel with the delay selected in the configuration file of Table 1.

There are three registers for each channel (or trigger tower), channel A, channel B, channel C, and channel D.

- First all registers are reset to zeros when the RESET signal is zero.
- Next, at the clock rising edge, the value of dly1_xx_x is copied into the register dly2_xx_x; the values of xx_x_clkd are copied into the register dly1_xx_x, the value of xx_x is copied in xx_x_clkd.

```
ELSIF (clock'EVENT AND clock = '1') THEN
      EM_A_clkd <= EM_A;
      EM_B_clkd <= EM_B;
      EM_C_clkd <= EM_C;
      EM_D_clkd <= EM_D;
      dly1_EM_A <= EM_A_clkd;
      dly1_EM_B <= EM_B_clkd;
      dly1_EM_C <= EM_C_clkd;
      dly1_EM_D <= EM_D_clkd;
      dly2_EM_A <= dly1_EM_A;
      dly2_EM_B <= dly1_EM_B;
      dly2_EM_C <= dly1_EM_C;
      dly2_EM_D <= dly1_EM_D;
```

- Change delay values based on detector, and/or electronics, and/or cable length

```
Select_Del_EM <= EM_del;
Select_Del_HD <= HD_del;
Select_Del_PS <= PS_del;
Select_Del_M1 <= M1_del;
```

- This synchronizes EM signals. EM_xx signal will get the value of one of the three registers conforming the selection made in the previous statement.

```
EM_AS  <= dly2_EM_A WHEN (Select_Del_EM = "10")
          ELSE dly1_EM_A WHEN (Select_Del_EM = "01")
          ELSE EM_A_clkd;
EM_BS  <= dly2_EM_B WHEN (Select_Del_EM = "10")
          ELSE dly1_EM_B WHEN (Select_Del_EM = "01")
          ELSE EM_B_clkd;
EM_CS  <= dly2_EM_C WHEN (Select_Del_EM = "10")
          ELSE dly1_EM_C WHEN (Select_Del_EM = "01")
          ELSE EM_C_clkd;
EM_DS  <= dly2_EM_D WHEN (Select_Del_EM = "10")
          ELSE dly1_EM_D WHEN (Select_Del_EM = "01")
          ELSE EM_D_clkd;
```

5.2.2 Coding of the Trigger-Word-Formatter module (VHDL)

Insert the header statement of Table 1, or Table 4 in case this code needs to be used or copied

The Trigger-Word-Formatter module builds four trigger words to be sent to four 3D-Flow processors by extracting the information from synchronized raw data. Any combination of bits available in the FPGA can be used, the same signal can be sent to several 3D-Flow processors, and the format can be changed at a later time.

The Load to the 3D-Flow processor signal is synchronized with the clock. The 32-bit trigger word is clocked out to the 3D-Flow processor at twice the speed of the standard clock (40 MHz) of the entire design.

The implementation for FPGA OR3T30 uses an internal PLL (Phase-Locked Loop) at 80 MHz. The circuit is different from the ASIC implementation. The FPGA implementation uses a different circuit made of the trigger-word formatter 32-bit register, connected to two 8-bit multiplexers 2:1, connected to two 8-bit registers, connected to one 8-bit multiplexer 2:1. The first set of multiplexers uses the clock at the select input, the second set of 8-bit registers uses clock_x2_delayed (@80 MHz) as strobe, and the last multiplexer is using clock_x2_delayed. The delays are necessary, and consequently the circuit is not elegant as it should be due to the limitation of the current FPGAs that cannot have a PLL @ 160 MHz. Future FPGAs will have PLL @ 160 MHz, and thus the circuit could be of the same type as the one for ASIC (which uses a counter @ 160 MHz to select the multiplexer) reported in the appendix.

- First, the trigger word is extracted from the synchronized raw data in the following manner (code shows only one channel out of four):

```
TEMP_3DF_A <= EM_AS(EM_trig_width -1 DOWNTO 0) &
```

```
                    HD_AS(HA_trig_width -1 DOWNTO 0)
                    & PS_AS & "000000000" & M1_AS(M1_trig_width - 1 DOWNTO 0)
                    & M1_BS(M1_trig_width - 1 DOWNTO 0)
                    & M1_CS(M1_trig_width - 1 DOWNTO 0);
```

- A counter @ 160 MHz is used to select data of the multiplexer that sends them to the 3D-Flow processor.

```
MUX_CNT: PROCESS (int_clk_x4, reset)
  BEGIN
       IF (reset = '0') THEN
              Mux_Count <= (others => '0');
       ELSIF (int_clk_x4'EVENT AND int_clk_x4 = '1') THEN
         IF (EnInData_delay = '1') THEN
       Mux_Count <= Mux_Count + 1;
         ELSE
         END IF;
       ELSE
       END IF;
END PROCESS MUX_CNT;
```

- The 32-bit of the trigger word is sent out, in four steps through a 8-bit data bus, to the 3D-Flow trigger processor (code shows only one channel out of four).

```
-- clocking the trigger-word to the trigger decision 3D-Flow processor.
CLK_TRI: PROCESS (int_clk_x4, reset)
         BEGIN
       IF (reset = '0') THEN
              TO_3DF_A  <= (others => '0');
       ELSIF (int_clk_x4'EVENT AND int_clk_x4 = '1') THEN
         IF (EnInData_delay = '1') THEN
              CASE Mux_count IS
WHEN "00" => TO_3DF_A <= TEMP_3DF_A(4 * Width_To3DF - 1 DOWNTO 3 * Width_To3DF);
WHEN "01" => TO_3DF_A <= TEMP_3DF_A(3 * Width_To3DF - 1 DOWNTO 2 * Width_To3DF);
WHEN "10" => TO_3DF_A <= TEMP_3DF_A(2 * Width_To3DF - 1 DOWNTO Width_To3DF);
WHEN "11" => TO_3DF_A <= TEMP_3DF_A(Width_To3DF - 1 DOWNTO 0);
WHEN OTHERS => NULL;
              END CASE;
```

5.2.3 Coding the Pipeline Buffer module (VHDL)

Insert the header statement of Table 1, or Table 4 in case this code needs to be used or copied

- At the clock rising edge, a new synchronized data is copied into the pipeline buffer at the LSB (Least Significant Bit) position, and the entire pipeline buffer is shifted one position to the left.

```
                    ELSIF (clock'EVENT AND clock = '1') THEN
PIPE_EM_A0(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A0(PIPE_depth - 2 DOWNTO 0) & EM_AS(0);
```

- The MSB (Most Significant Bit) of the pipeline buffer is copied into the 80-bit wide register “TO_IN_FIFO.” (Code is shown only for the first 12-bit channels out of 72 channels, last 8-bit are the value of the “Time_ID” counter)

```
TO_IN_FIFO(fifo_width - 1 DOWNTO 0) <=
            PIPE_EM_A0(127) & PIPE_EM_A1(127) & PIPE_EM_A2(127) & PIPE_EM_A3(127) &
            PIPE_EM_A4(127) & PIPE_EM_A5(127) & PIPE_EM_A6(127) & PIPE_EM_A7(127) &
            PIPE_EM_A8(127) & PIPE_EM_A9(127) & PIPE_EM_A10(127) & PIPE_EM_A11(127) &
```

5.2.4 Coding the FIFO and the output Serializer (VHDL)

Insert the header statement of Table 1, or Table 4 in case this code needs to be used or copied

- This code implements the FIFO read pointer. At the clock rising edge, if the FIFO is not empty and there is a request to read one data, the read pointer is incremented. (The write pointer is similar, but uses the Global Trigger signal “G_L0” as a condition to increment the write pointer).

```
-- FIFO read address
PROCESS (reset,clock,EnOutData)
BEGIN
  IF (reset = '0') THEN
     int_fifo_rdaddr <= (others => '0');

    ELSIF (clock'Event AND clock = '1') THEN

      IF EnOutData = '1' AND int_fifo_empty = '0' THEN
          int_fifo_rdaddr <= int_fifo_rdaddr + 1;
```

```
    END IF;
  END IF;
END PROCESS;
```

- The following code implements the update of the FIFO flags. A counter keeps track of how many data are present in the FIFO at any time. The counter is incremented when there is a write operation and the FIFO is not full, while it is decremented when there is a read operation and the FIFO is not empty.

```
-- fifo full/empty logic
PROCESS (clock, reset)
BEGIN
  IF reset = '0' THEN
     int_fifo_cnt <=  (OTHERS => '0');
    ELSIF (clock'EVENT AND clock = '1') THEN
      IF G_L0 = '1' AND int_fifo_full = '0' THEN
            int_fifo_cnt <= int_fifo_cnt + 1;
         END IF;
           ELSE
         IF EnOutData = '1' AND int_fifo_empty = '0' THEN
           int_fifo_cnt <= int_fifo_cnt - '1';
      END IF;
  END IF;
END PROCESS;
```

- This code writes a new data into the FIFO when a Global Trigger Accept signal is received and the FIFO is not full.

```
comb_proc: PROCESS (G_L0,TO_IN_FIFO   , int_fifo_wraddr)
BEGIN
  IF (reset = '0') THEN
           next_file <=  (OTHERS => (OTHERS => '0'));
      ELSIF   (wr_en = '1' AND G_L0 = '1' AND int_fifo_full = '0') THEN
              next_file(CONV_INTEGER(int_fifo_wraddr))   <= TO_IN_FIFO;
  END IF;
END PROCESS;
```

- This code sends data out of the FIFO serially from DataOut pin.

```
DataOut <= temp_out(fifo_width - 1);
```

- The code sends “St_burst_out” signal synchronized with first bit of output string of 80 bits.

```
PROCESS (reset,int_clk_x2,EnOutData,int_fifo_empty)
BEGIN
  IF (reset = '0') THEN
         St_burst <= '0';

    ELSIF (int_clk_x2'Event AND int_clk_x2 = '1') THEN

      IF EnOutData = '1' AND int_fifo_empty = '0' THEN
            St_burst <= '1';
         ELSE
            St_burst <= '0';
      END IF;
  END IF;
END PROCESS;
```

- This code reads out values from the FIFO when receiving "EnOutData" signal from the Highr-Level Trigger.
- (In more details) loads “temp_out” with FIFO value pointed by read_fifo_address ELSE load “temp_out” with shifted value.

```
PROCESS (reset,int_clk_x2,EnOutData)  -- MSB first shift register.
BEGIN
  IF (reset = '0') THEN
     temp_out <= (others => '0');

  ELSIF (int_clk_x2'EVENT AND int_clk_x2 = '1') THEN
    IF (EnOutData = '1' AND int_fifo_empty = '0') THEN
         temp_out <= next_file(CONV_INTEGER(int_fifo_rdaddr));
      ELSE

         temp_out <= temp_out(fifo_width - 2 downto 0) & '0';
    END IF;
  END IF;
END PROCESS;
```

- This signal assignment makes the FIFO Flags status available at the pin of the chip.

```
diff_fifo_addr <= int_fifo_cnt(fifo_depth - 1 DOWNTO 0);
int_fifo_full <= int_fifo_cnt(fifo_depth);
int_fifo_empty <= '1' WHEN int_fifo_cnt = "000000" ELSE '0';
```

5.2.5 Mapping the Level-0 front-end circuits into ORCA OR3T30 FPGA

The above "generic VHDL" style suitable for any FPGA or ASIC ASIC (providing that, until next year, FPGA's internal PLL be used instead of external high-frequency clocks), if kept as is, will be technology independent. The synthesis tools of different vendors will translate into gates for their technology. However, the user may further improve the layout for a particular technology in order to best optimize the silicon. (This effort, is not convenient for large designs such as the 3D-Flow chip because of the portability and the fact that it is more important to have a technology-independent design. In the long run, given the rapid advances in technology, it will also be cost effective, eliminating the need to spend many hours to save a few gates in an environment where the gates cost less every year.)

Since this is a small design, and the architecture of the ORCA Programmable Function Unit was known, the exercise of mapping the function into logic was not very complex.

The basic elements of the ORCA architecture used to implement the above functions are: a Programmable Logic Cell (PLCs), and Programmable Input/Output Cells (PICs). An array of PLCs is surrounded by PICs. Each PLC contains a Programmable Function Unit (PFU) containing 8 registers, a Supplemental Logic and Interconnect Cell (SLIC), local routing resources, and configuration RAM (used in our case to implement the 128 pipeline buffer).

Following is the resulting optimization, calculated for one trigger channel out of four that can be implemented in an OR3T30 FPGA device (which the synthesis tools from ORCA may not recognize from the above code).

Table 5. Mapping the Level-0 front-end circuit into ORCA OR3T30 FPGA.

Function	# of PFU	Comment
Input register	0	Use PIC registers
Variable delay	20	1 PFU per 4 input bits
3DF interface	32	
128-clock pipeline	80	1 per input bit
Counters (for 128 clock pipeline)	9	
32x80 FIFO	20	4 bit per PFU (use dual-port memory)
80-bits Parallel In, Serial Out regs	10	
5-bit read pointer	4	For FIFO read pointer
5-bit write pointer	4	For FIFO write pointer
Miscellaneous	3	

The total number of PFUs required is 182. The OR3T30 contains 196 PFU.

5.3 Front-end chip test results and timing information

Preliminary simulation has been performed with a preliminary test bench program reported in the Appendix and available on the web. Results meet the functional specifications. As can be determined from the timing report file available on the web, the timing of 80 MHz for the FPGA version mapped to OR3T30, with speed grade 7 does not meet the requirements for less than 2 ns. This problem will undoubtedly be solved with new release of ORCA Foundry tools (Currently the Beta version 9_35.97 was used).

5.3.1 Timing diagram of input signals synchronization

The signals on the top left of the Figure 10 are the inputs from the detector. The signals of the electromagnetic (e.g., indicated by em_a, em_b, ect.) and hadronic calorimeter are captured safely half the clock period (during the falling edge of the clock). Since the delay for the electromagnetic signals was selected to be zero in Table 1, at the first clock rising edge the signals are synchronized (e.g., em_as, em_bs, etc., shown in the center of Figure 11). For the signals from the muon detector, however, for which a delay of one was selected, the synchronization is done (see m1_as, m1_bs, etc., see at the bottom of Figure 10) at the second rising edge of the clock

.

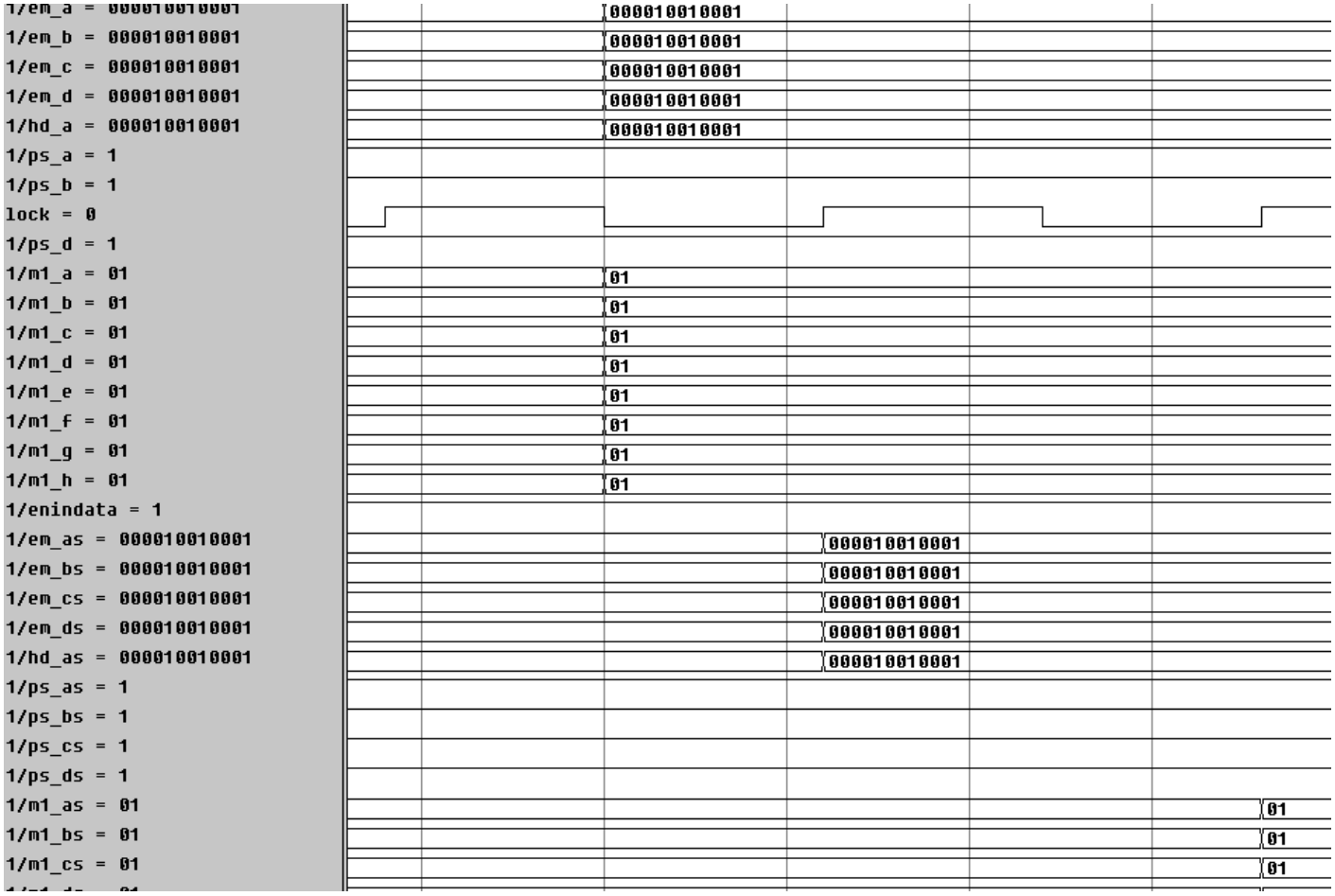


Figure 10. Timing diagram of input signals synchronization

5.3.2 Timing diagram of the 8-bit data path to the 3D-Flow trigger system

Figure 11 shows the timing diagram of the 8-bit data path to the 3D-Flow trigger system. The signals “to_3df_a,” is sending to the output in four steps (each being half the cycle of clk_x2 that is @ 80 MHz) the 32-bit value of the trigger-word register “temp_3df_a.”

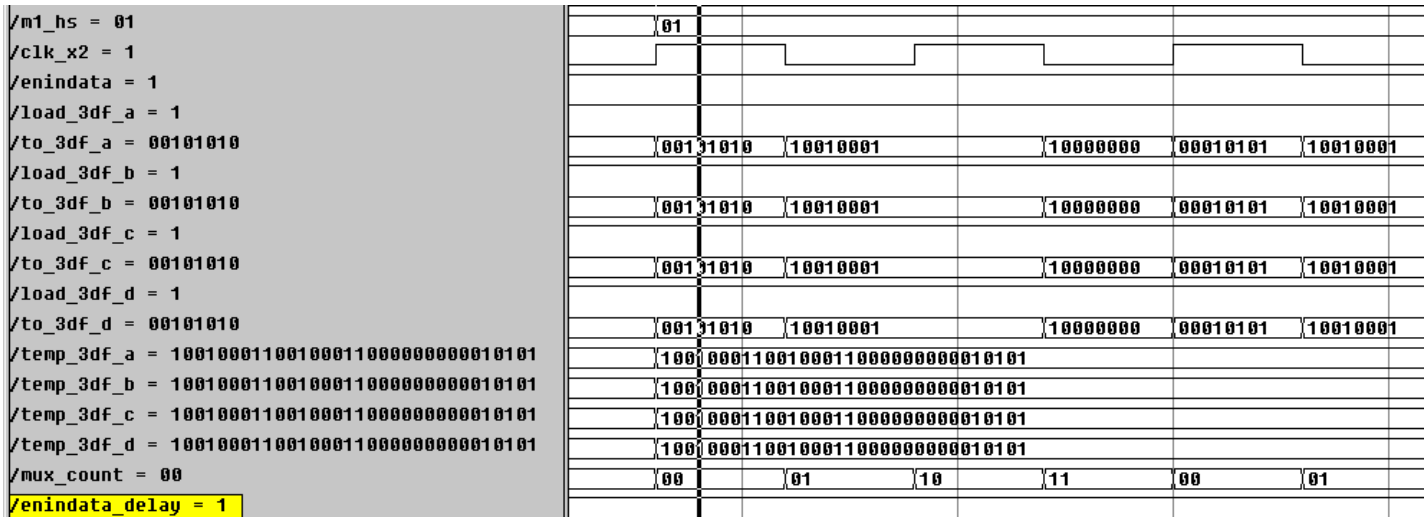


Figure 11. Timing diagram of the 8-bit data path to the 3D-Flow trigger.

5.3.3 Timing diagram of the pipeline buffer

Figure 12 shows the timing diagram of the pipeline buffer. The top left part of Figure 12 shows the pipeline buffer from pipe_hda5, to pipe_m1_g while they are being filled by data received from the sensors. Since the data of the test bench (as shown in the appendix) do not change, each pipeline buffer will be filled with the same value as shown in the right part of Figure 12.

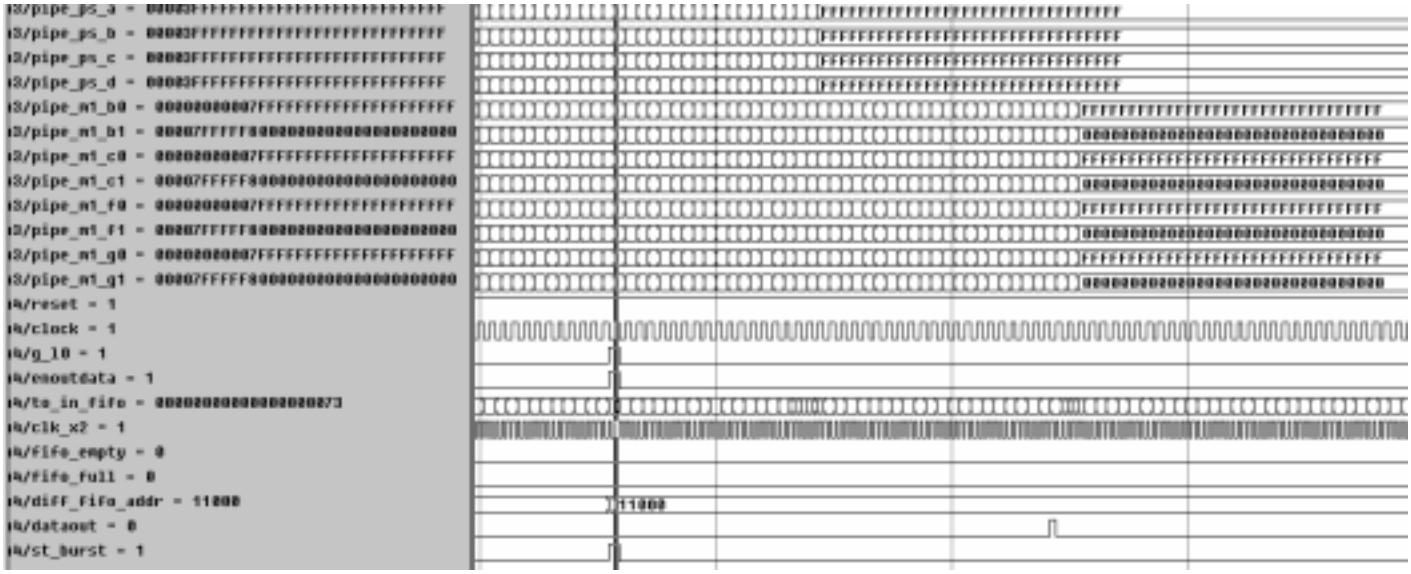


Figure 12. Timing diagram of the pipeline buffer

5.3.4 Timing diagram of the out FIFO and of the output Serializer

Figure 13 shows the timing diagram of out FIFO and the output serializer. The signal at the bottom of Figure 13 shows the content of the out FIFO, the signal “st_burst” when equal to one indicates the start of the serialization of the 80-bit output word on pin “dataout” at the clock frequency of clk_x2 (@80 MHz)

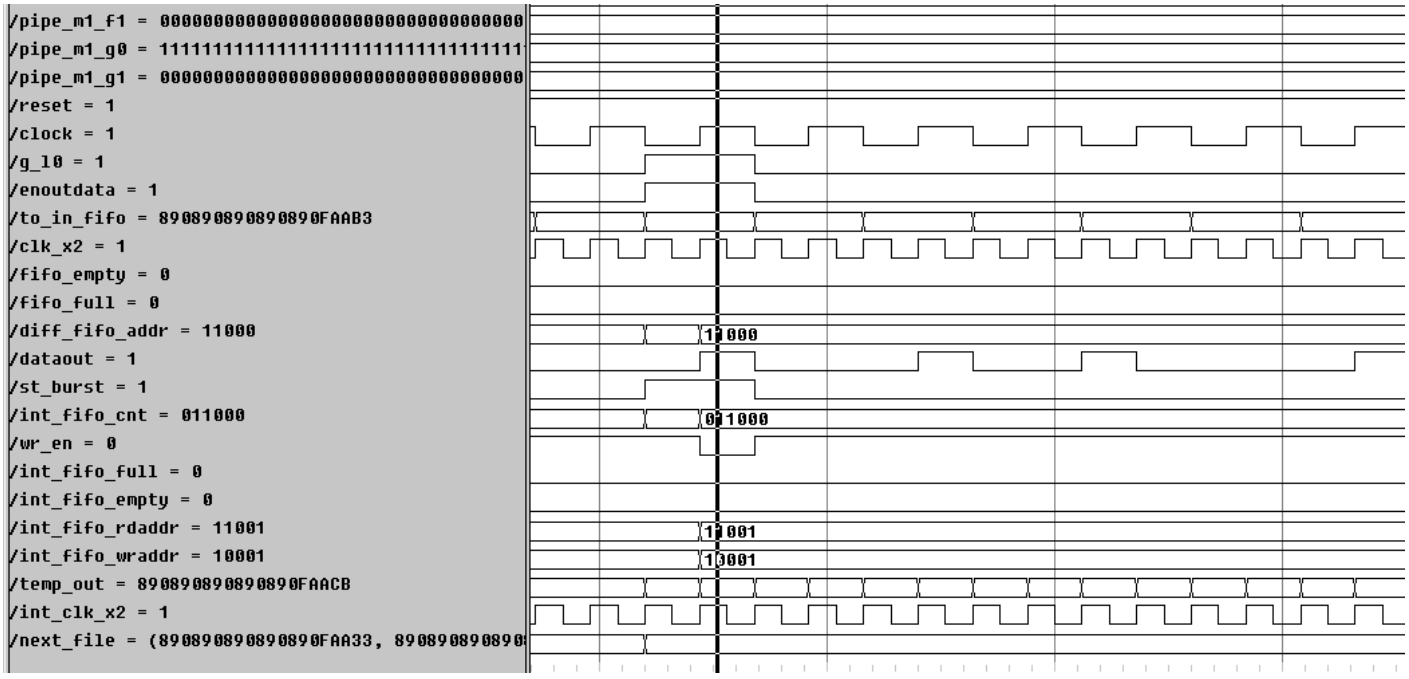


Figure 13. Timing diagram of the out FIFO and of the output Serializer

5.4 Description of all files of the entire project for generic-HDL and OR3T30 implementation

Table 6 shows how the files of the front-end 3D-Flow project are organized and how the can be accessed on the CERN website: http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm.

The files of the 3D-Flow front-end project can be downloaded by the users. Several third party tools for simulation should accept this code, in the case you encounter somme problems in a specific simulator, please let me know at crosetto@bonner-ibm9.rice.edu. (For the FPGA implementation, the ORCA Foundry tools version 95_95.35 has been used)

As it was described before in Section xx the 3D-Flow front-end project has been targeted to a generic-HDL and to a specific FPGA ORCA OR3T30. Files relative to both [[Mies, I will finish this paragraph later, now Joe is here]]

Table 6. 3D-flow front-end chip file organization and webaddres for downloading.

Test Bench of the Front-End chip: FE_3D-Flow

Behave.vhd	Version	Test description	Status	Date	Responsible
Script_tb_g1.do	Ver. 1	Script to compile all VHDL modules including Config+Top+Testbench	working	23/4/99	Dario Crosetto
FE_tbench_g2	Ver. 1	Preliminary test of the FE_3D-Flow chip using the Generic HDL, Technology independent code.	working	23/4/99	Dario Crosetto

Front-End Chip: FE_3D-Flow (Generic HDL code, synthesizable, Technology-independent)

Behave [.vhd]	Version	Test description	Status	Date	Responsible
Script_ch_g1.do	Ver. 1	Script to compile all VHDL modules including Config+Top	working	23/4/99	Dario Crosetto
FE_Top	Ver. 1	Top Level of the chip	working	23/4/99	Dario Crosetto
FE_Config	Ver. 1	Configuration Parameters	working	23/4/99	Dario Crosetto
FE_syncinput	Ver. 1	Module to synchronize input signals from detector	working	23/4/99	Dario Crosetto
FE_trig_formatter	Ver. 1	Module to format the trigger word	working	23/4/99	Dario Crosetto
FE_pipeline	Ver. 1	Module of the pipeline buffer	working	23/4/99	Dario Crosetto
FE_fifo	Ver. 1	Module of the out-FIFO and of the output Serializer	working	23/4/99	Dario Crosetto

Test Bench of the Front-End chip: FE_3D-Flow (for ORCA, OR3T30 implementation)

Behave.vhd	Version	Test description	Status	Date	Responsible
Script_tb_h1.do	Ver. 1	Script to compile all VHDL modules including Config+Top+Testbench	OK	23/4/99	Dario Crosetto
FE_tbench_h2	Ver. 1	Preliminary test of the FE_3D-Flow chip using the Back-annotated VHDL from ORCA Foundry Tools.	under test	23/4/99	Dario Crosetto

Chip: FE_3D-Flow (implemented with ORCA, OR3T30 FPGA)

Chip name	Version	Netlist [.edn]	Summary [.twr]	Back-anno [.vhd]	Back-anno timing[.sdf]	Pinout [.pad]	Bitstream [.bit]	Vendor	Status	Date	Responsible
FE_3D-Flow	Ver. 1	OFE1.edn	OFE1.twr	OFE1.vhd	OFE1.sdf	OEF1.pad	OFE1.bit	ORCA OR3T30	under test	24/4/99	Dario Crosetto

6 FROM DETECTOR SIGNALS TO GLOBAL LEVEL-0 TRIGGER DECISION UNIT

The front-end design (FPGA or ASIC) described in this article can be one component of a larger system for triggering and front-end data acquisition. What follows is the description of the logical layout and physical layout of the system embodying the front-end chip. Connections on printed board, and off-printed board between front-end chips in order to have no boundary limitation in the overall detector trigger system, are also described.

6.1 Logical layout

Figure 14 shows the logical layout of the entire front-end for the Level-0 trigger and of the front-end for the DAQ.

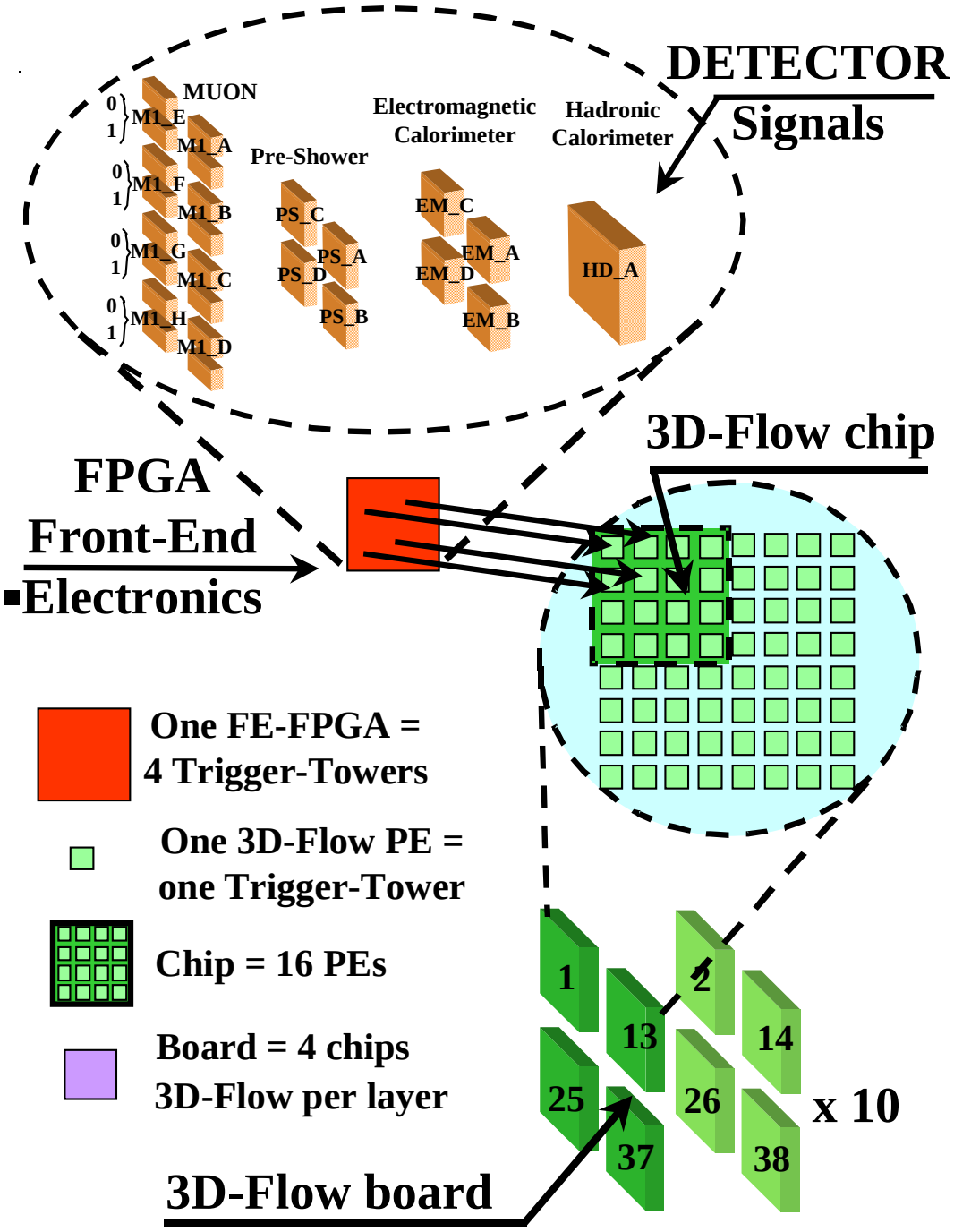


Figure 14. Logical layout of the functions, partitioned in components, that interface FE, Trigger, and DAQ.

6.2 Physical layout: a single type of board for several applications.

The modularity, flexibility, programmability, and scalability of the 3D-Flow system, including its front-end chip described in this article, are mantained all the way from the component to the crate(s). This also applies for the type of board used in the system. Only a single type of board is needed in a 3D-Flow system of any size. This board can change for each application from mixed signals, analog and digital, to a purely digital board, depending on the nature of the input signals received from the sensors. The complete description of the board can be found elsewhere[5], here follows only the description of the layout and the channels partitioning to the FPGA front-end chip with respect to the other chips on the board.

The board design, based upon an 80 MHz 3D-Flow processor and a 40 MHz FPGA with outputs to the 3D-Flow processors at 80 MHz, accommodates 64 trigger towers and 10 processing layers.

Every 25 ns each board received from the detector 64 ECAL + 16 HCAL, (since each HCAL is equivalent to an area of 4 ECALs), converted to digital with 12-bit resolution, and 192 bits, corresponding to 1 preshower + 2 Pads from muon station 1 x 64). This does not saturate the bandwidth of the 32-bit x 64 channels = 2048-bit every 25 ns bunch crossing that the 3D-Flow system could sustain per each board.

However, the front-end electronic FPGA chips synchronized all input channels listed above and store them into a pipeline buffer during the first level trigger decision time and rearrange the bits to be sent to the trigger unit in the following manner: The circuit of the trigger word formatter (see Section 5.2.2 and Figure 8) that is accommodated in the front-end FPGA, reduces the ECAL information from 12-bit to 8-bit, and increases the number of bits by duplicating information to different channels (e.g. sending the same 8-bit HCAL information to each of the 4 subtended ECAL blocks, and sending the same 2-bit Pads to 4 neighboring blocks), in order to save some bit-manipulation instructions to the 3D-Flow processors.

6.3 The 3D-Flow front-end chip layout in the signal processing board.

The 3D-Flow processing board[5] is built-in standard 9U x 5HP x 340 mm dimensions.

The FPGA front-end chip with the current design described in this article can be installed in either type of boards, the mixed signals, analog and digital, the purely digital board.

In both cases the digital information relative to four trigger towers (converted to digital by ADC converters in the mixed analog and digital board, or directly received in digital form via optical fibers in the purely digital board) is sent to the input of one FPGA.

Each of the 16 front-end FPGA chip (8 chips are assembled on the front and 8 are assembled on the rear of the board as shown in Figure 15) perform the following functions on four group of signals called "Trigger-Tower" (see Figure 8 and Section 5.1): [[message for Mies: the Figures 15 and 16 will be replaced late tonight with a single figure 15, consequently, all other Figures will be reduced by one]]

- synchronizes 72 inputs (4 x 12 bits ECAL, 12 bits HCAL, 4 x 1 PreSh, 4 x 2 Pads) every 25 ns;
- saves 72 raw-data every 25 ns in a 128 x 72 pipeline-stage digital buffer;

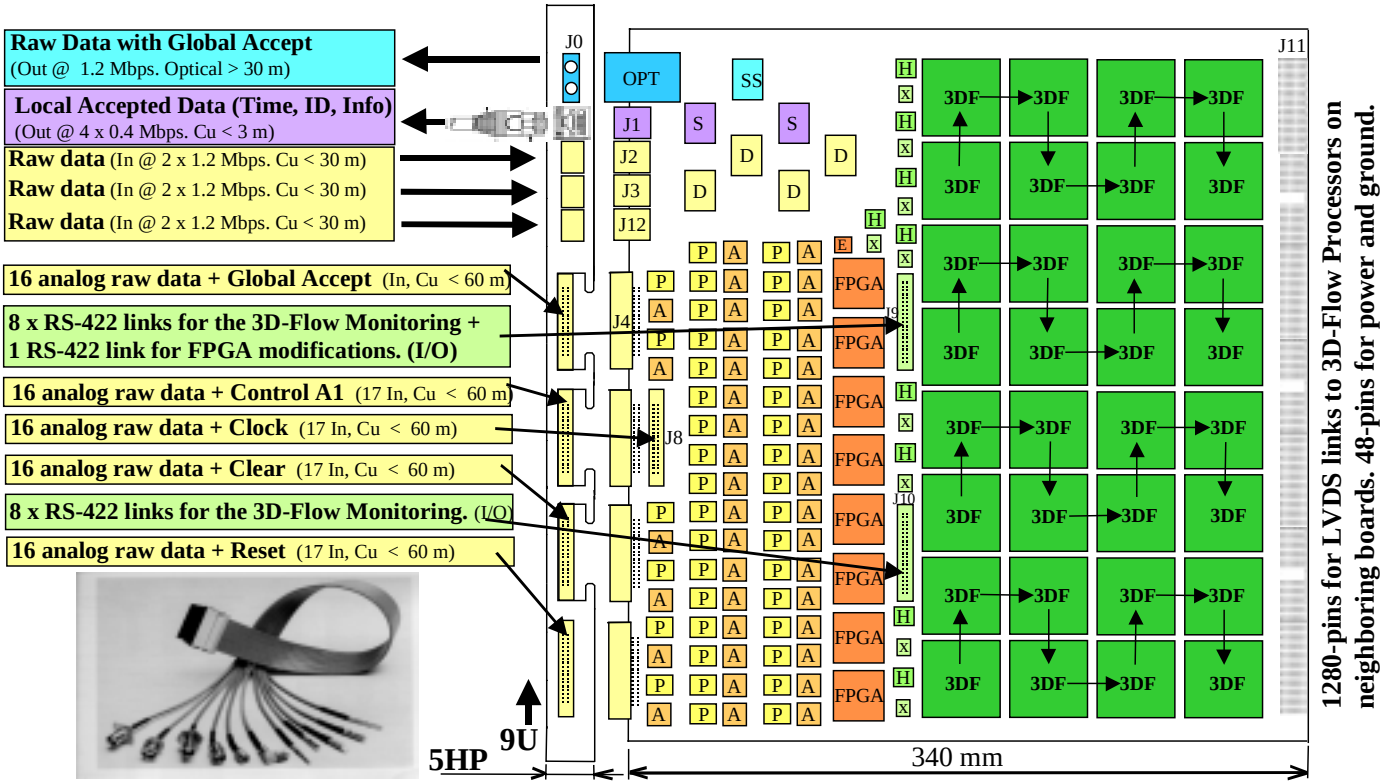
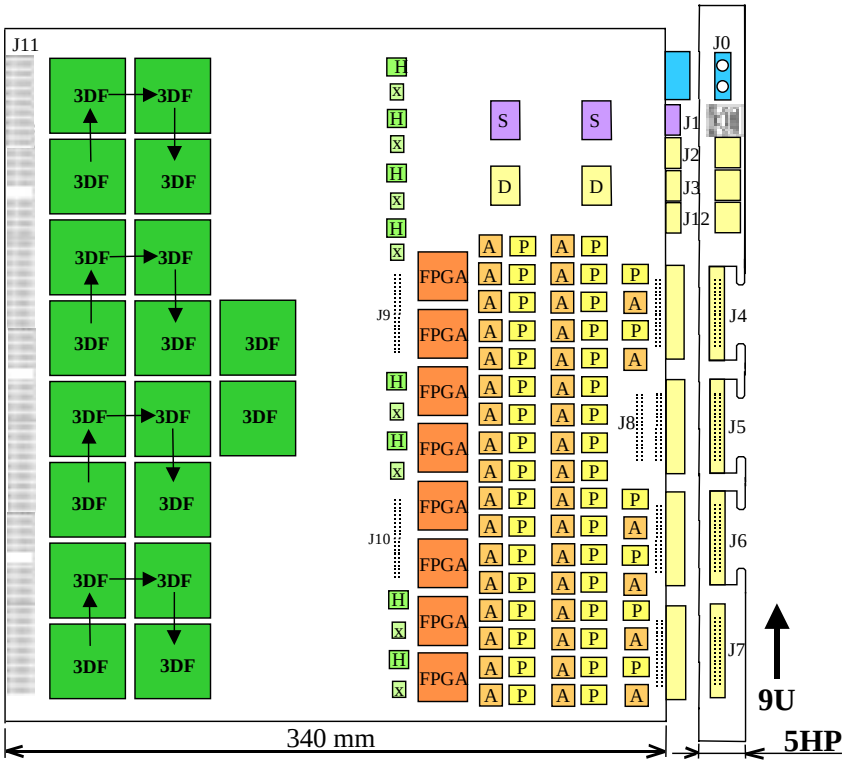


Figure 15. Mixed-signal processing board (front view).



COMPONENT LIST

#	Type	Device	Package
80	A	AD9042	ST-44
80	P	Preamplifier	
50	3DF	3D-FLOW	600-pin EBGA
16	FPGA	OR3T55 (OR3T30)	256-pin PBGA
17	H	MAX232	16-pin DW
17	X	MAX491	14-pin D
1	SS	HDMP-1032/1022 (AMCC-S3043 @ 2.4 GHz. VITESSE- VSC7214 @ 1.2 Gb/s)	64 Pin POFP
4	S	DS92LV1021	28-lead SSOP
6	D	HDMP-1034/1024	64 Pin POFP
1	OPTO	HFBR-53D5 (LUCENT –TC16-Type @ 2.4 Gb/s)	39.6 x 25.4 mm
1	E	AT7C010	20J PLCC
1	J0	HP- Opt. fiber	
1	J1	RJ-45	AMP-558342-1
3	J2.J3.J12	DB9 (or HSSDC) Equal. 30 m cable	AMP-748915-2 AMP-636000-1
4	J4-J7	17 coax ribbon	AMP-1-103167-4
1	J8	17 coax ribbon	AMP-1-103169-5
2	J9-J10	40 flat twisted p	AMP-103309-8
3	J11	Z-PACK type A	AMP-9-352153-2
4	J11	Z-PACK type B	AMP-9-352155-2

Figure 16. Mixed-signal processing board (rear view).

- Generates four trigger words to be sent to four 3D-Flow processors at 80 MHz. Currently the trigger word is defined as (however it can be changed at any time): 8-bit electromagnetic calorimenter, 8-bit hadronic calorimeter, 1-bit preshower, and 6-bit PADs from the muon station (see Figure 8);
- derandomizes accepted raw data into a FIFO;
- receives the global level-0 trigger at the average rate of 1 MHz and sends out the 80-bit raw data of the corresponding accepted events through a single output pin @ 80 MHz. Every FPGA chip (16 FPGA chips in total per each board, as shown in Figures 15 and 16) on the board is sending out one bit every 12.5 ns. The 16-bit word of raw data accepted by the global level-0 trigger decision unit are then serialized and sent out through an optical fiber @ 1.28 Gbps (12.5 ns/16 = 0.78125 ns period that is equivalent to 1.28 Gbps).

The physical layout is shown in Figure 15. Each of the 16 front-end FPGA chip receives four group of signals from the sub-destectors "Trigger-Tower" (see Figure 8 and Section 5.1). Four front-end FPGA chips send the trigger word formatted internally to the FPGA to one 3D-Flow chip (each 3D-Flow chip contains 16 processors. See Figure 14). In particular four FPGA chips which provide the optimized routing, send the trigger word to the 3D-Flow chip numbered 1 (See Figure 15), four other front-end FPGA send the trigger word to the 3D-Flow chip numbered 13,), four other front-end FPGA send the trigger word to the 3D-Flow chip numbered 25, and), four other front-end FPGA send the trigger word to the 3D-Flow chip numbered 37.

6.4 Connections between on board FE chips for no-boundary trigger implementation

In order to implement a system with no boundaries within a board, each FPGA chip has to connect (see Figure 9):

- M1_A signals of one component to M1_D of the adjacent component
- M1_E signals of one component to M1_H of the adjacent component

These connections can be easily implemented with Printed Circuit Board traces.

6.5 Connections between FE chips on different boards for no-boundary trigger implementation

In order to implement a system with no boundaries between boards or crates, one has to connect (see Figure 14):

- 4 x M1_A signals of components on one board to 4 x M1_D of components on adjacent board
- 4 x M1_E signals of components on one board to 4 x M1_H of components on adjacent board

These connections can be implemented with a cable (with connectors on the front panel of the board) between to adjacent boards (the boards may also be installed on different crates).

7 FRONT-END HARDWARE SUMMARY

The need of this study originated from generating the trigger word for the 3D-Flow level-0 trigger system. It would have been neither technically efficient, nor cost-effective to design a separate circuit to extract the trigger information from the full granularity of the DAQ (or higher-level trigger) signals.

The extraction of the level-0 trigger word, is well integrated (as shown in Section 5.2.2) into the circuit of the front-end that is performing the functions of input data synchronization, pipelining, derandomizing (FIFO). In fact, a design is more likely to turn out to be harmonious if all problems, from the front-end to the global level-0 decision unit, are considered as a whole problem to be solved rather than to be split into different stages with the risk of losing performance in the interface between two circuits designed by two different persons.

In summary, the complete design can be implemented with:

- 16 FPGAs per board would exploit the function of the front-end electronic and trigger word extraction of 64-trigger tower. The total calorimeter and muon station 1, front-end electronics will require 1536 FPGAs.
- Only about 375 additional OR3T30 FPGAs (The estimated cost of OR3T30 FPGA with speed grade 7 is about $80) are required to complete the FE for all subdetectors participating in the level-0 trigger. The calculation is as follows: the remaining subdetectors are the muon station 2, for 12,000 bits, and stations 3, 4, 5 for 6000-bits for a total of 30,000-bits. Assuming that the above function be implemented for 80-bit per FPGA OR3T30, we will need about 375 additional components.
- The mapping of the circuit into the FPGA has the following constraints: a) the ORCA PFU architecture is well optimized if the range of the variable delay that performs synchronization is limited from 0 to 2, b) the pipeline depth should not be greater than 128.

Purchasing about 2000 FPGA chips will provide maximum flexibility in downloading different circuits in the future.

8 CONCLUSIONS

The complete design of the front-end electronics has been made for a) ASIC implementation, and b) FPGA implementation. For the ASIC implementation, all VHDL source files and test results are provided.

Preliminary test results meet the functional requirements of LHCb and provide sufficient flexibility to allow future changes. Mapping to the FPGA component is currently being pursued. The design fits into an OR3T30 FPGA, current testing with the beta release of ORCA Foundry 9_35.57 shows that the speed of 80 MHz is missed by less than 2 ns. All reports (timing, netlist, etc.) are provided on the web site http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm With the new relese of ORCA Foundry software, there is no doubt that the 80 MHz requirements will be fully met.

The complete system design has been also provided, including how the problem was approached, and a solution has been provided to constrain the entire design to one type of FPGA, one type of ASIC, and one type of board, all of them replicated several times to build a system that is programmable, scalable, modular and that meets the requirements of several applications.

For the specific design of LHCb, 96 boarda (9U), about 2000 FPGAs (at a cost of about $80/each), and about 5000 3D-Flow ASICs in addition to all other commercially available components listed in Figures 15 and 16 of this article, will be sufficient to build a fully programmable system capable of sustaining an input data rate up to 960 GB/s, providing the programmability of executing a real-time algorithm (2x2, or 3x3, or 4x4, etc.) up to 20 steps (considering that 26 operations can be executed in each step).

Details have been provided on the evolution of the IC technology. As proven by the design of the technology-independent front-end chip with software tools that allow to target to FPGA and ASIC, the most cost-effective solution is to build the 3D-Flow in 0.18 μm CMOS technology @ 1.8 Volts, and accommodating 16 3D-Flow processors.

The difference in ~ 14,000 gate count/mm^2 for 0.35 μm to ~ 65,000 gate/count/mm^2 for 0.18 μm and the difference on power dissipation [gate/MHz] from 700 nW to 23 nW from 0.35 μm technology to 0.18 μm technology respectively (0.25 μm technology does not offer a noticeable power dissipation improvement @ 250 nW), indicates as the 0.18 μm technology @ 1.8 Volts power supply offers considerably more advantages with respect to 0.25 μm technology and that 16 3D-Flow processors in a chip would be the optimal implementation for a few years to come including the time the LHCb experiment needs to be ready for operation (considering that each 3D-Flow processor is approximately 100K gates, the total die size of the chip would be about 25 mm^2).

The design/verification methodology, which allows to verify the user's real-time system algorithm down to the gate-level simulation on a technology-independent platform, is a proof that the system can be implemented to any technology at any time.

.

Acknowledgments

The warmest acknowledgment goes to the SBIR office of the DOE of Dr. Rober Berger, for all its support, including a lot of moral support. Great support during all this project was also received from Albert Werbrouck, Sergio Conetti, and Billy Bonner, to whom I am very grateful. I would also like to acknowledge James Vorgert for his help in optimizing the mapping of the front-end design to ORCA FPGAs, Parisa Fathi from National Semiconductors for information regarding LVDS, and Mies de Vries for proofreading this paper.

APPENDIX A: VHDL CODE OF THE TOP-LEVEL MODULE OF THE FRONT-END CIRCUIT

```
--------------------------------------------------------------------------
--                                                                      --
-- Copyright (c) 1999 by 3D-Computing, Inc.                             --
--                                          All rights reserved.        --
--  Author       : Dario Crosetto                                       --
--                                                                      --
-- This source file is FREE for Universities, National Labs and         --
-- International Labs of non-profit organizations provided that the     --
-- above statements are not removed from the file,                      --
-- that the revision history is updated if changes are introduced, and  --
-- that any derivative work contains the entire above mentioned notice. --
--                                                                      --
--  Package name : FE_top.vhd
--
--  Project   : Front-End Electronics Logic
--  Purpose   : This file implements the front-end signals synchronization,
--              pipelining, derandomizing, trigger word formatter.
--              The code is for four trigger channels
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
```

```
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
-------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
LIBRARY work;
USE work.FE_config.ALL;

-------------------------------------------------------------------
--Entity Definition
-------------------------------------------------------------------

ENTITY FE_top IS
   PORT (      clock, reset   : IN STD_LOGIC;
        EM_A                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_B                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_C                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_D                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        HD_A                  : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        PS_A                  : IN std_logic;
        PS_B                  : IN std_logic;
        PS_C                  : IN std_logic;
        PS_D                  : IN std_logic;
        M1_A                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_B                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_C                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_D                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_E                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_F                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_G                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_H                  : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        Time_ID               : IN STD_LOGIC_VECTOR(Time_ID_width -1 DOWNTO 0);
        G_L0                  : IN std_logic;
        EnInData              : IN std_logic;
        EnOutData              : IN std_logic;
        clk_x2                 : IN STD_LOGIC; -- Replaced by the internal PLL –
        clk_x4                 : IN STD_LOGIC; -- Replaced by the internal PLL –

        fifo_empty             : OUT std_logic;
        fifo_full              : OUT std_logic;
        diff_fifo_addr         : OUT std_logic_vector(fifo_depth - 1 downto 0);
        LOAD_3DF_A             : OUT std_logic;
        TO_3DF_A               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_B             : OUT std_logic;
        TO_3DF_B               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_C             : OUT std_logic;
        TO_3DF_C               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_D             : OUT std_logic;
        TO_3DF_D               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        DataOut                : OUT std_logic;
        St_Burst               : OUT std_logic
        );
END FE_top;

-------------------------------------------------------------------
--ARCHITECTURE Definition
-------------------------------------------------------------------

ARCHITECTURE struct OF FE_top IS

        SIGNAL  EM_AS          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL  EM_BS          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL  EM_CS          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL  EM_DS          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL  HD_AS          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL  PS_AS          : STD_LOGIC;
        SIGNAL  PS_BS          : STD_LOGIC;
        SIGNAL  PS_CS          : STD_LOGIC;
        SIGNAL  PS_DS          : STD_LOGIC;
        SIGNAL  M1_AS          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL  M1_BS          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL  M1_CS          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL  M1_DS          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL  M1_ES          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL  M1_FS          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL  M1_GS          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL  M1_HS          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
```

```
        SIGNAL  TO_IN_FIFO      : STD_LOGIC_VECTOR(fifo_width -1 DOWNTO 0);

-------------------------------------------------------
-- component declaration
-------------------------------------------------------

COMPONENT FE_syncinput
    PORT (
        clock                   : IN STD_LOGIC;
        reset                   : IN STD_LOGIC;
        EM_A                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_B                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_C                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_D                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        HD_A                    : IN STD_LOGIC_VECTOR(adc_width-1 DOWNTO 0);
        PS_A                    : IN std_logic;
        PS_B                    : IN std_logic;
        PS_C                    : IN std_logic;
        PS_D                    : IN std_logic;
        M1_A                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_B                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_C                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_D                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_E                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_F                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_G                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_H                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        EnInData                : IN std_logic;

        EM_AS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_BS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_CS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_DS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        HD_AS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        PS_AS                   : OUT std_logic;
        PS_BS                   : OUT std_logic;
        PS_CS                   : OUT std_logic;
        PS_DS                   : OUT std_logic;
        M1_AS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_BS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_CS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_DS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_ES                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_FS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_GS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_HS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0)
    );

END COMPONENT;


COMPONENT FE_trig_formatter

    PORT (
        Clock                   : IN STD_LOGIC;
        Reset                   : IN STD_LOGIC;
        EM_AS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_BS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_CS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_DS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        HD_AS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        PS_AS                   : IN std_logic;
        PS_BS                   : IN std_logic;
        PS_CS                   : IN std_logic;
        PS_DS                   : IN std_logic;
        M1_AS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_BS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_CS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_DS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_ES                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_FS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_GS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_HS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        EnInData                : IN std_logic;
        clk_x4                  : IN STD_LOGIC; -- Replaced by the internal PLL -

        LOAD_3DF_A              : OUT std_logic;
        TO_3DF_A                : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_B              : OUT std_logic;
        TO_3DF_B                : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
```

```
        LOAD_3DF_C              : OUT std_logic;
        TO_3DF_C                : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_D              : OUT std_logic;
        TO_3DF_D                : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0)
        );

END COMPONENT;


COMPONENT FE_pipeline
    PORT (
        Clock                   : IN STD_LOGIC;
        Reset                   : IN STD_LOGIC;
        EM_AS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_BS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_CS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_DS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        HD_AS                   : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        PS_AS                   : IN std_logic;
        PS_BS                   : IN std_logic;
        PS_CS                   : IN std_logic;
        PS_DS                   : IN std_logic;
        M1_BS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_CS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_FS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_GS                   : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        G_L0                    : IN std_logic;
        Time_ID                 : IN STD_LOGIC_VECTOR(Time_ID_width -1 DOWNTO 0);

        TO_IN_FIFO              : OUT STD_LOGIC_VECTOR(fifo_width - 1 DOWNTO 0)
     );

END COMPONENT;


COMPONENT FE_fifo
     PORT (
        Reset                  : IN STD_LOGIC;
        Clock                  : IN STD_LOGIC;
        G_L0                   : IN std_logic;
        EnOutData              : IN std_logic;
        TO_IN_FIFO             : IN std_logic_vector(fifo_width - 1 DOWNTO 0);
        clk_x2                 : IN std_logic; -- Replaced by the internal PLL -

        fifo_empty             : OUT std_logic;
        fifo_full              : OUT std_logic;
        diff_fifo_addr         : OUT std_logic_vector(fifo_depth - 1 downto 0);
        DataOut                : OUT std_logic;
        St_Burst               : OUT std_logic
    );

 END COMPONENT;


BEGIN

u1: FE_syncinput
        PORT MAP(
        clock           =>      clock ,
        reset           =>      reset ,
        EM_A            =>      EM_A(adc_width - 1 DOWNTO 0),
        EM_B            =>      EM_B(adc_width - 1 DOWNTO 0),
        EM_C            =>      EM_C(adc_width - 1 DOWNTO 0),
        EM_D            =>      EM_D(adc_width - 1 DOWNTO 0),
        HD_A            =>      HD_A(adc_width - 1 DOWNTO 0),
        PS_A            =>      PS_A  ,
        PS_B            =>      PS_B  ,
        PS_C            =>      PS_C  ,
        PS_D            =>      PS_D  ,
        M1_A            =>      M1_A(M1_width - 1 DOWNTO 0),
        M1_B            =>      M1_B(M1_width - 1 DOWNTO 0),
        M1_C            =>      M1_C(M1_width - 1 DOWNTO 0),
        M1_D            =>      M1_D(M1_width - 1 DOWNTO 0),
        M1_E            =>      M1_E(M1_width - 1 DOWNTO 0),
        M1_F            =>      M1_F(M1_width - 1 DOWNTO 0),
        M1_G            =>      M1_G(M1_width - 1 DOWNTO 0),
        M1_H            =>      M1_H(M1_width - 1 DOWNTO 0),
        EnInData        =>      EnInData,
```

```
        EM_AS           =>      EM_AS(adc_width - 1 DOWNTO 0),
        EM_BS           =>      EM_BS(adc_width - 1 DOWNTO 0),
        EM_CS           =>      EM_CS(adc_width - 1 DOWNTO 0),
        EM_DS           =>      EM_DS(adc_width - 1 DOWNTO 0),
        HD_AS           =>      HD_AS(adc_width - 1 DOWNTO 0),
        PS_AS           =>      PS_AS,
        PS_BS           =>      PS_BS,
        PS_CS           =>      PS_CS,
        PS_DS           =>      PS_DS,
        M1_AS           =>      M1_AS(M1_width - 1 DOWNTO 0),
        M1_BS           =>      M1_BS(M1_width - 1 DOWNTO 0),
        M1_CS           =>      M1_CS(M1_width - 1 DOWNTO 0),
        M1_DS           =>      M1_DS(M1_width - 1 DOWNTO 0),
        M1_ES           =>      M1_ES(M1_width - 1 DOWNTO 0),
        M1_FS           =>      M1_FS(M1_width - 1 DOWNTO 0),
        M1_GS           =>      M1_GS(M1_width - 1 DOWNTO 0),
        M1_HS           =>      M1_HS(M1_width - 1 DOWNTO 0)
        );


    u2: FE_trig_formatter
        PORT MAP(
    clock               =>      clock ,
    reset               =>      reset ,
        EM_AS       =>      EM_AS(adc_width - 1 DOWNTO 0),
        EM_BS       =>      EM_BS(adc_width - 1 DOWNTO 0),
        EM_CS       =>      EM_CS(adc_width - 1 DOWNTO 0),
        EM_DS       =>      EM_DS(adc_width - 1 DOWNTO 0),
        HD_AS       =>      HD_AS(adc_width - 1 DOWNTO 0),
        PS_AS       =>      PS_AS,
        PS_BS       =>      PS_BS,
        PS_CS       =>      PS_CS,
        PS_DS       =>      PS_DS,
        M1_AS       =>      M1_AS(M1_width - 1 DOWNTO 0),
        M1_BS       =>      M1_BS(M1_width - 1 DOWNTO 0),
        M1_CS       =>      M1_CS(M1_width - 1 DOWNTO 0),
        M1_DS       =>      M1_DS(M1_width - 1 DOWNTO 0),
        M1_ES       =>      M1_ES(M1_width - 1 DOWNTO 0),
        M1_FS       =>      M1_FS(M1_width - 1 DOWNTO 0),
        M1_GS       =>      M1_GS(M1_width - 1 DOWNTO 0),
        M1_HS       =>      M1_HS(M1_width - 1 DOWNTO 0),
        EnInData    =>      EnInData,
        clk_x4      =>      clk_x4, -- Replaced by the internal PLL --

        LOAD_3DF_A  =>      LOAD_3DF_A,
        TO_3DF_A    =>      TO_3DF_A(Width_To3DF - 1 DOWNTO 0),
        LOAD_3DF_B  =>      LOAD_3DF_B,
        TO_3DF_B    =>      TO_3DF_B(Width_To3DF - 1 DOWNTO 0),
        LOAD_3DF_C  =>      LOAD_3DF_C,
        TO_3DF_C    =>      TO_3DF_C(Width_To3DF - 1 DOWNTO 0),
        LOAD_3DF_D  =>      LOAD_3DF_D,
        TO_3DF_D    =>      TO_3DF_D(Width_To3DF - 1 DOWNTO 0)
        );


    u3: FE_pipeline
        PORT MAP(
        clock       =>      clock ,
        reset       =>      reset ,
        EM_AS       =>      EM_AS(adc_width - 1 DOWNTO 0),
        EM_BS       =>      EM_BS(adc_width - 1 DOWNTO 0),
        EM_CS       =>      EM_CS(adc_width - 1 DOWNTO 0),
        EM_DS       =>      EM_DS(adc_width - 1 DOWNTO 0),
        HD_AS       =>      HD_AS(adc_width - 1 DOWNTO 0),
        PS_AS       =>      PS_AS,
        PS_BS       =>      PS_BS,
        PS_CS       =>      PS_CS,
        PS_DS       =>      PS_DS,
        M1_BS       =>      M1_BS(M1_width - 1 DOWNTO 0),
        M1_CS       =>      M1_CS(M1_width - 1 DOWNTO 0),
        M1_FS       =>      M1_FS(M1_width - 1 DOWNTO 0),
        M1_GS       =>      M1_GS(M1_width - 1 DOWNTO 0),

        G_L0        =>      G_L0,
        Time_ID     =>      Time_ID(Time_ID_width - 1 DOWNTO 0),
        TO_IN_FIFO  =>      TO_IN_FIFO(fifo_width - 1 DOWNTO 0)
    );

    u4: FE_fifo
        PORT MAP(
```

```
        Reset       =>      reset,
        Clock       =>      clock,
        G_L0        =>      G_L0,
        EnOutData   =>      EnOutData,
        TO_IN_FIFO  =>      TO_IN_FIFO,
        clk_x2      =>      clk_x2,   -- Replaced by the internal PLL --

        fifo_empty  =>      fifo_empty,
        fifo_full   =>      fifo_full,
        diff_fifo_addr  =>  diff_fifo_addr,
        DataOut     =>      DataOut,
        St_Burst    =>      St_Burst
    );

END struct;
```

VHDL code of the Input-Synchronizer module

```
--------------------------------------------------------------------------
--                                                                      --
-- Copyright (c) 1999 by 3D-Computing, Inc.                             --
--                                              All rights reserved.    --
--  Author       : Dario Crosetto                                       --
--                                                                      --
-- This source file is FREE for Universities, National Labs and         --
-- International Labs of non-profit organizations provided that the     --
-- above statements are not removed from the file,                      --
-- that the revision history is updated if changes are introduced, and  --
-- that any derivative work contains the entire above mentioned notice. --
--                                                                      --
--  Package name : FE_syncinput.vhd
--
--  Project   : Front-End Electronics Logic
--  Purpose   : This file implements the front-end signals              --
--              synchronization on the rising edge of the clock         --
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
--------------------------------------------------------------------------
--
LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
LIBRARY work;
USE work.FE_config.ALL;

-----------------------------------------------------------------
--Entity Definition
-----------------------------------------------------------------

ENTITY FE_syncinput IS
   PORT (clock, reset           : IN STD_LOGIC;
        EM_A                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_B                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_C                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        EM_D                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        HD_A                    : IN STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
        PS_A                    : IN std_logic;
        PS_B                    : IN std_logic;
        PS_C                    : IN std_logic;
        PS_D                    : IN std_logic;
        M1_A                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_B                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_C                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_D                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_E                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_F                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_G                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        M1_H                    : IN STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
        EnInData                : IN std_logic;

        EM_AS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_BS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_CS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
```

```
        EM_DS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        HD_AS                   : OUT std_logic_vector(adc_width - 1 DOWNTO 0);
        PS_AS                   : OUT std_logic;
        PS_BS                   : OUT std_logic;
        PS_CS                   : OUT std_logic;
        PS_DS                   : OUT std_logic;
        M1_AS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_BS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_CS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_DS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_ES                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_FS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_GS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_HS                   : OUT std_logic_vector(M1_width - 1 DOWNTO 0)
   );

END FE_syncinput;

-------------------------------------------------------------------
--ARCHITECTURE Definition
-------------------------------------------------------------------

ARCHITECTURE rtl OF FE_syncinput IS

    SIGNAL  Select_del_EM      : std_Logic_vector(1 DOWNTO 0);
    SIGNAL  Select_del_HD      : std_Logic_vector(1 DOWNTO 0);
    SIGNAL  Select_del_PS      : std_Logic_vector(1 DOWNTO 0);
    SIGNAL  Select_del_M1      : std_Logic_vector(1 DOWNTO 0);
    SIGNAL  EM_A_clkd          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly1_EM_A          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly2_EM_A          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  EM_B_clkd          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly1_EM_B          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly2_EM_B          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  EM_C_clkd          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly1_EM_C          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly2_EM_C          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  EM_D_clkd          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly1_EM_D          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly2_EM_D          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  HD_A_clkd          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly1_HD_A          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  Dly2_HD_A          : STD_LOGIC_VECTOR(adc_width - 1 DOWNTO 0);
    SIGNAL  PS_A_clkd          : STD_LOGIC;
    SIGNAL  Dly1_PS_A          : STD_LOGIC;
    SIGNAL  Dly2_PS_A          : STD_LOGIC;
    SIGNAL  PS_B_clkd          : STD_LOGIC;
    SIGNAL  Dly1_PS_B          : STD_LOGIC;
    SIGNAL  Dly2_PS_B          : STD_LOGIC;
    SIGNAL  PS_C_clkd          : STD_LOGIC;
    SIGNAL  Dly1_PS_C          : STD_LOGIC;
    SIGNAL  Dly2_PS_C          : STD_LOGIC;
    SIGNAL  PS_D_clkd          : STD_LOGIC;
    SIGNAL  Dly1_PS_D          : STD_LOGIC;
    SIGNAL  Dly2_PS_D          : STD_LOGIC;
    SIGNAL  M1_A_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_A          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_A          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  M1_B_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_B          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_B          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  M1_C_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_C          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_C          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  M1_D_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_D          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_D          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  M1_E_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_E          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_E          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  M1_F_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_F          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_F          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  M1_G_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_G          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_G          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  M1_H_clkd          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly1_M1_H          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
    SIGNAL  Dly2_M1_H          : STD_LOGIC_VECTOR(M1_width - 1 DOWNTO 0);
```

```
BEGIN

--Rising Edge triggered
    ADD_DLY: PROCESS (clock, reset)
          BEGIN
               IF (reset = '0') THEN

                     EM_A_clkd <= (others => '0');
                     EM_B_clkd <= (others => '0');
                     EM_C_clkd <= (others => '0');
                     EM_D_clkd <= (others => '0');
                     dly1_EM_A <= (others => '0');
                     dly1_EM_B <= (others => '0');
                     dly1_EM_C <= (others => '0');
                     dly1_EM_D <= (others => '0');
                     dly2_EM_A <= (others => '0');
                     dly2_EM_B <= (others => '0');
                     dly2_EM_C <= (others => '0');
                     dly2_EM_D <= (others => '0');
                     HD_A_clkd <= (others => '0');
                     dly1_HD_A <= (others => '0');
                     dly2_HD_A <= (others => '0');
                     PS_A_clkd <= '0';
                     PS_B_clkd <= '0';
                     PS_C_clkd <= '0';
                     PS_D_clkd <= '0';
                     dly1_PS_A <= '0';
                     dly1_PS_B <= '0';
                     dly1_PS_C <= '0';
                     dly1_PS_D <= '0';
                     dly2_PS_A <= '0';
                     dly2_PS_B <= '0';
                     dly2_PS_C <= '0';
                     dly2_PS_D <= '0';
                     M1_A_clkd <= (others => '0');
                     M1_B_clkd <= (others => '0');
                     M1_C_clkd <= (others => '0');
                     M1_D_clkd <= (others => '0');
                     M1_E_clkd <= (others => '0');
                     M1_F_clkd <= (others => '0');
                     M1_G_clkd <= (others => '0');
                     M1_H_clkd <= (others => '0');
                     dly1_M1_A <= (others => '0');
                     dly1_M1_B <= (others => '0');
                     dly1_M1_C <= (others => '0');
                     dly1_M1_D <= (others => '0');
                     dly1_M1_E <= (others => '0');
                     dly1_M1_F <= (others => '0');
                     dly1_M1_G <= (others => '0');
                     dly1_M1_H <= (others => '0');
                     dly2_M1_A <= (others => '0');
                     dly2_M1_B <= (others => '0');
                     dly2_M1_C <= (others => '0');
                     dly2_M1_D <= (others => '0');
                     dly2_M1_E <= (others => '0');
                     dly2_M1_F <= (others => '0');
                     dly2_M1_G <= (others => '0');
                     dly2_M1_H <= (others => '0');

               ELSIF (clock'EVENT AND clock = '1') THEN
                     EM_A_clkd <= EM_A;
                     EM_B_clkd <= EM_B;
                     EM_C_clkd <= EM_C;
                     EM_D_clkd <= EM_D;
                     dly1_EM_A <= EM_A_clkd;
                     dly1_EM_B <= EM_B_clkd;
                     dly1_EM_C <= EM_C_clkd;
                     dly1_EM_D <= EM_D_clkd;
                     dly2_EM_A <= dly1_EM_A;
                     dly2_EM_B <= dly1_EM_B;
                     dly2_EM_C <= dly1_EM_C;
                     dly2_EM_D <= dly1_EM_D;
                     HD_A_clkd <= HD_A;
                     dly1_HD_A <= HD_A_clkd;
                     dly2_HD_A <= dly1_HD_A;
                     PS_A_clkd <= PS_A;
                     PS_B_clkd <= PS_B;
```

```
                        PS_C_clkd <= PS_C;
                        PS_D_clkd <= PS_D;
                        dly1_PS_A <= PS_A_clkd;
                        dly1_PS_B <= PS_B_clkd;
                        dly1_PS_C <= PS_C_clkd;
                        dly1_PS_D <= PS_D_clkd;
                        dly2_PS_A <= dly1_PS_A;
                        dly2_PS_B <= dly1_PS_B;
                        dly2_PS_C <= dly1_PS_C;
                        dly2_PS_D <= dly1_PS_D;
                        M1_A_clkd <= M1_A;
                        M1_B_clkd <= M1_B;
                        M1_C_clkd <= M1_C;
                        M1_D_clkd <= M1_D;
                        M1_E_clkd <= M1_E;
                        M1_F_clkd <= M1_F;
                        M1_G_clkd <= M1_G;
                        M1_H_clkd <= M1_H;
                        dly1_M1_A <= M1_A_clkd;
                        dly1_M1_B <= M1_B_clkd;
                        dly1_M1_C <= M1_C_clkd;
                        dly1_M1_D <= M1_D_clkd;
                        dly1_M1_E <= M1_E_clkd;
                        dly1_M1_F <= M1_F_clkd;
                        dly1_M1_G <= M1_G_clkd;
                        dly1_M1_H <= M1_H_clkd;
                        dly2_M1_A <= dly1_M1_A;
                        dly2_M1_B <= dly1_M1_B;
                        dly2_M1_C <= dly1_M1_C;
                        dly2_M1_D <= dly1_M1_D;
                        dly2_M1_E <= dly1_M1_E;
                        dly2_M1_F <= dly1_M1_F;
                        dly2_M1_G <= dly1_M1_G;
                        dly2_M1_H <= dly1_M1_H;

                ELSE
                END IF;
END PROCESS ADD_DLY;

-- change constant based on detector, and/or electronics, and/or cable length
        Select_Del_EM <= EM_del;
        Select_Del_HD <= HD_del;
        Select_Del_PS <= PS_del;
        Select_Del_M1 <= M1_del;

-- This synchronizes EM
    EM_AS  <= dly2_EM_A WHEN (Select_Del_EM = "10")
                ELSE dly1_EM_A WHEN (Select_Del_EM = "01")
                ELSE EM_A_clkd;
    EM_BS  <= dly2_EM_B WHEN (Select_Del_EM = "10")
                ELSE dly1_EM_B WHEN (Select_Del_EM = "01")
                ELSE EM_B_clkd;
    EM_CS  <= dly2_EM_C WHEN (Select_Del_EM = "10")
                ELSE dly1_EM_C WHEN (Select_Del_EM = "01")
                ELSE EM_C_clkd;
    EM_DS  <= dly2_EM_D WHEN (Select_Del_EM = "10")
                ELSE dly1_EM_D WHEN (Select_Del_EM = "01")
                ELSE EM_D_clkd;

-- This synchronises HD
    HD_AS  <= dly2_HD_A WHEN (Select_Del_HD = "10")
                ELSE dly1_HD_A WHEN (Select_Del_HD = "01")
                ELSE HD_A_clkd;

-- This synchronises PS
    PS_AS  <= dly2_PS_A WHEN (Select_Del_PS = "10")
                ELSE dly1_PS_A WHEN (Select_Del_PS = "01")
                ELSE PS_A_clkd;
    PS_BS  <= dly2_PS_B WHEN (Select_Del_PS = "10")
                ELSE dly1_PS_B WHEN (Select_Del_PS = "01")
                ELSE PS_B_clkd;
    PS_CS  <= dly2_PS_C WHEN (Select_Del_PS = "10")
                ELSE dly1_PS_C WHEN (Select_Del_PS = "01")
                ELSE PS_C_clkd;
    PS_DS  <= dly2_PS_D WHEN (Select_Del_PS = "10")
                ELSE dly1_PS_D WHEN (Select_Del_PS = "01")
                ELSE PS_D_clkd;

-- This synchronises M1
    M1_AS  <= dly2_M1_A WHEN (Select_Del_M1 = "10")
```

```
                ELSE dly1_M1_A WHEN (Select_Del_M1 = "01")
                ELSE M1_A_clkd;

    M1_BS  <= dly2_M1_B WHEN (Select_Del_M1 = "10")
                ELSE dly1_M1_B WHEN (Select_Del_M1 = "01")
                ELSE M1_B_clkd;


    M1_CS  <= dly2_M1_C WHEN (Select_Del_M1 = "10")
                ELSE dly1_M1_C WHEN (Select_Del_M1 = "01")
                ELSE M1_C_clkd;

    M1_DS  <= dly2_M1_D WHEN (Select_Del_M1 = "10")
                ELSE dly1_M1_D WHEN (Select_Del_M1 = "01")
                ELSE M1_D_clkd;

    M1_ES  <= dly2_M1_E WHEN (Select_Del_M1 = "10")
                ELSE dly1_M1_E WHEN (Select_Del_M1 = "01")
                ELSE M1_E_clkd;

    M1_FS  <= dly2_M1_F WHEN (Select_Del_M1 = "10")
                ELSE dly1_M1_F WHEN (Select_Del_M1 = "01")
                ELSE M1_F_clkd;


    M1_GS  <= dly2_M1_G WHEN (Select_Del_M1 = "10")
                ELSE dly1_M1_G WHEN (Select_Del_M1 = "01")
                ELSE M1_G_clkd;

    M1_HS  <= dly2_M1_H WHEN (Select_Del_M1 = "10")
                ELSE dly1_M1_H WHEN (Select_Del_M1 = "01")
                ELSE M1_H_clkd;

END rtl;
```

VHDL code of the Trigger-Word-Formatter module

```
--------------------------------------------------------------------------------
--                                                                            --
-- Copyright (c) 1999 by 3D-Computing, Inc.                                   --
--                                          All rights reserved.              --
--  Author       : Dario Crosetto                                             --
--                                                                            --
-- This source file is FREE for Universities, National Labs and               --
-- International Labs of non-profit organizations provided that the           --
-- above statements are not removed from the file,                            --
-- that the revision history is updated if changes are introduced, and        --
-- that any derivative work contains the entire above mentioned notice.       --
--                                                                            --
--  Package name : FE_trig_formatter.vhd                                      --
--                                                                            --
--  Project   : Front-End Electronics Logic                                   --
--  Purpose   : This file implements the front-end trigger                    --
--              word formatter.                                               --
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
LIBRARY work;
USE work.FE_config.ALL;

-----------------------------------------------------------------
--Entity Definition
-----------------------------------------------------------------

ENTITY FE_trig_formatter IS

   PORT (clock, reset    : IN STD_LOGIC;
        EM_AS            : IN std_logic_vector(adc_width - 1 DOWNTO 0);
```

```
        EM_BS             : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_CS             : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        EM_DS             : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        HD_AS             : IN std_logic_vector(adc_width - 1 DOWNTO 0);
        PS_AS             : IN std_logic;
        PS_BS             : IN std_logic;
        PS_CS             : IN std_logic;
        PS_DS             : IN std_logic;
        M1_AS             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_BS             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_CS             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_DS             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_ES             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_FS             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_GS             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        M1_HS             : IN std_logic_vector(M1_width - 1 DOWNTO 0);
        EnInData          : IN std_logic;
        clk_x4            : IN STD_LOGIC;      -- Replaced by the internal PLL --

        LOAD_3DF_A        : OUT std_logic;
        TO_3DF_A          : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_B        : OUT std_logic;
        TO_3DF_B          : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_C        : OUT std_logic;
        TO_3DF_C          : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_D        : OUT std_logic;
        TO_3DF_D          : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0)
        );

END FE_trig_formatter;

-----------------------------------------------------------------
--ARCHITECTURE Definition
-----------------------------------------------------------------

ARCHITECTURE rtl OF FE_trig_formatter IS

        SIGNAL  TEMP_3DF_A      : STD_LOGIC_VECTOR(4 * Width_To3DF - 1 DOWNTO 0);
        SIGNAL  TEMP_3DF_B      : STD_LOGIC_VECTOR(4 * Width_To3DF - 1 DOWNTO 0);
        SIGNAL  TEMP_3DF_C      : STD_LOGIC_VECTOR(4 * Width_To3DF - 1 DOWNTO 0);
        SIGNAL  TEMP_3DF_D      : STD_LOGIC_VECTOR(4 * Width_To3DF - 1 DOWNTO 0);
        SIGNAL  Mux_Count       : STD_LOGIC_VECTOR(1 DOWNTO 0);
        SIGNAL  EnInData_delay  : STD_LOGIC;
        SIGNAL  int_clk_x4      : STD_LOGIC;


BEGIN

int_clk_x4 <= clk_x4;

-- Format the 32-bit output word to be sent to the trigger decision processor

   TEMP_3DF_A <= EM_AS(EM_trig_width -1 DOWNTO 0) &
                       HD_AS(HA_trig_width -1 DOWNTO 0)
                       & PS_AS & "000000000" & M1_AS(M1_trig_width - 1 DOWNTO 0)
                       & M1_BS(M1_trig_width - 1 DOWNTO 0)
                       & M1_CS(M1_trig_width - 1 DOWNTO 0);

   TEMP_3DF_B <= EM_BS(EM_trig_width -1 DOWNTO 0) &
                       HD_AS(HA_trig_width -1 DOWNTO 0)
                       & PS_BS & "000000000" & M1_BS(M1_trig_width - 1 DOWNTO 0)
                       & M1_CS(M1_trig_width - 1 DOWNTO 0)
                       & M1_DS(M1_trig_width - 1 DOWNTO 0);

   TEMP_3DF_C <= EM_CS(EM_trig_width -1 DOWNTO 0) &
                       HD_AS(HA_trig_width -1 DOWNTO 0)
                       & PS_CS & "000000000" & M1_ES(M1_trig_width - 1 DOWNTO 0)
                       & M1_FS(M1_trig_width - 1 DOWNTO 0)
                       & M1_GS(M1_trig_width - 1 DOWNTO 0);

   TEMP_3DF_D <= EM_DS(EM_trig_width -1 DOWNTO 0) &
                       HD_AS(HA_trig_width -1 DOWNTO 0)
                       & PS_DS & "000000000" & M1_FS(M1_trig_width - 1 DOWNTO 0)
                       & M1_GS(M1_trig_width - 1 DOWNTO 0)
                       & M1_HS(M1_trig_width - 1 DOWNTO 0);

--Clocking enable signal
ENBL_CLKD: PROCESS (int_clk_x4, reset)
         BEGIN
              IF (reset = '0') THEN
```

```
                        EnInData_delay <= '0';
                ELSIF (int_clk_x4'EVENT AND int_clk_x4 = '1') THEN
                        EnInData_delay <= EnInData;
                ELSE
                END IF;
        END PROCESS ENBL_CLKD;

        LOAD_3DF_A <= EnInData_delay;
        LOAD_3DF_B <= EnInData_delay;
        LOAD_3DF_C <= EnInData_delay;
        LOAD_3DF_D <= EnInData_delay;


-- clocking the trigger-word to the trigger decision processor.
        MUX_CNT: PROCESS (int_clk_x4, reset)
          BEGIN
                IF (reset = '0') THEN
                        Mux_Count <= (others => '0');
                ELSIF (int_clk_x4'EVENT AND int_clk_x4 = '1') THEN
                  IF (EnInData_delay = '1') THEN
                Mux_Count <= Mux_Count + 1;
                  ELSE
                  END IF;
                ELSE
                END IF;
        END PROCESS MUX_CNT;

CLK_TRI: PROCESS (int_clk_x4, reset)
          BEGIN
                IF (reset = '0') THEN
                        TO_3DF_A  <= (others => '0');
                ELSIF (int_clk_x4'EVENT AND int_clk_x4 = '1') THEN
                  IF (EnInData_delay = '1') THEN
                        CASE Mux_count IS
WHEN "00" => TO_3DF_A <= TEMP_3DF_A(4 * Width_To3DF - 1 DOWNTO 3 * Width_To3DF);
WHEN "01" => TO_3DF_A <= TEMP_3DF_A(3 * Width_To3DF - 1 DOWNTO 2 * Width_To3DF);
WHEN "10" => TO_3DF_A <= TEMP_3DF_A(2 * Width_To3DF - 1 DOWNTO Width_To3DF);
WHEN "11" => TO_3DF_A <= TEMP_3DF_A(Width_To3DF - 1 DOWNTO 0);
WHEN OTHERS => NULL;
                        END CASE;

                        CASE Mux_count IS
WHEN "00" => TO_3DF_B <= TEMP_3DF_B(4 * Width_To3DF - 1 DOWNTO 3 * Width_To3DF);
WHEN "01" => TO_3DF_B <= TEMP_3DF_B(3 * Width_To3DF - 1 DOWNTO 2 * Width_To3DF);
WHEN "10" => TO_3DF_B <= TEMP_3DF_B(2 * Width_To3DF - 1 DOWNTO Width_To3DF);
WHEN "11" => TO_3DF_B <= TEMP_3DF_B(Width_To3DF - 1 DOWNTO 0);
WHEN OTHERS => NULL;
                        END CASE;

                        CASE Mux_count IS
WHEN "00" => TO_3DF_C <= TEMP_3DF_C(4 * Width_To3DF - 1 DOWNTO 3 * Width_To3DF);
WHEN "01" => TO_3DF_C <= TEMP_3DF_C(3 * Width_To3DF - 1 DOWNTO 2 * Width_To3DF);
WHEN "10" => TO_3DF_C <= TEMP_3DF_C(2 * Width_To3DF - 1 DOWNTO Width_To3DF);
WHEN "11" => TO_3DF_C <= TEMP_3DF_C(Width_To3DF - 1 DOWNTO 0);
WHEN OTHERS => NULL;
                        END CASE;

                        CASE Mux_count IS
WHEN "00" => TO_3DF_D <= TEMP_3DF_D(4 * Width_To3DF - 1 DOWNTO 3 * Width_To3DF);
WHEN "01" => TO_3DF_D <= TEMP_3DF_D(3 * Width_To3DF - 1 DOWNTO 2 * Width_To3DF);
WHEN "10" => TO_3DF_D <= TEMP_3DF_D(2 * Width_To3DF - 1 DOWNTO Width_To3DF);
WHEN "11" => TO_3DF_D <= TEMP_3DF_D(Width_To3DF - 1 DOWNTO 0);
WHEN OTHERS => NULL;
                        END CASE;

                    ELSE
                    END IF;
                ELSE
                END IF;
        END PROCESS CLK_TRI;


END rtl;
```

VHDL source code of the Pipeline module

```
--------------------------------------------------------------------------------
--                                                                            --
-- Copyright (c) 1999 by 3D-Computing, Inc.                                   --
--                                                All rights reserved.        --
--  Author       : Dario Crosetto                                             --
--                                                                            --
-- This source file is FREE for Universities, National Labs and               --
-- International Labs of non-profit organizations provided that the           --
-- above statements are not removed from the file,                            --
-- that the revision history is updated if changes are introduced, and        --
-- that any derivative work contains the entire above mentioned notice.       --
--                                                                            --
--  Package name : FE_pipeline.vhd
--
--  Project   : Front-End Electronics Logic
--  Purpose   : This file implements the front-end pipelining.
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
--------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
LIBRARY work;
USE work.FE_config.ALL;

-----------------------------------------------------------------
--Entity Definition
-----------------------------------------------------------------

ENTITY FE_pipeline IS
   PORT (      clock, reset   : IN STD_LOGIC;
       EM_AS                     : IN std_logic_vector(adc_width - 1 DOWNTO 0);
       EM_BS                     : IN std_logic_vector(adc_width - 1 DOWNTO 0);
       EM_CS                     : IN std_logic_vector(adc_width - 1 DOWNTO 0);
       EM_DS                     : IN std_logic_vector(adc_width - 1 DOWNTO 0);
       HD_AS                     : IN std_logic_vector(adc_width - 1 DOWNTO 0);
       PS_AS                     : IN std_logic;
       PS_BS                     : IN std_logic;
       PS_CS                     : IN std_logic;
       PS_DS                     : IN std_logic;
       M1_BS                     : IN std_logic_vector(M1_width - 1 DOWNTO 0);
       M1_CS                     : IN std_logic_vector(M1_width - 1 DOWNTO 0);
       M1_FS                     : IN std_logic_vector(M1_width - 1 DOWNTO 0);
       M1_GS                     : IN std_logic_vector(M1_width - 1 DOWNTO 0);
       G_L0                      : IN std_logic;
       Time_ID                   : IN STD_LOGIC_VECTOR(Time_ID_width - 1 DOWNTO 0);

       TO_IN_FIFO                : OUT STD_LOGIC_VECTOR(fifo_width - 1 DOWNTO 0)
     );

END FE_pipeline;

-----------------------------------------------------------------
--ARCHITECTURE Definition
-----------------------------------------------------------------

ARCHITECTURE rtl OF FE_pipeline IS

       SIGNAL  PIPE_EM_A0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A2      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A3      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A4      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A5      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A6      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A7      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A8      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A9      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A10     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_A11     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_B0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_B1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_B2      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_B3      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_B4      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_B5      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
       SIGNAL  PIPE_EM_B6      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
```

```
        SIGNAL  PIPE_EM_B7      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_B8      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_B9      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_B10     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_B11     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C2      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C3      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C4      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C5      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C6      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C7      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C8      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C9      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C10     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_C11     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D2      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D3      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D4      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D5      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D6      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D7      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D8      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D9      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D10     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_EM_D11     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A2      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A3      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A4      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A5      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A6      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A7      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A8      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A9      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A10     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_HD_A11     : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_PS_A       : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_PS_B       : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_PS_C       : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_PS_D       : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_B0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_B1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_C0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_C1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_F0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_F1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_G0      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);
        SIGNAL  PIPE_M1_G1      : STD_LOGIC_VECTOR(PIPE_depth - 1 DOWNTO 0);

BEGIN

-- Pipelining the Data during Level-0 Trigger
    PIPE: PROCESS (clock, reset)
          BEGIN

            IF (reset = '0') THEN
                PIPE_EM_A0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A2(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A3(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A4(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A5(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A6(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A7(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A8(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A9(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A10(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_A11(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_EM_B0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B2(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B3(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B4(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
```

```
                PIPE_EM_B5(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B6(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B7(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B8(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B9(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B10(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_B11(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_EM_C0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C2(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C3(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C4(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C5(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C6(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C7(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C8(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C9(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C10(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_C11(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_EM_D0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D2(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D3(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D4(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D5(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D6(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D7(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D8(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D9(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D10(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_EM_D11(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_HD_A0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A2(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A3(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A4(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A5(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A6(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A7(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A8(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A9(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A10(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_HD_A11(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_PS_A(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_PS_B(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_PS_C(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_PS_D(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_M1_B0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_M1_B1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_M1_C0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_M1_C1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_M1_F0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_M1_F1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                PIPE_M1_G0(PIPE_depth - 1 DOWNTO 0) <= (others => '0');
                PIPE_M1_G1(PIPE_depth - 1 DOWNTO 0) <= (others => '0');

                        ELSIF (clock'EVENT AND clock = '1') THEN

        PIPE_EM_A0(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A0(PIPE_depth - 2 DOWNTO 0) & EM_AS(0);
        PIPE_EM_A1(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A1(PIPE_depth - 2 DOWNTO 0) & EM_AS(1);
        PIPE_EM_A2(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A2(PIPE_depth - 2 DOWNTO 0) & EM_AS(2);
        PIPE_EM_A3(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A3(PIPE_depth - 2 DOWNTO 0) & EM_AS(3);
        PIPE_EM_A4(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A4(PIPE_depth - 2 DOWNTO 0) & EM_AS(4);
        PIPE_EM_A5(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A5(PIPE_depth - 2 DOWNTO 0) & EM_AS(5);
        PIPE_EM_A6(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A6(PIPE_depth - 2 DOWNTO 0) & EM_AS(6);
        PIPE_EM_A7(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A7(PIPE_depth - 2 DOWNTO 0) & EM_AS(7);
        PIPE_EM_A8(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A8(PIPE_depth - 2 DOWNTO 0) & EM_AS(8);
        PIPE_EM_A9(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A9(PIPE_depth - 2 DOWNTO 0) & EM_AS(9);
        PIPE_EM_A10(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A10(PIPE_depth - 2 DOWNTO 0) & EM_AS(10);
```

```
PIPE_EM_A11(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_A11(PIPE_depth - 2 DOWNTO 0) & EM_AS(11);

PIPE_EM_B0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B0(PIPE_depth - 2 DOWNTO 0)  & EM_BS(0);
PIPE_EM_B1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B1(PIPE_depth - 2 DOWNTO 0)  & EM_BS(1);
PIPE_EM_B2(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B2(PIPE_depth - 2 DOWNTO 0)  & EM_BS(2);
PIPE_EM_B3(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B3(PIPE_depth - 2 DOWNTO 0)  & EM_BS(3);
PIPE_EM_B4(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B4(PIPE_depth - 2 DOWNTO 0)  & EM_BS(4);
PIPE_EM_B5(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B5(PIPE_depth - 2 DOWNTO 0)  & EM_BS(5);
PIPE_EM_B6(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B6(PIPE_depth - 2 DOWNTO 0)  & EM_BS(6);
PIPE_EM_B7(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B7(PIPE_depth - 2 DOWNTO 0)  & EM_BS(7);
PIPE_EM_B8(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B8(PIPE_depth - 2 DOWNTO 0)  & EM_BS(8);
PIPE_EM_B9(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_B9(PIPE_depth - 2 DOWNTO 0)  & EM_BS(9);
PIPE_EM_B10(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_B10(PIPE_depth - 2 DOWNTO 0) & EM_BS(10);
PIPE_EM_B11(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_B11(PIPE_depth - 2 DOWNTO 0) & EM_BS(11);

PIPE_EM_C0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C0(PIPE_depth - 2 DOWNTO 0)  & EM_CS(0);
PIPE_EM_C1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C1(PIPE_depth - 2 DOWNTO 0)  & EM_CS(1);
PIPE_EM_C2(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C2(PIPE_depth - 2 DOWNTO 0)  & EM_CS(2);
PIPE_EM_C3(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C3(PIPE_depth - 2 DOWNTO 0)  & EM_CS(3);
PIPE_EM_C4(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C4(PIPE_depth - 2 DOWNTO 0)  & EM_CS(4);
PIPE_EM_C5(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C5(PIPE_depth - 2 DOWNTO 0)  & EM_CS(5);
PIPE_EM_C6(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C6(PIPE_depth - 2 DOWNTO 0)  & EM_CS(6);
PIPE_EM_C7(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C7(PIPE_depth - 2 DOWNTO 0)  & EM_CS(7);
PIPE_EM_C8(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C8(PIPE_depth - 2 DOWNTO 0)  & EM_CS(8);
PIPE_EM_C9(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_C9(PIPE_depth - 2 DOWNTO 0)  & EM_CS(9);
PIPE_EM_C10(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_C10(PIPE_depth - 2 DOWNTO 0) & EM_CS(10);
PIPE_EM_C11(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_C11(PIPE_depth - 2 DOWNTO 0) & EM_CS(11);

PIPE_EM_D0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D0(PIPE_depth - 2 DOWNTO 0)  & EM_DS(0);
PIPE_EM_D1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D1(PIPE_depth - 2 DOWNTO 0)  & EM_DS(1);
PIPE_EM_D2(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D2(PIPE_depth - 2 DOWNTO 0)  & EM_DS(2);
PIPE_EM_D3(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D3(PIPE_depth - 2 DOWNTO 0)  & EM_DS(3);
PIPE_EM_D4(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D4(PIPE_depth - 2 DOWNTO 0)  & EM_DS(4);
PIPE_EM_D5(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D5(PIPE_depth - 2 DOWNTO 0)  & EM_DS(5);
PIPE_EM_D6(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D6(PIPE_depth - 2 DOWNTO 0)  & EM_DS(6);
PIPE_EM_D7(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D7(PIPE_depth - 2 DOWNTO 0)  & EM_DS(7);
PIPE_EM_D8(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D8(PIPE_depth - 2 DOWNTO 0)  & EM_DS(8);
PIPE_EM_D9(PIPE_depth - 1 DOWNTO 0) <=  PIPE_EM_D9(PIPE_depth - 2 DOWNTO 0)  & EM_DS(9);
PIPE_EM_D10(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_D10(PIPE_depth - 2 DOWNTO 0) & EM_DS(10);
PIPE_EM_D11(PIPE_depth - 1 DOWNTO 0) <= PIPE_EM_D11(PIPE_depth - 2 DOWNTO 0) & EM_DS(11);

PIPE_HD_A0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A0(PIPE_depth - 2 DOWNTO 0)  & HD_AS(0);
PIPE_HD_A1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A1(PIPE_depth - 2 DOWNTO 0)  & HD_AS(1);
PIPE_HD_A2(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A2(PIPE_depth - 2 DOWNTO 0)  & HD_AS(2);
PIPE_HD_A3(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A3(PIPE_depth - 2 DOWNTO 0)  & HD_AS(3);
PIPE_HD_A4(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A4(PIPE_depth - 2 DOWNTO 0)  & HD_AS(4);
PIPE_HD_A5(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A5(PIPE_depth - 2 DOWNTO 0)  & HD_AS(5);
PIPE_HD_A6(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A6(PIPE_depth - 2 DOWNTO 0)  & HD_AS(6);
PIPE_HD_A7(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A7(PIPE_depth - 2 DOWNTO 0)  & HD_AS(7);
PIPE_HD_A8(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A8(PIPE_depth - 2 DOWNTO 0)  & HD_AS(8);
PIPE_HD_A9(PIPE_depth - 1 DOWNTO 0) <=  PIPE_HD_A9(PIPE_depth - 2 DOWNTO 0)  & HD_AS(9);
PIPE_HD_A10(PIPE_depth - 1 DOWNTO 0) <= PIPE_HD_A10(PIPE_depth - 2 DOWNTO 0) & HD_AS(10);
PIPE_HD_A11(PIPE_depth - 1 DOWNTO 0) <= PIPE_HD_A11(PIPE_depth - 2 DOWNTO 0) & HD_AS(11);

PIPE_PS_A(PIPE_depth - 1 DOWNTO 0) <=  PIPE_PS_A(PIPE_depth - 2 DOWNTO 0)  & PS_AS;

PIPE_PS_B(PIPE_depth - 1 DOWNTO 0) <=  PIPE_PS_B(PIPE_depth - 2 DOWNTO 0)  & PS_BS;

PIPE_PS_C(PIPE_depth - 1 DOWNTO 0) <=  PIPE_PS_C(PIPE_depth - 2 DOWNTO 0)  & PS_CS;

PIPE_PS_D(PIPE_depth - 1 DOWNTO 0) <=  PIPE_PS_D(PIPE_depth - 2 DOWNTO 0)  & PS_DS;

PIPE_M1_B0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_B0(PIPE_depth - 2 DOWNTO 0)  & M1_BS(0);
PIPE_M1_B1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_B1(PIPE_depth - 2 DOWNTO 0)  & M1_BS(1);

PIPE_M1_C0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_C0(PIPE_depth - 2 DOWNTO 0)  & M1_CS(0);
PIPE_M1_C1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_C1(PIPE_depth - 2 DOWNTO 0)  & M1_CS(1);

PIPE_M1_F0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_F0(PIPE_depth - 2 DOWNTO 0)  & M1_FS(0);
PIPE_M1_F1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_F1(PIPE_depth - 2 DOWNTO 0)  & M1_FS(1);

PIPE_M1_G0(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_G0(PIPE_depth - 2 DOWNTO 0)  & M1_GS(0);
PIPE_M1_G1(PIPE_depth - 1 DOWNTO 0) <=  PIPE_M1_G1(PIPE_depth - 2 DOWNTO 0)  & M1_GS(1);

        ELSE
         END IF;

    END PROCESS PIPE;
```

```
    TO_IN_FIFO(fifo_width - 1 DOWNTO 0) <=
                PIPE_EM_A0(127) & PIPE_EM_A1(127) & PIPE_EM_A2(127) & PIPE_EM_A3(127) &
                PIPE_EM_A4(127) & PIPE_EM_A5(127) & PIPE_EM_A6(127) & PIPE_EM_A7(127) &
                PIPE_EM_A8(127) & PIPE_EM_A9(127) & PIPE_EM_A10(127) & PIPE_EM_A11(127) &

                PIPE_EM_B0(127) & PIPE_EM_B1(127) & PIPE_EM_B2(127)  & PIPE_EM_B3(127) &
                PIPE_EM_B4(127) & PIPE_EM_B5(127) & PIPE_EM_B6(127)  & PIPE_EM_B7(127) &
                PIPE_EM_B8(127) & PIPE_EM_B9(127) & PIPE_EM_B10(127) & PIPE_EM_B11(127) &

                PIPE_EM_C0(127) & PIPE_EM_C1(127) & PIPE_EM_C2(127)  & PIPE_EM_C3(127) &
                PIPE_EM_C4(127) & PIPE_EM_C5(127) & PIPE_EM_C6(127)  & PIPE_EM_C7(127) &
                PIPE_EM_C8(127) & PIPE_EM_C9(127) & PIPE_EM_C10(127) & PIPE_EM_C11(127) &

                PIPE_EM_D0(127) & PIPE_EM_D1(127) & PIPE_EM_D2(127)  & PIPE_EM_D3(127) &
                PIPE_EM_D4(127) & PIPE_EM_D5(127) & PIPE_EM_D6(127)  & PIPE_EM_D7(127) &
                PIPE_EM_D8(127) & PIPE_EM_D9(127) & PIPE_EM_D10(127) & PIPE_EM_D11(127) &

                PIPE_HD_A0(127) & PIPE_HD_A1(127) & PIPE_HD_A2(127)  & PIPE_HD_A3(127) &
                PIPE_HD_A4(127) & PIPE_HD_A5(127) & PIPE_HD_A6(127)  & PIPE_HD_A7(127) &
                PIPE_HD_A8(127) & PIPE_HD_A9(127) & PIPE_HD_A10(127) & PIPE_HD_A11(127) &

                PIPE_PS_A(127) & PIPE_PS_B(127) & PIPE_PS_C(127) & PIPE_PS_D(127) &

                PIPE_M1_B0(127) & PIPE_M1_B1(127) & PIPE_M1_C0(127) & PIPE_M1_C1(127) &
                PIPE_M1_F0(127) & PIPE_M1_F1(127) & PIPE_M1_G0(127) & PIPE_M1_G1(127) &
                Time_ID(Time_ID_width - 1 DOWNTO 0);


END rtl;
```

VHDL source code of the FIFO and Serializer module

```
--------------------------------------------------------------------------
--                                                                      --
-- Copyright (c) 1999 by 3D-Computing, Inc.                             --
--                                          All rights reserved.        --
--  Author       : Dario Crosetto                                       --
--                                                                      --
-- This source file is FREE for Universities, National Labs and         --
-- International Labs of non-profit organizations provided that the     --
-- above statements are not removed from the file,                      --
-- that the revision history is updated if changes are introduced, and  --
-- that any derivative work contains the entire above mentioned notice. --
--                                                                      --
--  Package name : FE_fifo.vhd                                          --
--                                                                      --
--  Project   : Front-End Electronics Logic                             --
--  Purpose   : This file implements the front-end Level-0 buffering    --
--              into the FIFOs and sends the data to the DAQ and to     --
--              the higher level trigger.                               --
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
--------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
LIBRARY work;
USE work.FE_config.ALL;

LIBRARY ORCA3;
USE ORCA3.ALL;

-----------------------------------------------------------------
--Entity Definition
-----------------------------------------------------------------

ENTITY FE_fifo IS
   PORT (
        Reset              : IN STD_LOGIC;
        clock              : IN STD_LOGIC;          -- FIFO write clock
        G_L0               : IN std_logic;          -- FIFO write enable
        EnOutData          : IN std_logic;          -- FIFO read enable
        TO_IN_FIFO         : IN std_logic_vector(fifo_width - 1 DOWNTO 0); --
        clk_x2             : IN STD_LOGIC;   -- Replaced by the internal PLL --
```

```
        fifo_empty          : OUT std_logic;
        fifo_full           : OUT std_logic;
        diff_fifo_addr      : OUT std_logic_vector(fifo_depth - 1 downto 0);
        DataOut             : OUT std_logic;  -- Out data (1-bit)
        St_Burst            : OUT std_logic
        );

END FE_fifo;

-------------------------------------------------------------------
--ARCHITECTURE Definition
-------------------------------------------------------------------

ARCHITECTURE rtl OF FE_fifo IS

    SIGNAL  int_fifo_cnt           : std_logic_vector(fifo_depth downto 0);
    SIGNAL  wr_en                  : std_logic;
    SIGNAL  int_fifo_full          : std_logic;
    SIGNAL  int_fifo_empty         : std_logic;
    SIGNAL  int_fifo_rdaddr        : std_Logic_vector(fifo_depth - 1 DOWNTO 0);
    SIGNAL  int_fifo_wraddr        : std_Logic_vector(fifo_depth - 1 DOWNTO 0);
    SIGNAL  temp_out               : std_Logic_vector(fifo_width - 1 DOWNTO 0);
    SIGNAL  int_clk_x2             : STD_LOGIC;      -- Replaced by the internal PLL

--Change depth
TYPE    regfile IS ARRAY (0 to 2**fifo_depth - 1) of std_logic_vector
        (fifo_width - 1 DOWNTO 0);
    SIGNAL  next_file              : regfile;

BEGIN

int_clk_x2 <= clk_x2;

-- FIFO write address
PROCESS (reset,clock,G_L0)
BEGIN
         wr_en <= '1';
  IF (reset = '0') THEN
     int_fifo_wraddr <= (others => '0');

    ELSIF clock'EVENT AND clock = '1' THEN

      IF G_L0 = '1'  AND int_fifo_full = '0' THEN
          int_fifo_wraddr <= int_fifo_wraddr + 1;
          wr_en <= '0';
    END IF;
  END IF;
END PROCESS;


-- FIFO read address
PROCESS (reset,clock,EnOutData)
BEGIN
  IF (reset = '0') THEN
     int_fifo_rdaddr <= (others => '0');

    ELSIF (clock'Event AND clock = '1') THEN

      IF EnOutData = '1' AND int_fifo_empty = '0' THEN
          int_fifo_rdaddr <= int_fifo_rdaddr + 1;
    END IF;
  END IF;
END PROCESS;


-- fifo full/empty logic
PROCESS (clock, reset)
BEGIN
  IF reset = '0' THEN
     int_fifo_cnt <=  (OTHERS => '0');

    ELSIF (clock'EVENT AND clock = '1') THEN

      IF G_L0 = '1' AND int_fifo_full = '0' THEN
            int_fifo_cnt <= int_fifo_cnt + 1;
         END IF;
           ELSE
```

```
          IF EnOutData = '1' AND int_fifo_empty = '0' THEN
            int_fifo_cnt <= int_fifo_cnt - '1';
       END IF;
  END IF;

END PROCESS;

-- writes data validated by the Global Level-0 (G_L0) trigger decision
--                          unit into the FIFO.
comb_proc: PROCESS (G_L0,TO_IN_FIFO   , int_fifo_wraddr)

BEGIN
  IF (reset = '0') THEN
            next_file <=  (OTHERS => (OTHERS => '0'));

       ELSIF   (wr_en = '1' AND G_L0 = '1' AND int_fifo_full = '0') THEN
                next_file(CONV_INTEGER(int_fifo_wraddr))   <= TO_IN_FIFO;
  END IF;
END PROCESS;


-- sends data out of FIFO seriallly from DataOut pin
         DataOut <= temp_out(fifo_width - 1);


-- sends Start_burst_out signal synchronized with first bit of string
PROCESS (reset,int_clk_x2,EnOutData,int_fifo_empty)
BEGIN
  IF (reset = '0') THEN
         St_burst <= '0';

    ELSIF (int_clk_x2'Event AND int_clk_x2 = '1') THEN

      IF EnOutData = '1' AND int_fifo_empty = '0' THEN
            St_burst <= '1';
          ELSE
            St_burst <= '0';
       END IF;
  END IF;
END PROCESS;


-- Read out values from the FIFO when receiving "EnOutData" signal
-- from the Level-1 Trigger (or in more details) loads temp_out with
-- FIFO value pointed by read_fifo_address
-- ELSE load temp_out with shifted value.
PROCESS (reset,int_clk_x2,EnOutData)  -- MSB first shift register.
BEGIN
  IF (reset = '0') THEN
     temp_out <= (others => '0');

  ELSIF (int_clk_x2'EVENT AND int_clk_x2 = '1') THEN
    IF (EnOutData = '1' AND int_fifo_empty = '0') THEN
         temp_out <= next_file(CONV_INTEGER(int_fifo_rdaddr));
      ELSE

         temp_out <= temp_out(fifo_width - 2 downto 0) & '0';
    END IF;
  END IF;
END PROCESS;


    diff_fifo_addr <= int_fifo_cnt(fifo_depth - 1 DOWNTO 0);
    int_fifo_full <= int_fifo_cnt(fifo_depth);
    int_fifo_empty <= '1' WHEN int_fifo_cnt = "000000" ELSE '0';
    fifo_full <= int_fifo_full;
    fifo_empty <= int_fifo_empty;


  END rtl;
```

APPENDIX B: VHDL CODE OF BEHAVIORAL TESTBENCH MODULE OF THE FRONT-END CIRCUIT

```
--------------------------------------------------------------------------------
```

```
--                                                                          --
-- Copyright (c) 1999 by 3D-Computing, Inc.                                 --
--                                                 All rights reserved.     --
--  Author       : Dario Crosetto                                           --
--                                                                          --
-- This source file is FREE for Universities, National Labs and             --
-- International Labs of non-profit organizations provided that the         --
-- above statements are not removed from the file,                          --
-- that the revision history is updated if changes are introduced, and      --
-- that any derivative work contains the entire above mentioned notice.     --
--                                                                          --
--  Package name : FE_testbench.vhd
--
--  Project   : Front-End Electronics Logic
--  Purpose   : This file implements the testbench for preliminary
--              testing of the chip that
--              performs the front-end signals synchronization,
--              pipelining, derandomizing, trigger word formatter.
--              The code is for four trigger channels.
--
--  Revisions :       D. Crosetto    2/12/99 created for one channel;
--                    D. Crosetto    4/23/99 modified for 4 channels;
--
------------------------------------------------------------------------------

LIBRARY IEEE;
USE IEEE.std_logic_1164.ALL;
USE IEEE.std_logic_arith.ALL;
USE IEEE.std_logic_unsigned.ALL;
LIBRARY work;
USE work.FE_config.ALL;

---------------------------------------------------------------------
--Entity Definition
---------------------------------------------------------------------

ENTITY FE_testbench IS
     END FE_testbench;

ARCHITECTURE behave OF FE_testbench IS

COMPONENT FE_top
   PORT (      clock, reset   : IN std_logic;
        EM_A                   : IN std_logic_VECTOR(adc_width - 1 downto 0);
        EM_B                   : IN std_logic_VECTOR(adc_width - 1 downto 0);
        EM_C                   : IN std_logic_VECTOR(adc_width - 1 downto 0);
        EM_D                   : IN std_logic_VECTOR(adc_width - 1 downto 0);
        HD_A                   : IN std_logic_VECTOR(adc_width - 1 downto 0);
        PS_A                   : IN std_logic;
        PS_B                   : IN std_logic;
        PS_C                   : IN std_logic;
        PS_D                   : IN std_logic;
        M1_A                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        M1_B                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        M1_C                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        M1_D                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        M1_E                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        M1_F                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        M1_G                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        M1_H                   : IN std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        Time_ID                : IN std_logic_VECTOR(Time_ID_width - 1 DOWNTO 0);
        G_L0                   : IN std_logic;
        EnInData               : IN std_logic;
        EnOutData              : IN std_logic;
        clk_x2                 : IN STD_LOGIC; -- Replaced by the internal PLL –
        clk_x4                 : IN STD_LOGIC; -- Replaced by the internal PLL –

        fifo_empty             : OUT std_logic;
        fifo_full              : OUT std_logic;
        diff_fifo_addr         : OUT std_logic_vector(fifo_depth - 1 DOWNTO 0);
        LOAD_3DF_A             : OUT std_logic;
        TO_3DF_A               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_B             : OUT std_logic;
        TO_3DF_B               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_C             : OUT std_logic;
        TO_3DF_C               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        LOAD_3DF_D             : OUT std_logic;
        TO_3DF_D               : OUT STD_LOGIC_VECTOR(Width_To3DF - 1 DOWNTO 0);
        DataOut                : OUT std_logic;
```

```
        St_Burst                : OUT std_logic
        );

END COMPONENT;

        SIGNAL clock           :  std_logic;
        SIGNAL reset           :  std_logic;
        SIGNAL clk_x2          :  std_logic;  -- Replaced by the internal PLL --
        SIGNAL clk_x4          :  std_logic;  -- Replaced by the internal PLL --
        SIGNAL int_clock       :  std_logic;
        SIGNAL int_clk_x2      :  std_logic;  -- Replaced by the internal PLL --
        SIGNAL int_clk_x4      :  std_logic;  -- Replaced by the internal PLL --
        SIGNAL int_reset       :  std_logic;
        SIGNAL EM_A            :  std_logic_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL EM_B            :  std_logic_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL EM_C            :  std_logic_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL EM_D            :  std_logic_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL HD_A            :  std_logic_VECTOR(adc_width - 1 DOWNTO 0);
        SIGNAL PS_A            :  std_logic;
        SIGNAL PS_B            :  std_logic;
        SIGNAL PS_C            :  std_logic;
        SIGNAL PS_D            :  std_logic;
        SIGNAL M1_A            :  std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL M1_B            :  std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL M1_C            :  std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL M1_D            : std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL M1_E            :  std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL M1_F            :  std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL M1_G            :  std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL M1_H            :  std_logic_VECTOR(M1_width - 1 DOWNTO 0);
        SIGNAL Time_ID         :  std_logic_VECTOR(Time_ID_width - 1 DOWNTO 0);
        SIGNAL Time_ID_count   :  std_logic_VECTOR(Time_ID_width - 1 DOWNTO 0);
        SIGNAL G_L0            :  std_logic;
        SIGNAL EnInData        :  std_logic;
        SIGNAL EnOutData       :  std_logic;

        CONSTANT ADC_23        : std_logic_vector (11 DOWNTO 0) := "000000010111";
        CONSTANT ADC_145       : std_logic_vector (11 DOWNTO 0) := "000010010001";

        CONSTANT M1_2          : std_logic_vector(1 DOWNTO 0)         := "10";
        CONSTANT M1_1          : std_logic_vector(1 DOWNTO 0)         := "01";

        CONSTANT clock_period           : TIME :=   24    ns;
        CONSTANT clk_x2_period          : TIME :=   12    ns;        -- Replaced by PLL --
        CONSTANT clk_x4_period          : TIME :=    6    ns;        -- Replaced by PLL --


 BEGIN

--setting up the clocks
clock <= int_clock;
int_clock <= '1' WHEN int_clock = 'U'
                ELSE NOT(int_clock) AFTER clock_period/2;

clk_x2 <= int_clk_x2;                                       -- Replaced by the internal PLL --
int_clk_x2 <= '1' WHEN int_clk_x2 = 'U'
                 ELSE NOT(int_clk_x2) AFTER clk_x2_period/2;

clk_x4 <= int_clk_x4;                                       -- Replaced by the internal PLL --
int_clk_x4 <= '1' WHEN int_clk_x4 = 'U'
                 ELSE NOT(int_clk_x4) AFTER clk_x4_period/2;

--setting up the reset
reset <= int_reset;
int_reset <= '0' WHEN int_reset = 'U' ELSE '1' AFTER clock_period*4;


PROCESS BEGIN


-- initializing Time_ID count
Time_ID_count(Time_ID_width - 1 DOWNTO 0) <= (others => '0');
G_L0              <= '0';
EnOutData       <= '0';

        for j in 10 to 16 loop for k in 0 to 3 loop  --

EM_A              <= ADC_23;
EM_B              <= ADC_23;
EM_C              <= ADC_23;
```

```
EM_D               <= ADC_23;
HD_A               <= ADC_23;
PS_A               <=  '1';
PS_B               <=  '1';
PS_C               <=  '1';
PS_D               <=  '1';
M1_A               <= M1_2;
M1_B               <= M1_2;
M1_C               <= M1_2;
M1_D               <= M1_2;
M1_E               <= M1_2;
M1_F               <= M1_2;
M1_G               <= M1_2;
M1_H               <= M1_2;
G_L0               <= '1';
Time_ID            <= Time_ID_count;
EnInData           <= '1';

WAIT UNTIL int_clock'EVENT AND int_clock = '0' ;
Time_ID_count <= (Time_ID_count + "00000001");

end loop; end loop;


G_L0               <=  '0';

       for i in 2 to 255 loop for j in 30 to 51 loop for k in 0 to 3 loop

EM_A               <= ADC_145;
EM_B               <= ADC_145;
EM_C               <= ADC_145;
EM_D               <= ADC_145;
HD_A               <= ADC_145;
PS_A               <=  '1';
PS_B               <=  '1';
PS_C               <=  '1';
PS_D               <=  '1';
M1_A               <= M1_1;
M1_B               <= M1_1;
M1_C               <= M1_1;
M1_D               <= M1_1;
M1_E               <= M1_1;
M1_F               <= M1_1;
M1_G               <= M1_1;
M1_H               <= M1_1;
Time_ID            <= Time_ID_count;
EnInData           <= '1';

WAIT UNTIL int_clock'EVENT AND int_clock = '0' ;
Time_ID_count <= (Time_ID_count + "00000001");

end loop; end loop;

EnOutData       <= '1', '0' AFTER clock_period ;
G_L0               <= '1', '0' AFTER clock_period ;

       end loop;

wait;

end process;

FE_top_inst: FE_top

   PORT MAP (
    clock                  => clock,
    reset                  => reset,
       EM_A                => EM_A,
       EM_B                => EM_B,
       EM_C                => EM_C,
       EM_D                => EM_D,
       HD_A                => HD_A,
       PS_A                => PS_A,
       PS_B                => PS_B,
       PS_C                => PS_C,
       PS_D                => PS_D,
       M1_A                => M1_A,
       M1_B                => M1_B,
       M1_C                => M1_C,
```

```
            M1_D                => M1_D,
            M1_E                => M1_E,
            M1_F                => M1_F,
            M1_G                => M1_G,
            M1_H                => M1_H,
            Time_ID             => Time_ID,
            G_L0                => G_L0,
            EnInData            => EnInData,
            EnOutData           => EnOutData,
            clk_x2              => clk_x2,
            clk_x4              => clk_x4,

            fifo_empty          => OPEN,
            fifo_full           => OPEN,
            diff_fifo_addr      => OPEN,
            LOAD_3DF_A          => OPEN,
            TO_3DF_A            => OPEN,
            LOAD_3DF_B          => OPEN,
            TO_3DF_B            => OPEN,
            LOAD_3DF_C          => OPEN,
            TO_3DF_C            => OPEN,
            LOAD_3DF_D          => OPEN,
            TO_3DF_D            => OPEN,
            DataOut             => OPEN,
            St_Burst            => OPEN
            );

    END behave;
```

APPENDIX C: FRONT-END CIRCUIT MAPPED TO LUCENT TECHNOLOGIES ORCA OR3T30 FPGA

Web site .edn file
See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

Front-End signals PIN assignment to OR3T30

See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

MAP to OR3T30 Report file

See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

Place and Route Report file

See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

APPENDIX D: BACK-ANNOTATED VHDL CODE OBTAINED FROM ORCA FOUNDRY TOOLS

See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

APPENDIX E: BACK-ANNOTATED TIMING INFORMATION OBTAINED FROM ORCA FOUNDRY TOOLS

See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

APPENDIX F: TRACE OUTPUT FROM ORCA FOUNDRY TOOLS REPORTING TIMING INFORMATION

See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

APPENDIX G: BIT FILE GENERATED FOR ORCA OR3T30 FPGA CHIP FOR CIRCUIT TESTING ON REAL HARDWARE

See web site:
http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm

References

[1] S. Conetti, D. Crosetto, “The LHCb Calorimeter Trigger and its implementation with the 3D-Flow system,” LHCb 98-13
[2] G. Corti, B. Cox, and D. Crosetto, “An Implementation of the L0 Muon Trigger Using the 3D-Flow,” note LHCb 98-001
[3] LHCb Technical Proposal, CERN/LHCC 98-4 20 February 1998.
[4] http://rapid:rapid@catalog.rapid.org/scripts/isynch.dll?panel=TclScript&file=sipcTop.tcl (for Virtual Components catalog)
[5] http://lhcb.cern.ch/notes/99-004.ps D. Crosetto, “LHCb base-line level-0 trigger 3D-Flow implementation.” LHCb 99-004, TRIG, 17 February 1999.
[6] http://lhcb.cern.ch/electronics/simulation/vhdlmodels.htm