Configuring Block Memory and Basic Software Debugging

Acknowledgement

With help from some Xilinx XUP training material.

Two Xilinx IP cores are used in this design. The first being the on chip BRAM memory controller, and the other being the Block Memory Generator. The controller interfaces the memory block to the processor bus. The Memory Generator instantiates the BRAM memory allowing you to configure the memory as required.

Goal

- Use IP integrator to connect and configure BRAM in a MicroBlaze system
- Be able to use a C program in SDK to interact with the BRAM
- Use some software debugging tools in an embedded processor environment.

Requirements

- Xilinx Vivado software
- Xilinx SDK software
- Xilinx Nexys 4 DDR board and a programming cable
- Enough disk space for the project files

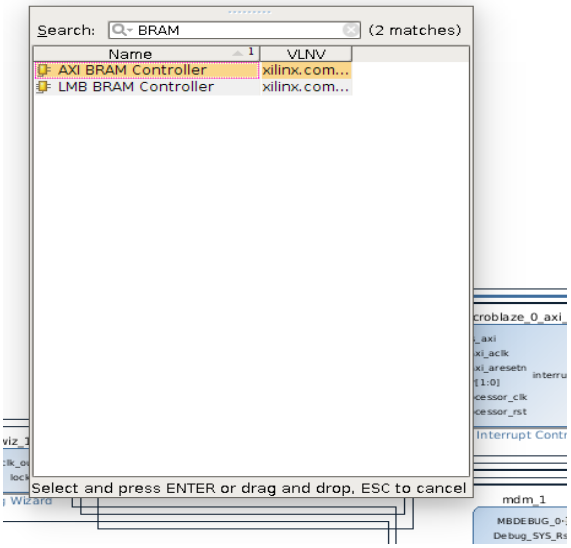
Background

The FPGA contains a number of memory blocks in the fabric that can be configured in many different ways, such as the aspect ratio (number of words versus the width of a word), the number of memory ports (up to two) and the behaviour of the memory, such as when there are simultaneous accesses to the same location on both ports. The BRAM can be attached to a variety of BRAM Interface Controllers. The BRAM Block structural HDL is generated by the design tools based on the configuration of the BRAM interface controller IP.

1. Adding a BRAM to Your Design

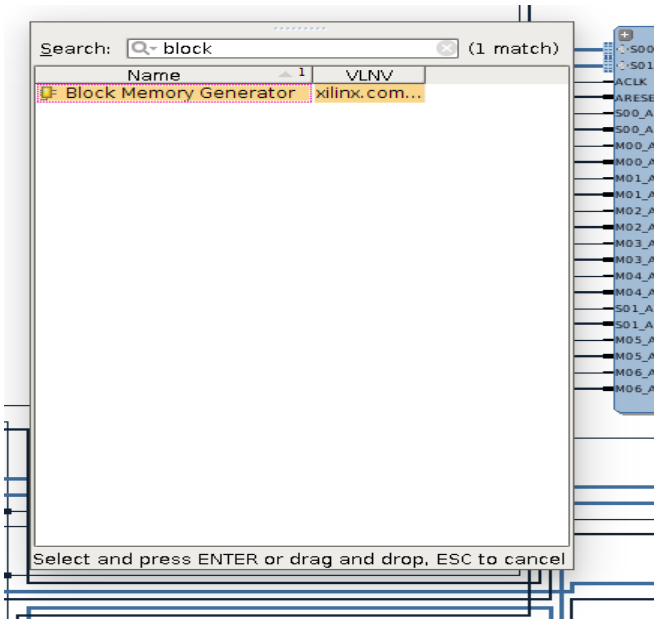
1. Start with the MicroBlaze design in the previous tutorial. You might want to copy the design tree first and then work with the copied directory so that you can preserve the previous design.
2. Under IP Integrator in the Flow Navigator, Open Block Design to see your design.

3. Open the IP Catalog from the Window menu at the top. Type in 'BRAM controller' in the Search box. Choose the one specified for an AXI interface.



4. Select Add IP to Block Design

5. Type in 'Block memory generator' and select the AXI4 Block Memory Generator.

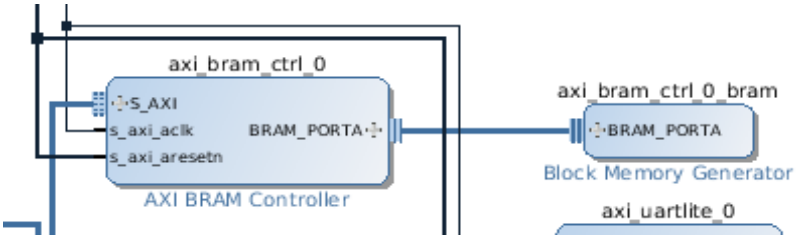


6. Add IP to Block Design

7. Go back to the IP integrator window and connect the BRAM controller to the processor bus. To do this, you will see a message on the top pane of the integrator window that says 'run connection automation'. Click that and connect the BRAM controller as a slave on the AXI bus. Clock Connection should be Auto.

8. The controller can be used to interface to both ports of a BRAM configured as a dual-port memory. In this case, we will use only one port. Double click on the BRAM controller to open the Re-customize window. Select only one interface.

9. Run Connection Automation again and click on the BRAM_PORTA item. Click OK for blk_mem_gen_0. This will connect PORTA of the BRAM to the AXI bus using the BRAM controller. You should now have the controller and BRAM connected like this:



10. We will need to increase the MicroBlaze memory size to run a larger program. Go to the Address Editor and increase the ilmb and dlmb sizes to 16K.

11. From the Tools menu, Validate Design.

12. Now Generate Block Design under IP Integrator in the Flow Navigator to rebuild the design files.

13. Next Run Synthesis, Run Implementation and Generate Bitstream.

14. You can now open up the address editor to see the address space of the BRAM controller. Note the starting address that has been assigned (0xC0000000) and that its default size is 4K words. Using a C program you can now read and write to that specified address space. A modified version of the helloworld.c program has been provided. Locate the original helloworld.c program that is in your design. It will be in ../microblaze.sdk/SDK/SDK_Export/mb_simple/src. Your pathname may be slightly different depending on how you named your design. Replace it with the new version. Inspect the new version to see what it does.

15. Export your design and start the SDK.

2. Accessing the BRAM from your C Program

1. When the SDK starts you may see an error in the Console window indicating that your program is too large. The linker places the various pieces of your program in the available memories. The default has been incorrectly selected. In the Project Explorer window, select your

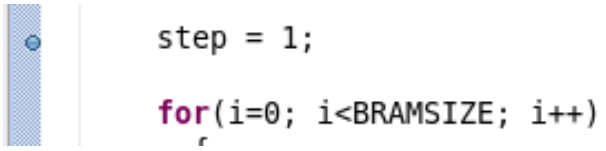
application project, likely *mb_simple*. Right click on it and select Generate linker script, or after selecting your application project, select Generate linker script under the Xilinx Tools menu.

2. In the Basic tab you will see pull-down options to select the memory to place the Code, Data and Heap and Stack. The defaults for Data, Heap and Stack have been placed in the BRAM that you just added. Using the pull-down, put these sections into the microblaze local memory. We want the BRAM to be available for other uses than just the program. However, you can now see how to add more memory for your program.
3. Select Generate and overwrite the existing linker script. Note that the linker script can be viewed in the src directory of your application project. It is called *lscript.ld.*
4. Your design will be recompiled and there should be no errors.
5. Examine the helloworld.c program and note the additional code to write and read to the new memory. Note also that the *print* function has been changed into a macro defined as *xil_printf.* This is because *print* can only output strings and we need the *printf* capability to output some numbers. We use *xil_printf* because it is significantly smaller than the full *printf*. The difference is that *xil_printf* cannot output floating-point numbers. However, *xil_printf* is still larger, which is why we needed to add memory for our program.
6. Program the FPGA and run your program. Verify that the new memory exists.

3. Basic Software Debugging in SDK

In this section, you will try some basic features in the gdb debugger interface of the SDK.

1. Right-click on the *mb_simple* project in the Project Explorer view and select **Debug As > Launch on Hardware**, or do this through the Run menu after you have selected your project. The elf file (binary) of your program will be downloaded and a dialog box will appear to switch to the Debug perspective, which is a different layout of windows for debugging in the SDK.
2. Click **Yes** to change perspectives.
3. Right click in the **Variables** tab and select **Add Global Variables**. The global variables will be displayed. Select *swt*, *i*, and *step* and click OK. Note that *swt* is the pointer to the switches and *i* and *step* are some counter variables.
4. In the *helloworld.c* source window, you can double click on the bar on the left to set breakpoints. Click beside all of the statements assigning to *step.*

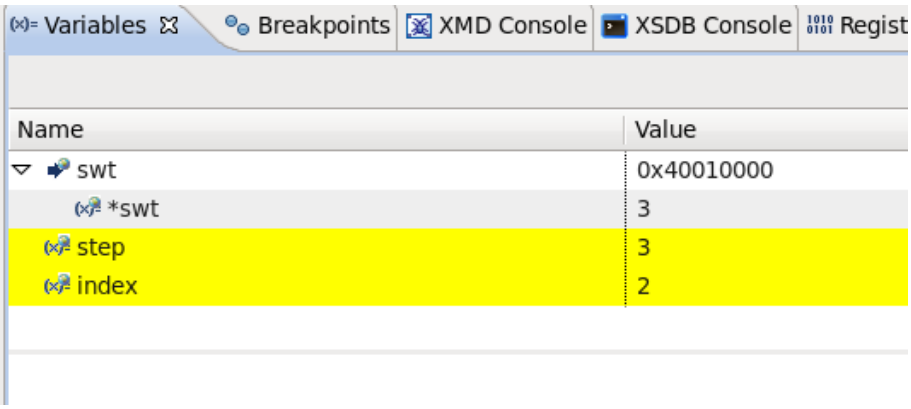


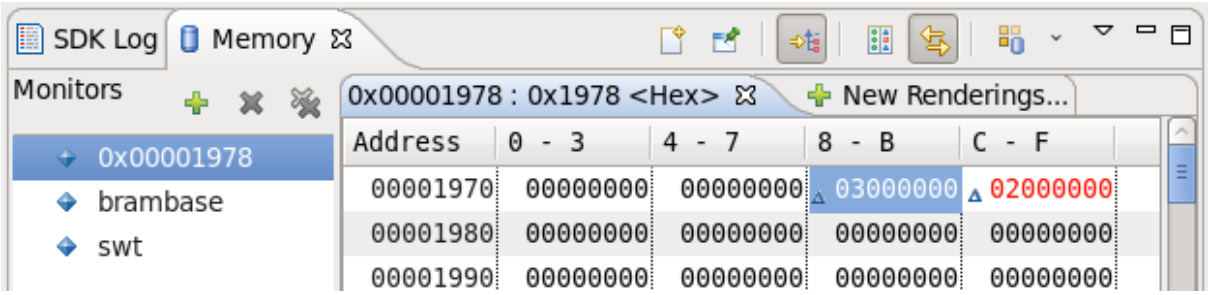
5. Click on the memory tab. If you do not see it, go to **Window > Show View > Memory**.
6. Click the + sign to add a Memory Monitor. Monitor the BRAM by entering *brambase*. Do the same for the switch register *swt*. This will allow you to monitor those memoy locations as they change.

   You are also supposed to be able to enter *&i* and *&step*, which would be addresses for those variables. This may not work. Another way to find those addresses is to use the symbol table generated by the compiler. This is in the .elf file for your program. Go back to Project Explorer in the C/C++ Perspective. You can change perspectives via **Window > Open Perspective**. Under *Debug* in your application, double click on the .elf file to open it. Part of that view will be the symbol table. Click in the window and search (cntrl-F) for *index* and *step*. The hex number on the left will be memory address for those variables, probably 0x197c and 0x1978, respectively.

   If you are running linux, you can go to the directory microblaze.sdk/SDK/SDK_Export/mb_simple/Debug. Here you will find *mb_simple.elf*. You can run the following command to get the symbol table in a sorted order:
   % mb-nm mb_simple.elf | sort -nr
7. Return to the Debug perspective.
8. You can enter 0x1978 in the Memory Monitor. Since the monitor displays several values, 0x197c will be next to 0x1978.
9. When you launched the session, the execution should have paused at *main()*. You can continue execution by hitting the Resume button  and execution will continue to the next breakpoint.

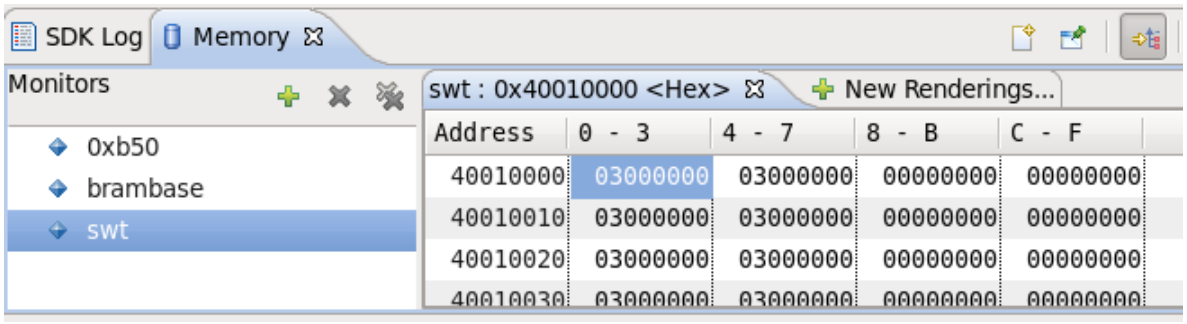
10. After a few iterations of the first loop, you should see the following:





You can see *step* and *index* in both the variable view and the Memory monitor view. Observe that in the memory view, the value is organized in byte address order, so the least significant byte comes first. Don't know how to change that...

11. Also, see that you can read the switch gpio register. Here, both switches are on. Change the switches and observe how the values change when you step the program.

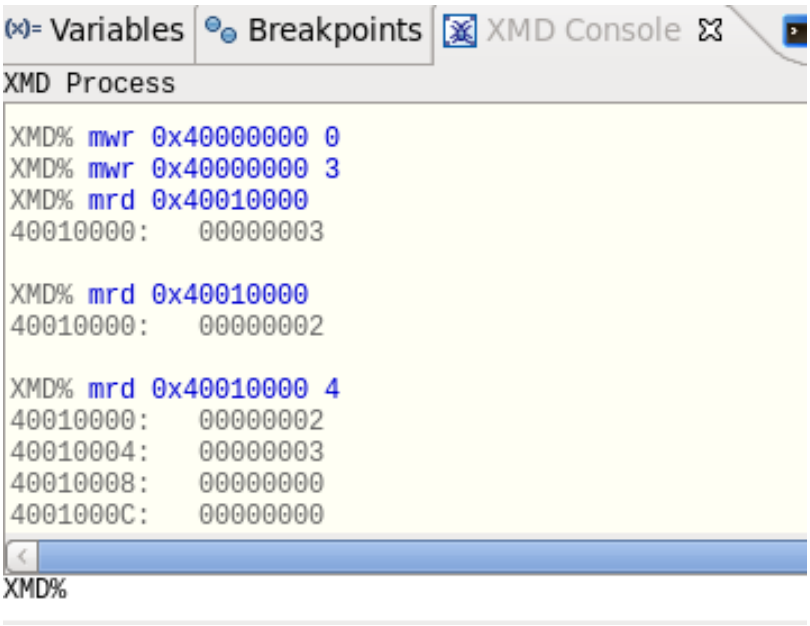


12. Continue stepping through the program. Watch *brambase* in the memory monitor to see the BRAM being changed in the writing loop. If you get tired of iterating through a loop, you can double click on the breakpoint and remove it before resuming.

13. You can read and modify the memory directly using the XMD Console. Go to the XMD Console tab. You can access the gpio registers to turn the LEDs on and off, as well as read the

switches.

The sequence shown in the figure is two writes to the LED gpio register, turning them off and then both on. Then there is a read of the switch gpio register with both switches shown on and then one switch turned off before the next read. The final read command shows that you can dump a number of memory locations at once, four in this case.



14. When you are done, click the Terminate button, which is the square to the right of the Resume button.