



OpenCores.Org

log_anal

Internal Logic State Analyzer



Author: Ernest Jamro

jamro@agh.edu.pl

Rev. 1.4

January 6, 2005

Revision History

Rev.	Date	Author	Description
1.0	04 Dec 02	Ernest Jamro	First Draft
1.1	09 Oct 03	Ernest Jamro	Logic Analyzer Data width up to 64 bits
1.2	8 Jan 04	Ernest Jamro	Logic Analyzer trigger width up to 64 bits Run-Length Coding Added
1.3	2 Feb 04	Ernest Jamro	Data Clock Enable trigger-like function added
1.4	6 Jan 05	Erenst Jamro	Additional LA identifications registers added, also previous registers addresses have been changed

Contents

1	Introduction	3
	Abstract	3
	Main Features	3
2	IO Ports	5
3	Registers	7
	List of Registers	7
	Internal Dual Port Memory	15
	Status Register	8
	Stop_Counter Register	11
	Tirg_Value Register	11
	Trig_Care Register	11
	Trig_Value64 & TrigCare64 Registers	12
	CED_Control Register	12
	CED_Value Register	13
	CED_Care Register	14
	CED_Value64 Register	14
	CED_Care64 Register	14
4	Operation	16
	Data Acquisition	16
	Data transfer	17
	Watching the Recorded Signals	18
	Run Length Coding	20
5	Implementation	21
	Xilinx	21

1

Introduction

Abstract

The internal Logic state Analyser (LA) is a simplified version of a standard logic state analyzer, however it is build in the prototyped circuit and therefore allows for probing internal signals. The LA at first writes probed signals into its internal memory, and then allows for off-line transfer through WISHBONE bus to a PC where the probed data can be watched. As during design prototyping watched signals are very often changed, the LA is mainly intended for FPGAs and works similarly to Xilinx ChipScope.

Main Features

Internal memory for on-line data probing and off-line probed data transfer.

Generic probed signals number: 8, 16, 32 or 64 bits.

Generic acquired data depth (internal memory size) (16 to 64k).

Software programmable single trigger value (and don't care).

Software programmable trigger place.

Separate trigger bus with generic width 1 to 64 bit.

Acquired trigger clock enable for easy additional trigger logic.

Separate Data Capture Clock Enable or Additional Trigger, width 0 to 64-bit

Generic (implement or not) run-length coding for acquired data compression

Generic single or double clock operation (separate or not clock for data acquisition and system interface).

WISHBONE compatible.

An internal Logic state Analyser (LA) is a device that works similarly to an analog oscilloscope but only digital (zero or one) signals are recorded only at certain moments (rising system clock edges). The LA samples signals at different time (one sample per clock cycle) and therefore allows for checking if the device works properly and how the real signal states behave. Recorded data (different signals states at different time intervals) should be then read off-line from the LA through 8-bit WISHBONE bus to your personal computer (PC). Then a special module written in vhd1 (la_view.vhd) reads the recorded data and displays them in a VHDL simulator. This allows for not only

a very convenient probing internal signals but also for some additional functions related with VHDL simulation. Firstly, real (internal) signals recorded by the LA can be compared with your simulation and the difference can be easily detected. Secondly, the LA data can be used as a stimulus for simulation. The latest may be a very convenient simulation approach when typing in long stimulus vectors is a very time-consuming procedure.

The LA is mainly intended for testing FPGA designs as probed signals can be easily changed by connecting different signals to the LA and then reconfiguring a FPGA. The LA has a similar functionality as the ChipScope sold by Xilinx. A trigger value can be changed by a proper write to the LA through WISHBONE control interface. Similarly, a trigger place (whether in the beginning, somewhere in the middle or in the end of the sampled data) can be also freely set without reprogramming the FPGA.

The LA has separate data and trigger buses. This introduces little additional logic but allows the LA to be trigger by input signals which need not be observed. Besides the LA may have two separate clock signals: the first for acquiring data and the second for interfacing the LA by the WISHBONE bus. The LA data acquisition clock frequency may be much higher than the system frequency and this allows for data sampling several times during a system (wishbone) clock cycle. The LA contains a clock enable inputs for data and trigger. Therefore in the case when input data are acquired at a low frequency, the clock enable signals rather than the two separate clocking should be used. Besides these clock enable signals might be used when only selected data sequence is to be acquired.

2

IO Ports

Core Parameters:

Parameter	Value	Default	Description
data_width	8, 16, 32, 64	16	Width of the observed data (the number of different one-bit signals to be watched)
mem_adr_width	4÷16	9	Sampled data depth (internal memory address width). Number of samples= $2^{\text{mem_adr_width}}$
adr_width	5÷18	11	WISHBONE interface address width: $\text{adr_width} = 1 + \text{mem_adr_width} + \log_2(\text{data_width}/8)$ adr_width must be ≥ 7
trig_width	1÷64	8	Trigger bus width
ce_dwidth	1÷64	8	Data capture clock enable or additional trigger logic
ced_type	0÷3	2	Data capture clock enable logic input selection 0 – data capture clock enable is always 1 (advance clock enable and additional trigger logic is not implemented) 1 – bus <i>ce_data</i> is clock enable (CE) logic input (implement advance clock enable logic) 2- bus <i>data</i> is the CE logic input, <i>ce_data</i> input is ignored (<i>ce_dwidth</i> must not be greater than <i>data_width</i>) 3- signal <i>trig</i> is CE logic input, <i>ce_data</i> input is ignored (<i>ce_dwidth</i> must not be greater than <i>trig_width</i>)
two_clocks	0, 1	0	0 – single clock operation – the LA signal (<i>clk</i>) is the same as <i>wb_clk_i</i> (this option has a slightly reduced area) 1 – the LA has two separate clocks
use_run_length_coding	0, 1	1	implement (1) or not (0) run-length coding for acquired data compression. See also <i>ced_control</i> register bit RLC

Control WISHBONE Slave interface signals

Port	Width	Direction	Description
wb_clk_i	1	Input	Clock input. For <i>two_clocks</i> =0 also trigger and analyzed data are latched on the rising edge of this

Port	Width	Direction	Description
			signal
wb_dat_i	8	Input	Input data bus
wb_dat_o	8	Output	Output data bus
wb_adr_i	<i>adr_width</i>	Input	Address bus
wb_we_i	1	Input	Write Enable
wb_stb_i	1	Input	Strobe
wb_ack_o	1	Output	Acknowledge

Other Control Signals

Port	Width	Direction	Description
arst	1	Input	Active high asynchronous reset signal (needed mainly for simulation purpose)

The Logic Analyzer Interface

Port	Width	Direction	Description
clk	1	Input	Separate clock for the LA interface. Should be the same as <i>wb_clk_I</i> when generic <i>two_clocks</i> = 0
data	<i>data_width</i>	Input	observed signals bus
ce_data	<i>ce_dwidth-1</i>	Input	Data capture Clock Enable or additional trigger logic
trig	<i>trig_width</i>	Input	Trigger input bus
ce_trig	1	Input	Clock Enable for trigger bus – Trigger is valid only when <i>ce_trig</i> ='1' and <i>clk</i> rises

3

Registers

List of Registers

Name	Address	Width	Access	Description
LA ID	0	8-bit	Rd	Constant value 0x45 to identify the LA
gen_mem_adr_width	1	8-bit	Rd	bits: 4-0 – generic – mem_adr_width bit 5: generic - use_run_length_coding bits: 7-6 – generic - ce_dtype
gen_data_width	2	8-bit	Rd	generic data_width - 1
gen_trig_width	3	8-bit	Rd	generic trigger bus width - 1
gen_ce_dwidth	4	8-bit	Rd	generic ce_dwidth-1
reserved	5-7	24-bit	Rd	Reserved for future use
status	8	8-bit	Rd/Wr	Status Register which sets/indicates the state of the LA and the trigger place
reserved	9	8-bits	Rd	Unknown
ced_control	0x0A	5-bit	Wr	Data Capture and Additional Trigger output function, Run- Length-Coding Control
reserved	0x0B	8-bit	Rd	Unknown
stop_counter	0x0C-0x0D	mem_adr_width - bit	Rd	Shows the place where the sampled data will be written to the internal memory.
trig_value	0x10 – 0x13	min(trig_width, 32) bit	Rd/Wr	Trigger Value (the value written to this register is then compared with the input trig)
trig_care	0x14 – 0x17	min(trig_width, 32) bits	Rd/Wr	Trigger Care (the value written to this register states that the corresponding trig_value bit is consider or ignored)
trig_value64	0x18 – 0x1B	trig_width - 32 bit	Rd/Wr	Trigger Value the Most Significant bits. This register is used for trig_width>32)
trig_care64	0x1C – 0x1F	trig_width - 32 bit	Rd/Wr	Trigger Care the MSB. This register is valid for trig_width>32
dce_valu	0x20 –	min(32,	Rd/Wr	Clock Enable (or additional trigger)

Name	Address	Width	Access	Description
e	0x23	ce_dwidth		active value – this value is then compared with the <code>ce_data</code> . This register (and the next dce registers) is implemented only when <code>ced_type>0</code>
dce_care	0x24 – 0x27	$\min(32, ce_dwidth)$	Rd/Wr	Clock Enable (or additional trigger) care – the corresponding input bit <code>ce_data</code> is considered (1) or not (0)
dce_value64	0x28 – 0x2B	$ce_dwidth - 32$	Rd/Wr	Dce_value register upper 32-bit – valid when <code>ce_dwidth>32</code>
dce_care64	0x2C – 0x2F	$ce_dwidth - 32$	Rd/Wr	Dce_care register upper 32-bit – valid when <code>ce_dwidth>32</code>
none	0x30 – (M-1)	$M - 0x30$	None	Must not be written and undefined when read.
internal memory	M – $2 * M - 1$	M-byte	Rd	Sampled data internal memory

where: $M = 2^{\text{adr_width}-1}$ (the size of the internal memory)

LA Identification Constant

Address: 0. Constant value: 0x4A.

Constant identification value used to check if LA is present in the specified address location.

Gen_Mem_Adr_Width Constant

Address: 1, Value: bits: 4-0 – generic – `mem_adr_width`, bit 5: generic `use_run_length_coding`, bits: 7-6 – generic – `ce_dtype`,

Constant values the same as the generic values defined in the `log_anal` entity in VHDL. This constant value should be then read (from the file) by the `la_view.vhd` in order to properly display captured data.

Gen_Data_Width Constant

Address: 2, Value: `data_width-1`

Constant values the same as the generic value defined in the `log_anal` entity in VHDL. This constant value should be then read by the `la_view.vhd` in order to properly display captured data. Example: `data_width= 16`, then the data read from address 2= 0x0F.

Gen_Trig_Width Constant

Address: 3, Value: `trig_width-1`

Constant values the same as the generic value defined in the `log_anal` entity in VHDL. This constant value should be then read by the `la_view.vhd` in order to properly display captured data.

Gen_CE_DWidth Constant

Address: 4, Value: *ce_dwidth-1*

Constant values the same as the generic value defined in the *log_anal* entity in VHDL. This constant value may be then read by the *la_view.vhd* in order to properly display captured data.

Status Register

Address: 0x08. Reset value: 0x00.

The status register defines the current operation mode of the LA and consists from 3 sections: Run bit – the trigger has been detected and now data are acquired, Finish bit – the data acquisition is finished (or not), Trig_Place – the trigger place in comparison to the acquired data.

Bit #	Access	Description
7 Run	Rd Wr	1 – the LA is in data acquisition mode and the trigger has been detected. 0 – the trigger has not been detected or the data acquisition is finished (see Finish bit). Note: The data are written to the internal memory whenever the Finish=0, this allows for recording data before the trigger. 1 – forces the LA to behave as if the trigger has occurred. Note: If the trigger place is not set for the beginning (Trig_Place=000000), the data before the trigger are undefined. This does not hold when the finish bit has been '0' for a long enough time. Consequently, it is recommended to write only 10000000 to the Status Register. 0 – forces the LA to wait for the trigger (normal operation).
6 Finish	Rd Wr	1 – the LA has finished the data acquisition (the internal memory is no more written). 0 – the LA is in the acquisition mode (data are written to the internal memory). 1 – forces the LA to stop data acquisition. Note: This might caused that the data in the internal memory are undefined (at the beginning in the <i>la_view</i> module). 0 – sets the LA into the acquire mode (normal operation)
5÷0 Trig Place	Rd Wr	The trigger place – where the trigger is placed in the watched data. 000000 – at the beginning, 111111 – at the end, 100000 – in the half, etc. <i>Finish</i> =0, <i>Run</i> = 0 – the same value as previously written <i>Finish</i> =0, <i>Run</i> = 1 – indicates how much data are still to be acquired (acquisition stops when the <i>Trig_Place</i> up-counter overflows). <i>Finish</i> = 1 – the <i>Trig_Place</i> up-counter has overflowed and should be zero (or almost zero). 000000 – the trigger is placed at the beginning of the acquired data (the data acquisition process starts a few moments after the trigger,

Bit #	Access	Description
		therefore the trigger moment is not recorded) 000001 – the trigger is placed almost at the beginning (the data acquisition is started a few moments before the trigger, and therefore the trigger moment is recorded) xxxxxx – the trigger is placed in the middle of the acquired data, the greater the number the closer to the end 111111 – the trigger is placed at the end of the sampled data (the trigger moment can be observed)

Note1: The standard size of the *Trig_Place* section is 6-bits. Nevertheless for *mem_adr_width*<6 the size of the *Trig_Place* is defined by the *mem_adr_width* parameter, and the LSBs are fill with zeros.

Note2: The actual size of the *Trig_Place* counter is *mem_adr_width*-bits, and the Status Registers shows only 6 MSBs of the counter. Consequently every tick of the *Trig_Place* section is equivalent to $2^{(adr_width-6)}$ data samples. See: *Stop_Counter* Register.

Note3: The reset value of the Status Register is 0x00 and the reset value of the *Trig_Value* and *Trig_Care* Registers is 0x0...00. Consequently the LA starts to acquire data just after the reset (signal: *arst*) is deactivated.

Note4: In order to properly configure the LA for data acquisition at first the *Trig_Value* and *Trig_Care* registers and then the Status Register should be properly written. Otherwise the LA may be triggered by the old version of the trigger.

Note5: When a trigger is not placed at the beginning, the trigger condition is ignored until a proper amount of data is sampled before the trigger. (See: *trig_counter_down* internal *log_anal* signal). This ensures that only valid data are observed by the LA.

Examples:

writebyte address data_to_be _written – writebyte instruction format used in this document.

writebyte 8 0x01 – standard acquisition – trigger at the beginning (the trigger moment is sampled).

writebyte 8 0x10 – standard acquisition – the trigger in ¼ if the acquired data.

writebyte 8 0x20 – standard acquisition – trigger in the half of the acquired data.

writebyte 8 0x3F – standard acquisition – trigger at the end.

writebyte 8 0x80 – force data acquisition just after the write instruction is executed.

readbyte address – readbyte instruction format used in this document

readbyte 8, result: 0x40 – the LA has finished data acquisition and is ready for data reading.

readbyte 8, result: 0x00 – acquisition mode, the trigger has not been detected, the trigger will be at the beginning.

readbyte 8, result: 0x3F – acquisition mode, the trigger has not been detected, the trigger will be at the end.

readbyte 8, result: 0xA0 – acquisition mode, the trigger has been detected and the half of the data has been acquired.

Stop_Counter Register

Address 0x0C-0x0D, Reset value: 0x0000

The width of the Stop_Counter Register is defined by the *mem_adr_width* parameter. This register is read only register (writes are ignored). It's function depends slightly on the *Finish* bit in the Status Register.

Finish= 0. The Stop_Counter Register shows the position where the acquired data are written to the internal memory (actually, the address of the next write).

Finish= 1. The Stop_Counter Register indicated the address where the last data was written to the internal memory (actually: Stop_Counter-1 does).

The Stop_Counter value should be read in order to obtain where is the beginning of the data sequence recorded inside the internal memory and should be therefore included in the *la_data.bin* file, see Operation/Data transfer Section. The Stop_Counter might be also used to indicate how quickly the data acquisition occurs.

Trig_Value Register

Address 0x10-0x13, Reset value: 0x0...00

The width of this register depends on the *trig_width* parameter. The read value is the same as the last written value. Each bit of this register specifies the trigger activation value on the corresponding *trig* input (see also: Trig_Care Register).

Trig_Care Register

Address 0x14-0x17, Reset value: 0x0000

The width of this register depends on the *trig_width* parameter. The read value is the same as the last written value. Each bit of this register specifies if the corresponding bit of the Trig_Value Register and *trig* input is taken into account (1) or ignored (0) while evaluating the trigger condition. Consequently the following condition must be satisfied to activate the trigger:

$$\begin{aligned} \text{trigger} = & (\text{NOT} (\text{trig}(0) \text{ XOR } \text{trig_value}(0)) \text{ OR NOT } \text{trig_care}(0)) \text{ AND} \\ & \text{AND} (\text{NOT} (\text{trig}(1) \text{ XOR } \text{trig_value}(1)) \text{ OR NOT } \text{trig_care}(1)) \text{ AND} \\ & \dots \text{AND} \\ & (\text{NOT} (\text{trig}(\text{trig_width}-1) \text{ XOR } \text{trig_value}(\text{trig_width}-1)) \text{ OR} \end{aligned}$$

NOT trig_care(trig_width-1))

It should be noted that the **trigger is ignored** just after the status register write (e.g. after command: start data acquisition). At first the internal memory must be filled with valid captured data and then the trigger is taken into account. The trigger ignore time is the greatest when the trigger is at the end of the captured data, and is almost zero (1 sample must be recorded) when the trigger is at the beginning. This condition is very important when additional clock enable logic is used.

Trig_Value64 & TrigCare64 Registers

Address 0x18-0x1B & 0x1C-0x1F, Reset value: 0x0...00

Same as TrigCare and TrigValue but used when trigger width is greater than 32-bit.

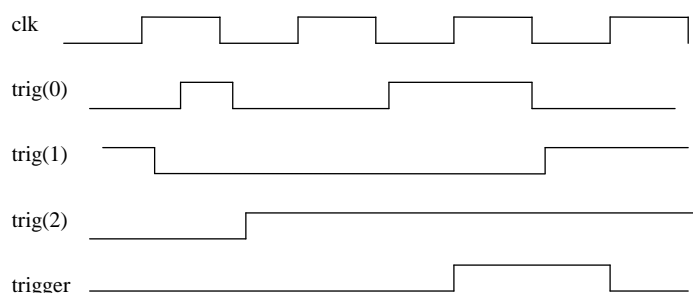
Example:

For *trig_width*= 3 and trigger condition: *X01*

writebyte 0x10 0b0101 // *Trig_Value Register Write*

writebyte 0x14 0b0011 // *Trig_Care Register Write*

the following result is obtained for the following input trig data.



CED_Control Register

Address 0x0A, Reset value: 0x00

Function of the Clock Enable Data (ced) logic- this register and whole Data Clock Enable logic is implemented only when generic dce_type>0.

Bit #	Access	Description
0 ced_active_low	Wr	The DCE logic active low, i.e. is activated when ced_care and ced_value condition is (=0) or is not (1) satisfied. 0- active high 1- active low
1 ced_reg	Wr	0 – direct logic 1 – additional flip-flop delay on the ced logic- this allows e.g. for

Bit #	Access	Description
		edge detection
2 ced_dat	Wr	0 – The data are captured regardless of the dce logic. 1 – The data are captures only when ced logic is 1 – see ced_care and ced_value registers
3 and_trig	Wr	0 – The AND-gate for trigger is not active. Trigger logic is defined only by the trig input and trig_value and trig_care logic or by the or_trig bit. 1 – trigger is active only when both trig and dce output functions are active. This bit is ignored when or_trig=1.
4 or_trig	Wr	0 – The OR-gate for trigger is not active. Trigger logic is defined only by the trig input and trig_value and trig_care logic or by the and_trig bit. 1- trigger is active when trig or dce output functions are active
5 RLC (not)	Wr	0 – implement data compression (RLC) (valid only when use_run_length_coding=1) 1 – do not use Run Length Coding (however the number of acquired signals is still decreased by 1 when use_run_length_coding=1).

Summing up the following functions are implemented

$\text{Clock_Enable_Data_Capture} = \text{ced} \text{ OR } \text{NOT ce_dat}$

$\text{Activate_Trigger} = (\text{ced} \text{ AND } \text{or_trig}) \text{ OR } (\text{trig_out} \text{ AND } (\text{NOT and_trig} \text{ OR } \text{ced}))$

Where

$\text{ced} = \text{ced_active_low} \text{ XOR } [\text{Flip_Flop_Output}] \text{ AND_REDUCE}(\text{NOT ced_care} \text{ OR } \text{NOT (ce_data XOR ced_value)})$

$\text{trig_out} = \text{AND_REDUCE}(\text{NOT trig_care} \text{ OR } \text{NOT (trig XOR trig_value)})$

It should be noted that this register can be written at any time (e.g. during data capture) and the LA will work properly and will update immediately the capture mode.

CED_Value Register

Address 0x20 – 0x23, Reset value: 0x0...00

The width of this register depends on the *ce_dwidth* parameter and is implemented when *ced_type*>0. The logic function of this register is similar like for Trig_Value register. The difference is that different input *ce_data* (not trig) is the input logic. The result of the logic can be used as captured data clock enable (data is captured or not) or can be used as an additional trigger – to extend the trigger width or condition. The output function is software configured by the *dce_control* register.

It should be noted that the CE logic is a very sophisticated function especially when combined with Run Length Coding. This can cause that captured memory is never filled, e.g. when captured data are the same for the clock enable condition.

In order to record the trigger moment the additional clock enable for captured data is generated when the trigger condition is met and the logic analyser is not in the run mode (the trigger moment). This may however cause that during trigger ignore time (see trigger register) the captured data are filled with the trigger rather than clock enable signals.

CED_Care Register

Address 0x24-0x27, Reset value: 0x0...00

Enable (1) or not (0) each bit of the dce_value. See trig_care and dce_value registers description.

CED_Value64 Register

Address 0x28 – 0x2B, Reset value: 0x0...00

The upper 32 bit of the dce_value register – valid when ce_dwidth>32

CED_Care64 Register

Address 0x2C-0x2F, Reset value: 0x0...00

The upper 32 bit of the dce_care register – valid when ce_dwidth>32

Example 1

Data, trig and ce_data input are connected together and the following input is used:

D0÷D7- Data bus

D8÷D15- Address bus

D16- Stb – Strobe active data and bus (driven by a bus master)

D17 – Ack – Acknowledge – active when a slave device is ready for data transfer

D18 – RNW – Read not write – 1 for reading, 0 –for writing.

In order to capture only a write cycles to the memory location A0÷AF triggered by a read from address location A0 when the read value is 50÷53, the following should be set:

trig_value= 0x6A050

trig_care = 0x6FFFC

ced_control= 0x2 (ce_dat='1')

ced_value= 0x2A000

ced_care= 0x2F000

Example 2

Input is the same as in the Example 1.

Trigger when address bus was (in the previous clock cycle) different than 5x and strobe was active and now address is equal 51 and strobe is 1.

trig_value= 15100

trig_care= 1FF00

ced_control= B (and_trig=1, ced_reg=1, ced_active_low=1)

ced_value= 05000

ced_care= 1F000

Internal Dual Port Memory

Address: $M - (2 \cdot M - 1)$ *Size:* M

The data probed by the LA are written to the internal memory and then these data should be read out off-line to the PC to file la_data.bin. The size of the internal memory M [Bytes] is defined by the *mem_adr_width* and *data_width* generic parameters. $M = 2^{\text{mem_adr_width}} \cdot \text{data_width} / 8 = 2^{\text{adr_width}-1}$.

4

Operation

The normal operation of the LA consists from three different procedures:

- Data acquisition when data (watched signals) are sampled into the internal memory.
- Data transfer, when previously acquired data are transferred to your PC through the WISHBONE bus.
- Signal watching, when the previously transferred data are displayed on your PC by a VHDL simulator and the `la_view.vhd` file.

Data Acquisition

Log_anal entity instantiation

In order to see what happens inside the prototyped circuit you should instantiate the `log_anal` entity into your design and connect the *data* input to the signals which states are to be watched. Then the *trig* input should be connected to the signals that will trigger the LA. In most cases the *data* and *trig* signals are the same. Then the LA should also be connected to the WISHBONE bus through which the LA trigger and operation setting are written and then the acquired data are transferred to a personal computer. Instantiating the LA into a circuit might influence the normal operation of your circuit as the LA uses standard CLB and routing resources of a FPGA. Nevertheless it might be the only way to learn what happens inside your real circuit. Besides every input (*data* and *trig*) is connected directly to the flip-flop in order to reduce propagation time and to allow Place & Route tool to optimise rather your circuit than the LA routing.

After the `log_anal` has been instantiated into your design, the standard design procedure should be invoked: synthesis, place & route, simulation and design downloading to the FPGA. Every time a different signal or trigger is used, the above procedure should be repeated. This might be time consuming especially when a lot of different signals are to be watch to spot an error. In this case, it is encourage to use inputs multiplexers (not included in the source code) which configured by the WISHBONE bus might switch a proper signals to the LA inputs.

The `log_anal` entity consists from the following sub-entities:

`la_trig` –main function is trigger logic (it is included inside the `log_anal.vhd`).

`la_mem` – is a technology dependent description of the dual port (dual clock for *two_clocks=1*) synchronous SRAM.

Example for Xilinx Virtex family:

la_mem – internal memory, this entity should be replaced by the technology dependent dual port synchronous memory. Default la_mem entity divides large memory to smaller BlockRAM which size depends on the technology and therefore is defined by a constant values: constant *BRAM_max_data_width*, *BRAM_size* which should be adjust to proper values whenever the default la_mem entity is used for the synthesis.

la_bram – this file should be used only when default la_mem entity is used. This entity contains a synthesable (only by Xilinx XST) single clock (parameter *two_clocks*= 0) VHDL description. For two independent clocks (*two_clocks*=1) a BlockRAM components (RAMB4_S?_S?) are directly instantiated.

Clocks and Clock Enables (CEs)

The signals connected to the *data* and *trig* inputs are sampled on the rising edge of the *clk* input. It is strongly recommended to uses single clock operation (parameter: *two_clocks*= 0) whenever the data sampling clock (signal *clk*) is the same as the WISHBONE clock (signal *wb_clk_i*), as it will reduce the circuit area and the WISHBONE access time.

In the case when input *data* and *trig* should be sampled several times per clock cycle or WISHBONE clock differs from the clock for the watched signals, two separate clocks (*two_clocks*=1) should be used. It should be noted that data sampling clock frequency should not be very small (especially must not be gated – $f = 0$ Hz) as the control register write time is at least sum of clocks periods: $T_{clk} + T_{wb_clk_i}$. For reading access time is constant and equal $2 \cdot T_{wb_clk_i}$. Consequently for $f_{clk} \ll f_{wb_clk_i}$ it is recommended to use the data and trigger clock enable (CE) signals.

In addition to the above CE consideration, the CE signals (*ce_data* and *ce_trig*) can be also used when only specific data sequence should be watched, e.g. when watching WISHBONE activity only when *wb_cyc*=1 or only cases when data transfer occurs *wb_ack*=1. This allows for reducing memory size.

Data transfer

The previously sampled data written to the internal memory should be transferred to a personal computer and written to the *la_data.bin* file. This file should contain not only sampled data (internal memory contents) but also the Control Registers' settings. Consequently the following two instructions should be executed by the wishbone master device:

readblock *file_name address_start address_stop append* – the readblock instruction format.

where: *file_name*= *la_data.bin* – default file name.

address_start – address from which data reading starts

address_stop – address of the last transferred data.

append – File append option: 0- create a new file, 1- append read data to the already created file

M- the size of the internal memory $M = 2^{adr_width-1}$.

adr_width – input parameter specified in the the log_anal by generic value.

Instructions:

```
readblock la_data.bin 0 0x3F 0
```

```
readblock la_data.bin M (2*M-1) 1
```

Watching the Recorded Signals

After the file *la_data.bin* has been created on your PC you can watch the recorded signals sequence employing a VHDL simulator and *la_view.vhd*. At first you should update the generics in the file *la_view.vhd* to properly watch the captured data.

The generic values are:

RLC_max_count - limits the maximum number of the same signal displayed during simulation. The same signal values can be repeated millions of times and Run Length Coding module can compress them very efficiently using only 2 samples. Unfortunately the VHDL simulator often cannot display such long sequence. Besides such waveform is often difficult to analyse by the user (nothing is changing for a very long time). Consequently the maximum number of clock cycles for which no change is detected is limited to *RLC_max_count* values. This generic value refers only to the viewing process, and this value can be changed at any time without the need for rerunning the LA hardware capture procedure. To disable this reduction, use *RLC_max_count=-1*.

trigger_same_as_data – The LA has separate data and trigger bus. In the case when these buses are the same (which often the case), dedicated logic inside the *la_view.vhd* displays when the trigger condition is satisfy – signal trigger is then equal '1'.

file_name - the file name which contains captured data (by default "la_data.bin")

Afterwards you should start the VHDL simulator. During simulation the *la_view* reads the *la_data.bin* file and assigns the acquired signals at *data* input (*log_anal*) to signal name *d*. Very often watched signals connected to the *data* bus have different names: e.g. *rd*, *wr*, *ack*, *adr(3:0)*, and it would be rather difficult to tract which index of the *d* signal corresponds to the signal of interest. Therefore it is strongly encouraged to define your own signals, and then assign a corresponding *d* signal to it. Therefore the *la_view.vhdl* should be edited in the following way for the previously given example of used signals:

Example:

The signal assignment for the *log_anal*:

```
data(6 downto 0) <= adr & ack & wr & rd;
```

The signal assignment inside the *la_view*:

```
signal rd, wr, ack: std_logic;  
signal adr: std_logic_vector(3 downto 0);  
begin --(architecture)  
  rd<= d(0);  
  wr<= d(1);  
  ack<= d(2);  
  adr<= d(6 downto 3);
```

It should be noted that by default OPB (by IBM) bus is defined.

The simulation is automatically stopped when all recorded samples are shown. The following report is presented:

O.K. All acquired data in the LA has already been shown.

Advance design simulation examples

The acquired data might be also used for real signals sequence simulation, i.e. the data recorded by the LA are then used to stimulate your design. For example: a device is connected to a PC by the parallel port. It is rather difficult (or at least time consuming) to specify real signal sequence issued by the parallel port. However the LA can be easily used to record the real signals sequence and then the data can be used as the stimulus for simulation. In this case you should place your top-level entity inside the *la_view* and feed the inputs with the proper *d* signal.

Another simulation approach is to compare the real signals obtained by the LA with a simulation results. In this case the *la_view* should be combined with the simulated circuit. Then the simulation result should be compared at every rising clock with the signals obtained by the LA. The following VHDL code might be included inside your code:

```
process(clk)  
  variable error: std_logic:= '0';  
  begin  
    if clk'event and clk='1' then  
      for i in 0 to data_width-1 loop  
        error:= error OR ( d_from_LA(i) XOR d_simulated(i) );  
      end loop;  
    end if;  
  end process;
```

Run Length Coding

For generic: *use_run_length_ciding*= 1, the run length ciding compression is implemented. Consequently if input *data* does not change for several clock cycles then the memory writes is reduced only to the data and the number of repeats of the data. For example for *data*= 001E for consecutive 9 clock cycles only two memory writes occur (001E and 8007 (MSB=1 and count value -2)) instead of the 9 memory writes. In this way the virtual memory size is enlarged especially when input data are often repeated. The drawback of the run length ciding is that the most significant bit (*data_width-1*) is used for data compression and not for data acquisition, therefore **the acquired data width is reduced by 1**. The run length coding logic occupies little additional resources: (*data_width-1*) 2-input xor gates; (*data_width-1*)-input AND gate, *data_width*-bit counter and *data_width* flip-flops. These additional resources however significantly enlarge the number of required data, therefore the memory size can be reduced. Consequently it is strongly recommended to implement the run-length coding. The only exception is that the input sequence is completely random and changes every clock cycle, in this case the run-length coding reduces the acquired data width.

The RLC is very efficient compression, and in the case when a dead-lock occurs the LA cannot work properly – the captured data memory is never filled. Consequently an additional setting: bit RLC (not) in *ced_control* register can be used to stop using the RLC.

5 Implementation

Xilinx

Optimal Generic Selection

Generic *mem_adr_width* and *data_width* should be selected with respect to the FPGA built-in memory size.

Precaution: Constant Values: *BRAM_max_data_width*, *BRAM_size* specified in the *la_mem* entity define a single BRAM maximum data width and size and should be specified according to FPGA family.

Virtex (BRAM 4kb)

BRAM_max_data_width= 16, *BRAM_size*= 4kb (4096)

# BRAM	data_width	mem_adr_width	adr_width
1	8	9	10
	16	8	10
2	8	10	11
	16	9	11
	32	8	11
4	8	11	12
	16	10	12
	32	9	12

Virtex II (BRAM 16 kb)

BRAM_max_data_width= 32, *BRAM_size*= 16kb (16 384) (parity bits not included)

# BRAM	data_width	mem_adr_width	adr_width
1	8	11	12
	16	10	12
	32	9	12

2	8	12	13
	16	11	13
	32	10	13
4	8	13	14
	16	12	14
	32	11	14

Implementation results for Virtex Family (XCV300PQ240-6)

For the following log_anal parameters:

data_width:= 16, mem_adr_width:= 9, adr_width:= 11 trig_width:= 8, two_clocks:= 0

Implementation result:

Number of Slices: 78 out of 3,072 2%

Number of Slices containing

unrelated logic: 0 out of 78 0%

Number of Slice Flip Flops: 40 out of 6,144 1%

Total Number 4 input LUTs: 116 out of 6,144 1%

Number used as LUTs: 105

Number used as a route-thru: 11

IOB Flip Flops: 34

Number of Block RAMs: 2 out of 16 12%

Number of GCLKs: 1 out of 4 25%

Total equivalent gate count for design: 34,245

Minimum period is 13.492ns.