Synchronous FIFO with LUT, BRAM or DRAM Implementation Modes on FPGA Controller on PL/PS

Summer, 2019 Project Final Report

Giorgi Aptsiauri

San Diego State University

Author Note

Project Funded by SDSU Georgia

Project Mentor: Dr. Amir Alimohammad

Contents

Learning Objectives Fulfilled

In this project, I researched:

- A Verilog design implementation on Xilinx FPGA boards.
- Using PL/PS sides of Xilinx FPGA evaluation boards such as *MiniZed* and *ZedBoard*, separately and together (PL/PS).
- Creating/Using AXI interfaces for data exchange between PL and PS sides of the FPGA boards.
- Building block design diagrams using *Vivado IDE* that synthesize and implement successfully for the aforementioned boards.
- Utilizing different memory types on Xilinx FPGAs, particularly, from PL or PS side, using Verilog HDL or C programming language running on a single ARM core, respectively.
- Lastly and most importantly, using all previous items to understand and build a synchronous FIFO with read/write capability in the same clock cycle, with three memory implementation modes (LUT, BRAM, DRAM), design running either on FPGA fabric or as a bare-metal application on a single ARM core. More information about this in the ***Summary*** section.

FIFO Summary

The FIFO I designed for this project works on ***MiniZed board*** and supports the following combinations of controller side and memory types:

1. PL-LUT: FIFO controller written In Verilog and running on PL, memory type used is distributed RAM, i.e. LUTs.
2. PL-BRAM: FIFO controller written In Verilog and running on PL, memory type used is block RAM residing on FPGA fabric, i.e. BRAM.
3. PS-DRAM: FIFO controller written in C and running on PS, memory type used is dynamic RAM residing on the FPGA board, *but* as an independent chip.
4. PS-DRAM: FIFO controller written in C and running on PS, memory type used is block RAM residing on FPGA fabric, i.e. BRAM.

NOTE: not all combinations are feasible to implement, for example, PL-DRAM because DRAM chip is not connected to the FPGA fabric directly on *MiniZed* board which was the main board used for this project. So, the *Memory Interface Generator* in *Vivado* block design tool would not work to generate signals to control the DRAM chip.

Similar argument goes for PS-LUT. There is no easy way to create an LUT-based memory in the FPGA fabric and be able to read/write from and to it from an ARM core using C.

Downloads Required

The project was built using ***Vivado IDE 2019.1.1*** version, so this or later version is required. ***Putty*** is also required for PS_BRAM and PS_DRAM testing.

Download the code zip file from here .

How to build and run the FIFO?

PL_LUT and PL_BRAM

Principle of Operation

PL_LUT and PL_BRAM FIFOs are very similar. In fact, the code for these two versions are identical. The only difference, however, is how RAM, i.e., memory for the FIFO is instantiated. To use BRAM or LUTs for the memory, Verilog ***ramstyle*** directives, "*block"* or *"distributed"* are used, respectively. For example, open the following file, from the *Sources* tab, to view how ***ramstyle*** directive is used for BRAM instantiation:

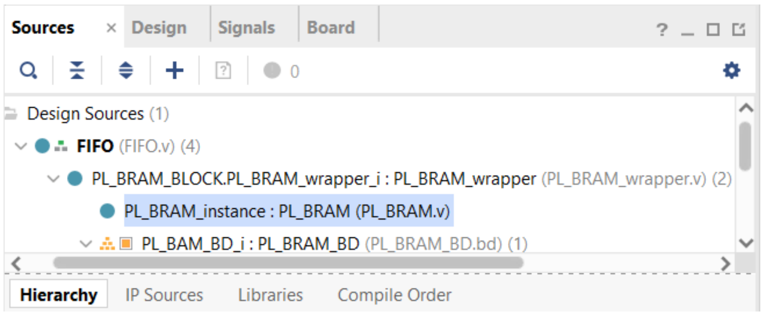
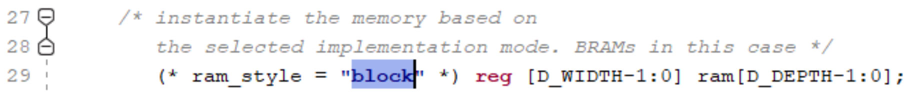

Sources
Design
Signals
Board
0
Design Sources (1)
FIFO (FIFO.v) (4)
PL_BRAM_BLOCK.PL_BRAM_wrapper_i : PL_BRAM_wrapper (PL_BRAM_wrapper.v) (2)
PL_BRAM_instance : PL_BRAM (PL_BRAM.v)
PL_BAM_BD_i : PL_BRAM_BD (PL_BRAM_BD.bd) (1)
Hierarchy
IP Sources
Libraries
Compile Order

Figure 1: PL_BRAM.v constains FIFO controller and BRAM memory instantiation code.

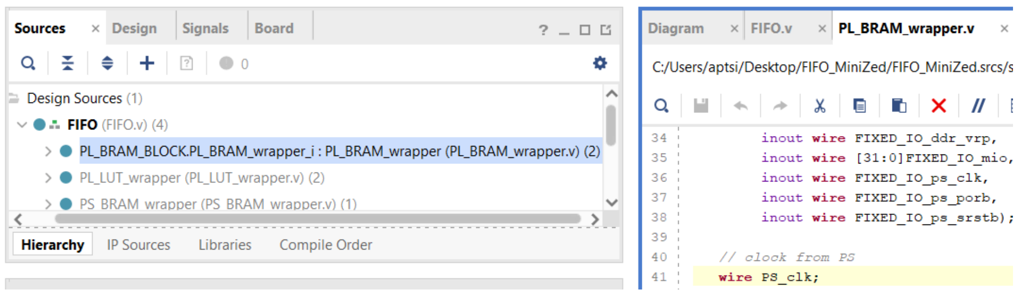
Line 29 (Fig. 2) instantiates BRAM for the FIFO which is controlled by the rest of the code in the same file. As you can see, "*block*" is used for BRAM.



```
27  /* instantiate the memory based on
28     the selected implementation mode. BRAMs in this case */
29     (* ram_style = "block" *) reg [D_WIDTH-1:0] ram[D_DEPTH-1:0];
```

Figure 2: ramstyle directive usage example

A very important note is about the system clock. MiniZed does not have an on-board clock on PL side. So, we have to use a PS clock. For that, in the block design of PL_BRAM version, for example, clock signal is extracted from the Zynq Processing System and used as an



Design Sources (1)
FIFO (FIFO.v) (4)
PL_BRAM_BLOCK.PL_BRAM_wrapper_i : PL_BRAM_wrapper (PL_BRAM_wrapper.v) (2)
PL_LUT_wrapper (PL_LUT_wrapper.v) (2)
PS_BRAM_wrapper (PS_BRAM_wrapper.v) (1)

C:/Users/aptsi/Desktop/FIFO_MiniZed/FIFO_MiniZed.srcs/s

```
        inout wire FIXED_IO_ddr_vrp,
        inout wire [31:0]FIXED_IO_mio,
        inout wire FIXED_IO_ps_clk,
        inout wire FIXED_IO_ps_porb,
        inout wire FIXED_IO_ps_srstb);

    // clock from PS
    wire PS_clk;
```

Figure 3: clock extraction from Zynq PS.

input clock to the FIFO. Fig. 3 illustrates a snippet of code from the code which handles clock extraction from PS.

This FIFO can be used as a part of bigger project which needs a customizable FIFO. For that, that project must be able to control the signals of this FIFO, like read/write enable, clock, reset, and input data.

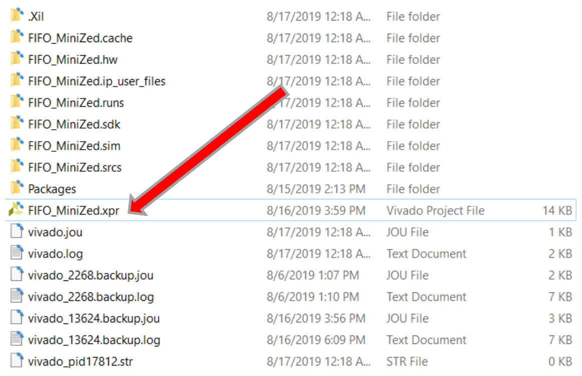
For the behavior of this FIFO, please see *appendix A* for **simulation results**.

Steps

PL_LUT and PL_BRAM versions of this FIFO are the easiest to build and run. Now, we will take a look at how we can build these two and download it to a **MiniZed** board.

Step 1: unarchive the downloaded project .zip file to desktop or any folder which *does not contain spaces* in its path. Remember, *Vivado* requires project directories to be short and not to contain any space. Thus, desktop is a safe place to unarchive this project.

Step 2: double-click on ***FIFO_MiniZed.xpr*** to open the project in *Vivado*.



.Xil	8/17/2019 12:18 A...	File folder	
FIFO_MiniZed.cache	8/17/2019 12:18 A...	File folder	
FIFO_MiniZed.hw	8/17/2019 12:18 A...	File folder	
FIFO_MiniZed.ip_user_files	8/17/2019 12:18 A...	File folder	
FIFO_MiniZed.runs	8/17/2019 12:18 A...	File folder	
FIFO_MiniZed.sdk	8/17/2019 12:18 A...	File folder	
FIFO_MiniZed.sim	8/17/2019 12:18 A...	File folder	
FIFO_MiniZed.srcs	8/17/2019 12:18 A...	File folder	
Packages	8/15/2019 2:13 PM	File folder	
FIFO_MiniZed.xpr	8/16/2019 3:59 PM	Vivado Project File	14 KB
vivado.jou	8/17/2019 12:18 A...	JOU File	1 KB
vivado.log	8/17/2019 12:18 A...	Text Document	2 KB
vivado_2268.backup.jou	8/6/2019 1:07 PM	JOU File	2 KB
vivado_2268.backup.log	8/6/2019 1:10 PM	Text Document	7 KB
vivado_13624.backup.jou	8/16/2019 3:56 PM	JOU File	3 KB
vivado_13624.backup.log	8/16/2019 6:09 PM	Text Document	7 KB
vivado_pid17812.str	8/17/2019 12:18 A...	STR File	0 KB

Figure 4: step 2, open the project in Vivado

Step 3: in *Project Manager*, under *Sources*, double-click on ***FIFO (FIFO.v)*** to open the top-level module.

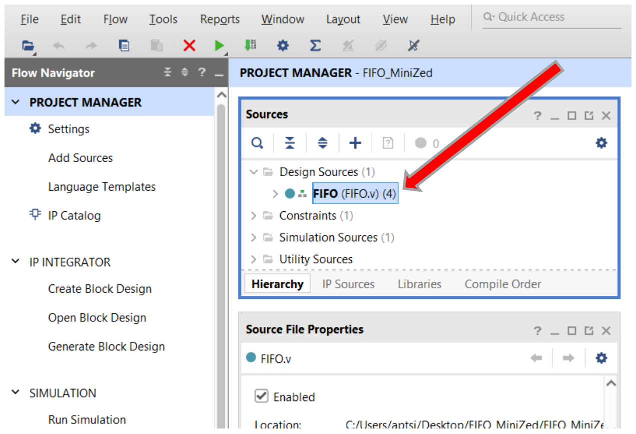


Figure 5: top-level module

Step 4: now, you are going to open the top-level module of the FIFO (Fig. 6) which contains the rest of the code under it in *Vivado*'s file hierarchy. If you take a look at *line 33*, it is already set to compile as *PL_LUT*. For the sake of demonstration, let us change to *PL_BRAM* and proceed. To compile the FIFO in *PL_BRAM* implementation mode, update the code on *line 33*, from

```
parameter MODE = `PL_LUT;          // SPECIFY default implementation mode here
```

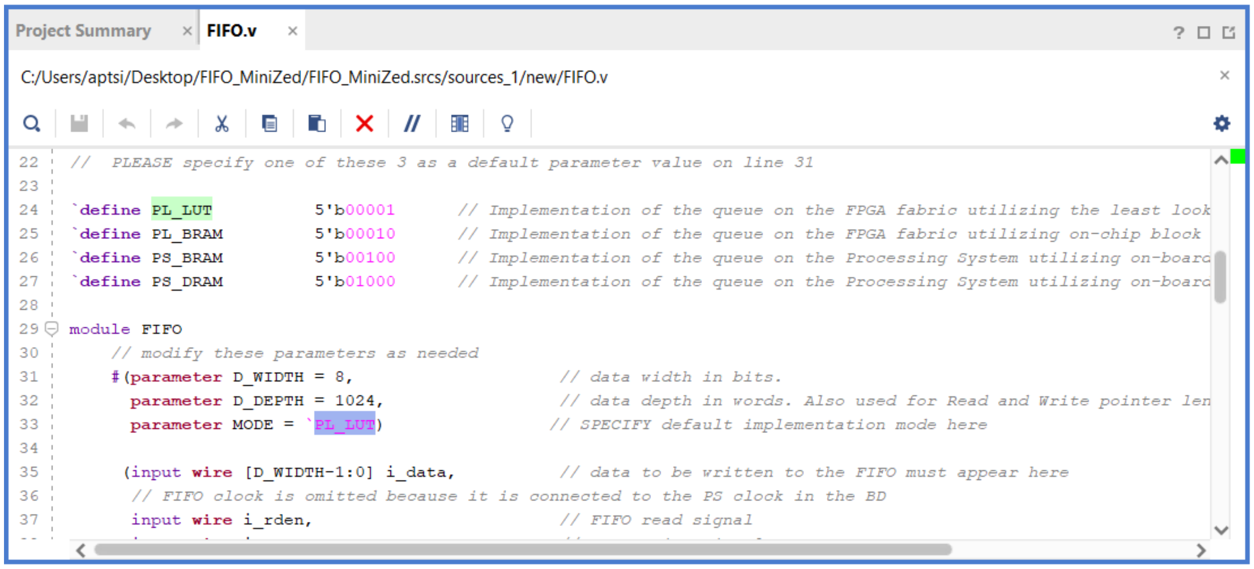
to

```
parameter MODE = `PL_BRAM;          // SPECIFY default implementation mode here
```

and press *Ctrl+s* to save the file. Note, how *Vivado* updated the file hierarchy under the *Sources* tab as you changed this line of code and saved it.

Note, the constants "*PL_LUT*" and "*PL_BRAM*" are defined as macros at the beginning of this file using *one-hot encoding*. Same goes for "*PS_BRAM*" and "*PS_DRAM*" constants, but more on this later.

NOTE that simply changing the implementation mode to "*PS_BRAM*" and "*PS_DRAM*" will NOT work. Now, we are only describing how to make the FIFO work on FPGA fabric (i.e. PL).



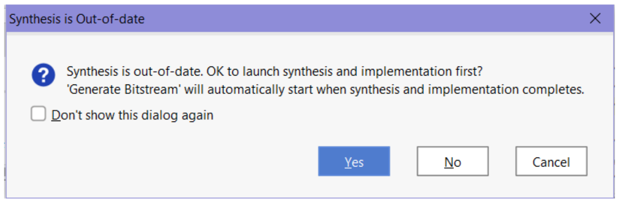
C:/Users/aptsi/Desktop/FIFO_MiniZed/FIFO_MiniZed.srcs/sources_1/new/FIFO.v

```
//  PLEASE specify one of these 3 as a default parameter value on line 31

`define PL_LUT          5'b00001      // Implementation of the queue on the FPGA fabric utilizing the least look
`define PL_BRAM         5'b00010      // Implementation of the queue on the FPGA fabric utilizing on-chip block
`define PS_BRAM         5'b00100      // Implementation of the queue on the Processing System utilizing on-boarc
`define PS_DRAM         5'b01000      // Implementation of the queue on the Processing System utilizing on-boarc

module FIFO
    // modify these parameters as needed
    #(parameter D_WIDTH = 8,                    // data width in bits.
      parameter D_DEPTH = 1024,                 // data depth in words. Also used for Read and Write pointer len
      parameter MODE = `PL_LUT)                // SPECIFY default implementation mode here

     (input wire [D_WIDTH-1:0] i_data,          // data to be written to the FIFO must appear here
      // FIFO clock is omitted because it is connected to the PS clock in the BD
      input wire i_rden,                        // FIFO read signal
```

Figure 6: updating implementation mode

Step 5: after step 4, the code is ready to be compiled as PL_BRAM. Now in the bottom left corner, click on *Generate Bitstream* to start synthesis, implementation and bitstream generation tasks.

Click “Yes” to tell *Vivado* to start *Synthesis* task all over again.

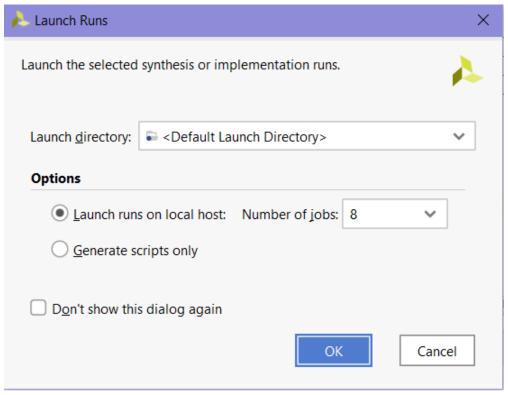

Synthesis is Out-of-date

Synthesis is out-of-date. OK to launch synthesis and implementation first?
'Generate Bitstream' will automatically start when synthesis and implementation completes.

☐ Don't show this dialog again

Figure 7: new synthesis task required after editing the code

Now, click “OK” to start the process.


Launch Runs

Launch the selected synthesis or implementation runs.

Launch directory: <Default Launch Directory>

Options

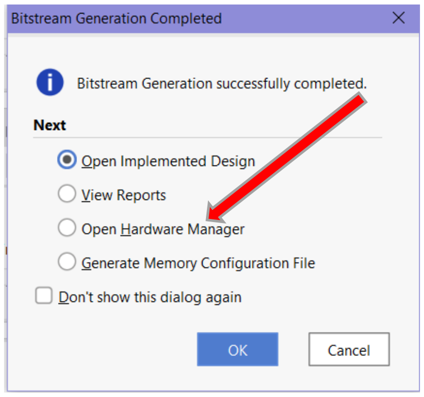
● Launch runs on local host: Number of jobs: 8

○ Generate scripts only

☐ Don't show this dialog again

This process will take a few minutes.

Step 6: Once bitstream generation finishes, you will get a window like in Fig. 8. Now, we need to download the code to ***MiniZed*** board. Connect your board to the computer using a Mini USB cable. Choose "*Open Hardware Manager*" as it is in Fig. 3, and press *OK*. Next, click on "Program Device". And, the FIFO which uses *BRAM* must be running on your board, at this point.


Bitstream Generation Completed

Bitstream Generation successfully completed.

Next

[✓ Open Implemented Design]
View Reports
Open Hardware Manager
Generate Memory Configuration File

Don't show this dialog again

Figure 8: bitstream generation finished successfully

PS_BRAM

Principle of Operation

PS_BRAM version of this FIFO uses Zynq PS where from a C code, AXI transactions are initiated to GP0 and GP1 ports. We use GP0 port for writing to the FIFO and GP1 for reading from it. This way, the maximum achievable bandwidth is massive enough to accommodate most applications' needs. These ports are then connected to two BRAM controllers through AXI interconnect which separately control BRAM itself which is configured as dual-port memory. The low and high addresses of these ports are given in *Address Editor* in Xilinx block design tool. They are configured as per *Zynq* manual and, thus, are guaranteed to work.

The given C code for PS_BRAM also contains testing part which is located in the "main()" block.

Note that C code was adapted from the Verilog version, so as many features of the FIFO are preserved as it was possible. Once again, the features of the FIFO are demonstrated in *Appendix A: FIFO simulation results,* at the end of this report.

You may view the block design for PS_BRAM in *Vivado*, under *Sources*. See Fig. 9.

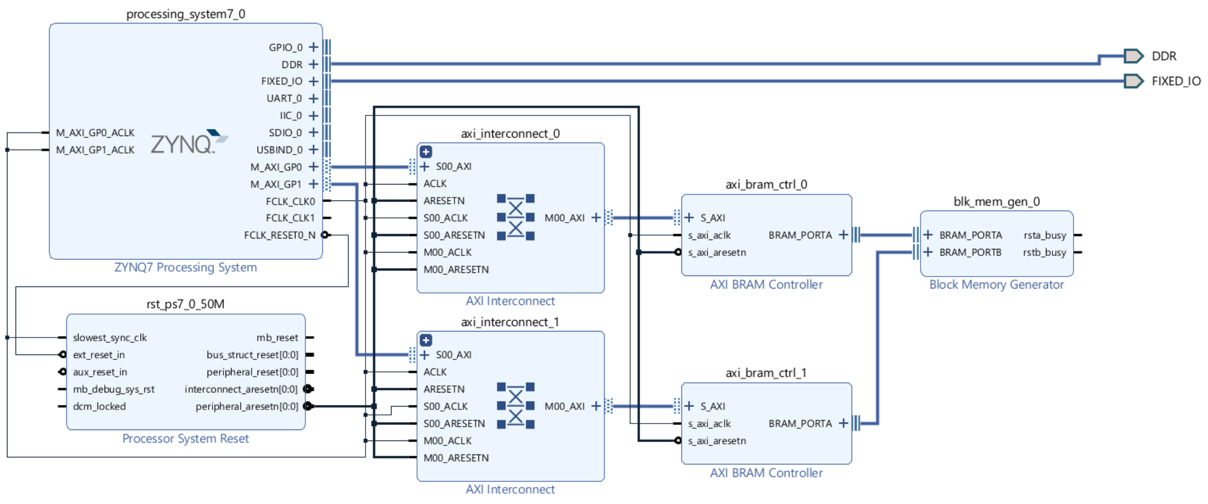


Figure 9: PS_BRAM BD

Steps

I will skip detailed steps as it was already shown in the previous section.

Step 1: change the implementation mode to PS_BRAM in *FIFO.v* file.

Step 2: generate bitstream and download it to the board.

Step 3: export the generated hardware to the project directory as shown in Fig. 10.

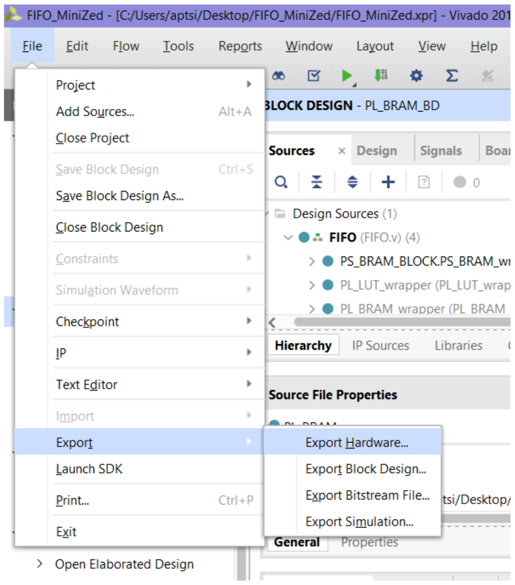


Figure 10: export hardware

Check "*include bitstream*" and click OK. Click on YES or OK on every prompt.

Step 4: Click on "*Launch SDK*" in *File > Launch SDK*. Again, click on YES or OK on every prompt. At this point, Xilinx SDK will open up, where we will create a project, add C code, resolve dependencies and run it on the ARM core.

Step 5: Create an application project as shown in Fig. 11.

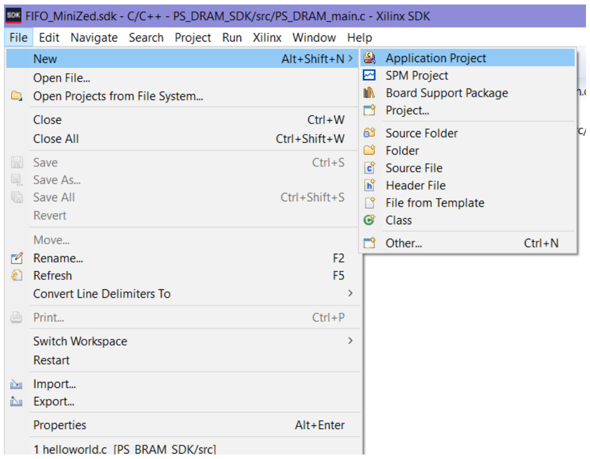
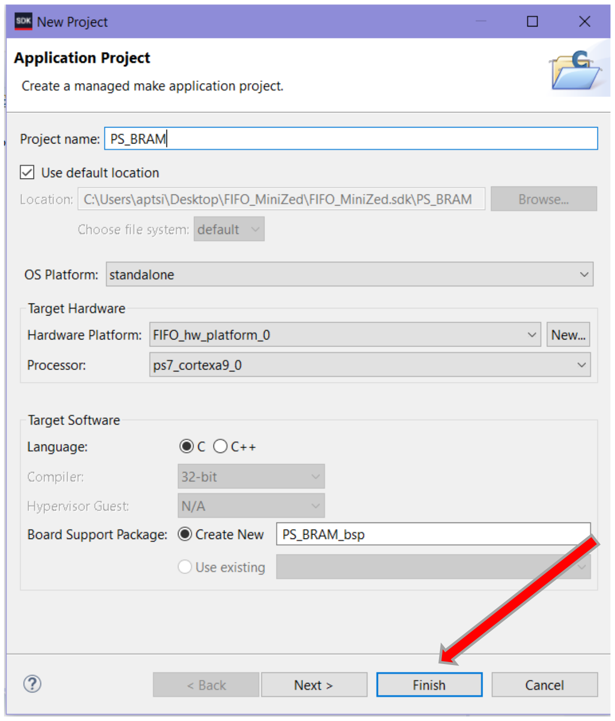


Figure 11: creating application project

Name the project "PS_BRAM" as shown in Fig. 12, and press Finish.


New Project

Application Project

Create a managed make application project.

Project name: PS_BRAM

[✓] Use default location

Location: C:\Users\aptsi\Desktop\FIFO_MiniZed\FIFO_MiniZed.sdk\PS_BRAM

Choose file system: default

OS Platform: standalone

Target Hardware

Hardware Platform: FIFO_hw_platform_0

Processor: ps7_cortexa9_0

Target Software

Language: [✓ C] C++

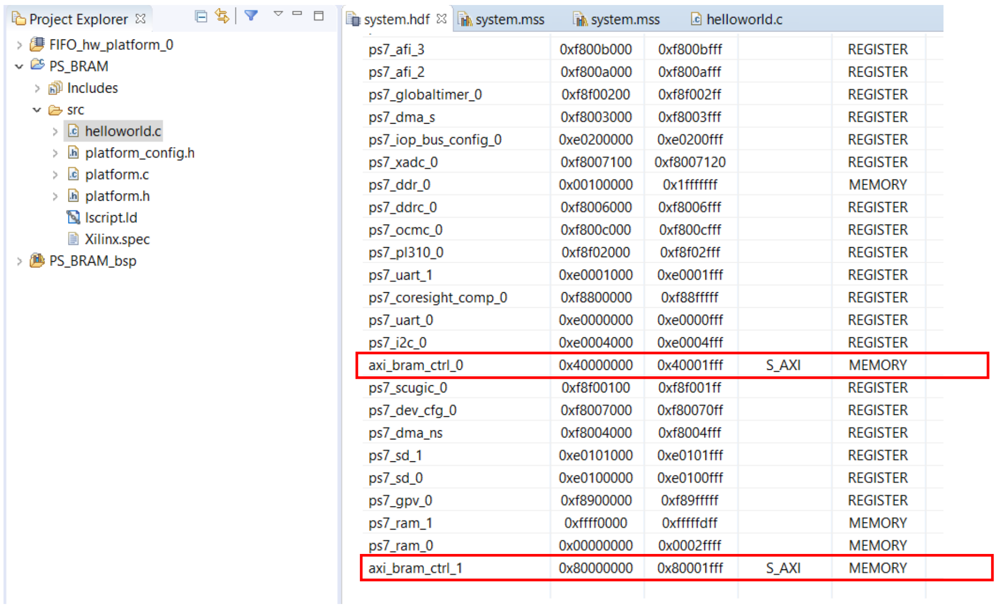
Compiler: 32-bit

Hypervisor Guest: N/A

Board Support Package: [✓ Create New] PS_BRAM_bsp / Use existing

Figure 12: PS_BRAM application project

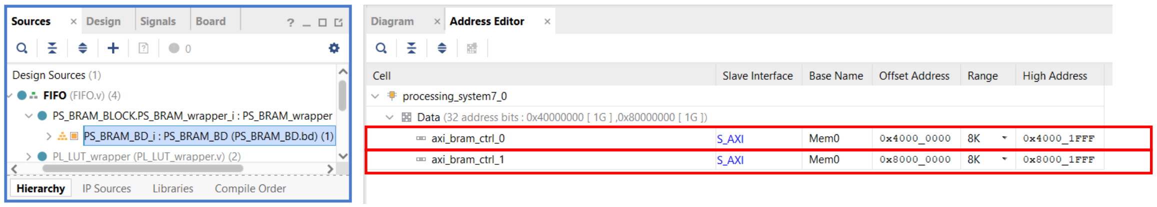
Step 6: now, if you look under *"system.hdf"* file, you will see BRAM memory read/write ports that we set up in the block design in *Xilinx Vivado IDE*. See Fig. 13. You may convince yourself that the low and high addresses of these ports match the ones we set up in *Vivado* by opening the PS_BRAM block diagram in *Vivado* and checking address editor. You may check Fig. 14, which shows address editor information.



```
FIFO_hw_platform_0
PS_BRAM
  Includes
  src
    helloworld.c
    platform_config.h
    platform.c
    platform.h
    lscript.ld
    Xilinx.spec
PS_BRAM_bsp
```

ps7_afi_3	0xf800b000	0xf800bfff		REGISTER
ps7_afi_2	0xf800a000	0xf800afff		REGISTER
ps7_globaltimer_0	0xf8f00200	0xf8f002ff		REGISTER
ps7_dma_s	0xf8003000	0xf8003fff		REGISTER
ps7_iop_bus_config_0	0xe0200000	0xe0200fff		REGISTER
ps7_xadc_0	0xf8007100	0xf8007120		REGISTER
ps7_ddr_0	0x00100000	0x1fffffff		MEMORY
ps7_ddrc_0	0xf8006000	0xf8006fff		REGISTER
ps7_ocmc_0	0xf800c000	0xf800cfff		REGISTER
ps7_pl310_0	0xf8f02000	0xf8f02fff		REGISTER
ps7_uart_1	0xe0001000	0xe0001fff		REGISTER
ps7_coresight_comp_0	0xf8800000	0xf88fffff		REGISTER
ps7_uart_0	0xe0000000	0xe0000fff		REGISTER
ps7_i2c_0	0xe0004000	0xe0004fff		REGISTER
axi_bram_ctrl_0	0x40000000	0x40001fff	S_AXI	MEMORY
ps7_scugic_0	0xf8f00100	0xf8f001ff		REGISTER
ps7_dev_cfg_0	0xf8007000	0xf80070ff		REGISTER
ps7_dma_ns	0xf8004000	0xf8004fff		REGISTER
ps7_sd_1	0xe0101000	0xe0101fff		REGISTER
ps7_sd_0	0xe0100000	0xe0100fff		REGISTER
ps7_gpv_0	0xf8900000	0xf89fffff		REGISTER
ps7_ram_1	0xffff0000	0xfffffdff		MEMORY
ps7_ram_0	0x00000000	0x0002ffff		MEMORY
axi_bram_ctrl_1	0x80000000	0x80001fff	S_AXI	MEMORY

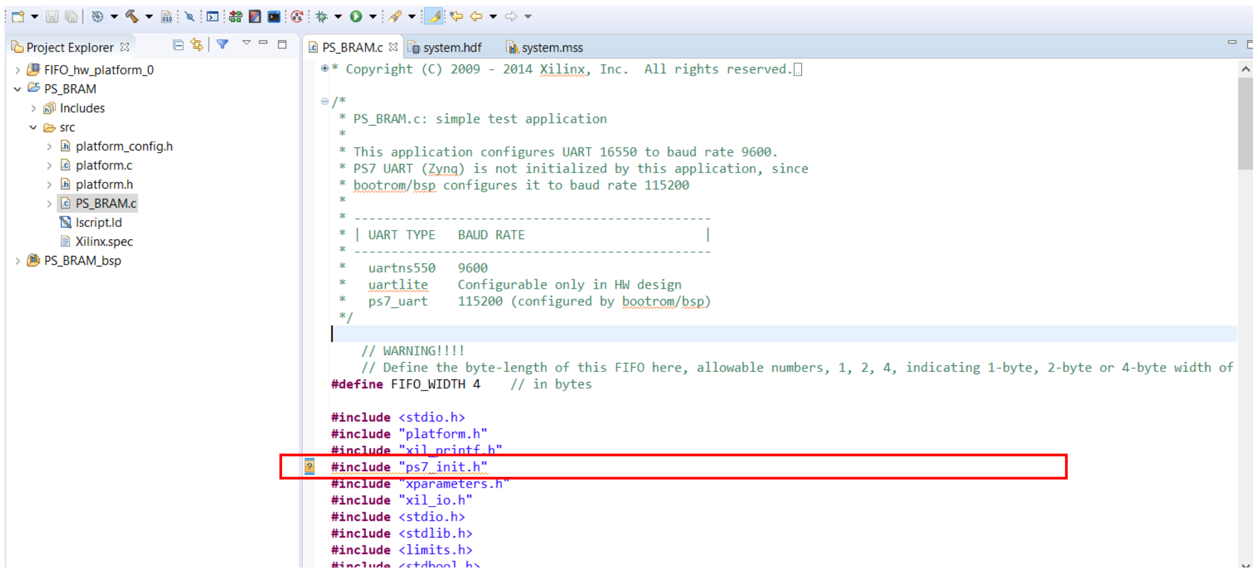
Figure 13: BRAM read/write ports


Sources
Design
Signals
Board
Design Sources (1)
FIFO (FIFO.v) (4)
PS_BRAM_BLOCK.PS_BRAM_wrapper_i : PS_BRAM_wrapper
PS_BRAM_BD_i : PS_BRAM_BD (PS_BRAM_BD.bd) (1)
PL_LUT_wrapper (PL_LUT_wrapper.v) (2)
Hierarchy
IP Sources
Libraries
Compile Order
Diagram
Address Editor

Cell	Slave Interface	Base Name	Offset Address	Range	High Address
processing_system7_0					
Data (32 address bits : 0x40000000 [1G] ,0x80000000 [1G])					
axi_bram_ctrl_0	S_AXI	Mem0	0x4000_0000	8K	0x4000_1FFF
axi_bram_ctrl_1	S_AXI	Mem0	0x8000_0000	8K	0x8000_1FFF

Figure 14: ports match

With this at hand, you may copy the contents of "*PS_BRAM.c*" file, that you downloaded, to "*PS_BRAM > src > helloworld.c*" file under Project Explorer of SDK. And, lastly, change the file name from "*helloworld.c*" to "*PS_BRAM.c*". See Fig. 14 for final results.



```
* Copyright (C) 2009 - 2014 Xilinx, Inc.  All rights reserved.

/*
 * PS_BRAM.c: simple test application
 *
 * This application configures UART 16550 to baud rate 9600.
 * PS7 UART (Zynq) is not initialized by this application, since
 * bootrom/bsp configures it to baud rate 115200
 *
 * ------------------------------------------------
 * | UART TYPE   BAUD RATE                        |
 * ------------------------------------------------
 *   uartns550   9600
 *   uartlite    Configurable only in HW design
 *   ps7_uart    115200 (configured by bootrom/bsp)
 */

    // WARNING!!!!
    // Define the byte-length of this FIFO here, allowable numbers, 1, 2, 4, indicating 1-byte, 2-byte or 4-byte width of
#define FIFO_WIDTH 4    // in bytes

#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "ps7_init.h"
#include "xparameters.h"
#include "xil_io.h"
#include <stdio.h>
#include <stdlib.h>
#include <limits.h>
#include <stdbool.h>
```

Figure 15: code ready to download to the ARM core

As you can see, everything is fine in the SDK, except for *“ps7_init.h”*.

Step 7: resolve dependencies and download the code to the board. To resolve the *"ps7_init.h"* dependency, drag it from *“FIFO_hw_platform_0”* under *Project Explorer* to “*PS_BRAM* > *src”*. This will resolve the dependency.

Step 8: now the PS_BRAM version is ready to be launched on the ARM core. For this, simply press the green play-like button which will launch the program on the board. Optionally, you may set up a Putty serial session before downloading the code to the board. This way, you will see the FIFO in action as items are read from it in the order they were written to it.

PS_DRAM

Principle of Operation

PS_DRAM is almost identical to PS_BRAM version, except, there is no complicated block design. In fact, the *Zynq* PS configuration for PS_DRAM is nothing but what we get by running design automation after placing Zynq Processing System in the workplace. This is enough because DDR memory get connected to the PS after running design automation.

After that, Xilinx SDK gives us a low and high address of the DRAM memory. In fact, we can write to the memory and read from it just like we would do in regular C programming, using pointers.

Steps

The steps to build and run PS_DRAM version of this FIFO is almost the same as for PS_BRAM. Thus, we will skip most of it. Follow steps 1-5 from PS_BRAM, but, set the implementation mode to PS_DRAM in FIFO.v top-level wrapper. During project creating in SDK, name it "PS_DRAM".

After copying the contents of "PS_DRAM.c" from the downloaded file to "helloworld.c" in your newly created project in SDK, you may change its name to "PS_DRAM.c" for readability. This time, you should not have any unresolved dependencies, which is good. If you have, it should be easy to resolve them because all you would need to do is to find those files in the project tree and copy them to the *src* folder of your PS_DRAM project.

Finally, you may start a Putty session, and click the green play-like button at the top to download the code to the PS. In your serial connection, you would see messages as items are read from the FIFO in the same order as they were written to it.

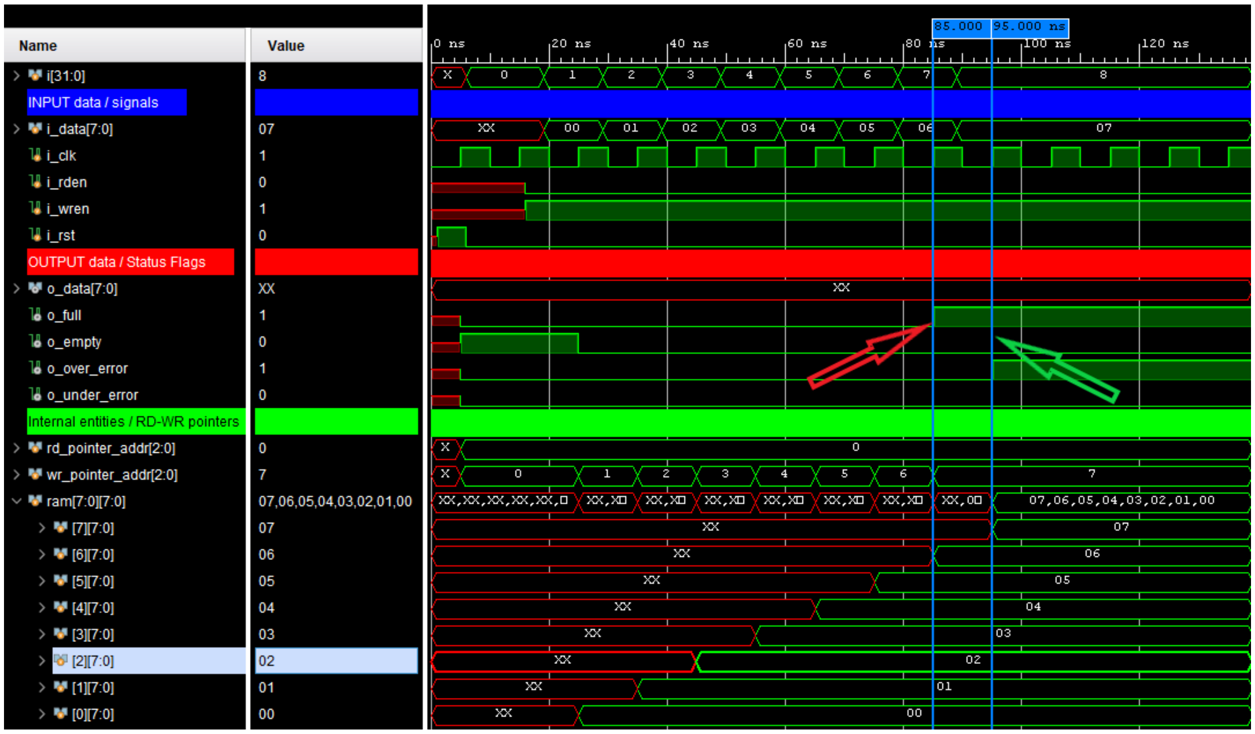
Appendix A: FIFO simulation results

Step 1) Simply Writing to the FIFO

Code snippet from the testbench:

```
// generate clock
  initial i_clk = 0;
  always #5 i_clk = ~i_clk;   // period of 10

  initial begin
    // resetting the FIFO before writing anything to it
    #1 i_rst = 1; #5 i_rst = 0;

    for (i = 0; i < 8; i = i + 1)
    begin
      @(posedge i_clk);
      #1  i_wren = 1; i_rden = 0;
      #3  i_data = i;
    end
  #50 $stop;
  End
```

Waveform from Xilinx Vivado simulator:



Name	Value
i[31:0]	8
INPUT data / signals	
i_data[7:0]	07
i_clk	1
i_rden	0
i_wren	1
i_rst	0
OUTPUT data / Status Flags	
o_data[7:0]	XX
o_full	1
o_empty	0
o_over_error	1
o_under_error	0
Internal entities / RD-WR pointers	
rd_pointer_addr[2:0]	0
wr_pointer_addr[2:0]	7
ram[7:0][7:0]	07,06,05,04,03,02,01,00
[7][7:0]	07
[6][7:0]	06
[5][7:0]	05
[4][7:0]	04
[3][7:0]	03
[2][7:0]	02
[1][7:0]	01
[0][7:0]	00

In an attempt to write 8 items to the FIFO, we got "o_full" flag asserted as the 7th item was written (indicated by the red arrow at ***85ns***). At this point, any further write the FIFO will cause the “o_over_error” to be set which indicates to the upper level module that an overflow occurred in the FIFO memory.

The reason the FIFO filled up at the 7th item originates from how the “o_full” flag is computed in the HDL code, namely: `assign o_full = (next_wr_addr == rd_pointer_addr);`

When the write pointer catches up with the read pointer, and there is only one unwritten cell of memory in the FIFO, and that’s when it can definitely be deduced that the FIFO is full. This way of computing the “o_full” signal comes at the cost of leaving one last memory cell unused. This also implies that the FIFO will be able to hold “the-memory-depth-minus-one” number of items at maximum.

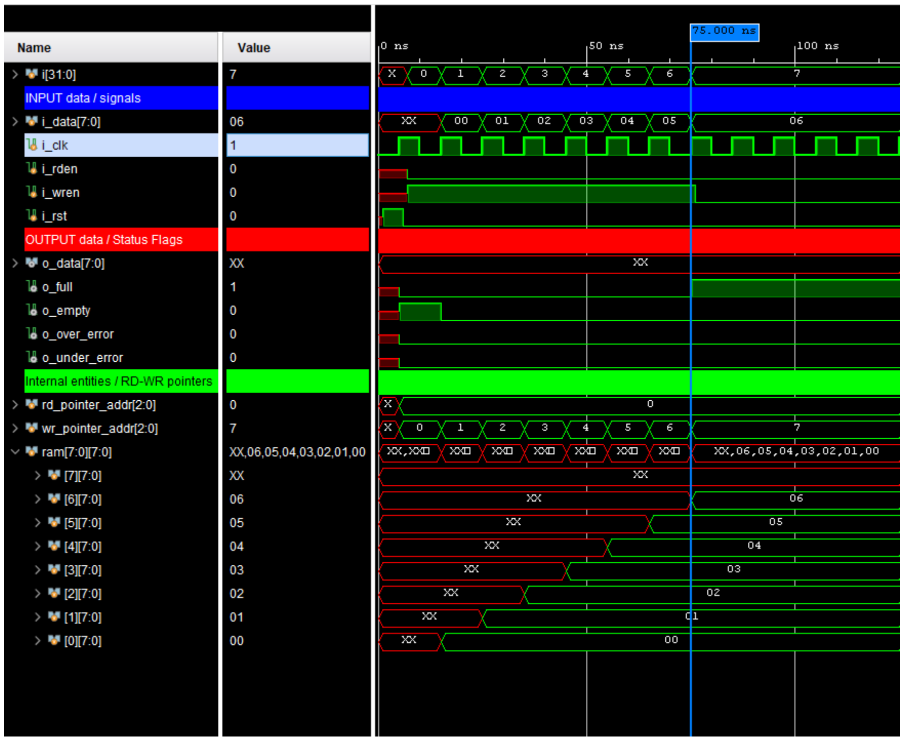
Additionally, as the FIFO was full and a write was still made, the overflow flag, “o_over_error”, was set (indicated by the green arrow at ***95ns***)! Note, that even though the error was bound to occur, the input data was still written to the FIFO, i.e., ‘07’ was written to ram[07]. At this point, the FIFO is in an error state and it must be reset via the “i_rst” signal to come back to a working state.

Step 2) Writing to the FIFO until it fills up

Code snippet from the testbench:

```
// generate clock
  initial i_clk = 0;
  always #5 i_clk = ~i_clk;   // period of 10

  initial begin
    #1 i_rst = 1; #5 i_rst = 0;
    #1  i_wren = 1; i_rden = 0;

    for (i = 0; i < 7; i = i + 1)
    begin
      @(posedge i_clk);
      i_data = i;
    end

    // disable the write signal
    #1  i_wren = 0; i_rden = 0;
```

Waveform from Xilinx Vivado simulator:



Note, at ***75ns***, as the seventh item is written to the FIFO, the "o_full" signal goes high. At this moment, the 8th memory cell, ram[7], is empty and the reason is the same as above, it is just my way of computing the "o_full" flag. As the 7th item was written, I set the "i_wren" signal to low, to prevent FIFO from entering an error state.

Now that the FIFO is full, I would like to test an interesting feature of this design, namely, the ability to perform reading and writing simultaneously in the same clock cycles ***while it is full***.

Please observe the following snippet from the HDL which describes how the write pointer is handled:

```
// handle the write pointer
always @(posedge i_clk)
if (i_rst)
    begin
        wr_pointer_addr <= 0;
        o_over_error  <= 0;
    end else if (i_wren)
    begin
        // Update the FIFO write address pointer any time a write is made to
        // the FIFO and it's not FULL.
        //
        // OR any time a write is made to the FIFO at the same time a
        // read is made from the FIFO.
        if ((!o_full)||(i_rden))
```

```
15.             wr_pointer_addr <= (wr_pointer_addr + 1'b1);
16.         else
17.             o_over_error <= 1'b1;
18.     end
```

Namely, the comment (line 9 to line 13) and line 14. Even though the FIFO can be full, if the "i_rden" signal is high, i.e., a read is taking place in the same clock cycle as a write is requested, the FIFO can *legally* process a write without entering an error state.

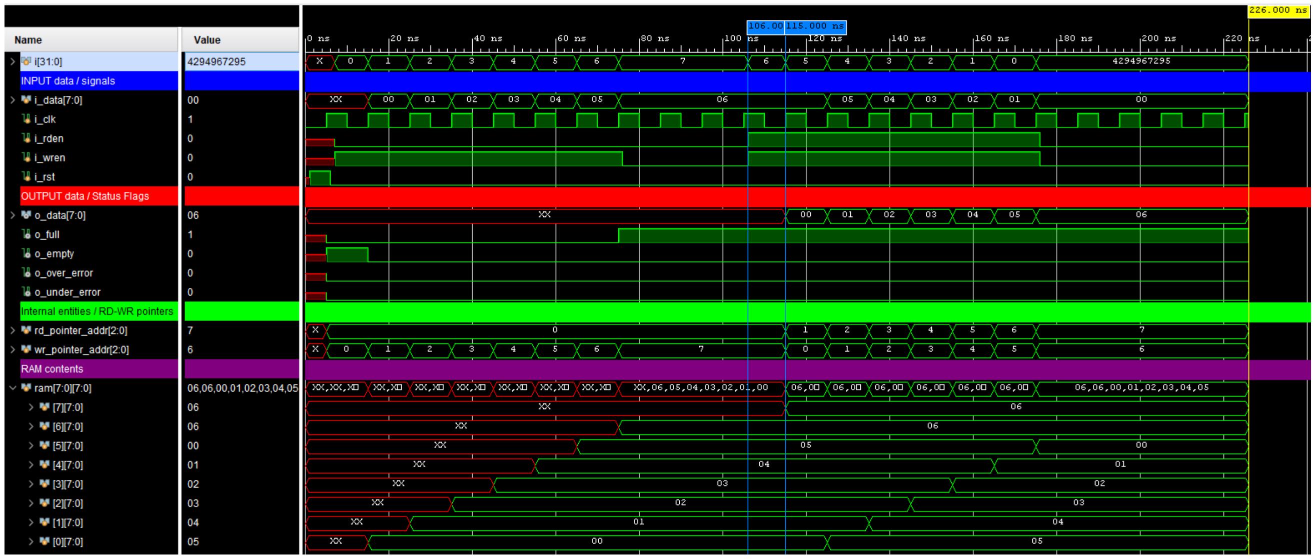
Let us now test it in the following step. The next step will also demonstrate how the read and write pointers wrap around to form a "circular FIFO".

Step 3) Writing to the FIFO while it is full and a read is takin place in the same clock cycle

Code snippet from the testbench:

```
// generate clock
    initial i_clk = 0;
    always #5 i_clk = ~i_clk;   // period of 10

    initial begin
        #1 i_rst = 1; #5 i_rst = 0;
        #1  i_wren = 1; i_rden = 0;

        for (i = 0; i < 7; i = i + 1)
        begin
            @(posedge i_clk);
            i_data = i;
        end

        // disable the write signal
        #1  i_wren = 0; i_rden = 0;

        // a gap on the waveform
        repeat (3) @(posedge i_clk);

        // enable both, write signal and read signal
        #1  i_wren = 1; i_rden = 1;

        //
        for (i = 6; i >= 0; i = i - 1)
        begin
            @(posedge i_clk);
            i_data = i;
        end

        // disable both, write signal and read signal
        #1  i_wren = 0; i_rden = 0;
```

Waveform from Xilinx Vivado simulator:



This waveform is the same as the last one until ***106ns***. Then, as the FIFO is full, I enable both “i_wren” and “i_wren” to perform read and write from and to the FIFO in the same clock cycles. In this manner, observe that the FIFO stays full (“o_full” flag is high), BUT, overflow does not occur.

A similar procedure can be followed to produce similar results when the FIFO is empty, i.e., read and write in the same clock cycles while there is only one item written to the FIFO.