



WISHBONE SCOPE SPECIFICATION

Dan Gisselquist, Ph.D.
dgisselq (at) ieee.org

March 24, 2017

Copyright (C) 2017, Gisselquist Technology, LLC

This project is free software (firmware): you can redistribute it and/or modify it under the terms of the GNU General Public License as published by the Free Software Foundation, either version 3 of the License, or (at your option) any later version.

This program is distributed in the hope that it will be useful, but WITHOUT ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE. See the GNU General Public License for more details.

You should have received a copy of the GNU General Public License along with this program. If not, see <http://www.gnu.org/licenses/> for a copy.

Revision History

Rev.	Date	Author	Description
0.3	6/22/2015	Gisselquist	Minor updates to enhance readability
0.2	6/22/2015	Gisselquist	Finished Draft
0.1	6/22/2015	Gisselquist	First Draft

Contents

	Page
1 Introduction	1
2 Operation	2
3 Registers	4
3.1 Control Register	4
3.2 Data Register	5
4 Clocks	6
5 Wishbone Datasheet	7
6 I/O Ports	8

Tables

Table		Page
3.1.	List of Registers	4
3.2.	Control Register	5
5.1.	Wishbone Datasheet	7
6.1.	List of IO ports	8

Preface

This project began, years ago, for all the wrong reasons. Rather than pay a high price to purchase a Verilog simulator and then to learn how to use it, I took working Verilog code, to include a working bus, added features and used the FPGA system as my testing platform. I arranged the FPGA to step internal registers upon command, and to make many of those registers available via the bus.

When I then needed to make the project run in real-time, as opposed to the manually stepped approach, I generated a scope like this one. I had already bench tested the components on the hardware itself. Thus, testing and development continued on the hardware, and the scope helped me see what was going right or wrong. The great advantage of the approach was that, at the end of the project, I didn't need to do any hardware in the loop testing. All of the testing that had been accomplished prior to that date was already hardware in the loop testing.

When I left that job, I took this concept with me and rebuilt this piece of infrastructure using a Wishbone Bus.

Dan Gisselquist, Ph.D.

1.

Introduction

The Wishbone Scope is a debugging tool for reading results from the chip after events have taken place. In general, the scope records data until some (programmable) holdoff number of data samples after a trigger has taken place. Once the holdoff has been reached, the scope stops recording and asserts an interrupt. At this time, data may be read from the scope in order from oldest to most recent. That's the basics, now for two extra details.

First, the trigger and the data that the scope records are both implementation dependent. The scope itself is designed to be easily reconfigurable from one build to the next so that the actual configuration may even be build dependent.

Second, the scope is built to be able to run off of a separate clock from the bus that commands and controls it. This is configurable, set the parameter "SYNCHRONOUS" to '1' to run off of a single clock. When running off of two clocks, it means that actions associated with commands issued to the scope, such as manual triggering or being disabled or released, will not act synchronously with the scope itself—but this is to be expected.

Third, the data clock associated with the scope has a clock enable line associated with it. Depending on how often the clock enable line is enabled may determine how fast the scope is PRIMED, TRIGGERED, and eventually completes its collection.

Finally, and in conclusion, this scope has been an invaluable tool for testing, for figuring out what is going on internal to a chip, and for fixing such things. I have fixed interactions over a PS/2 connection, Internal Configuration Access Port (ICAPE2) interfaces, mouse controller interactions, bus errors, quad-SPI flash interactions, and more using this scope.

2.

Operation

So how shall one use the scope? The scope itself supports a series of states:

1. RESET

Any write to the control register, without setting the high order bit, will automatically reset the scope. Once reset, the scope will immediately start collecting.

2. PRIMED

Following a reset, once the scope has filled its memory, it enters the PRIMED state. Once it reaches this state, it will be sensitive to a trigger.

3. TRIGGERED

The scope may be TRIGGERED either automatically, via an input port to the core, or manually, via a wishbone bus command. Once a trigger has been received, the core will record a user configurable number of further samples before stopping.

4. STOPPED

Once the core has STOPPED, the data within it may be read back off.

Let's go through that list again. First, before using the scope, the holdoff needs to be set. The scope is designed so that setting the scope control value to the holdoff alone, with all other bits set to zero, will reset the scope from whatever condition it was in, freeing it to run. Once running, then upon every clock enabled clock, one sample of data is read into the scope and recorded. Once every memory value is filled, the scope has been PRIMED. Once the scope has been PRIMED, it will then be responsive to its trigger. Should the trigger be active on an input clock with the clock-enable line set, the scope will then be TRIGGERED. It will then count for the number of clocks in the holdoff before stopping collection, placing it in the STOPPED state. ¹ If the holdoff is zero, the last sample in the buffer will be the sample containing the trigger. Likewise if the holdoff is one less than the size of the memory, the first sample in the buffer will be the one containing the trigger.

There are two further commands that will affect the operation of the scope. The first is the MANUAL trigger command/bit. This bit may be set by writing the holdoff to the control register while setting this bit high. This will cause the scope to trigger as soon as it is primed. If the RESET_n bit is also set so as to prevent an internal reset, and if the scope was already primed, then manual trigger command will cause it to trigger immediately.

¹You can even change the holdoff while the scope is running by writing a new holdoff value together with setting the RESET_n bit of the control register. However, if you do this after the core has triggered it may stop at some other non-holdoff value!

The last command that can affect the operation of the scope is the `DISABLE` command/bit in the control register. Setting this bit will prevent the scope from triggering, or if `TRIGGERED`, it will prevent the scope from generating an interrupt.

Finally, be careful how you set the clock enable line. If the clock enable line leaves the clock too often disabled, the scope might never prime in any reasonable amount of time.

So, in summary, to use this scope you first set the holdoff value in the control register. Second, you wait until the scope has been `TRIGGERED` and `STOPPED`. Finally, you read from the data register once for every memory value in the buffer and you can then sit back, relax, and study what took place within the FPGA. Additional modes allow you to manually trigger the scope, or to disable the automatic trigger entirely.

3.

Registers

This scope core supports two registers, as listed in Tbl. 3.1: a control register and a data register. Each register will be discussed in detail in this chapter.

Name	Address	Width	Access	Description
CONTROL	0	32	R/W	Configuration, control, and status of the scope.
DATA	1	32	R(/W)	Read out register, to read out the data from the core. Writes to this register reset the read address to the beginning of the buffer, but are otherwise ignored.

Table 3.1: List of Registers

3.1 Control Register

The bits in the control register are defined in Tbl. 3.2. The register has been designed so that one need only write the holdoff value to it, while leaving the other bits zero, to get the scope going. On such a write, the `RESET_n` bit will be a zero, causing the scope to internally reset itself. Further, during normal operation, the high order nibble will go from 4'h8 (a nearly instantaneous reset state) to 4'h0 (running), to 4'h1 (`PRIMED`), to 4'h3 (`TRIGGERED`), and then stop at 4'h7 (`PRIMED`, `TRIGGERED`, and `STOPPED`). Finally, user's are cautioned not to adjust the holdoff between the time the scope triggers and the time it stops—just to guarantee data coherency.

While this approach works, the scope has some other capabilities. For example, if you set the `MANUAL` bit, the scope will trigger as soon as it is `PRIMED`. If you set the `MANUAL` bit and the `RESET_n` bit, it will trigger immediately if the scope was already `PRIMED`. However, if the `RESET_n` bit was not also set, a reset will take place and the scope will start over by first collecting enough data to be `PRIMED`, and only then will the `MANUAL` trigger take effect.

A second optional capability is to disable the scope entirely. This might be useful if, for example, certain irrelevant things might trigger the scope. By setting the `DISABLE` bit, the scope will not automatically trigger. It will still record into its memory, and it will still prime itself, it will just not trigger automatically. The scope may still be manually `TRIGGERED` while the `DISABLE` bit is set. Likewise, if the `DISABLE` bit is set after the scope has been `TRIGGERED`, the scope will continue to its natural stopped state—it just won't generate an interrupt.

Bit #	Access	Description
31	R/W	RESET_n. Write a '0' to this register to command a reset. Reading a '1' from this register means the reset has not finished crossing clock domains and is still pending.
30	R	STOPPED, indicates that all collection has stopped.
29	R	TRIGGERED, indicates that a trigger has been recognized, and that the scope is counting for holdoff samples before stopping.
28	R	PRIMED, indicates that the memory has been filled, and that the scope is now waiting on a trigger.
27	R/W	MANUAL, set to invoke a manual trigger.
26	R/W	DISABLE, set to disable the internal trigger. The scope may still be TRIGGERED manually.
25	R	RZERO, this will be true whenever the scope's internal address register is pointed at the beginning of the memory.
20-24	R	LGMEMLen, the base two logarithm of the memory length. Thus, the memory internal to the scope is given by $1 \ll \text{LGMEMLen}$.
0-19	R/W	Unsigned holdoff

Table 3.2: Control Register

There are two other interesting bits in this control register. The RZERO bit indicates that the next read from the data register will read from the first value in the memory, while the LGMEMLen bits indicate how long the memory is. Thus, if LGMEMLen is 10, the FIFO will be $(1 \ll 10)$ or 1024 words long, whereas if LGMEMLen is 14, the FIFO will be $(1 \ll 14)$ or 16,384 words long.

3.2 Data Register

This is perhaps the simplest register to explain. Before the core stops recording, reads from this register will produce reads of the bits going into the core, save only that they have not been protected from any meta-stability issues. This is useful for reading what's going on when the various lines are stuck. After the core stops recording, reads from this register return values from the stored memory, beginning at the oldest and ending with the value holdoff clocks after the trigger. Further, after recording has stopped, every read increments an internal memory address, so that after $(1 \ll \text{LGMEMLen})$ reads (for however long the internal memory is), the entire memory has been returned over the bus. If you would like some assurance that you are reading from the beginning of the memory, you may either check the control register's RZERO flag which will be '1' for the first value in the buffer, or you may write to the data register. Such writes will be ignored, save that they will reset the read address back to the beginning of the buffer.

4.

Clocks

This scope supports two clocks: a wishbone bus clock, and a data clock. If the internal parameter “SYNCHRONOUS” is set to zero, proper transfers will take place between these two clocks. Setting this parameter to a one will save some flip flops and logic in implementation. The speeds of the respective clocks are based upon the speed of your device, and not specific to this core.

5.

Wishbone Datasheet

Tbl. 5.1 is required by the wishbone specification, and so it is included here. The big thing to notice

Description	Specification																				
Revision level of wishbone	WB B4 spec																				
Type of interface	Slave, Read/Write, pipeline reads supported																				
Port size	32-bit																				
Port granularity	32-bit																				
Maximum Operand Size	32-bit																				
Data transfer ordering	(Irrelevant)																				
Clock constraints	None.																				
Signal Names	<table border="1"> <thead> <tr> <th>Signal Name</th> <th>Wishbone Equivalent</th> </tr> </thead> <tbody> <tr> <td><code>i_wb_clk</code></td> <td><code>CLK_I</code></td> </tr> <tr> <td><code>i_wb_cyc</code></td> <td><code>CYC_I</code></td> </tr> <tr> <td><code>i_wb_stb</code></td> <td><code>STB_I</code></td> </tr> <tr> <td><code>i_wb_we</code></td> <td><code>WE_I</code></td> </tr> <tr> <td><code>i_wb_addr</code></td> <td><code>ADR_I</code></td> </tr> <tr> <td><code>i_wb_data</code></td> <td><code>DAT_I</code></td> </tr> <tr> <td><code>o_wb_ack</code></td> <td><code>ACK_O</code></td> </tr> <tr> <td><code>o_wb_stall</code></td> <td><code>STALL_O</code></td> </tr> <tr> <td><code>o_wb_data</code></td> <td><code>DAT_O</code></td> </tr> </tbody> </table>	Signal Name	Wishbone Equivalent	<code>i_wb_clk</code>	<code>CLK_I</code>	<code>i_wb_cyc</code>	<code>CYC_I</code>	<code>i_wb_stb</code>	<code>STB_I</code>	<code>i_wb_we</code>	<code>WE_I</code>	<code>i_wb_addr</code>	<code>ADR_I</code>	<code>i_wb_data</code>	<code>DAT_I</code>	<code>o_wb_ack</code>	<code>ACK_O</code>	<code>o_wb_stall</code>	<code>STALL_O</code>	<code>o_wb_data</code>	<code>DAT_O</code>
	Signal Name	Wishbone Equivalent																			
	<code>i_wb_clk</code>	<code>CLK_I</code>																			
	<code>i_wb_cyc</code>	<code>CYC_I</code>																			
	<code>i_wb_stb</code>	<code>STB_I</code>																			
	<code>i_wb_we</code>	<code>WE_I</code>																			
	<code>i_wb_addr</code>	<code>ADR_I</code>																			
	<code>i_wb_data</code>	<code>DAT_I</code>																			
	<code>o_wb_ack</code>	<code>ACK_O</code>																			
<code>o_wb_stall</code>	<code>STALL_O</code>																				
<code>o_wb_data</code>	<code>DAT_O</code>																				

Table 5.1: Wishbone Datasheet

is that this core acts as a wishbone slave, and that all accesses to the wishbone scope registers become 32-bit reads and writes to this interface. You may also wish to note that the scope supports pipeline reads from the data port, to speed up reading the results out.

What this table doesn't show is that all accesses to the port take a single clock. That is, if the `i_wb_stb` line is high on one clock, the `i_wb_ack` line will be high the next. Further, the `o_wb_stall` line is tied to zero.

6.

I/O Ports

The ports are listed in Table 6.1. At this point, most of these ports should have been well defined and

Port	Width	Direction	Description
<code>i_clk</code>	1	Input	The clock the data lines, clock enable, and trigger are synchronous to.
<code>i_ce</code>	1	Input	Clock Enable. Set this high to clock data in and out.
<code>i_trigger</code>	1	Input	An active high trigger line. If this trigger is set to one on any clock enabled data clock cycle, once the scope has been PRIMED, it will then enter into its TRIGGERED state.
<code>i_data</code>	32	Input	32-wires of ... whatever you are interested in recording and later examining. These can be anything, only they should be synchronous with the data clock.
<code>i_wb_clk</code>	1	Input	The clock that the wishbone interface runs on.
<code>i_wb_cyc</code>	1	Input	Indicates a wishbone bus cycle is active when high.
<code>i_wb_stb</code>	1	Input	Indicates a wishbone bus cycle for this peripheral when high. (See the wishbone spec for more details)
<code>i_wb_we</code>	1	Input	Write enable, allows indicates a write to one of the two registers when <code>i_wb_stb</code> is also high.
<code>i_wb_addr</code>	1	Input	A single address line, set to zero to access the configuration and control register, to one to access the data register.
<code>i_wb_data</code>	32	Input	Data used when writing to the control register, ignored otherwise.
<code>o_wb_ack</code>	1	Output	Wishbone acknowledgement. This line will go high on the clock after any wishbone access, as long as the wishbone <code>i_wb_cyc</code> line remains high (i.e., no ack's if you terminate the cycle early).
<code>o_wb_stall</code>	1	Output	Required by the wishbone spec, but always set to zero in this implementation.
<code>o_wb_data</code>	32	Output	Values read, either control or data, headed back to the wishbone bus. These values will be valid during any read cycle when the <code>i_wb_ack</code> line is high.

Table 6.1: List of IO ports

described earlier in this document. The only new things are the data clock, `i_clk`, the clock enable for the data, `i_ce`, the trigger, `i_trigger`, and the data of interest itself, `i_data`. Hopefully these are fairly self explanatory by this point. If not, just remember the data, `i_data`, are synchronous to the clock, `i_clk`. On every clock where the clock enable line is high, `i_ce`, the data will be recorded until the scope has stopped. Further, the scope will stop some programmable holdoff number of clock enabled data clocks after `i_trigger` goes high. Further, `i_trigger` need only be high for one clock cycle to be noticed by the scope.