A Trigger- and Data-Readout-Module for the ARA1 firmware

Kael Hanson, Thomas Meures

July 20, 2011

Contents

1 General functionality (Introduction)

The described module has different purposes. It is supposed to:

1. receive all trigger signals and process them properly to allow the correct association of data to the given trigger signals,
2. communicate with the history_buffer and the block manager, to lock or unlock the associated data blocks for each trigger,
3. store the data-block addresses and do the actual digitization and readout of the data belonging to a trigger.

For this purpose it consists of different submodules, shown in figure 1.1.

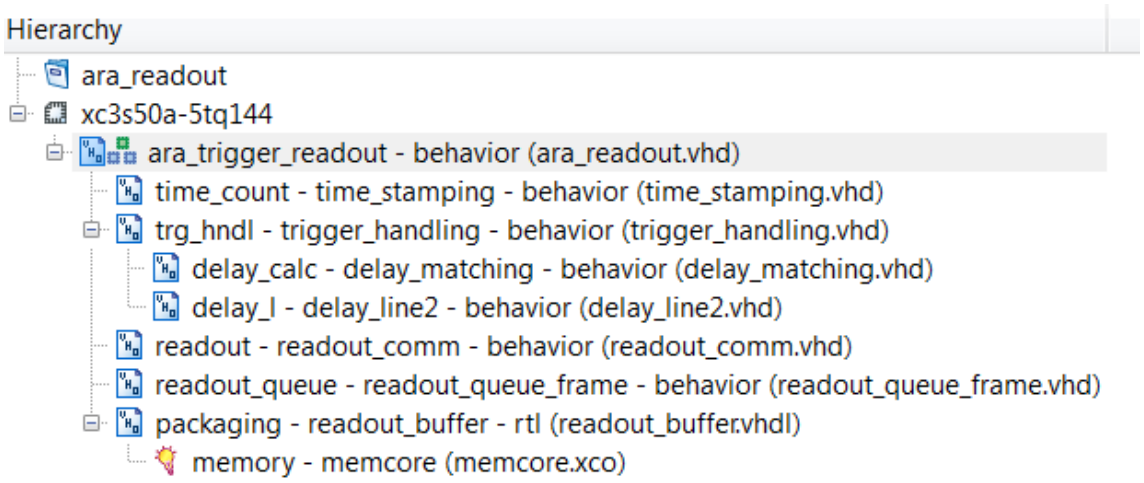

Figure 1.1 The design flow of the ARA trigger-readout-module.

In short terms, *the time_stamping* module provides the system timestamp, initialyzed at a signal of choice. The output consists of two timestamps, which are just different parts of the overall timestamp (see fig. 1.2).
The *trigger_handling* module receives all the different triggers implemented in the ARA-firmware and processes them, according to different parameters in the firmware configuration, to provide one combined trigger signal with the information about the delay and the combination of triggers in each clock cycle.
These signals are passed to the *readout_comm* module. Here the block address of the data corresponding to the triggers are requested from the *history_buffer*, using the information coming from the *trigger_handling* module, and the blocks are assigned to be locked by the *block_manager¿* In addition to that the block addresses are associated with a timestamp and the trigger combination and sent to *readout_queue_frame* module.
In the *readout_queue_frame* module, the block addresses are stored into a block RAM, with the trigger combination and the timestamp. From this block RAM the information are passed to the *readout_buffer*.
Here the actual digitization and readout of the data, stored at the given block address, is controlled.

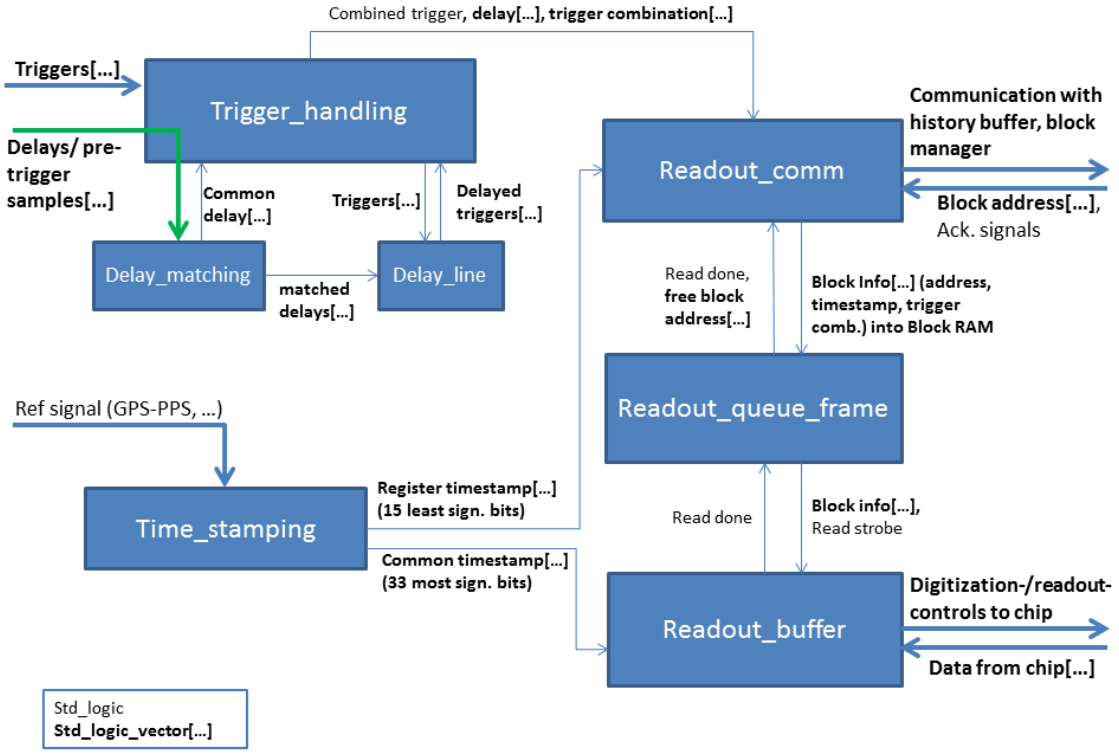

Figure 1.2 The module network inside the ARA trigger-readout-module (only the most important signals are shown).

2 The used modules

2.1 Time_stamping

In this module a timestamp, initialized with an external reference signal of any kind (ex.: GPS-PPS signal, or some signal once a week), will be incremented with each clock cycle. The timestamp is saved in a 48 bit vector, thus, at a clock speed of 100 MHz, it can be incremented for about a month time (more precisely 32.5 days), before rolling over and restarting at zero. This timestamp will be devided into a 15 bit **register_timestamp** (least significant part) and a 33 bit **common_timestamp** (most significant part), which feed into different modules in the design (see fig. 1.2).
The **register_timestamp** is sufficiently long to count for 327 μs, which should be sufficient for the readout of multiple blocks. In fact it is chosen to not roll over more then once for each data block, between being stored in the readout queue and being actually read out. A roll over will be recognized in the main module and it will be accounted for it.

The correct implementation concerning an external refernce still has to be done, once the reference is chosen for the firmware.

2.2 Trigger_handling

Here all triggers which are implemented arrive and are processed. Since they can have different delay times and pre-trigger-samples to be read out, they will all be delayed by means of the *delay_matching* and the *delay_line* modules, in order to fit the highest possible delay in the current configuration. This is one possibility to avoid conflicts in later data processing.

1. The *delay_matching* module takes the delays of all triggers from the configuration, determines which is the highest and provides the difference of all other delays to this highest delay, as well as the highest delay itself.

2. In the *delay_line* module, the trigger signals are delayed to the given delay difference by means of shift registers. The output of this module will be the triggers, all having the same delay to the time when corresponding data is buffered on the chip.

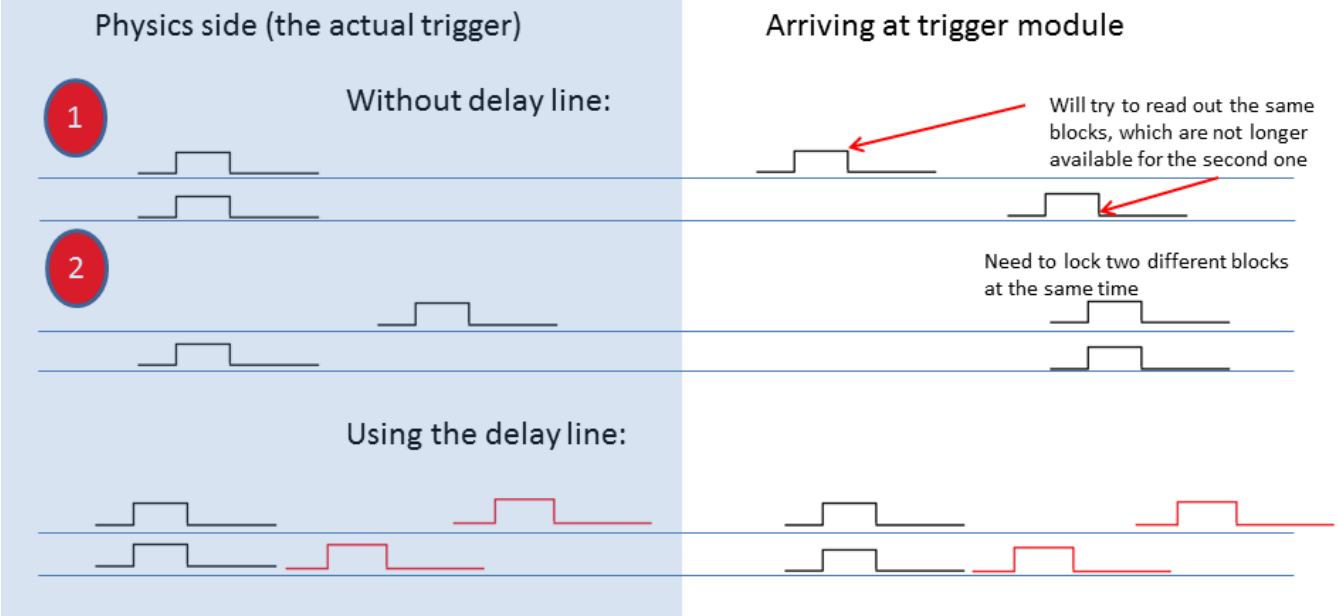

Figure 2.1 An illustration of the functionality of the delay handling.

In the *trigger_handling* module, these triggers are combined to one trigger signal, being the "**OR**" of all triggers. Moreover a vector of the delayed triggers is created showing the given trigger combination at each clock cycle. These two signals are passed together with their delay (which is now the common delay for all of them) to the *readout_comm* module.

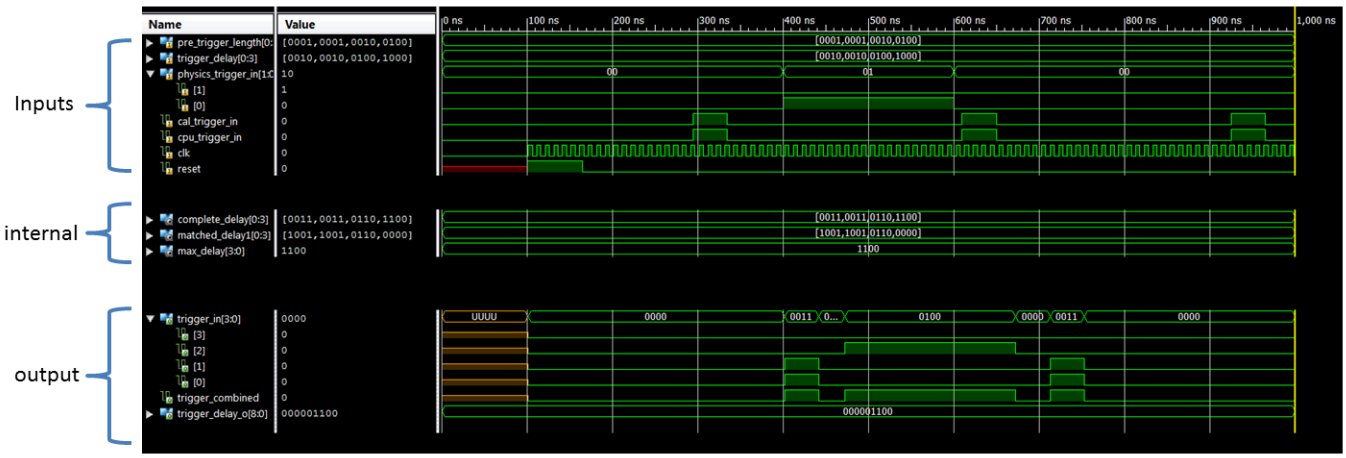

Figure 2.2 An example simulation of the behavior of the *trigger_handling* module.

2.3 Readout_comm

This module connects the *trigger_handling* with the *readout_queue_frame* and does the communication with the *block_manager* and the *history_buffer*.
The signals coming from the *trigger_handling* are directly passed to the *history_buffer*, requesting the address of the data block written when the trigger was asserted. Moreover a direct connection is passing the information which are going to be written into the *readout_queue_frame*. These information consist of the timestamp at the trigger time, the **trigger combination** and the block address coming from the history buffer.
A small state machine, switching between an **idle** state and a **reading** or **liberating** state, will ensure that all signals are passed at the right time and do the communication with the *block_manager*. The three states have the following functionality:

1. In the **idle** state all outgoing control signals are set to zero and either a **trigger_processed** or a **readout_done** signal are awaited to switch to the corresponding state.

2. The **reading** state is chosen when **triggerprocessed** signal occurs. In this state the system awaits the answer from the *history_buffer* (the **history_ack** signal), because together with this signal the the requested block address is provided. In this moment all lock signals are sent to the *block_manager* and a write strobe is sent to the *readout_queue_frame*. As long as there is an incoming trigger signal high in the moment the **history_ack** signal occurs, the machine will stay in the reading state. If not it will switch back to the **idle** state.

3. The **liberating** state has the lowest priority in the state machine. This is ensured by adding a statement telling the machine to switch directly back to the **reading** state. If this doesn't happen the address of the block to be liberated is sent along with the liberating control signals to the *block_manager*. When the *block_manager* sends back an acknowledge signal, everyting is switched back to the **idle** state.

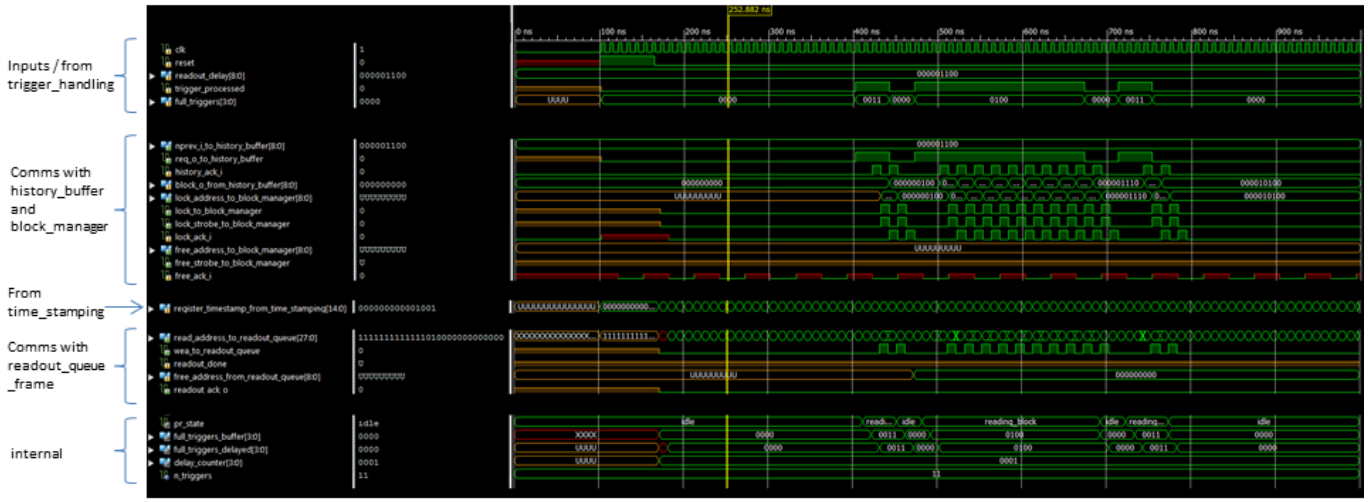

Figure 2.3 An example simulation of the behavior of the *readout_comm* module.

2.4 Readout_queue_frame

The *readout_queue_frame* is a frame around a block RAM, which is used to store the block addresses of blocks, locked to be digitized and read out. The block RAM contains 512 blocks of 36 bits. In these

35	occupation bit
$34...9 + n^o triggers + 1$	register time stamp
$9 + n^o triggers...9$	trigger combination
8...0	block address

Table 2.1 Use of the block RAM.

36 bits all needed information about the block will be saved as shown in table 2.1. It has two access lines, each with in and output. By this conflicts between writing and reading can be avoided easily.
When a write signal occurs, the block information coming fom the *readout_comm* module are written into the block RAM and the write address is incremented. Then the writing part is waiting for the next write signal.
On the other side the moment a part of the block RAM is occupied, the readout start signal is sent to the *readout_buffer* module, along with the block information. After getting a **readout_done** signal back, the block RAM content is deleted, the block address is sent to the *readout_comm* module, to be liberated and also on this side the address counter is incremented. This procedure is repeated as long as a block in the block RAM is found occupied.

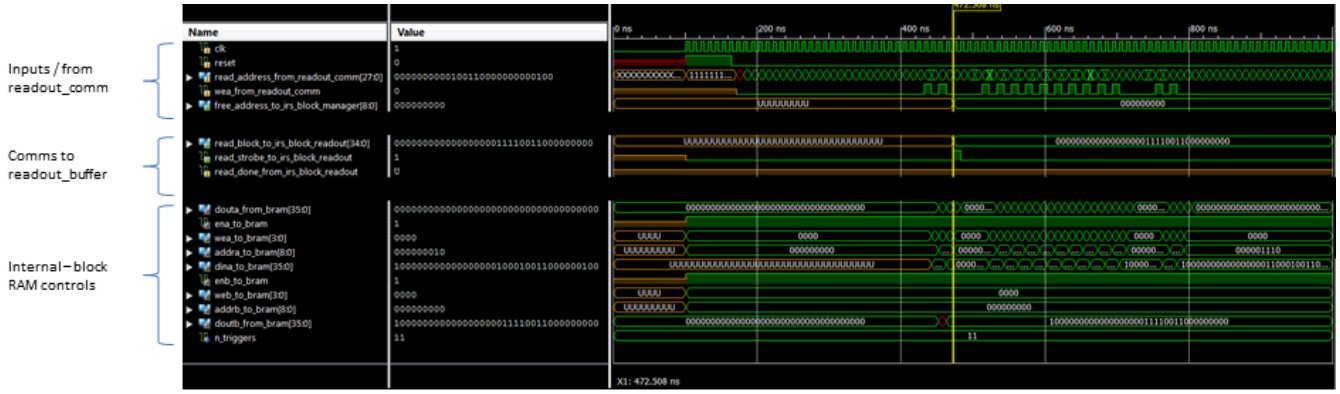

Figure 2.4 An example simulation of the behavior of the *readout_queue_frame* module.

2.5 Readout_Buffer

From this module the digitization and readout on the chip is controlled and the read data are transferred to the event interface, from where they go to a computer for further processing and finally to the data storage.
The core of this module is a block RAM of 2024 blocks, each 16 bits wide. The digitized data is written as 4 12 bit words into 3 16 bit blocks of this block RAM in the format shown in table 2.2.

Further the module consists basically of two parts. One part controls the digitization and collection of data from the IRS2 chip. The other part is dealing with the event interface, to hand the collected data over to the *dda_eval_event_fifo* module.

The first part is based on a state machine of seven states. It works in the following way:

1. The default state is the **ROS_IDLE** state. Here all control signals are set to 0. The machine will leave this state when a start signal, coming form the *readout_queue_frame*, switches high. A write control signal, going to the block RAM, is set high and the state of the system switches to **ROS_CLK1**.

2. **ROS_CLK1**, **ROS_CLK2**, **ROS_CLK3:** In these three states an event header is started with writing the timestamp of the block into the block RAM. This has to be devided into three states because the timestamp is 48 bits long. After that, the system switches to the **ROS_HDR1** state.

3. In the **ROS_HDR1** state, the other components of the header, i. ex. the block ID and the trigger combination, are written into the memory and the system is switched to the actual data readout.

4. In the **ROS_SAMPLE_START** state the blcok address and the sample and channel number of the data to be read out is sent to the IRS2 chip. Moreover a **read_enable** signal and the control signals to start the digitization are sent. The write control signal to the block RAM is switched low because from now on the 12 bit words of data have to be combined to a 16 bit vector, before being written to the memory. From this state the system switches directly to the **ROS_SAMPLE_WAIT** state.

5. In the **ROS_SAMPLE_WAIT** state the system waits for a response from the IRS2 chip. Once this response arrives, the 12 bit data of the samples is packed into 16 bit vectors and saved in the block RAM. Now the system switches forth and back between the last two states, until all samples for all channels in one block are saved in the block RAM. The form how they are saved can be seen in table 2.2. When all data is saved, the system sends out a **done** signal and swithces back to the **IDLE** state.

	block number	bit number	content
Header	0		number of words in this block
	1		low 16-bits of the system clock
	2		mid 16-bits of the system clock
	3		hi 16-bits of the system clock
	4		trigger / block ID word:
		8..0	block ID
		12..9	trigger pattern
Data	5	11..0	sample[0]
		15..12	sample[1] low 4 bits
	6	7..0	sample[1] hi 8 bits
		15..8	sample[2] low 8 bits
	7	3..0	sample[2] hi 4 bits
		15..4	sample[3]
	...	...	Etc. (it repeats - 4 12-bit words packed into 3 16-bit)

Table 2.2 Use of the block RAM in the *readout_buffer*.

The second part of the module handles the data transfer to the *dda_eval_event_fifo* module. The functionality of this part is very simple. One process is just checking, if data has been written to a block RAM address. If this is the case and if the FIFO on the other side is not occupied, the data is directly sent out (along with the needed control signals).

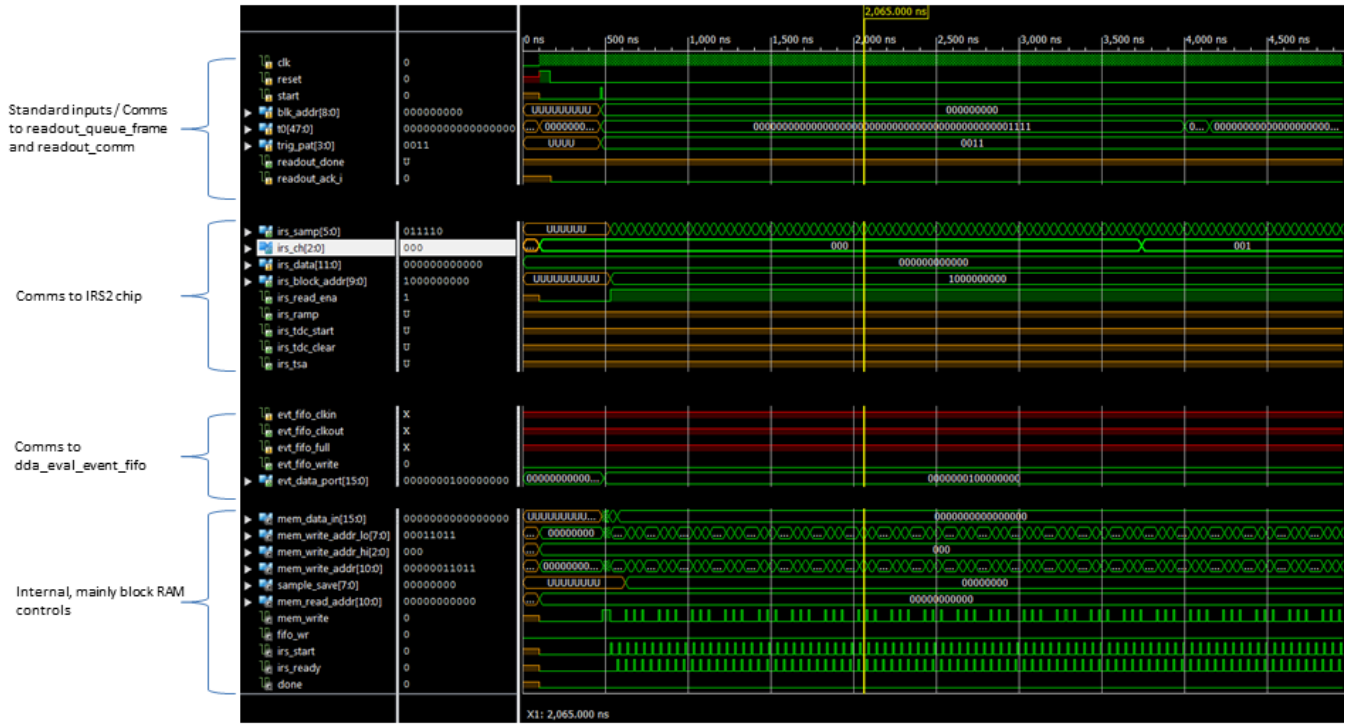

Figure 2.5 An example simulation of the behavior of the *readout_buffer* module.

3 Design properties and performance

3.1 Device utilization summary

ara_trigger_readout Project Status			
Project File:	DDA_EVAL.xise	**Parser Errors:**	No Errors
Module Name:	ara_trigger_readout	**Implementation State:**	Programming File Generated
Target Device:	xc3s700a-4fg484	• **Errors:**	No Errors
Product Version:	ISE 13.1	• **Warnings:**	117 Warnings (117 new)
Design Goal:	Balanced	• **Routing Results:**	All Signals Completely Routed
Design Strategy:	Xilinx Default (unlocked)	• **Timing Constraints:**	All Constraints Met
Environment:	System Settings	• **Final Timing Score:**	0 (Timing Report)

Device Utilization Summary				[-]
Logic Utilization	**Used**	**Available**	**Utilization**	**Note(s)**
Number of Slice Flip Flops	150	11,776	1%	
Number of 4 input LUTs	197	11,776	1%	
Number of occupied Slices	161	5,888	2%	
Number of Slices containing only related logic	161	161	100%	
Number of Slices containing unrelated logic	0	161	0%	
Total Number of 4 input LUTs	213	11,776	1%	
Number used as logic	189			
Number used as a route-thru	16			
Number used as Shift registers	8			
Number of bonded IOBs	95	372	25%	
Number of BUFGMUXs	1	24	4%	
Number of RAMB16BWEs	1	20	5%	
Average Fanout of Non-Clock Nets	2.91			

Figure 3.1 The utilization summary of the complete trigger and readout module on an Xilinx Spartan 3S device.

3.2 simulations