FPGA **DESIGN**

Interfacing Over AXI

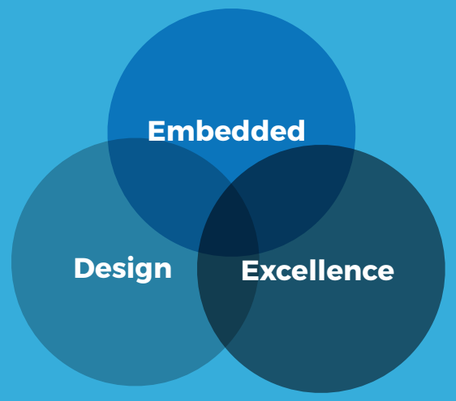
Using A Simple Address Data Bus

Roland Bodlovic

BYTESNAP
embedded design excellence

Contents

ByteSnap Design is an embedded electronics consultancy



Introduction

This ebook is a step-by-step guide to a method of using AXI interconnect whilst avoiding some of the overhead in design and simulation.

FPGA vendors offer powerful tools that help with this process; however this means another layer is added in to managing IP. Time has to be spent supervising and generating IP outputs before they can be incorporated into the mainline project. These tools build on the AXI (ARM eXtensible Interface) interconnect, allowing transfer and integration of re-usable approved (tested) IP building blocks. It is part of the AMBA open standards from ARM.

There are some excellent articles* available online that take the designer step by step through generating AXI IP for slave and master plus using the licensed BFM for test.

FPGA Design Challenges with AXI

When looking at the overall design flow with AXI, there are some challenges - such as how to simulate the design. This isn't trivial, especially if you haven't done it before. This walkthrough guide is here to help you.

BFMs (Bus Functional Models) for AXI are available but usually require a paid license. AXI is very flexible and, as such, writing a test suite that's verified will take time. If the custom IP needs to support burst access, the testing complexity increases.

Is there something simpler that can help interface to AXI without having to handle the AXI bus directly and remove the worry about potential issues that are not related to the function of the IP? Can we reduce the overhead - especially for smaller IP? Can it be included in the mainline project, but also be managed as a piece of separate IP, if the designer chooses?

The BRAM Interface

Well - let's answer those queries. One solution is to use the BRAM interface and roll your own address decoder.

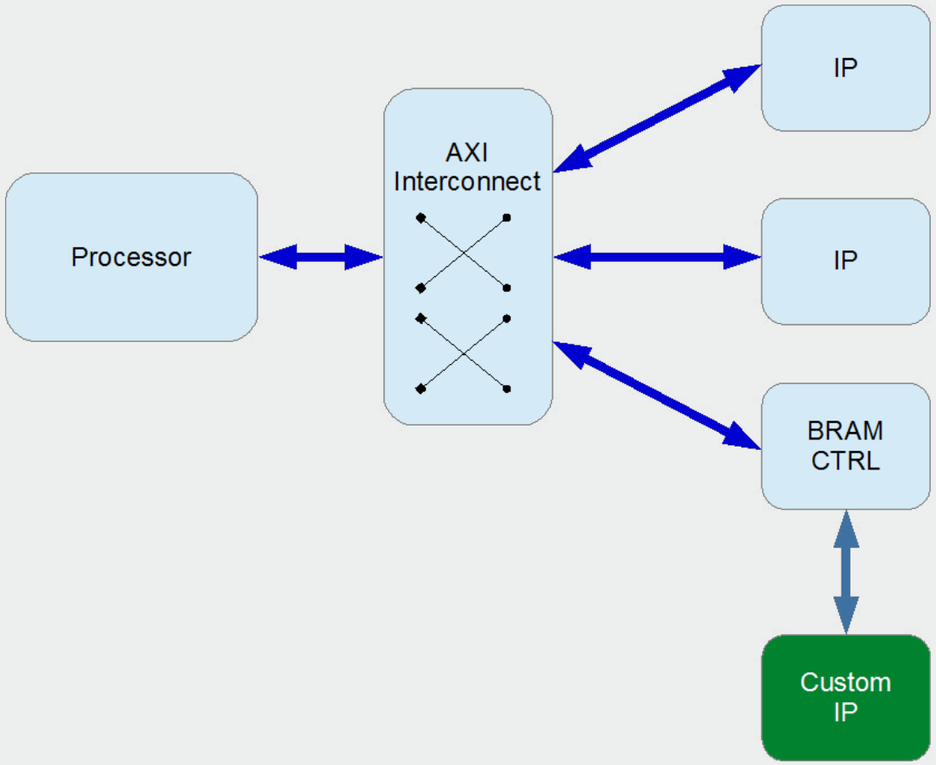


Figure 1: System Diagram

Advantages

- BRAM interface supports AXI burst access, which means high throughput is possible. DMA engines can be used with little effort
- Test bench simplified by a well-defined simple address and data bus model
- IP Simulation can be performed in the mainline project using different source directories for each IP - which allows the IP source to be extracted and shared later or even packaged with an IP-XACT layer
- Choice of address, registers and BRAM areas is placed into the hands of the designer and is easily defined with a case statement

Disadvantages

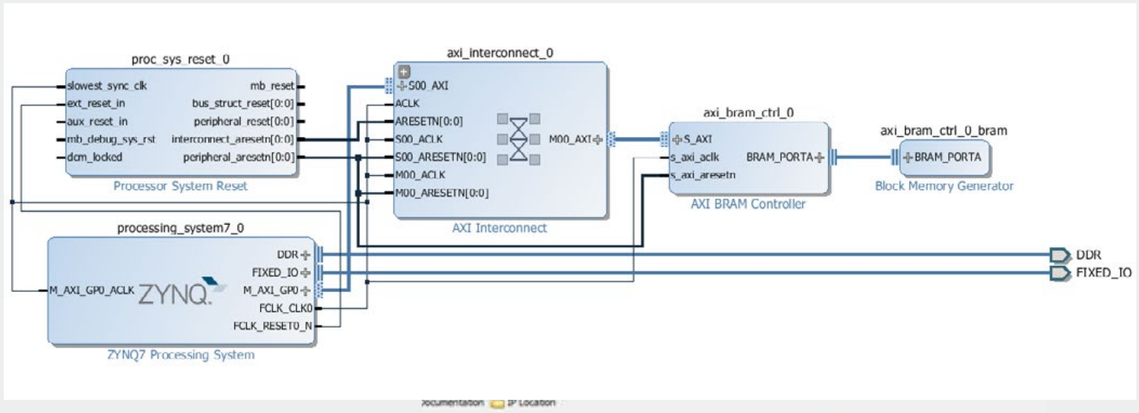
- May take up slightly more resources as the BRAM controller will have to be instantiated in the design
- Read/write access must obey BRAM access; both an advantage (it's a simple read/write access) and disadvantage (no method to hold the bus off if data takes a long time to come back)
- It is possible to get around data not ready issues by using status register bits or interrupts which can indicate when a slow read is ready
- AXI re-ordering is not supported. The current transaction must complete before a new one can occur

How to interface to AXI using the BRAM controller

Generate block diagram

OK, now the walkthrough. The first thing we must do is create the block diagram for our project - assuming you are starting from scratch.
The Vivado tools support this quite well and there are many tutorials available online to help.

The block diagram must have a BRAM controller. Probably the best way to do this is to just add the IP and let the tools automatically connect and configure it. For this project, we are only going to use 1 BRAM port.
Finally, make sure the address is set up. Once we have our system setup, it's usually a good idea to try it before moving on to the next step.



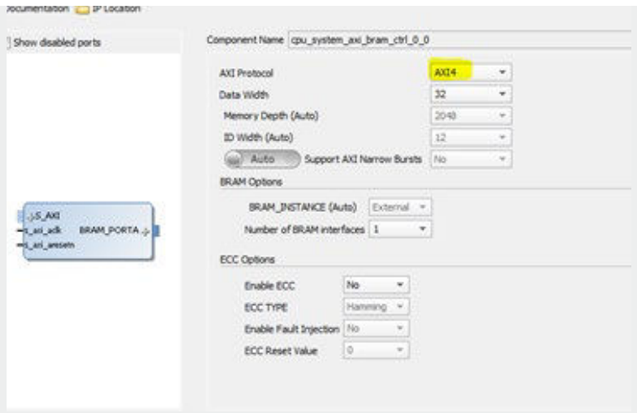


Figure 1a:
Block Diagram

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

Figure 2: Address list

In this example, the BRAM controller is configured as a full fat AXI interconnect so that it supports DMA burst access. More on this later.

Synthesising this design and running it in the Zynq is relatively straightforward; generate a top level wrapper and then synthesise.

Ensure the constraints are correct for your target PCB. Once you have generated the bitstream, upload it.

A quick test of the BRAM memory is advisable to ensure everything is connected correctly before we start modifying the design.

Vivado offers a memory test application to help. Launch the SDK and create the test app.

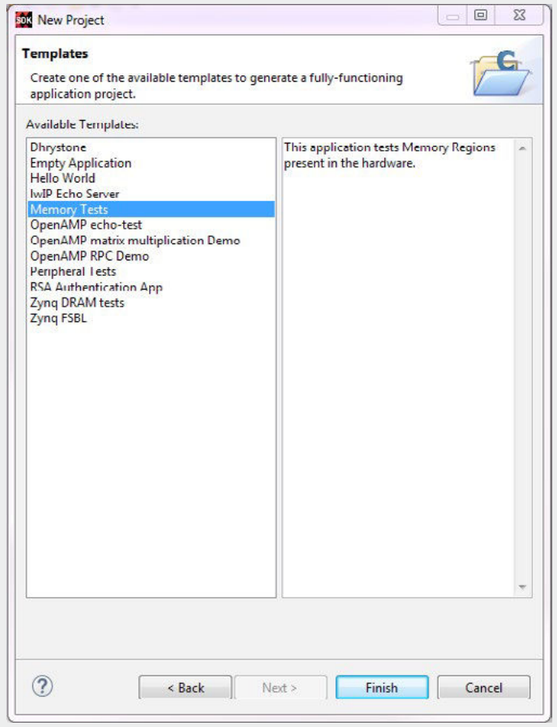


Figure 3: SDK template projects

Figure 4: Memory test results

```
--Starting Memory Test Application--
NOTE: This application runs with D-Cache disabled.As a result, cacheline requests will not be generated
Testing memory region: axi_bram_ctrl_0
    Memory Controller: axi_bram_ctrl
          Base Address: 0x40000000
                  Size: 0x00002000 bytes
           32-bit test: PASSED!
           16-bit test: PASSED!
            8-bit test: PASSED!
Testing memory region: ps7_ddr_0
    Memory Controller: ps7_ddr
          Base Address: 0x00100000
                  Size: 0x3ff00000 bytes
           32-bit test: PASSED!
           16-bit test: PASSED!
            8-bit test: PASSED!
Testing memory region: ps7_ram_1
    Memory Controller: ps7_ram
          Base Address: 0xffff0000
                  Size: 0x0000fe00 bytes
           32-bit test: PASSED!
           16-bit test: PASSED!
            8-bit test: PASSED!
--Memory Test Application Complete--
```

Notice that axi_bram_ctrl has passed the test.

Create new TOP LEVEL wrapper

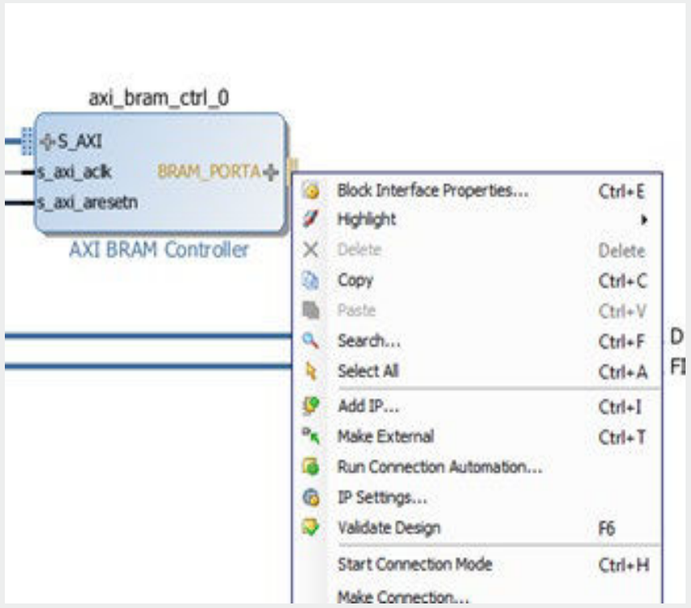
The next few steps will create a new top level interface for use with our custom IP.

To do this we will need a new top level source file that is in our source directory and instantiate the CPU system. We then have full access to the BRAM interface signals.

Having a HDL top level source file allows user edits without it being overwritten by Vivado tools when the block diagram is updated.

Add a new top level VHDL file to the project and open the cpu_system block diagram. Remove the BRAM and make the BRAM port signals external. This exposes the BRAM interface to the top level.

Figure 5: Make BRAM interface external



Next, we need to instantiate the CPU system.

Use 'View Instantiation Template' function on the CPU_system block diagram and copy the contents into the new top level VHDL file.

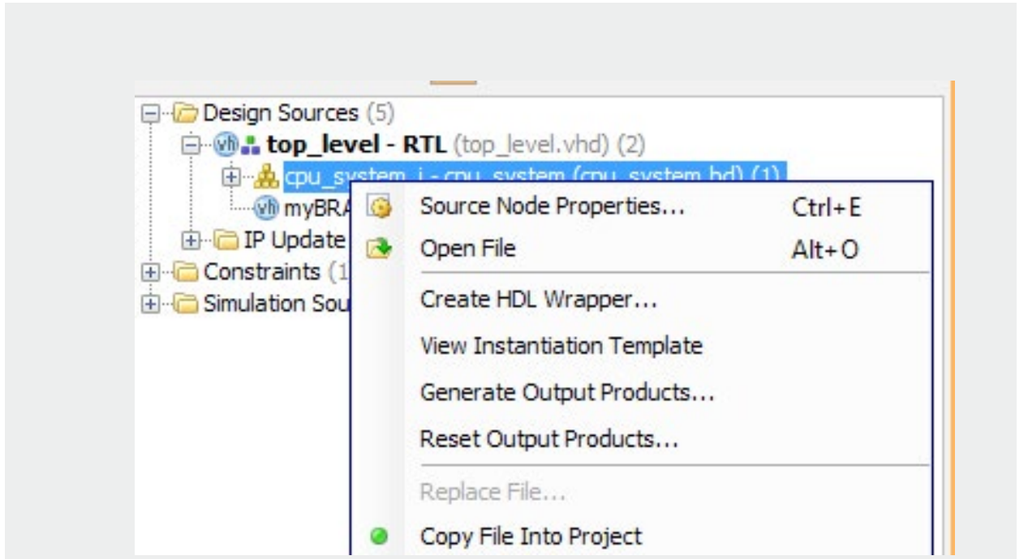


Figure 6: Make BRAM interface external

Figure 6a: CPU_system instantiation

```
cpu_system_i : component cpu_system
    port map(
        BRAM_PORTA_addr(12 downto 0) => BRAM_PORTA_addr(12 downto 0),
        BRAM_PORTA_clk               => BRAM_PORTA_clk,
        BRAM_PORTA_din(31 downto 0)  => BRAM_PORTA_din(31 downto 0),
        BRAM_PORTA_dout(31 downto 0) => BRAM_PORTA_dout(31 downto 0),
        BRAM_PORTA_en                => BRAM_PORTA_en,
        BRAM_PORTA_rst               => BRAM_PORTA_rst,
        BRAM_PORTA_we(3 downto 0)    => BRAM_PORTA_we(3 downto 0),
        DDR_addr(14 downto 0)        => DDR_addr(14 downto 0),
        DDR_ba(2 downto 0)           => DDR_ba(2 downto 0),
        DDR_cas_n                    => DDR_cas_n,
        DDR_ck_n                     => DDR_ck_n,
        DDR_ck_p                     => DDR_ck_p,
        DDR_cke                      => DDR_cke,
        DDR_cs_n                     => DDR_cs_n,
        DDR_dm(3 downto 0)           => DDR_dm(3 downto 0),
        DDR_dq(31 downto 0)          => DDR_dq(31 downto 0),
        DDR_dqs_n(3 downto 0)        => DDR_dqs_n(3 downto 0),
        DDR_dqs_p(3 downto 0)        => DDR_dqs_p(3 downto 0),
        DDR_odt                      => DDR_odt,
        DDR_ras_n                    => DDR_ras_n,
        DDR_reset_n                  => DDR_reset_n,
        DDR_we_n                     => DDR_we_n,
        FIXED_IO_ddr_vrn             => FIXED_IO_ddr_vrn,
        FIXED_IO_ddr_vrp             => FIXED_IO_ddr_vrp,
        FIXED_IO_mio(53 downto 0)    => FIXED_IO_mio(53 downto 0),
        FIXED_IO_ps_clk              => FIXED_IO_ps_clk,
        FIXED_IO_ps_porb             => FIXED_IO_ps_porb,
        FIXED_IO_ps_srstb            => FIXED_IO_ps_srstb
    );
```

The top_level.vhd ports can be copied from the CPU system block diagram template minus the BRAM interface.

The first thing to try is adding a BRAM in VHDL to this interface and check the behaviour is the same as before.

Check out the BRAM HDL inference template from the Xilinx documentation UG901 chapter 3. It can be easily modified to support the four byte write strobes.

Once created, instance the new BRAM in the top level and connect it to the system instance.

```
myBRAM_i : entity work.myBRAM
    port map(
        clk  => BRAM_PORTA_clk,
        we   => BRAM_PORTA_we,
        en   => BRAM_PORTA_en,
        addr => BRAM_PORTA_addr,
        di   => BRAM_PORTA_din,
        do   => BRAM_PORTA_dout
    );
```

Figure 7: Block RAM instantiation

The full example is provided in **top_level.vhd** and **myBRAM.vhd**.

We can see the connectivity by exploring the elaborated design in Vivado.

Notice the **cpu_system_i** is connected to the **myBRAM_i**.
The remaining signals are ports going off chip.

This is identical to before; we removed the BRAM from the block diagram and pushed it on level above in the hierarchy.

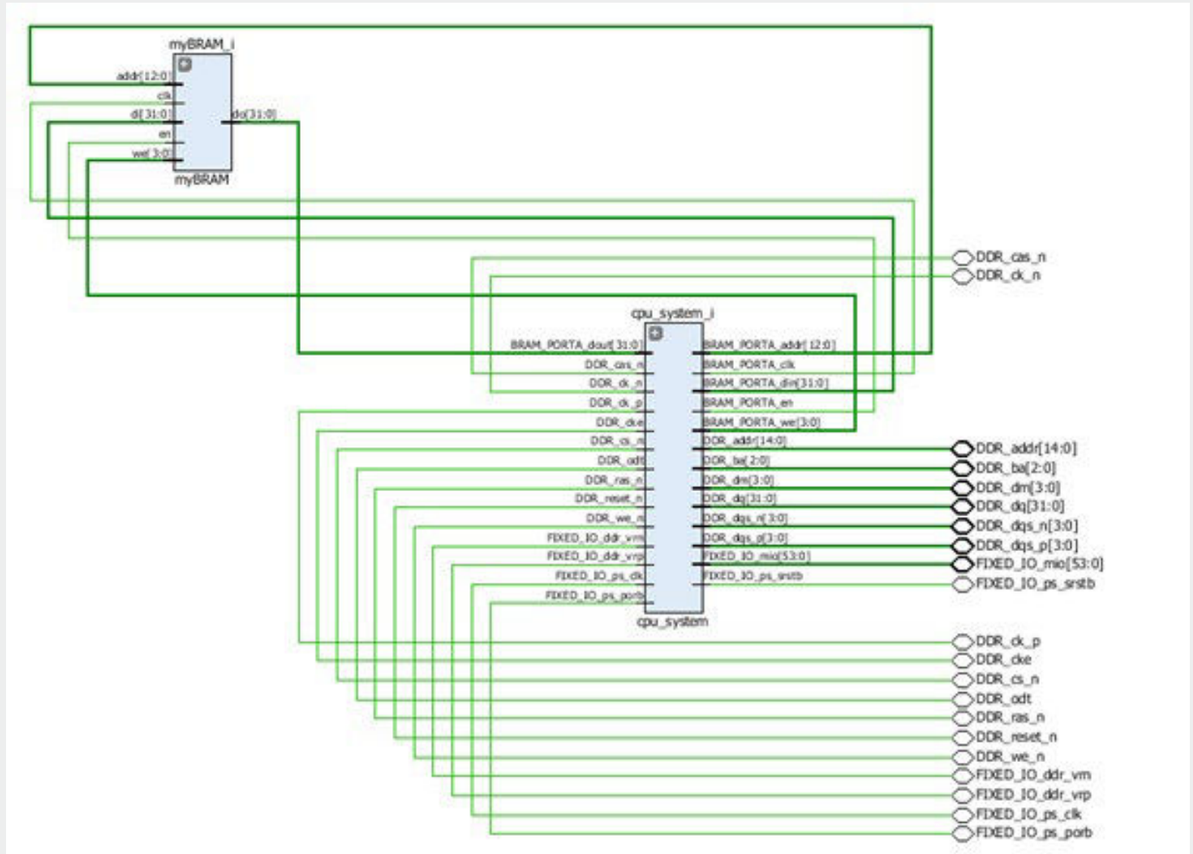


Figure 7a: Elaborated design with new BRAM interface

Running the same test app as before should give exactly the same results.

We now have full control over the BRAM interface and a simple bus to which we can attach any custom peripheral to the CPU without directly interfacing with the AXI interconnect.

TIP : When creating a new top level and exporting the bitstream, the SDK will create a new hardware platform. Remember to retarget the BSP for the new HW platform, otherwise the application project will be using an older bitstream.

Customise the new BRAM

So far, we have exposed the BRAM interface at the top level of our design and re-implemented the BRAM instance testing it functions correctly.

The next step is to modify our BRAM entity with some initialisation values that represent text, which we can read and print out in a console window in the SDK. This demonstrates how easy it is to make modifications.

We can look up the hex values for “Hello World !!!” and let the synthesis tool initialise the BRAM contents with these values when the FPGA is configured.

```
type ram_type is array (0 to 511) of std_logic_vector(31 downto 0);
signal RAM : ram_type := (
    X"6c6c6548", X"6f57206f", X"20646c72", X"00212121", others => X"00000000"
);
```

Figure 8: Block RAM initialisation

After synthesis, bitstream generation and programming, we can write some simple C code to read this out in SDK by creating a pointer to our BRAM address and casting it to char.

```
int main()
{

    volatile char *myBRAM = XPAR_BRAM_0_BASEADDR;
    int i;

    init_platform();

    printf("%.*s", 15, myBRAM);

    cleanup_platform();
    return 0;
}
```

Figure 9a: Custom BRAM result

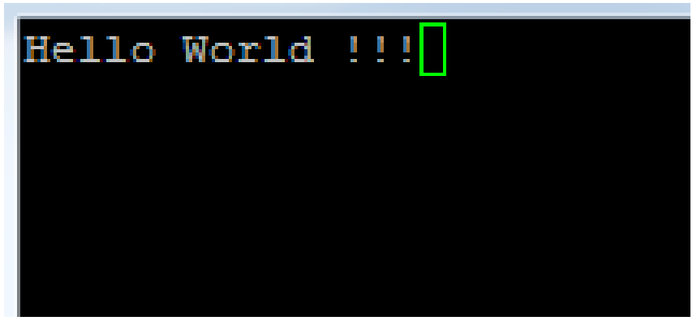


Figure 9b: Custom BRAM result

Changing the text is as simple as placing in new BRAM instance initialisation values.

Peripheral interfacing

Up to this point, we've added a BRAM to the BRAM interface and added some custom initialisation values; this is relatively simple.
Now we'll use the BRAM interface to talk with our customer peripheral.

To do this, we need to decode the BRAM address bus and emulate the BRAM interface timings in our code to make successful transfers.

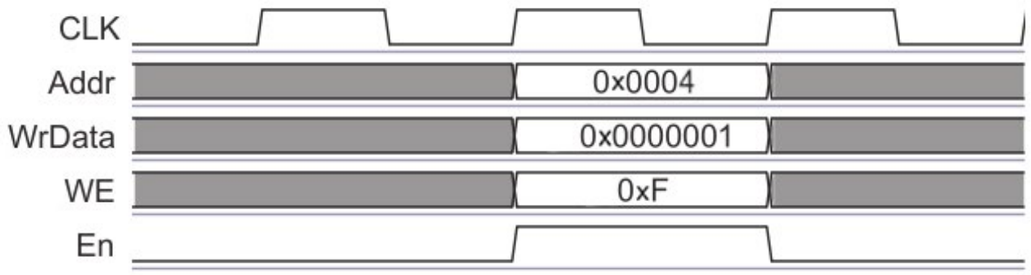


Figure 10: Writes

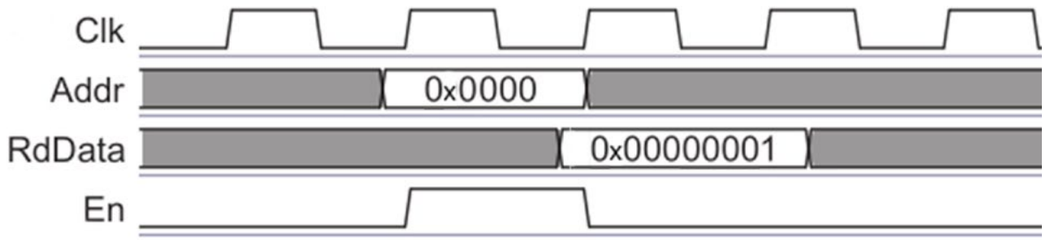


Figure 11: Reads

For writes, the address has to be decoded and written in the same clock cycle.

For reads, the address is decoded but data is presented on the following clock cycle back to the BRAM interface on the data out bus.

Decoding the address bus

There are a number of ways to decode the address bus,

The method described here uses a case statement to generate read and write strobes which are used to clock data in and out of the data buses.

The strobes can also be used to determine a read/write event has occurred to a particular address location, and are useful for FIFO read/write events, counters, reset triggers etc...

An address decoder in one location makes for easier code maintenance when adding more registers, FIFOs, buffers etc. later.

Decoding the address bus: to start, we need the required signals that will make up the registers and strobes of a simple FIFO.

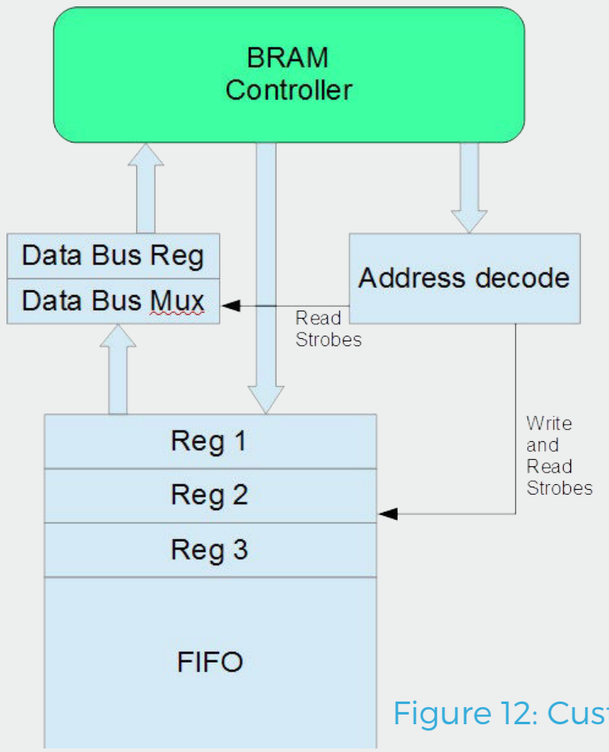


Figure 12: Custom BRAM result

Here, we will create a status and control register to allow the CPU to reset the FIFO and also monitor whether the FIFO is full or empty.

A couple of memory locations will also be required, to allow reading and writing of data.

This gives us a queue register which tells the software how full the FIFO is - useful for DMA transfers.

```
signal status_reg     : STD_LOGIC_VECTOR(1 downto 0)  := (others => '0')
signal ctrl_reg       : STD_LOGIC_VECTOR(1 downto 0)  := (others => '0')
signal fifo_read_reg  : STD_LOGIC_VECTOR(31 downto 0) := (others => '0')
signal fifo_write_reg : STD_LOGIC_VECTOR(11 downto 0) := (others => '0')
signal fifo_queue_reg : STD_LOGIC_VECTOR(11 downto 0) := (others => '0')

signal ctrl_wr       : STD_LOGIC;
signal fifo_read_rd  : STD_LOGIC;
signal fifo_write_wr : STD_LOGIC;

signal read_bus : STD_LOGIC_VECTOR(31 downto 0);
```

Figure 13: Custom BRAM result

```
ADDR_DECODE_PROC : process(ADDR, WE, EN, status_reg, ctrl_reg, fifo_read_reg, fifo_queue_reg)
    variable address     : std_logic_vector(15 downto 0);
    variable we_combined : std_logic;
begin
    ctrl_wr       <= '0';
    fifo_read_rd  <= '0';
    fifo_write_wr <= '0';

    we_combined := WE(0) or WE(1) or WE(2) or WE(3);

    read_bus <= x"FF1111FF";

    if (EN = '1') then
        address := "000" & ADDR(12 downto 2) & "00";
        case address is
            when x"0000" =>
                read_bus                           <= (others => '0');
                read_bus(status_reg'high downto 0) <= status_reg;
            when x"0004" =>
                read_bus                           <= (others => '0');
                read_bus(ctrl_reg'high downto 0)   <= ctrl_reg;
                ctrl_wr                            <= we_combined;
            when x"0008" =>
                read_bus     <= fifo_read_reg;
                fifo_read_rd <= '1';
            when x"000C" =>
                fifo_write_wr <= we_combined;
            when x"0010" =>
                read_bus                                 <= (others => '0');
                read_bus(fifo_queue_reg'high downto 0)   <= fifo_queue_reg;
            when others =>
        end case;
    end if;
end process;
        READ_BUS_REG_PROC : process(clk)
        begin
            if (rising_edge(CLK)) then
                DO <= read_bus;
            end if;
        end process;

        WRITE_BUS_REG_PROC : process(CLK)
        begin
            if (rising_edge(CLK)) then
                if (ctrl_reg(0) = '1') then
                    ctrl_reg    <= (others => '0');
                else
                    if (ctrl_wr = '1') then
                        ctrl_reg(1 downto 0) <= DI(1 downto 0);
                    end if;
                end if;
            end if;
        end process;
```

Figure 14: Custom BRAM result

The code opposite emulates a BRAM interface using a case statement for the decode logic. Strobes are generated to increment the FIFO counters and pointers as the FIFO registers are read/written by the CPU.

The address is aligned to a 4 byte boundary due to the 32-bit bus. From the CPU's point of view, these addresses are accessed starting at the BRAM controller offset address (0x4000 0000).

Therefore, to access the control register, we would need to write to location 0x4000 0004.

Summary

- Simple code, just a case statement and a process to register the read bus plus some additional strobes.
- **The code synthesis into a multiplexer and a register for data.**
- Easy to change and add more registers.
- Easy to maintain. It's all HDL.
- Making changes does not require running any other tools, Just add in a register and synthesise/simulate (provided enough address and data bits already exist).
- Adding extra I/O signals just requires a change to the peripheral source and the top level HDL file to bring them out to pins. This doesn't interfere with the BLOCK diagram.
- Multiple peripherals can be multiplexed together onto one BRAM interface bus.
- Can run the BRAM interface and the core of the peripheral at different clock speeds by adding synchronisers to registers in the IP.

Peripheral testbench

Before we **synthesis** our design, it's worth creating and running a test bench. Since we are using a BRAM access cycle for reading and writing to the peripheral, it's very simple to create a couple of procedures to perform a CPU read and write cycle.

```
procedure cpu_read_data(addr_in          : in  std_logic_vector(31 downto 0);
                        data_read        : out std_logic_vector(31 downto 0);
                        signal p_addr    : out std_logic_vector(12 downto 0);
                        signal p_data_rd : in  std_logic_vector(31 downto 0);
                        signal p_en      : out std_logic) is
begin
    p_addr <= addr_in(12 downto 0);
    p_en   <= '1';
    wait until clk = '1';
    p_addr <= (others => '0');
    p_en   <= '0';
    wait until clk = '1';
    data_read := p_data_rd;
end cpu_read_data;
```

Figure 15: Custom BRAM result

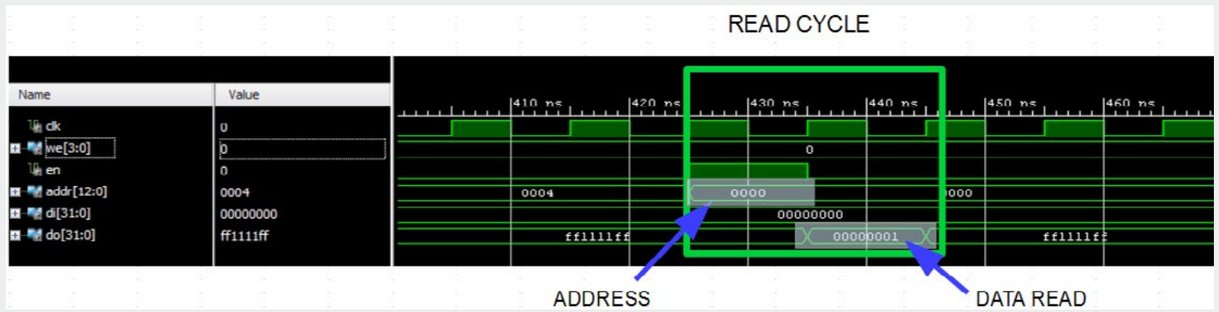


Figure 16: TestBench Read Cycle

The read procedure simulates a read cycle by placing writing an address to the peripheral, waiting one clock period then read the data.

The write is simpler as everything is performed in one clock period. The address and data to be written is placed onto the bus at the same time along with the enable and write strobe. The main loop in the testbench calls these procedures to access the peripheral.

```
procedure cpu_write_data(addr_in          : in  std_logic_vector(31 downto 0);
                         data_write       : in  std_logic_vector(31 downto 0);
                         signal p_addr    : out std_logic_vector(12 downto 0);
                         signal p_data_wr : out std_logic_vector(31 downto 0);
                         signal p_en      : out std_logic;
                         signal p_we      : out std_logic_vector(3 downto 0)) is
begin
    p_addr    <= addr_in(12 downto 0);
    p_data_wr <= data_write;
    p_we      <= "1111";
    p_en      <= '1';
    wait until clk = '1';
    p_en      <= '0';
    p_we      <= "0000";
    p_data_wr <= (others => '0');
end cpu_write_data;
```

Figure 17: TestBench Write Cycle

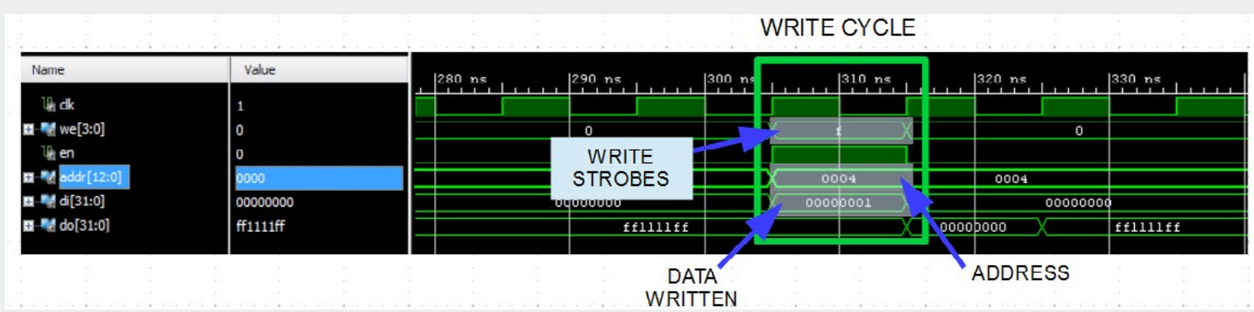


Figure 18: TestBench Write Cycle

We can also use assert or report functions to print messages to the simulator console window.

```
CPU_INTERFACE_PROC : process
    variable read_addr : unsigned(31 downto 0);
    variable data_read : std_logic_vector(31 downto 0);

begin
    we   <= "0000";
    en   <= '0';
    addr <= (others => '0');
    di   <= (others => '0');
    report ("-- Reset.");

    wait until (rst_n = '1');

    wait for 100 ns;
    for i in 0 to 10 loop
        wait until clk = '1';
    end loop;

    report ("-- Clear all registers and data");
    cpu_write_data(X"00000004", X"00000001", addr, di, en, we);

    -- Wait 100 AXI clk period for clear to complete and show a clean write cycle.
    for i in 0 to 10 loop
        wait until clk = '1';
    end loop;

    -- Check empty flag is true and full flag is false
    cpu_read_data(X"00000000", data_read, addr, do, en);
    if (data_read(1 downto 0) /= "01") then
        report ("-- ERROR : FAILED Check empty flag is true and full flag is false") severity failure;
    end if;

    for i in 0 to 10 loop
        wait until clk = '1';
    end loop;

    -- Check FIFO queue is 0
    cpu_read_data(X"00000010", data_read, addr, do, en);
    if (data_read /= X"00000000") then
        report ("-- ERROR : FAILED Check FIFO queue is 0") severity failure;
    end if;

...<SNIP>...
```

Figure 19: Custom BRAM result

This can be expanded with **TEXTIO** interface as the peripheral complexity increases.

The advantage here is that the read and writes are relatively simple procedures. Simulation for this peripheral has been moved outside of the IP flow altogether. Since the interface is simple, the testbench complexity is reduced.

Back into software

Once we are satisfied with the results from the testbench, we can synthesise the design, create a bitstream and configure the FPGA.
Next, bring up the SDK and write a simple test program to access the new peripheral completing the testing.

```
int main() {

      volatile unsigned int *status_reg = XPAR_BRAM_0_BASEADDR;
      volatile unsigned int *ctrl_reg = XPAR_BRAM_0_BASEADDR + 0x4;
      volatile unsigned int *fifo_read_reg = XPAR_BRAM_0_BASEADDR + 0x8;
      volatile unsigned int *fifo_write_reg = XPAR_BRAM_0_BASEADDR + 0xC;
      volatile unsigned int *fifo_queue_reg = XPAR_BRAM_0_BASEADDR + 0x10;

      int i;

      init_platform();


      printf("-- Clear all registers and data\n\r");
      *ctrl_reg = 0x1;


      printf("-- Check empty flag is true and full flag is false\n\r");
      if (*status_reg != 1) {
            printf("-- ERROR : FAILED Check empty flag is true and full flag is false : %x \n\r", *status_reg);
      }

      printf("-- Check FIFO queue is 0\n\r");
      if (*fifo_queue_reg != 0) {
            printf("-- ERROR : FAILED Check FIFO queue is 0\n\r");
      }

      printf("-- Fill the FIFO with a count\n\r");
      for (i=0;i<2048;i++) *fifo_write_reg = i;

      printf("-- Check FIFO queue is 2048\n\r");
      if (*fifo_queue_reg != 2048) {
            printf("-- ERROR : FAILED Check FIFO queue is 2048\n\r");
      }

      printf("-- Read all FIFO values back and check\n\r");
      for (i=0; i<2048; i++) printf(" %d ", *fifo_read_reg);

<Snip ...>
```

Figure 20: Custom BRAM result

```
-- Clear all registers and data
-- Check empty flag is true and full flag is false
-- Check FIFO queue is 0
-- Fill the FIFO with a count
-- Check FIFO queue is 2048
-- Read all FIFO values back and check
 0   1   2   3   4   5   6   7   8   9   10  11  12  13  14  15  16  17  18  19  20  21  22  23  24  25  26  27  28  29  30
31   32  33  34  35  36  37  38  39  40  41  42  43  44  45  46  47  48  49  50  51  52  53  54  55  56  57  58  59
 60  61  62  63  64  65  66  67  68  69  70  71  72  73  74  75  76  77  78  79  80  81  82  83  84  85  86  87  88
  89  90  91  92  93  94  95  96  97  98  99  100 101 102 103 104 105 106 107 108 109 110 111 112 113
 114 115 116 117 118 119 120 121 122 123 124 125 126 127 128 129 130 131 132 133 134 135 136
```

Figure 21: Custom BRAM result

Once a peripheral has been created, it's fairly easy to add more peripherals by splitting up the addresses into blocks on a power of 2 boundary, then create enable signals to select the peripheral you want to access.

For example, if we allocate 64K words of address space to the BRAM controller, this can be split up by decoding the most significant 2 bits of the used 18 bit address signals into 4 peripheral select lines.

These select lines can then be used to control data out multiplexors and address decoders of 4 peripherals.

Summary

There you have it - we've covered using the BRAM controller to avoid interfacing directly to AXI.

We have generated a simple test bench to simulate our peripheral and can easily access the peripherals from simple memory mapped registers in software.

Finally, since we used a BRAM controller with full AXI, it can support the CDMA - giving fast data access between the custom peripherals and other memory mapped structures, such as DDR memory.

Hopefully, this proves useful to you. If you have any technical enquiries feel free to contact the experts here at ByteSnap!

About the author

ROLAND BODLOVIC
Technical Manager, ByteSnap Design

Roland is an experienced electronics systems design engineer who has successfully completed numerous projects over the past 18 years.

Prior to joining ByteSnap Design, Roland worked in several industries - covering defence, aerospace, pro-audio and scientific instrumentation - which has provided a broad understanding across multiple engineering disciplines. He has spent the last 6 years in senior positions leading mixed-skilled teams.

Roland loves technical challenges and keeping up to date with the latest tools and technologies around.

About Us

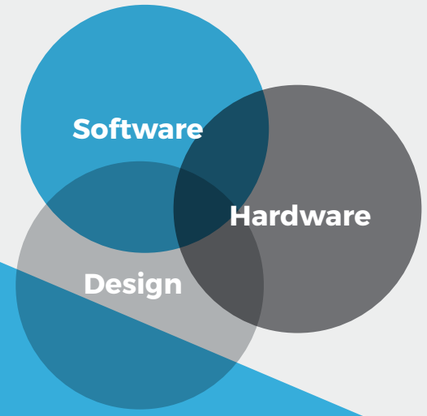
ByteSnap Design is an award-winning specialist in innovative embedded systems development, encompassing hardware and software design, with an international client list.

“We are a Full Service Software, Hardware and Electronics Design Consultancy

ISO 9001:2008 certified, ByteSnap Design is an NXP Approved Engineering Consultant and an ARM Connected Community Partner. The team’s experience ranges from electronic design through to BSP porting and mobile app development.

ByteSnap won 2016 Design Team of the Year and Consultancy of the Year in 2013 at the British Engineering Excellence Awards (BEEA). The team was ‘Highly Commended’ for design work on electric vehicle charging posts for the London 2012 Olympic Games at the BEEA 2012. ByteSnap also won European Design Team of the Year at the 2011 ELEKTRA awards.

The consultancy is expert in electronic circuit design, FPGA design, microcontroller design, Linux and embedded software development; designing hardware products from wireless sensors to ruggedised tablets with multiple software projects such as developing Android BSPs through to video processing applications.



CONNECT WITH US
LinkedIn
Facebook
Twitter
Google+

FPGA **DESIGN**

Interfacing Over AXI

Using A Simple Address Data Bus

ByteSnap Design
2 Devon Way
Longbridge Technology Park
Birmingham B31 2TS

info@bytesnap.co.uk

www.bytesnap.co.uk

+ 44 121 222 5433

BYTESNAP
embedded design excellence