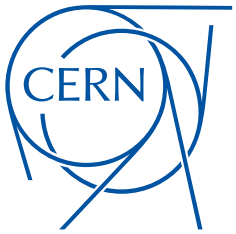
CONV-TTL-BLO HDL Guide

Gateware v2.1
April 8, 2014



Theodor-Adrian Stana (CERN/BE-CO-HT)

Revision history

Date	Version	Change
04-07-2013	0.1	First draft
26-07-2013	0.2	Second draft
07-08-2013	1.02	Added pulse rejection to *conv_pulse_gen*
14-08-2013	1.02	Changed name of *elma_i2c* to *vbcp_wb*
20-11-2013	1.04	Added MultiBoot module, gateware release v1.0
14-02-2014	2.00	Version 2.0 of gateware, with diagnostics support including: unique board ID and temperature readout, input pulse counters, pulse time-tagging and manual pulse triggering.
08-04-2014	2.10	Version 2.1 of gateware, bringing down the max. allowed input pulse frequency, changing the ERR LED behaviour, adding system errors and changing the pulse timetag FIFO for a timetag ring buffer

Contents

List of Figures

List of Tables

List of Abbreviations

DAC	Digital-to-Analog Converter
FPGA	Field-Programmable Gate Array
FF	Flip-Flop
FSM	Finite-State Machine
IC	Integrated Circuit
I^2C	Inter-Integrated Circuit (bus)
PLL	Phase-Locked Loop
RTM	Rear-Transition Module
RAM	Random-Access Memory
SPI	Serial Peripheral Interface
SysMon	(ELMA) System Montior
VCXO	Voltage-controlled oscillator
WRPC	White-Rabbit PTP Core

1 Introduction

This document details the HDL implemented on the Spartan-6 FPGA on the CONV-TTL-BLO board. The HDL (mostly implemented in VHDL) handles the following aspects of the CONV-TTL-BLO capabilities:

- pulse detection (on pulse rising edge)
- fixed-width pulse generation with pulse rejection
- diagnostics via I^2C
  - converter board ID
  - gateware version
  - unique board ID and temperature readout
  - state of on-board switches and RTM detection lines
  - input pulse counters
  - input pulse time-tagging
  - manual pulse triggering
- remote reprogramming via I^2C

Figure 1 shows a simplified block diagram of the HDL gateware. The blocks in this figure implemented as part of the CONV-TTL-BLO gateware are presented in the sections that follow.

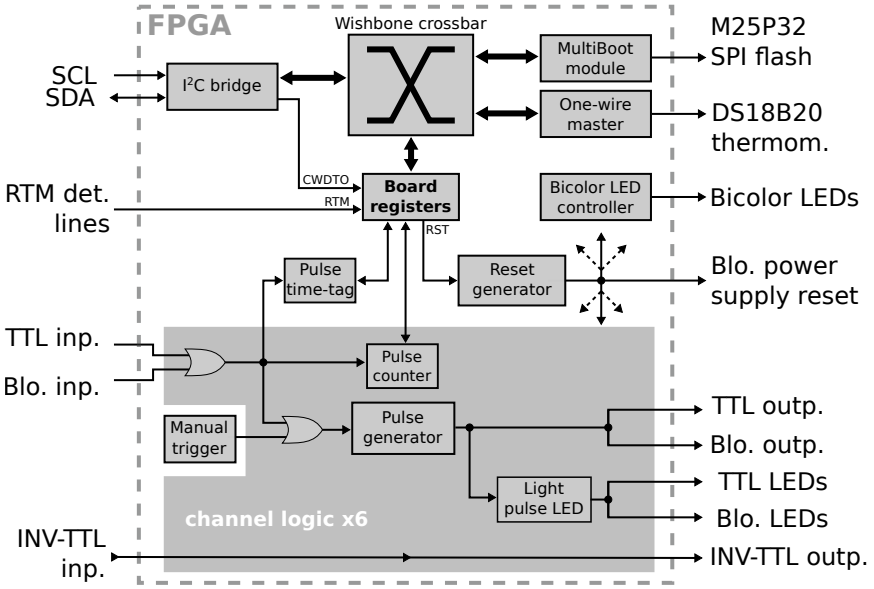


Figure 1: Block diagram of FPGA gateware

Additional documentation

- CONV-TTL-BLO User Guide [1]
- CONV-TTL-BLO Hardware Guide [2]

2 FPGA clocks

There are two clock signals input to the FPGA (Figure 2). The first is a 20 MHz signal from a VCXO. The second clock signal with a frequency of 125 MHz is generated on-board via a Texas Instruments PLL IC from a 25 MHz VCXO.

Two DACs are provided on-board for controlling the two VCXOs. The DACs can be controlled via SPI, and if White Rabbit is available, the White Rabbit PTP Core (WRPC) can control these DACs to discipline these clocks.

Table 1 lists the clock domains used in the gateware.

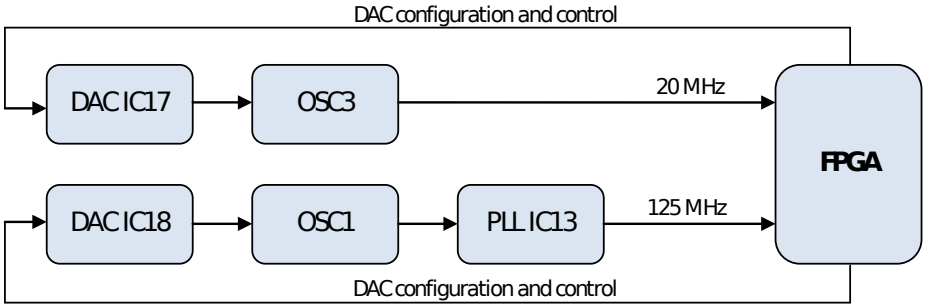


Figure 2: FPGA clock inputs

Table 1: Clock domains

Clock domain	Frequency	Comments
clk_20_vcxo_i	20 MHz	Global clock input to all sequential logic.
clk_125	125 MHz	Time-tagging logic and WR reference clock.

3 Memory-mapped peripherals

This section details the various peripherals mapped on the internal Wishbone bus. Access to these peripherals is made through the two serial lines on the VME P1 connector (SERCLK, SERDAT). A protocol based on I^2C is used to access these peripherals. The protocol, as well as the bridge component translating I^2C accesses into Wishbone accesses, are defined in the bridge component's documentation.

The complete memory map of the gateware can be found in Appendix A.

3.1 I^2C to Wishbone bridge

The *wb_i2c_bridge* module implements a bridge to translate I^2C accesses on the VME P1 connector into Wishbone accesses on the FPGA. The module provides one I^2C slave interface for connecting to an ELMA SysMon and one Wishbone master interface.

Details about the module's implementation can be found in its documentation.

3.2 Converter board registers

A set of registers are implemented as general-purpose registers for converter boards. These are status and control registers implemented utilizing *wbgen2* [3]. Appendix A.1 presents the converter board registers.

On the status registers side, there is one general status register (SR – see Appendix A.1.2) that contains details about the gateware version, the state of the on-board switches and RTM detection lines, as well as the state of the communication watchdog timer. Then, there are six pulse counter registers (CHxPCR – see Appendix A.1), one per each channel, which are updated with the current values of the input pulse counters. These are followed by the time-tagging logic (Section 9) registers.

The logic also contains one control register (CR – see Appendix A.1.3), which contains two bits for remotely resetting the FPGA logic and six bits used by the manual pulse triggering (Section 8).

3.3 MultiBoot control

The MultiBoot module offers the remote reprogramming capabilities for the CONV-TTL-BLO board. It offers a set of registers for controlling writing a bitstream to the M25P32 flash chip and for issuing the remote reprogramming command.

For information on the module, refer to its documentation. The memory map of the module is also present in this manual, for quick reference (see Appendix A.2).

3.4 One-wire master

The one-wire master provides access to the DS18B20 thermometer chip [4] on the CONV-TTL-BLO. It provides two registers for software control of the module.

Note that the FPGA does not control the one-wire thermometer line in any way. Accessing the thermometer is done through software only.

More details about how to access the one-wire master module can be found in its documentation [5].

4 Reset generator

Entity	*reset_gen*	
Generics	*g_reset_time*	Reset time in *clk_i* cycles
Ports	*clk_i*	Clock signal
	rst_i	Active-high reset input
	rst_n_o	Active-low reset output
Usage	Global reset generation	100 *ms* reset

The reset generator module (*reset_gen*) implemented inside the FPGA generates a predefined-width reset signal when power is applied to the FPGA, or when an external reset is triggered via the *rst_i* pin.

When a power-on reset occurs on the Xilinx FPGA, a counter inside the *reset_gen* module starts counting up. While this counter is counting up, the active-low reset signal is kept low, resetting synchronous logic inside the FPGA. When the counter reaches the value of the reset width (specified via the *g_reset_time* generic), the reset signal is de-asserted, the counter is disabled and the *reset_gen* module remains inactive.

The module reactivates on the power-on reset, or when a reset is triggered externally, via the *rst_i* pin. The *rst_i* pin is tied in the design to the second bit in the control register (CR – see Appendix A.1.3), which has to be first unlocked by writing the RST_UNLOCK bit. Both these registers are implemented in the top-level file of the design.

Note that the VHDL of this module is Xilinx and XST-specific and porting to a different FPGA architecture is not guaranteed to provide the same results. The *reset_gen* module has an initial value set for the counter signal after power-up, which is guaranteed by XST to be set after the FPGA's GSR signal is de-asserted.

By default, the reset time is set to 100 ms.

5 Bicolor LED controller

Entity	*bicolor_led_ctrl*	
Generics	*g_NB_COLUMN*	Number of columns
	g_NB_LINE	Number of lines
	g_CLK_FREQ	Frequency (in Hz) of *clk_i* signal
	g_REFRESH_RATE	LED refresh rate (in Hz)
Ports	*rst_n_i*	Active-low reset input
	clk_i	Clock signal input
	led_intensity_i(6..0)	7-bit LED intensity vector
	led_state_i(..)	LED state vector, two bits per LED
	column_o(..)	LED column vector, one bit per column
	line_o(..)	LED line vector, one bit per line
	line_oen_o(..)	LED line enable vector, one bit per line
Usage	Light bicolor LEDS	

The *bicolor_led_ctrl* block controls the lighting of a bicolor LED matrix. Based on the refresh rate given via the *g_REFRESH_RATE* generic, the clock frequency (*g_CLK_FREQ* generic) and the number of lines and columns, the module lights each LED in the LED matrix sequentially at the refresh rate given by the user.

Figure 3 shows an example of controlling a three-line, two-column red-and-green LED matrix. The FPGA ouputs for the columns (C) are connected to buffers and serial resistors and then to the LEDs. The FPGA outputs for lines (L) are connected to tri-state buffers and then to the LEDs. The FPGA outputs for line output enables (L_OEN) are connected to the output enable of the tri-state buffers.

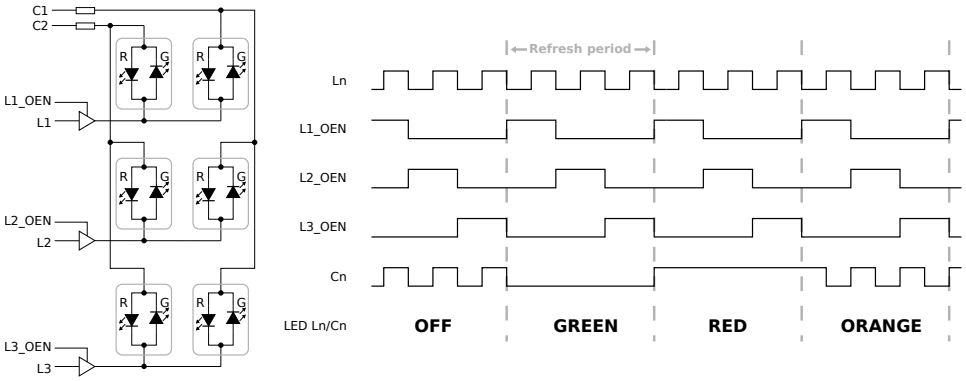


Figure 3: 3x2 bicolor LED matrix control

The two-bit *led_state_i* vector can be used to control the color of each LED. Table 2 lists the values that should be input on *led_state_i* to get the needed color, as well as constant definitions provided in the *bicolor_led_ctrl_pkg.vhd* file for setting the color of the LED via *led_state_i*.

Table 2: LED state input

State	Constant	Value
Off	c_LED_OFF	00
Green	c_LED_GREEN	01
Red	c_LED_RED	10
Orange	c_LED_RED_ORANGE	11

Each LED's two-bit state is connected to *led_state_i* on a column-first, line-second basis.

5.1 Board-level view

There are twelve bicolor LEDs on the CONV-TTL-BLO; they are connected in a two-line, six-column pattern controlled by a *bicolor_led_ctrl* block. Table 3 shows the *led_state_i* connections for the bicolor status LEDs in the CONV-TTL-BLO gateware.

Table 3: LED state vector connections in the gateware

Line	Column	LED	LED state bits
1	1	WHITE_RABBIT_ADDR	*1..0*
1	2	WHITE_RABBIT_GMT	*3..2*
1	3	WHITE_RABBIT_LINK	*5..4*
1	4	WHITE_RABBIT_OK	*7..6*
1	5	MULTICAST_ADDR_1	*9..8*
1	6	MULTICAST_ADDR_2	*11..10*
2	1	I2C	*13..12*
2	2	TTL	*15..14*
2	3	ERR	*17..16*
2	4	PW	*19..18*
2	5	MULTICAST_ADDR_4	*21..20*
2	6	MULTICAST_ADDR_8	*23..22*

The states of the used LEDs can be found in Table 1 of the CONV-TTL-BLO User Guide [1]. They are controlled by combinatorial multiplexers. The selection signals to these multiplexers are set throughout the logic.

6 Pulse generator

Entity	*ctblo_pulse_gen*	
Generics	*g_pwidth*	Width of the output pulse in *clk_i* cycles
	g_duty_cycle_div	Duty cycle divider ratio
Ports	*clk_i*	Clock signal
	rst_n_i	Active-low reset signal
	en_i	Pulse generator enable
	gf_en_n_i	Active-low glitch filter enable
	trig_a_i	Pulse trigger
	pulse_err_p_o	Pulse error
	pulse_o	Pulse output
Usage	Output pulse	1.2 μs pulses with min. period of 6 μs and one-cycle wide glitch filter

The *conv_pulse_gen* block generates pulses on the rising edge of the *trig_a_i* input. The pulse width is configurable via the *g_pwidth* generic. The block also incorporates a glitch filter with a configurable length (*g_gf_len*) that can be used to avoid pulses generated because of glitches at the *trig_a_i* input.

Pulse widths at the output are limited internally to 1/5 duty cycle, to safeguard the blocking output transformers.

Six *conv_pulse_gen* blocks (one per channel) are used for generating blocking and TTL pulses at the outputs, based on trigger inputs arriving on the channels. The *conv_pulse_gen* blocks are configured for 1.2 μs pulses (*g_pwidth = 24*, considering the 50 ns clock input).

6.1 Implementation

Figure 4 shows the implementation of the *conv_pulse_gen* block. It employs a finite-state machine (FSM) that is used to generate a fixed-width pulse at the output.

An external glitch filter (*gc_glitch_filt* from the OHWR general-cores library) can be enabled by the user by flipping SW1.1. Enabling this external glitch filter will mean the input trigger is sampled with the 20 MHz clock prior to it being input to the glitch filter (see Figure 1).

Regardless of whether the glitch filter is enabled or not, the FSM reacts to the rising edge of one of its two start inputs. A rising edge on an input starts the internal counter, which counts up to a maximum value in order to assure a pulse with the length *g_pwidth*. However, since the glitch filter adds a delay to the input pules, the behavior of the outputs is different depending on whether the filter is enabled or not.

With the glitch filter disabled, the input pulse enables the input flip-flop, which starts pulse generation. The generated pulse signal is then synchronized in the *clk_20_vcxo_i* domain and input to the synchronous FSM, which

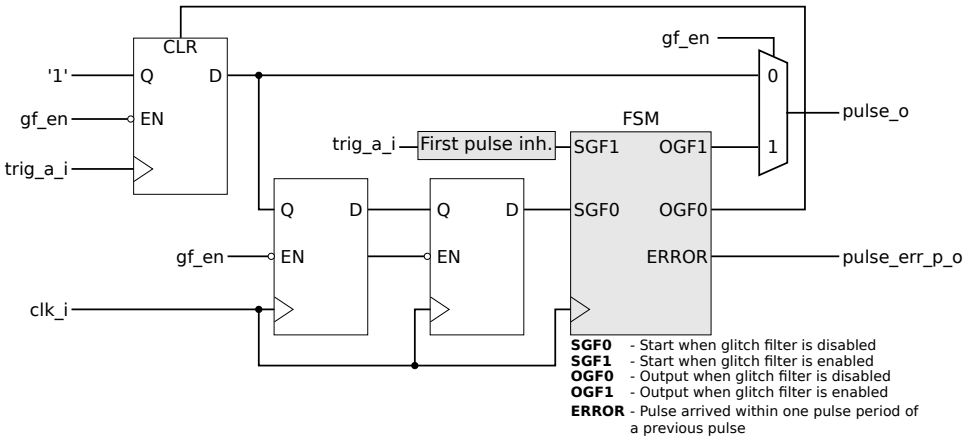


Figure 4: Pulse generator block

extends the pulse to the appropriate pulse width. The rising edge on *SGF0* triggers the counter, and when the counter reaches the value corresponding to the selected pulse width, it sets the *OGF0* output, which will reset the input flip-flop, thus ending the pulse.

The output of the pulse-triggered flip-flop is fed into the synchronizer since this is guaranteed to be a signal which is longer than two clock cycles, thus being detected by the FSM. Had the trigger input been fed directly to the synchronization FFs, a shorter-than-one-cycle glitch would trigger a pulse but not the synchronization FFs, and an output signal can potentially be extended to a pulse length which is unsafe for the controlling MOSFET on the CONV-TTL-BLO board.

With the glitch filter enabled, the rising edge on *SGF1* sets *OGF1*, and this will be kept high until the counter reaches the value corresponding to the pulse width. The input trigger signal is synchronized into the 20 MHz clock domain outside the *conv_pulse_gen* block, and is fed directly to the input of the FSM.

Before being fed to the FSM, however, the glitch-filtered signal is passed through a *pulse inhibit circuit*, which inhibits the first pulse when the board is in TTL-BAR repetition mode. In this mode, an unconnected channel is always HIGH and until the *no signal detect* block outside the *conv_pulse_gen* block (see the next subsection) triggers, the continuous HIGH signal will trigger a pulse, due to the reset state of the rising-edge detector on the FSM input.

After the pulse generation period, the FSM goes into a pulse rejection state, where the pulse reset is kept high. If any pulses arrive on the input while the FSM is in this rejection state, they are not replicated at the output. The pulse rejection phase lasts for *g_duty_cycle_div*g_pwidth*, yielding a maximum duty cycle of $1/g\_duty\_cycle\_div$ for input pulses.

Should a pulse rising-edge arrive anywhere within the generation or rejection phase in the FSM, the *pulse_err_p_o* output is set high for one clock cycle. This signal can be used to, e.g., count the number of pulses that were rejected on a board. In the actual top-level design, however, missed pulse counting is not implemented, the *pulse_err_p_o* output simply asserts the SR.PMISSE bit (see Appendix A.1.2).

Note that due to the fact that the counter starts counting up from zero and delays in the glitch filter when it is enabled, the maximum value of the internal counter is not *g_pwidth*. Instead, the counter counts up to a pair of VHDL constants defined in the code. These constants assure the pulse at the output is kept high for a number of *g_pwidth* cycles of the *clk_i* signal.

6.2 Board-level view

Figure 5 shows the pulse replication mechanism on the CONV-TTL-BLO. Here, the *PG* block is the *conv_pulse_gen* block with the necessary settings. This block can either be triggered via a pulse arriving on the TTL or blocking channel, or by a manual trigger pulse arriving from the *conv_man_trig* component (see Section 8).

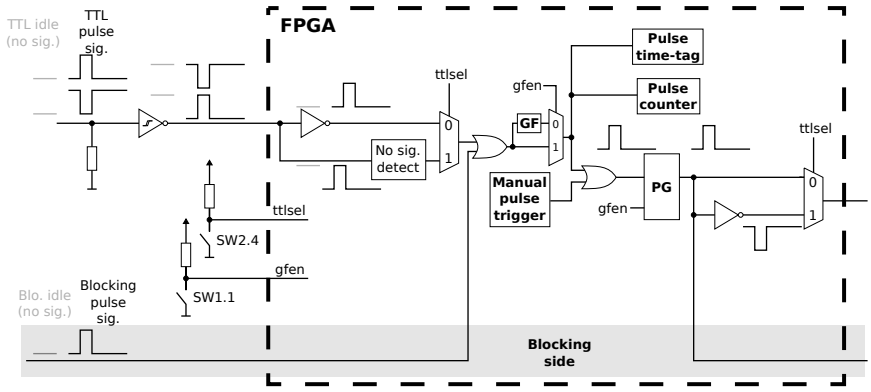


Figure 5: Board-level view of pulse replication mechanism

Since the *conv_pulse_gen* block expects a rising edge at its *trig_a_i* input in order to generate a pulse at the output, logic external to the block caters for the different types of signals that arrive on CONV-TTL-BLO inputs.

Most of this external logic is on the TTL pulse side, where both TTL and TTL-BAR pulses may arrive. As described in Section 4.3 of the CONV-TTL-BLO User Guide [1], if a wire is not plugged in when TTL-BAR pulses are input, a continuous logic high level on the line would inhibit pulses arriving on the blocking side from triggering a pulse generation. This is why the *no signal detect* block has been implemented.

The block's implementation is shown in Figure 6. It is implemented as a counter which keeps the *en_o* signal high as long as it does not reach its

maximum value. The counter counts up when the *cnt* input is high. By setting the maximum value of the counter to 1999, it disables the line to the multiplexer if this stays high for 100 μs, thus allowing for blocking pulses at the input of the OR gate. The line is re-enabled as soon as it goes back low, i.e., when a wire has been plugged into the channel.

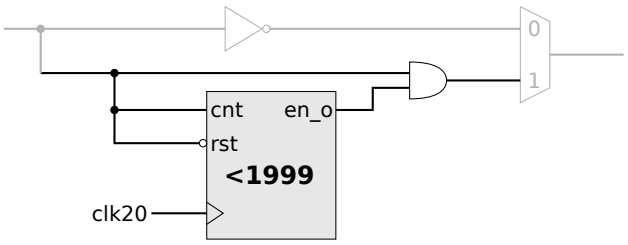


Figure 6: No signal detect block

Both the *no signal detect* block and the glitch filter are generated for each channel in the top-level VHDL file of the design.

7 Pulse counters

There are a total of six pulse counters implemented in the logic. Their implementation is achieved via a single process – *p_pulse_cnt*.

Figure 7 presents the implementation of the pulse counters. When a pulse arrives on either the TTL or blocking side, it is resynchronized in the 20 MHz clock domain and passed through a rising edge detector. When a rising edge occurs on the pulse, the counter is incremented by one and stored to the channel pulse counter register (CHxPCR – see Appendix A.1) register.

Note that this register is implemented outside of the *conv_regs* component, since it is a read-write register. When the register is written by the *conv_regs* component, the *load* output is asserted and the register is loaded with the value received via I^2C.

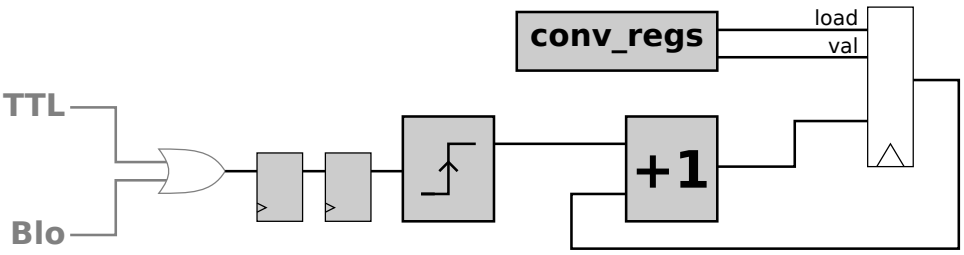


Figure 7: Pulse counter implementation

8 Manual pulse trigger

Entity	*conv_man_trig*	
Generics	*g_nr_chan*	Pulse repeater number of channels
	g_gf_len	Pulse generator glitch filter length
Ports	*clk_i*	Clock signal
	rst_n_i	Active-low reset signal
	reg_ld_i	MPT in the CR written
	reg_i	Value of MPT field in the CR
	trig_o	Trigger output
Usage	Trigger pulse generator	Two clock-cycle pulse output

The manual pulse triggering mechanism is achieved via the *conv_man_trig* module. This module generates a pulse that is input to the *conv_pulse_gen* module and that triggers a 1.2 μs pulse on the TTL and blocking outputs.

Since manual pulse generation is a delicate feature which is only used when a debug pulse is generated on a channel, a "password" is needed in order to manually trigger a pulse. This "password" is obtained from the MPT field in the control register (CR – see Appendix A.1.3), where a magic sequence of numbers should be written. Once this magic sequence has been input, the next write to the register should be the channel number. If a valid channel number is written, a pulse is generated on this channel.

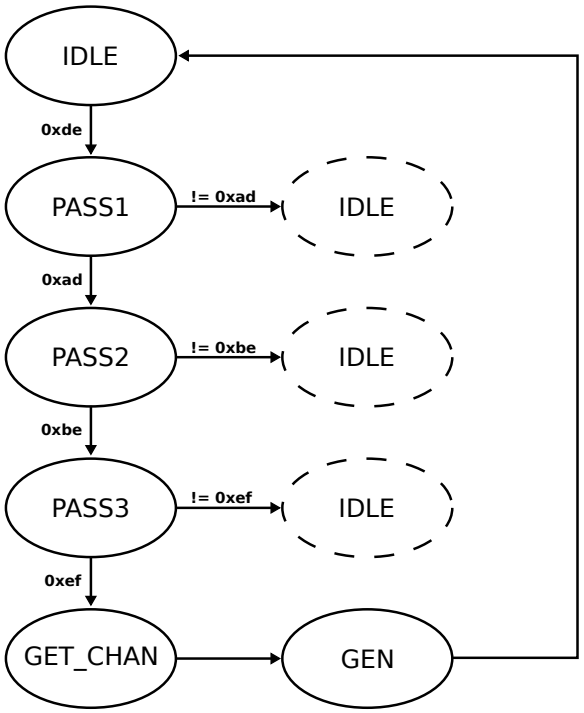


Figure 8: FSM of the *conv_man_trig* component

The *conv_man_trig* takes the value of the MPT field in the control register as an input and via the state-machine shown in Figure 8, it checks that the values written to the MPT field correspond to the magic sequence (Table 4). The FSM advances on writes to the MPT field of the CR. Once the magic sequence has been received, the next write to the MPT field sets the *trig_o* output, which is input to the *trig_a_i* input of *conv_pulse_gen* (see Section 6) after it is ORed together with the TTL and blocking inputs as shown in Figure 5. Should an invalid channel number be input, no error is reported and no pulse is generated.

Table 4: Magic sequence to initiate manual pulse triggering

Byte 0	**Byte 1**	**Byte 2**	**Byte 3**
0xde	0xad	0xbe	0xef

To generate a long enough pulse to be detected by the *conv_pulse_gen* component when its glitch filter is enabled, the *conv_man_trig* should have knowledge of the length of the pulse generator's glitch filter, which can be supplied via the *g_gf_len* generic. With the value of this generic, the *conv_man_trig* component extends the *trig_o* pulse to the necessary number of cycles for the *conv_pulse_gen* to detect a pulse. This extension is done in the *GEN* state (Figure 8).

9 Pulse time-tagging

The time-tagging architecture is shown in Figure 9. There are two clock domains in the design. The time-tag controller and the time from the WRPC are both in the 125 MHz clock domain, while the ring buffer is in both the 20 MHz clock domain and the 125 MHz clock domain.

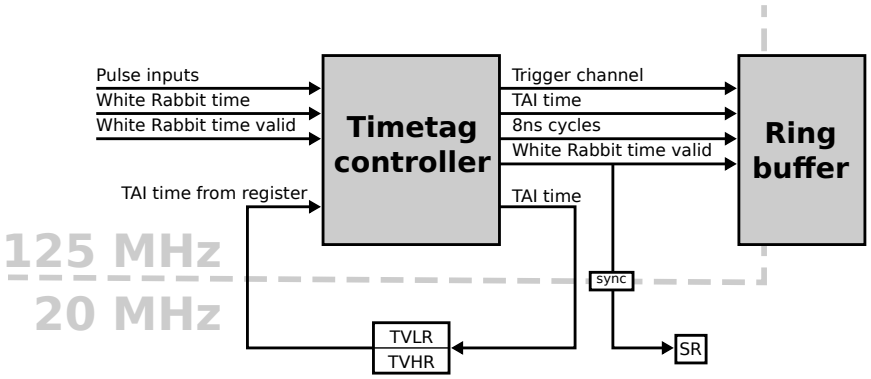


Figure 9: Pulse time-tagging architecture

The timetag controller and ring buffer components are connected at the top-level. The outputs of the ring buffer are connected directly to the *conv_regs* component at the top level also.

9.1 Timetag controller

The time-tag controller in Figure 9 is the *conv_pulse_timetag* VHDL component. It is designed to be connected directly to the ring buffer as shown above. As opposed to what the simplified architecture above shows, the time-tag controller also implements the local time counters.

The block's design is presented in Figure 10. This figure shows the functioning when the *conv_pulse_timetag* component is clocked from the 125 MHz clock, as is the case in the converter board designs. Note however that the component can work with any clock rate by changing the *g_clk_rate* generic.

A free-running counter inside the block counts the ticks of the *clk_i* signal to count the seconds. When it reaches the value *g_clk_rate-1* (125 mega in the figure), it resets and sends a "tick" to the TAI seconds counter, which then increments.

As seen in Figure 1, the pulse inputs are derived from the OR gate which ORs together the TTL and blocking inputs. Since these pulses can be asynchronous, they are synchronized in the 125 MHz domain and passed through rising edge detectors. A rising edge on any pulse channel triggers the *buf_wr_req_p_o* signal. As the port's name suggests, the *buf_wr_req_p_o* signal is a one-cycle pulse that triggers a write to the ring buffer.

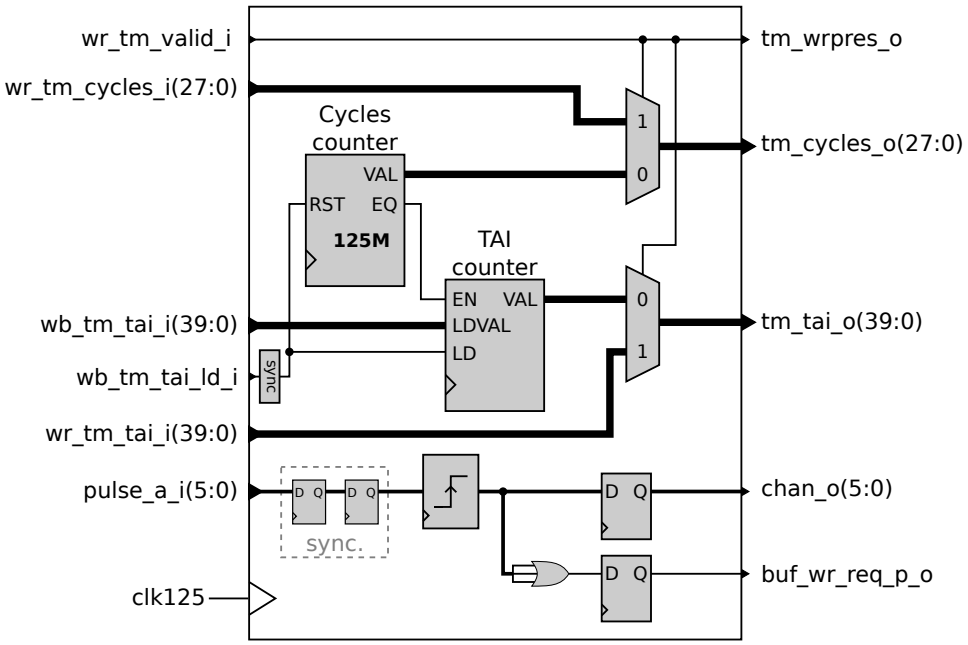


Figure 10: Timetag controller logic

All the output ports are connected externally directly to the ring buffer, therefore when the *buf_wr_req_p_o* output pulses, they are written to the ring buffer. As shown in Figure 9, the *tm_wpres_o* signal is also reflected in the board status register (SR – see Appendix A.1.2).

Note that due to the synchronization logic, rising edge detector and the latching of the ORed pulse rising edge detection signal, the *buf_wr_req_p_o* signal is set between three and four cycles after the pulse signal actually arrives on the input.

Due to the two clock domains in the design, some synchronization logic is needed. This is achieved via *gc_sync_ffs* components from the *general-cores* library [6] in the case of the WR time valid signals storage to the SR and in the case of the TAI time value load pulses from the *conv_regs* to the *conv_pulse_timetag* components.

The TAI time signal is not synchronized before being connected to the *conv_regs* component, since its rate of change of once per second is considered too slow to present any problem of synchronization when read by the user.

9.2 Ring buffer

The mechanics of the ring buffer are presented in the **Ring buffer mechanics** section of the CONV-TTL-BLO user guide [1]. This section gives a few more details about the implementation of the ring buffer, which can be found in the *conv_ring_buf.vhd* file.

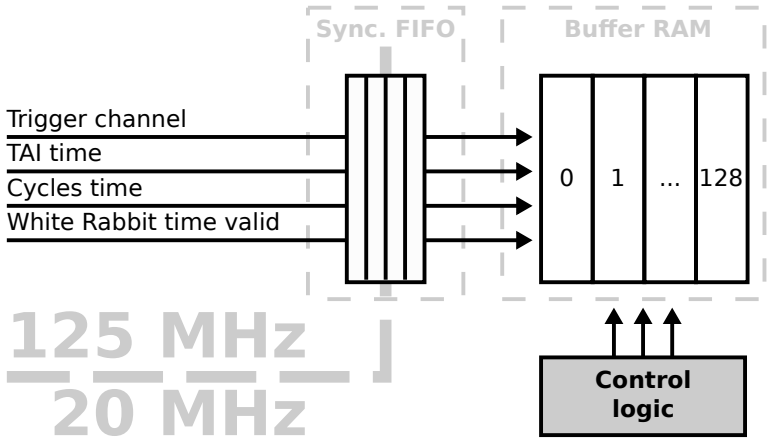


Figure 11: Ring buffer implementation

A high-level view of the implementation is shown in Figure 11. This figure is oriented towards the converter board designs, but the ring buffer is generic and can be used in other designs by properly instantiating it.

The ring buffer is implemented via a dual-clock, asynchronous FIFO, whose purpose is to synchronize data between the read and write clock domains, a dual-port RAM clocked with the read clock, and control logic for the buffer RAM. In the case of the converter board designs, the write clock is the 125 MHz clock and the read clock is the 20 MHz clock.

The buffer control logic controls when the read and write pointers get incremented and when the empty and full signals are set. The setting of the empty and full signals are based on a buffer count signal, which gets incremented when a write is performed and the buffer is not full, and decremented when a read is performed and the buffer is not empty.

10 Folder structure

Gateware files are organized on a per type-of-project basis. There are two different types of projects for CONV-TTL-BLO gateware: the *release project* and *test projects*. The release project is the latest production gateware version, that goes on the CONV-TTL-BLO board used in the field. Test projects are meant to be downloaded to a CONV-TTL-BLO for testing the CONV-TTL-BLO system under long-term test conditions. The projects present in the repository at the time of writing of this document are presented in Table 5.

Table 5: Gateware projects in the repository

Project	Description
conv_ttl_blo	Design-wide release project to be used in the field
regtest	Long-term test for testing the I^2C communication by writing to a RAM on the FPGA
pulsetest	Long-term test for testing pulse repetition on the CONV-TTL-BLO

The folder structure for the project is presented below.

→ conv-ttl-blo-gw/
- → doc/
  - → hdlguide/
- → ip_cores/
  - → general-cores/
- → modules/
  - → Release/
    - → conv_man_trig.vhd
    - → conv_regs.vhd
    - → conv_regs.wb
    - → conv_pulse_timetag.vhd
    - → ctblo_pulse_gen.vhd
  - → pulsetest/
    - → pulse_gen_gp.vhd
    - → [...]
  - → conv_pulse_gen.vhd
  - → reset_gen.vhd
- → sim/

- → syn/
  - → Release/
  - → pulsetest/
  - → regtest/
- → top/
  - → Release/
    - → conv_ttl_blo.ucf
    - → conv_ttl_blo.vhd
  - → pulsetest/
    - → pulsetest.ucf
    - → pulsetest.vhd
  - → regtest/
    - → regtest.ucf
    - → regtest.vhd

Table 6: Folder structure

Folder	**Description**
doc/	Documentation files
ip_cores/	IP cores used in the design
modules/	Project-specific modules instantiated in the top-level design Organized on a project type basis
sim/	Module simulation files
syn/	ISE project file and synthesis output files, including binaries to download to the FPGA Organized on a project type basis
top/	Top-level .vhd and .ucf files Organized on a project type basis

As can be seen from the folder structure above, gateware files are organized in the *modules/*, *syn/* and *top/* folders following this project convention. Files in these folders (where relevant) are organized in the *Release/*, *pulsetest/* and *regtest/* folders, where the *Release/* folder of course represents the release gateware and the other two are test projects, as their names suggest.

One place where the project structure is not necessarily enforced is the *sim/* folder. This folder is meant to contain files relevant for simulation of various modules within the design and as such can be composed of folders named after the component to be simulated.

11 Getting around the code

Ports and signals usually follow the coding guideline at [7]. Most of the top-level ports of the gateware are lower-case versions of their schematics counterparts. The exceptions from this are due to either net names that could not be syntactically represented in VHDL, or net names that have been made clearer in VHDL code.

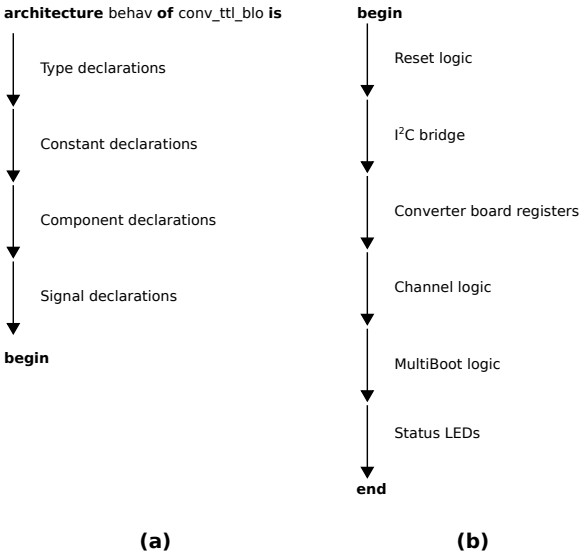


Figure 12: VHDL architecture of the release gateware

Code in the top-level files is organized in code sections. A code section is a piece of code pertaining to a certain part of the design, where component instantiations and input and output port assignments are made. For example, there is a section pertaining to pulse repetition, where there is a generate block to generate the logic necessary for pulse repetition on each channel, including the pulse status LEDs.

The VHDL architecture of the top-level file of the release gateware is shown in Figure 12. Table 7 lists the code sections of the top-level file.

Table 7: Code sections in the FPGA gateware

Code section	Description
Reset logic	– *reset_gen* instantiation
I^2C bridge	– *wb_i2c_bridge* instantiation – logic for blinking the I2C bicolor LED on the front panel – generate the CWDTO bit register

Code section	Description
Converter boards registers	– *conv_regs* instantiation – connect the switch lines to the SR – connect the RTM detection lines to the SR – implement the RST, RST_UNLOCK and WR-PRES bit registers
Channel logic	– connect inputs to internal signals – instantiation of the pulse time-tagging controller and manual pulse trigger components VHDL *generate* statements then generate the channel logic: – glitch filters and selection between filtered and non-filtered based on the glitch filter selection switch – synchronization flip-flops on the input signals – input pulse counter logic – no signal detect block (Figure 6) – *conv_pulse_gen* instantiation – pulse output connections – process to light pulse LEDs on pulse output
MultiBoot logic	– *wb_xil_multiboot* instantiation
Status LEDs	– *bicolor_led_ctrl* instantiation – connecting the *led_state_i* input of the component to the relevant control signals

Appendices

A Memory map

Table 8 shows the complete memory map of the gateware. The following sections list the memory map of each peripheral.

In order to convert address values to register index values for SNMP access, the following formula should be used:

$$reg.index = \frac{addr}{4} + 1$$

Table 8: CONV-TTL-BLO memory map

Peripheral	Address range		Description
Board registers	0x000	0x020	Coverter board registers
MultiBoot	0x040	0x050	MultiBoot module
Thermometer	0x080	0x084	Thermometer chip

A.1 Converter board registers

Base address: 0x000

Offset	Reset	Name	Description
0x0	0x54424c4f	BIDR	Board ID Register
0x4	(1)	SR	Status Register
0x8	0x00000000	CR	Control Register
0xc	0x00000000	CH1PCR	Channel 1 Pulse Counter Register
0x10	0x00000000	CH2PCR	Channel 2 Pulse Counter Register
0x14	0x00000000	CH3PCR	Channel 3 Pulse Counter Register
0x18	0x00000000	CH4PCR	Channel 4 Pulse Counter Register
0x1c	0x00000000	CH5PCR	Channel 5 Pulse Counter Register
0x20	0x00000000	CH6PCR	Channel 6 Pulse Counter Register
0x24	0x00000000	TVLR	Time Value Low Register
0x28	0x00000000	TVHR	Time Value High Register
0x2c	0x00000000	TBMR	Tag Buffer Meta Register
0x30	0x00000000	TBCYR	Tag Buffer Cycles Register
0x34	0x00000000	TBTLR	Tag Buffer TAI Low Register
0x38	0x00000000	TBTHR	Tag Buffer TAI High Register
0x3c	0x00020000	TBCSR	Tag Buffer Control and Status Register

Note (1): The reset value of the SR cannot be specified, since it is based on the gateware version, the state of the on-board switches and whether an RTM is plugged in or not.

A.1.1 BIDR – Board ID Register

31	30	29	28	27	26	25	24
BIDR[31:24]							
23	22	21	20	19	18	17	16
BIDR[23:16]							
15	14	13	12	11	10	9	8
BIDR[15:8]							
7	6	5	4	3	2	1	0
BIDR[7:0]							

- **BIDR** [*read-only*]: ID register bits
  Reset value: 0x54424c4f
- **Unimplemented bits**: write as '0', read undefined

A.1.2 SR – Status Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	PMISSE	I2C_ERR
23	22	21	20	19	18	17	16
WRPRES	I2C_WDTO	RTM[5:0]					
15	14	13	12	11	10	9	8
SWITCHES[7:0]							
7	6	5	4	3	2	1	0
GWVERS[7:0]							

- **GWVERS** [*read-only*]: Gateware version
  Leftmost nibble hex value is major release decimal value
  Rightmost nibble hex value is minor release decimal value
  e.g.
  0x11 – v1.1
  0x2e – v2.14

- **SWITCHES** [*read-only*]: Status of on-board switches (see Section **??**)
  0 – switch is ON
  1 – switch is OFF

- **RTM** [*read-only*]: RTM detection lines [8]
  0 – line active
  1 – line inactive

- **I2C_WDTO** [*read/write*]: Communication watchdog timer status
  1 – timeout occured
  0 – no timeout
  This bit can be cleared by writing a '1' to it

- **WRPRES** [*read-only*]: White Rabbit present
  1 – White Rabbit present
  0 – White Rabbit not present

- **I2C_ERR** [*read/write*]: I2C communication error
  1 – attempted to address non-existing address
  0 – idle
  This bit can be cleared by writing a '1' to it

- **PMISSE** [*read/write*]: Pulse missed error
  1 – input pulse rejected to safeguard blocking output stage
  0 – idle
  This bit can be cleared by writing a '1' to it

- **Unimplemented bits**: write as '0', read undefined

A.1.3 CR – Control Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	MPT[7:6]	
7	6	5	4	3	2	1	0
MPT[5:0]						RST	RST_UNLOCK

- **RST_UNLOCK** [*read/write*]: Reset unlock bit
  1 – Reset bit unlocked
  0 – Reset bit locked

- **RST** [*read/write*]: Reset bit
  1 – initiate logic reset
  0 – no reset

- **MPT** [*write-only*]: Manual Pulse Trigger
  Write the following sequence to trigger a pulse:
  0xde – Byte 1 of magic sequence
  0xad – Byte 2 of magic sequence
  0xbe – Byte 3 of magic sequence
  0xef – Byte 4 of magic sequence
  Number in range 1..6 – trigger a pulse

- **Unimplemented bits**: write as '0', read undefined

A.1.4 CH1PCR – Channel 1 Pulse Counter Register

31	30	29	28	27	26	25	24
CH1PCR[31:24]							
23	22	21	20	19	18	17	16
CH1PCR[23:16]							
15	14	13	12	11	10	9	8
CH1PCR[15:8]							
7	6	5	4	3	2	1	0
CH1PCR[7:0]							

- **CH1PCR** [*read/write*]: Pulse counter value

- **Unimplemented bits**: write as '0', read undefined

A.1.5 CH2PCR – Channel 2 Pulse Counter Register

31	30	29	28	27	26	25	24
CH2PCR[31:24]							
23	22	21	20	19	18	17	16
CH2PCR[23:16]							
15	14	13	12	11	10	9	8
CH2PCR[15:8]							
7	6	5	4	3	2	1	0
CH2PCR[7:0]							

- **CH2PCR** [*read/write*]: Pulse counter value
- **Unimplemented bits**: write as '0', read undefined

A.1.6 CH3PCR – Channel 3 Pulse Counter Register

31	30	29	28	27	26	25	24
CH3PCR[31:24]							
23	22	21	20	19	18	17	16
CH3PCR[23:16]							
15	14	13	12	11	10	9	8
CH3PCR[15:8]							
7	6	5	4	3	2	1	0
CH3PCR[7:0]							

- **CH3PCR** [*read/write*]: Pulse counter value
- **Unimplemented bits**: write as '0', read undefined

A.1.7 CH4PCR – Channel 4 Pulse Counter Register

31	30	29	28	27	26	25	24
CH4PCR[31:24]							
23	22	21	20	19	18	17	16
CH4PCR[23:16]							
15	14	13	12	11	10	9	8
CH4PCR[15:8]							
7	6	5	4	3	2	1	0
CH4PCR[7:0]							

- **CH4PCR** [*read/write*]: Pulse counter value
- **Unimplemented bits**: write as '0', read undefined

A.1.8 CH5PCR – Channel 5 Pulse Counter Register

31	30	29	28	27	26	25	24
CH5PCR[31:24]							
23	22	21	20	19	18	17	16
CH5PCR[23:16]							
15	14	13	12	11	10	9	8
CH5PCR[15:8]							
7	6	5	4	3	2	1	0
CH5PCR[7:0]							

- **CH5PCR** [*read/write*]: Pulse counter value
- **Unimplemented bits**: write as '0', read undefined

A.1.9 CH6PCR – Channel 6 Pulse Counter Register

31	30	29	28	27	26	25	24
CH6PCR[31:24]							
23	22	21	20	19	18	17	16
CH6PCR[23:16]							
15	14	13	12	11	10	9	8
CH6PCR[15:8]							
7	6	5	4	3	2	1	0
CH6PCR[7:0]							

- **CH6PCR** [*read/write*]: Pulse counter value
- **Unimplemented bits**: write as '0', read undefined

A.1.10 TVLR – Time Value Low Register

31	30	29	28	27	26	25	24
TVLR[31:24]							
23	22	21	20	19	18	17	16
TVLR[23:16]							
15	14	13	12	11	10	9	8
TVLR[15:8]							
7	6	5	4	3	2	1	0
TVLR[7:0]							

- **TVLR** [*read/write*]: TAI seconds counter bits 31..0
  Writing this field resets the internal cycles counter.
- **Unimplemented bits**: write as '0', read undefined

A.1.11 TVHR – Time Value High Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
TVHR[7:0]							

- **TVHR** [*read/write*]: TAI seconds counter bits 39..32
  Writing this field resets the internal cycles counter.
- **Unimplemented bits**: write as '0', read undefined

A.1.12 TBMR – Tag Buffer Meta Register

31	30	29	28	27	26	25	24
WRTAG	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	-	CHAN[5:0]					

- **CHAN** [*read-only*]: Channel mask
  Mask for the channel(s) that triggered time-tag storage:
  bit 0 – channel 1
  bit 1 – channel 2
  ...
  bit 5 – channel 6
- **WRTAG** [*read-only*]: White Rabbit present
  1 - Current time tag generated with White Rabbit
  0 - Current time tag generated with internal counter
- **Unimplemented bits**: write as '0', read undefined
- **A read from this register advances the buffer read pointer, if the ring buffer is not empty**

A.1.13 TBCYR – Tag Buffer Cycles Register

31	30	29	28	27	26	25	24
-	-	-	-	TBCYR[27:24]			
23	22	21	20	19	18	17	16
TBCYR[23:16]							
15	14	13	12	11	10	9	8
TBCYR[15:8]							
7	6	5	4	3	2	1	0
TBCYR[7:0]							

- **TBCYR** [*read-only*]: Cycles counter
  Value of the 8-ns cycles counter when time tag was taken.
- **Unimplemented bits**: write as '0', read undefined

A.1.14 TBTLR – Tag Buffer TAI Low Register

31	30	29	28	27	26	25	24
TBTLR[31:24]							
23	22	21	20	19	18	17	16
TBTLR[23:16]							
15	14	13	12	11	10	9	8
TBTLR[15:8]							
7	6	5	4	3	2	1	0
TBTLR[7:0]							

- **TBTLR** [*read-only*]: Lower part of TAI seconds counter
  Value of the TAI seconds counter bits 31..0 when time tag was taken.
- **Unimplemented bits**: write as '0', read undefined

A.1.15 TBTHR – Tag Buffer TAI High Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	-	-
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
TBTHR[7:0]							

- **TBTHR** [*read-only*]: Upper part of TAI seconds counter
  Value of the TAI seconds counter bits 39..32 when time tag was taken.
- **Unimplemented bits**: write as '0', read undefined

A.1.16 TBCSR – Tag Buffer Control and Status Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	CLR	EMPTY	FULL
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	USEDW[6:0]						

- **USEDW** [*read-only*]: Buffer counter
  Number of samples in the ring buffer
- **FULL** [*read-only*]: Buffer full
  1 – buffer full
  0 – buffer is not full
- **EMPTY** [*read-only*]: Buffer empty
  1 – buffer empty
  0 – buffer is not empty
- **CLR** [*write-only*]: Clear tag buffer
  1 – clear
  0 – no effect
- **Unimplemented bits**: write as '0', read undefined

A.2 MultiBoot controller

Base address: 0x040

Offset	Reset	Name	Description
0x0	0x00000000	CR	Control Register
0x4	0x00000000	SR	Status Register
0x8	0x00000000	GBBAR	Golden Bitstream Base Address Register
0xc	0x00000000	MBBAR	MultiBoot Bitstream Base Address Register
0x10	0x10000000	FAR	Flash Access Register

A.2.1 CR – Control Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	IPROG	IPROG_UNLOCK
15	14	13	12	11	10	9	8
-	-	-	-	-	-	-	-
7	6	5	4	3	2	1	0
-	RDCFGREG	CFGREGADR[5:0]					

- **CFGREGADR** [*read/write*]: Configuration register address
  Address of FPGA configuration register to read.
- **RDCFGREG** [*write-only*]: Read FPGA configuration register
  1 – Start FPGA configuration register sequence.
  0 – No effect.
- **IPROG_UNLOCK** [*read/write*]: Unlock bit for the IPROG command
  1 – Unlock IPROG bit.
  0 – No effect.
- **IPROG** [*read/write*]: Start IPROG sequence
  1 – Start IPROG configuration sequence
  0 – No effect
  This bit needs to be unlocked by writing the IPROG_UNLOCK bit first.
  A write to this bit with IPROG_UNLOCK cleared has no effect.
- **Unimplemented bits**: write as '0', read undefined

A.2.2 SR – Status Register

31	30	29	28	27	26	25	24
-	-	-	-	-	-	-	-
23	22	21	20	19	18	17	16
-	-	-	-	-	-	WDTO	IMGVALID
15	14	13	12	11	10	9	8
CFGREGIMG[15:8]							
7	6	5	4	3	2	1	0
CFGREGIMG[7:0]							

- **CFGREGIMG** [*read-only*]: Configuration register image
  Image of the FPGA configuration register at address CFGREGADR (see Configuration Registers section in Xilinx UG380 [9]); validated by IMGVALID bit

- **IMGVALID** [*read-only*]: Configuration register image valid
  1 – CFGREGIMG valid
  0 – CFGREGIMG not valid;

- **WDTO** [*read/write*]: MultiBoot FSM stalled at one point and was reset by FSM watchdog
  1 – FSM watchdog fired
  0 – FSM watchdog has not fired

- **Unimplemented bits**: write as '0', read undefined

A.2.3 GBBAR – Golden Bitstream Base Address Register

31	30	29	28	27	26	25	24
BITS[31:24]							
23	22	21	20	19	18	17	16
BITS[23:16]							
15	14	13	12	11	10	9	8
BITS[15:8]							
7	6	5	4	3	2	1	0
BITS[7:0]							

- **BITS** [*read/write*]: Bits of GBBAR register
  31..24 – Read or fast-read OPCODE of the flash chip (obtain it from the flash chip datasheet)
  23..0 – Golden bitstream address in flash

- **Unimplemented bits**: write as '0', read undefined

A.2.4 MBBAR – MultiBoot Bitstream Base Address Register

31	30	29	28	27	26	25	24
BITS[31:24]							
23	22	21	20	19	18	17	16
BITS[23:16]							
15	14	13	12	11	10	9	8
BITS[15:8]							
7	6	5	4	3	2	1	0
BITS[7:0]							

- **BITS** [*read/write*]: Bits of MBBAR register
  31..24 – Read or fast-read OPCODE of the flash chip (obtain it from the flash chip datasheet)
  23..0 – MultiBoot bitstream start address in flash

- **Unimplemented bits**: write as '0', read undefined

A.2.5 FAR – Flash Access Register

31	30	29	28	27	26	25	24
-	-	-	READY	CS	XFER	NBYTES[1:0]	
23	22	21	20	19	18	17	16
DATA[23:16]							
15	14	13	12	11	10	9	8
DATA[15:8]							
7	6	5	4	3	2	1	0
DATA[7:0]							

- **DATA** [*read/write*]: Flash data field
  23..16 – DATA[2]; after an SPI transfer, this register contains the value of data byte 2 read from the flash
  15..8 – DATA[1]; after an SPI transfer, this register contains the value of data byte 1 read from the flash
  7..0 – DATA[0]; after an SPI transfer, this register contains the value of data byte 0 read from the flash

- **NBYTES** [*read/write*]: Number of DATA fields to send and receive in one transfer:
  0x0 – Send 1 byte (DATA[0])
  0x1 – Send 2 bytes (DATA[0], DATA[1])
  0x2 – Send 3 bytes (DATA[0], DATA[1], DATA[2])

- **XFER** [*write-only*]: Start transfer to and from flash
  1 – Start transfer
  0 – Idle

- **CS** [*read/write*]: Chip select bit
  1 - Flash chip selected (CS pin low)
  0 - Flash chip not selected (CS pin is high)
- **READY** [*read-only*]: Flash access ready
  1 - Flash access completed
  0 - Flash access in progress
- **Unimplemented bits**: write as '0', read undefined

A.3 Thermometer module

Base address: 0x080

Offset	Default	Name	Description
0x00	0x00000000	OWCSR	One-Wire Control and Status Register
0x04	0x00000004	OWCDR	One-Wire Clock Divider Registers

For details on the bits of the thermometer module access registers, see the OneWire Master module's documentation [5].

Note that the OWCDR should be set accordingly for proper functioning of the one-wire timings. The value for the current version of the gateware is `OWCDR = 0x00130063`.

References

[1] T.-A. Stana, "CONV-TTL-BLO User Guide." http://www.ohwr.org/documents/263, 06 2013.

[2] T.-A. Stana, "CONV-TTL-BLO Hardware Guide." http://www.ohwr.org/documents/282, 07 2013.

[3] "Wishbone Slave Generator." http://www.ohwr.org/projects/wishbone-gen/wiki.

[4] Maxim Integrated, "DS18B20 – Programmable Resolution 1-Wire Digital Thermometer." http://datasheets.maximintegrated.com/en/ds/DS18B20.pdf.

[5] I. Jeras, "sockit_owm, 1-wire (onewire) master," 2011. http://opencores.org/websvn,filedetails?repname=sockit_owm&path=%2Fsockit_owm%2Ftrunk%2Fdoc%2Fsockit_owr.pdf.

[6] "Platform-independent Core Collection webage on Open Hardware Repository." http://www.ohwr.org/projects/general-cores/wiki.

[7] P. Loschmidt, N. Simanić, C. Prados, P. Alvarez, and J. Serrano, "Guidelines for VHDL Coding," 04 2011. http://www.ohwr.org/documents/24.

[8] "Rear Transition Module detection." http://www.ohwr.org/projects/conv-ttl-blo/wiki/RTM_board_detection.

[9] Xilinx, "UG380 - Spartan-6 Configuration Guide." http://www.xilinx.com/support/documentation/user_guides/ug380.pdf, Jan. 2013. v2.5.