ECE532S Digital Systems Design

Tutorial 3 - Adding Memory to the Microblaze Project

Last Updated: July, 2019

In this tutorial, we build a MicroBlaze-based system similar to the one built in the previous tutorial, but we also include a connection from the processor to some additional memory (separate from the program memory instantiated by the *automated block connection*). In the previous design, the MicroBlaze was connected to a program and data Block RAM (BRAM) through the *Local Memory Bus* connection interface standard. This connection allows for fast access to BRAM memory from the MicroBlaze, but this memory is not shareable with other cores in our system. If we'd like to have a memory that is accessible from the MicroBlaze and some other core in our system, we usually rely on the *AXI* connection interface standard. The *AXI* standard explicitly allows multiple masters (devices that can issue read and write requests) and is commonly used in industry.

For this tutorial, we will show how to connect a microblaze to an AXI-based memory. In the first part of the tutorial, this memory will take the form of a BRAM, which recall from the previous tutorial is an SRAM-based memory available within the logic of the FPGA. Next, we will replace this BRAM memory with an AXI-based *memory controller* that can connect to the DDR memory on the Nexsys board (a 128 MB Micron MT47H64M16HR-25 DDR2 module). DDR memory has the advantage of offering larger storage size, though with the downside of longer access latency and less predictable bandwidth. Controllers that access DDR memory can also assert back pressure, which is a way of saying that it can stall masters from issuing new requests until it is ready to receive them; if your design needs to read/write every cycle, BRAMs should be used. Despite the downsides of DDR memory vs. SRAM-based memories, we often have to use DDR memory since the BRAM storage on an FPGA is limited. Make sure to consider these pros/cons when choosing what memory to use in your system.

1 Installing the Digilent Board Files

In previous tutorials, we created a project using constraints files to specify how signals within our design are to connect to the pins of the FPGA. For this tutorial, we will be using a *Board File* instead. Board Files contains these pin constraints, but also configuration details for the DDR memory. The Xilinx memory controller, called the *memory interface generator* (or MIG for short) is highly configurable and requires detailed information about the memory to function correctly. Rather than enter all of these configuration details by hand, we use a *Board File* provided by Digilent that has all these configuration settings saved.

The Digilent board files are not shipped with the default Xilinx install, so we need to download the board files and add them to your Xilinx install. Note, this is only possible if you have write access to the Xilinx install directory, steps to add the board files using the TCL console will be covered later. Download the source files from the Digilent github repo: www.github.com/digilent/vivado-boards/archive/master.zip. Open the zip file and navigate to the *new/board_files* directory. Extract the contents of this directory to the following folder:

```
<Vivado Install Directory>/Xilinx/Vivado/<Vivado Version>/data/boards/board_files
```

We've now added all of the Digilent board files to the Vivado install. Note, alternatively you can simply add the *nexsys4_ddr* directory, as this corresponds to the Nesys 4 DDR board (the *nexsys_video* corresponds to the Nexsys Video board which you may want to use for the project). The zip file provided with this tutorial contains these two board files in case the github link is unavailable, though they may be out of date.

Invoke the Vivado IDE to start creating a new project. Once Vivado opens, but before starting the *Create New Project* wizard, we can use the TCL Console to add board files too. If you have write access to the Xilinx install directory, the previously described steps should be followed to install the board files; if so, skip to section 2. Otherwise, open the TCL Console at the bottom of the Vivado interface (see Figure 1). Enter the following command into the console:

```
set_param board.repoPaths <path to extracted board files>/board_files/
```

This will add the board files to the current instance of Vivado; you may have to repeat this step each time you run Vivado.

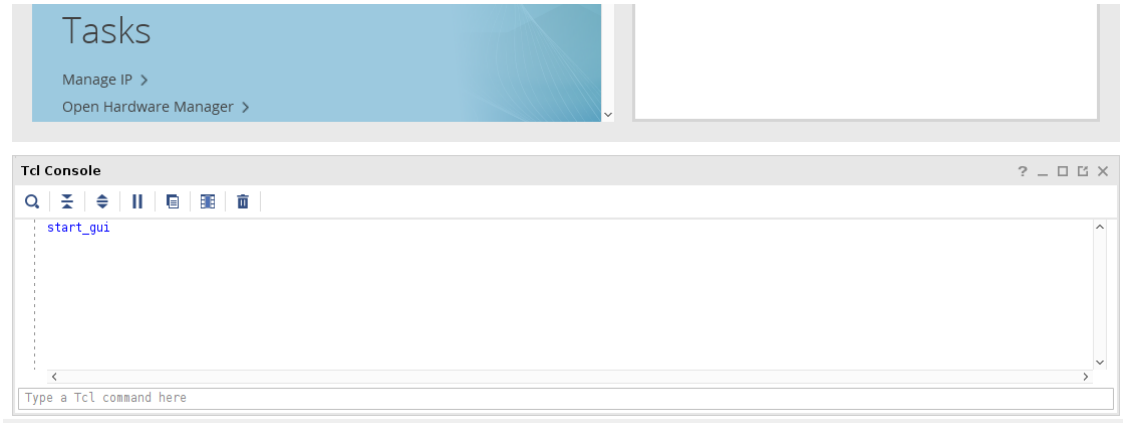

Figure 1: The TCL Console within the Vivado opening window

2 Creating a Vivado Project Using a Board File

From the *Getting Started* page, select **Create New Project**. In the *Project Name* dialog box, type the project name and location; make sure to choose a directory to which you extracted the contents of the provided *zip* file for this tutorial. In the *Project Type* dialog box, select **RTL Project**. In the *Add Sources* dialog box, ensure that the Target language is set to **Verilog**; we don't have any source to add to this project. We don't have anything to add in the *Add Constrains and Add Existing IP* dialog box, since we are using a board file instead of a constraints file. Note, it is possible to use both a board file and a constraints file in the same project; details will be provided later in the tutorial. In the *Default Part* dialog box, switch to the *Boards* selection option at the top to choose from the installed board files. From the *Vendor* drop-dwon menu, select **digilentinc.com**, and then select the **Nexsys4 DDR** option in the list (see Figure 2 for reference). Click **Next** and then click **Finish** to finish creating the project.

Follow the instructions from tutorial 2 to create a new *Block Diagram* within your new Vivado project. In the block design editor, you should notice a new *Board* tab within the sources window, as

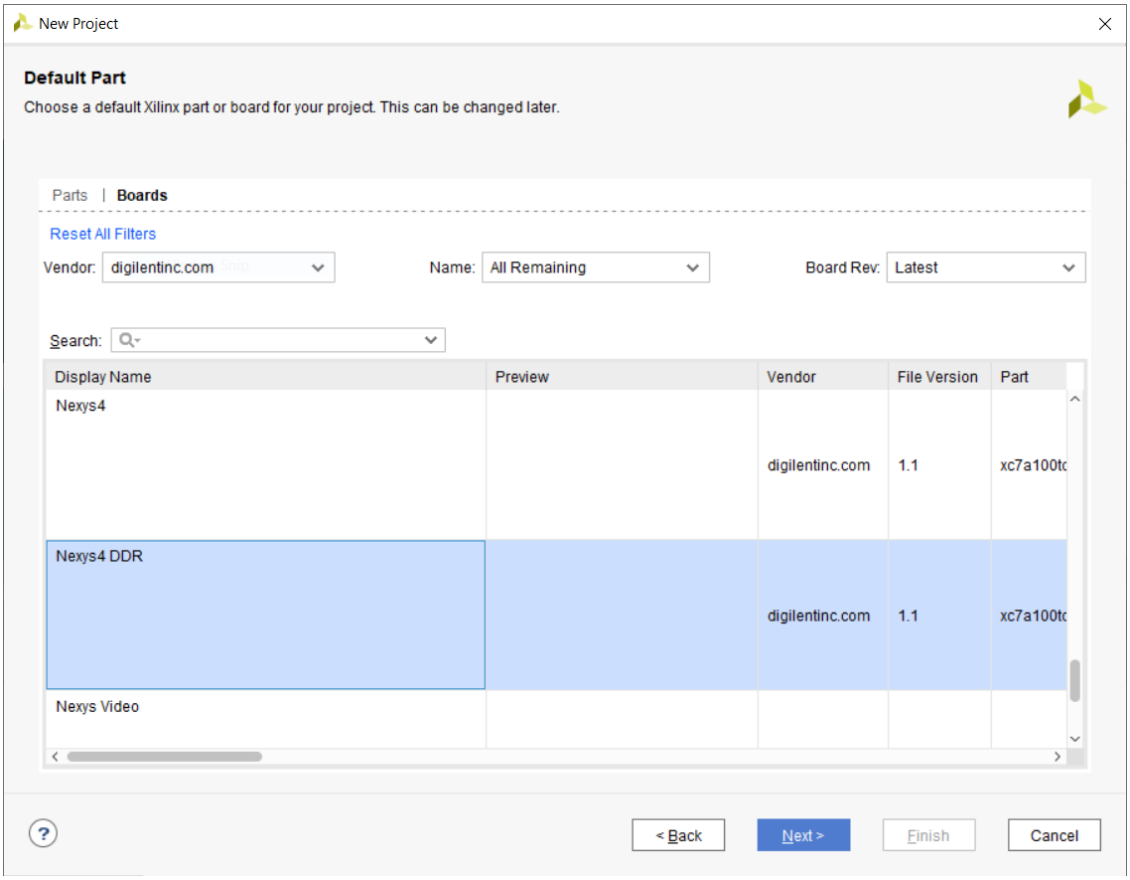

Figure 2: The board selection dialog window

per Figure 3. This tab lists all of the interfaces defined within the board file. To connect something to one of these interfaces, double click the interface and select from the list of cores capable of connecting to that interface. While this way of connecting to the interfaces defined in the board file does work, it is more common to instantiate the core itself first, and then indicate that the core should be connected to the interface; we'll show how to do this later in the tutorial. If you don't know which cores are capable of connecting to the interfaces you'd like to use, this is a good way to find out. once an interface is connected, the icon beside the interface changes from unconnected ( ) to connected ( ) . if you think a signal should be connected but your project doesn't seem to be working, you can check this *Board* tab to ensure the signal is being connected correctly.

3 Recreating the Simple MicroBlaze Project

We need to create the simple MicroBlaze project from tutorial 2 again, but there are some differences now that we are using a board file. Add the *MicroBlaze* IP and click **Run Block Automation**, as we did in tutorial 2. Set the *Local memory* to **32KB**, check the **Interupt Controller** option, and select the *Clock Connection* option of **New Clocking Wizard (100 Mhz)**. Click **OK**.

As with tutorial 2, we need to setup the *Clocking Wizard* configuration to connect to the Nexsys board's clock pin. **Double click** the *Clocking Wizard* to configure this core. you should notice a new tab called *Board*; this tab allows you to connect the pins of this core to any compatible pins defined in the board file. For the input labelled *CLK_IN1*, use the drop-down menu to select

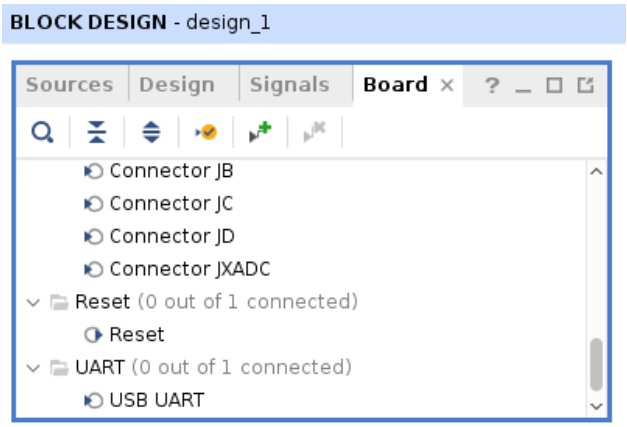

Figure 3: The Board tab within the Block Design editor window

sys_clock. This will automatically connect that pin of the Clock Wizard core to the *sys_clock* pin defined in the board file. For the input labelled *EXT_RESET_IN*, use the drop-down menu to select the **reset** pin. See Figure **??** for reference. If you go to the second tab of the configuration window, you should notice that Vivado automatically changed *clk_in1* to be a *Single ended clock capable pin*, which we had to do manually in tutorial 2. In the *Output Clocks* tab, the *Reset Type* still needs to be modified to **Active Low**, since this is an output of the core and not specified by the board file (i.e. this is a user decision).

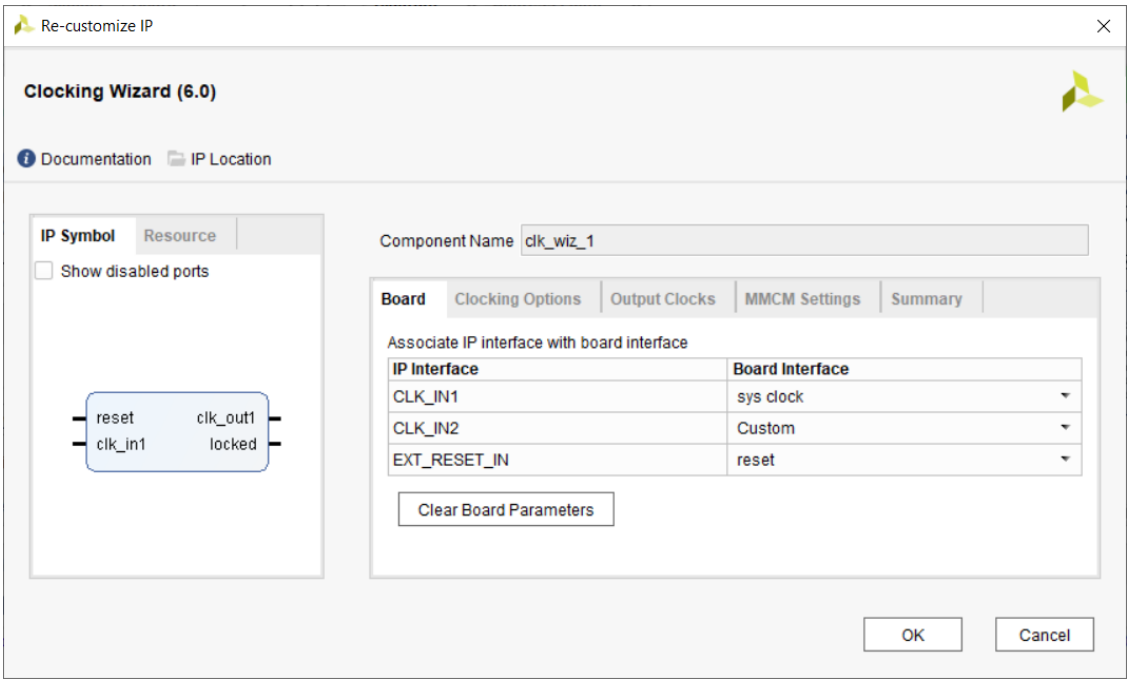

Figure 4: Clocking Wizard configuration dialog to select board file signals

As with tutorial 2, the last step we need to do in order to make this a fully functional system is to connect the clock and reset signals to external ports. To do this, we'll use *Connection Automation* again (Designer Assistance available. Run Connection Automation). In the left pane, we see the clock and reset signals of

the *Clock Wizard*, and the ext_reset_n signal of the *Processor System Reset* core. For each signal, there should be a drop down menu to select from compatible board file signals to connect to. In our case, each of these signals will have one option. Select the *sys_clock* signal for the clock input and the *reset* signal for each of the reset inputs. Note that we don't have to indicate that the reset is *Active Low* as we did in tutorial 2, since the board file has this information. Remember to actually tick the checkbox beside all the signals for which you want connection automation to run. See Figure 5 for reference. We should now have a fully connected design.

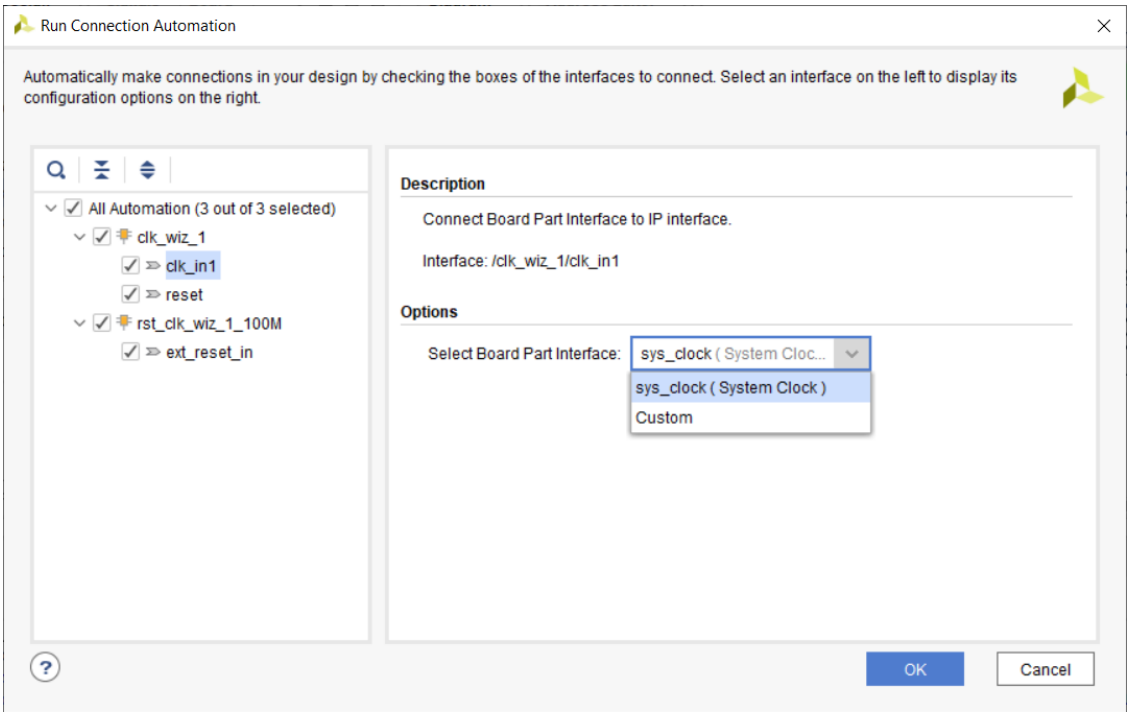

Figure 5: Connection automation for the clocking wizard

Note, if you wanted to synthesize the design as is, you would need to connect something to the inputs of the *Concat* block, otherwise you will get an error. A block that is commonly used to tie off unconnected signals is the the *Constant* block, which simply has an output that is some constant value. We will be adding the UART core to use the interrupt, so there is no need for us to tie this signal to a constant value at this time.

To add the AXI Uartlite peripheral to our project, **Right click** anywhere in the block diagram, select **Add IP** and search for and select the AXI Uartlite. Connect the **interrupt** signal of the AXI Uartlite to the **In0** signal of *Concat*, and configure the *Concat* core such that it has no unconnected inputs (as in tutorial 2). Run *Connection Automation* again to handle the remaining Uartlite signals. In the *Connection Automation* window, select to run *Connection Automation* for all signals by selecting all the check boxes in the left pane. Click the *uart* signal specifically to see that the board signal for the *usb_uart* is automatically selected for that connection; if it is not, select it from the drop-down menu. See Figure 6 for reference.

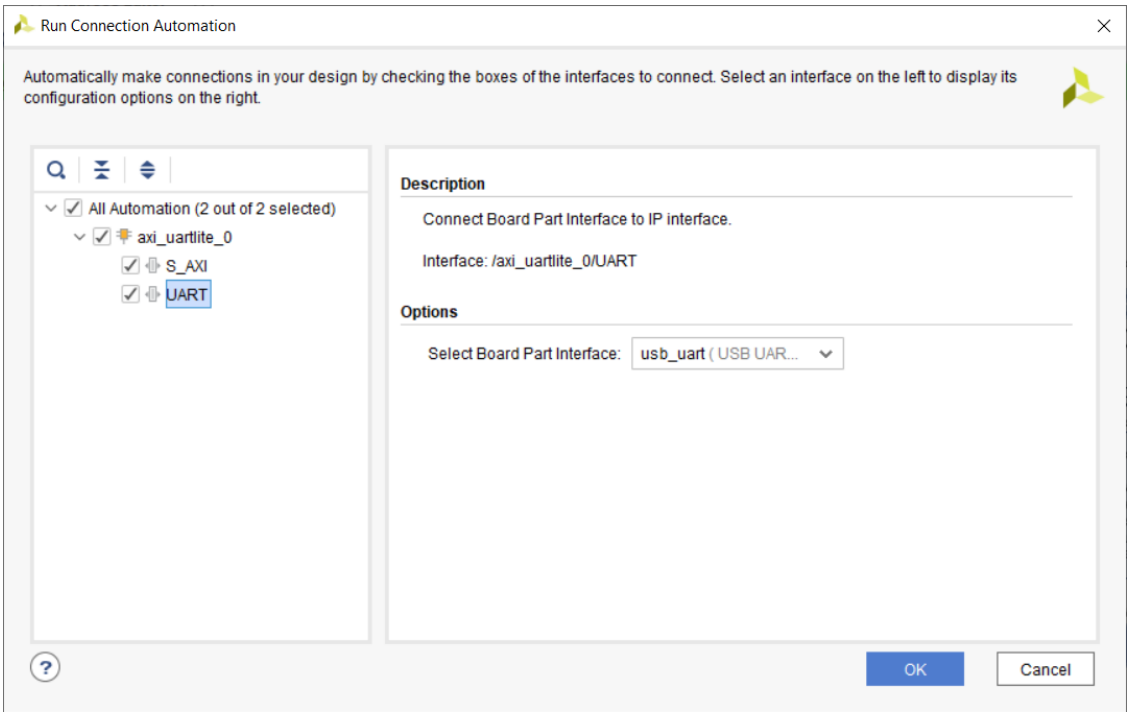

Figure 6: Connection automation for the uartlite

4 Adding an AXI BRAM Controller

To add an AXI BRAM to the MicroBlaze project, **right click** anywhere in the block diagram, select **Add IP** and search for and select the *AXI BRAM Controller*. Mdify the configuration of the BRAM controller by **double clicking** it to bring up the configuration window. We want to change the number of output BRAM interfaces to one, since we only need to interface with a single BRAM (See Figure 7 for reference). We can now run *Connection Automation* again to finish the connectivity for the newly added BRAM. In the *Connection Automation* window, select the check box to enable connection automation for all connections. Click on the *BRAM_PORTA* signal, and select **New Blk_Mem_Gen** in the *Blk_Mem_Gen* pull-dwon menu. This will instruct *Connection Automation* to create a new BRAM to connect the BRAM controller to. alternatively, you could have created a *Block Memory Generator* core before running *Connection Automation*. See Figure 8 for reference.

By default, the generated BRAM is created with a size of 8KB. To increase the size of the BRAM open the *Address Editor* view and expand the *Data* section of the *microblaze_0* core (or whatever you happen to call your MicroBlaze instance). You can change the *Range* pull-dwon menu for the *axi_bram_ctrl_0* entry to the size you'd like your BRAM to be. When the block diagram is synthesized, this *Range* parameter is propagated to the BRAM core and the size is set automatically. Increase the size to 16K, see Figure 9 for details.

Follow the steps for tutorial 2 to validate the block diagram, generate an HDL wrapper, ans synthesize, implement, and generate the bitstream for the design. Note, we don't need to check that the signal names in out wrapper match the names in our constraints file, since we don't have a constraints file. If, in a future project, you choose to connect some signals using the board file and some using a constraints file, you would need to check the top-level names of only those signals whose connections are not determined in the board files. Once the bitstream generation is complete,

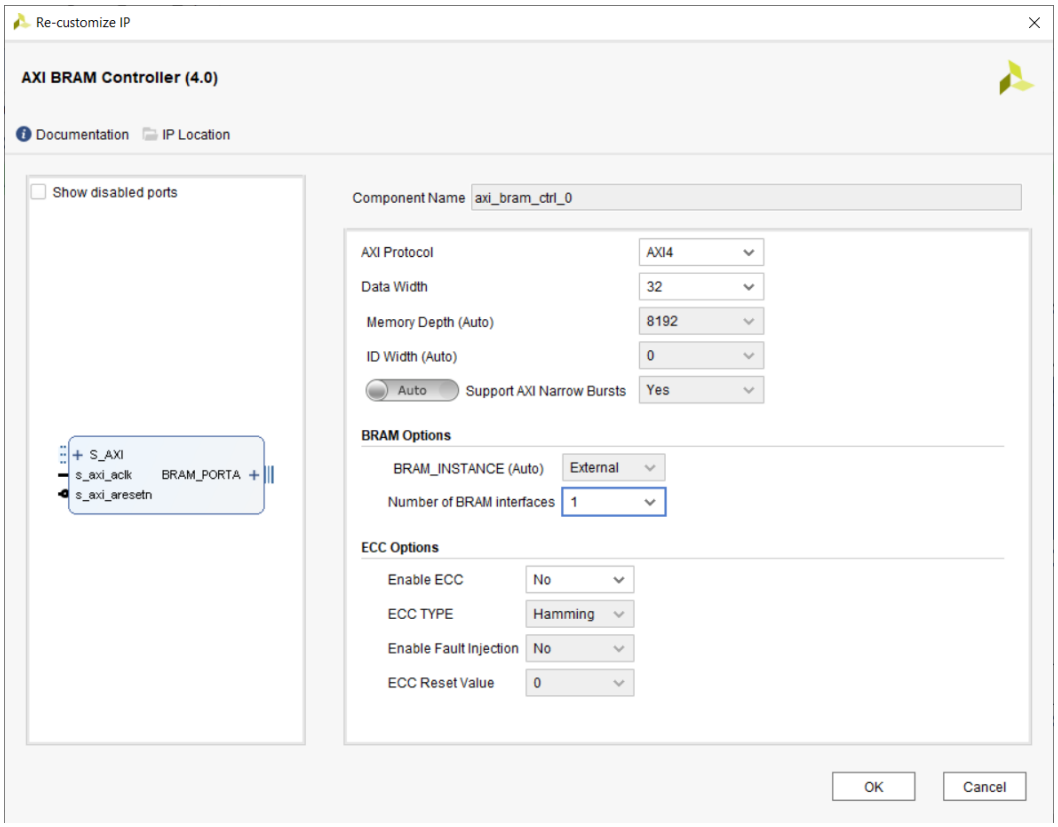

Figure 7: Configuration window for the AXI BRAM Controller

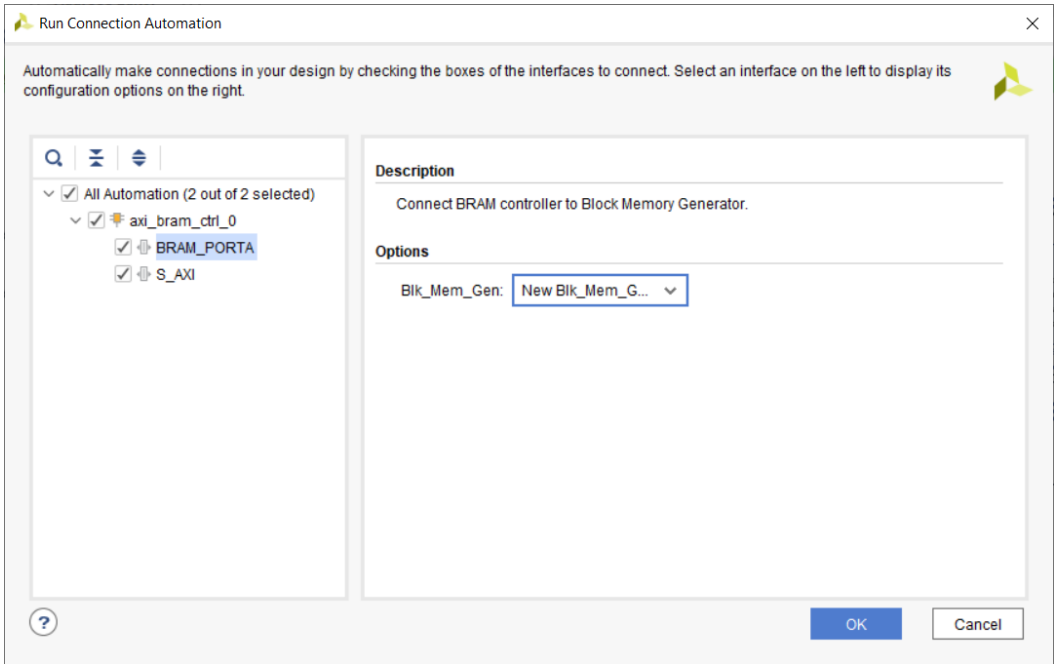

Figure 8: Connection automation for the AXI BRAM Controller

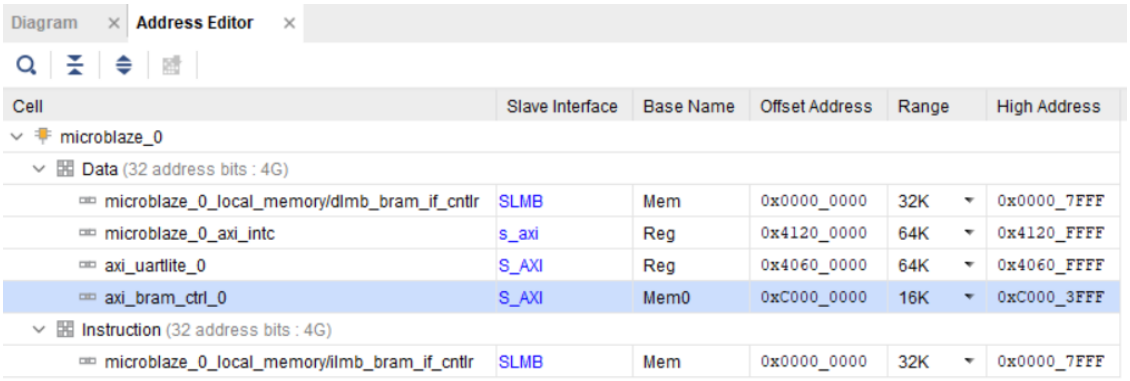

Figure 9: Address Editor with Range field modified

export the hardware and **Launch SDK** in order to test the system with software. When *Vivado SDK* launches, create a *Hello World* template program. Open the *helloworld.c* source file and modify it's contents to that of Figure 10. For more details on these steps, refer to the instructions in tutorial 2.

The application simply writes to the BRAM memory with increasing values and then reads back the values to make sure the writes were performed correctly. Note, the macro:

```
XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR
```

is defined automatically in the file *xparameters.h*, though make sure to check that file to ensure the macro has the same name. This program also uses the function *xil_printf*, which is a Xilinx version of the printf function. The *xil_printf* function has smaller memory footprint than the actual printf function, though it doesn't support floating point numbers. In some cases, applications we develop (if you want to use the full printf function for example) will be too large to fit into the instructions memory created when you ran *Block Automation* to create your MicroBlaze project. In this case, we'll need to change the linker script in order direct the assembler to store the code in a different memory. Now that we've added a memory, we can demonstrate that feature.

Right Click on your project name in the left pane and select **Generate Linker Script**; the window that appears is shown in Figure 11. On the left side is listed the various memories connected to your MicroBlaze, and on the right side we see the various parts of the program that we can assign to be stored in particular memory devices. Select the drop-down menu labelled *Place Data Sections in* and note that we have the BRAM controller listed here. If you select the drop-down menu labelled *Place Code Sections in*, you won't see the BRAM controller. This is because the BRAM Controller is only connected to the Data AXI interface in our original project, so the instruction port of the MicroBlaze cannot access our BRAM. We don't need to change the linker script at this, so simply press **Cancel**. These steps were simply to show you how you would move sections of the program to different memories for future projects. Note, the *Heap* is used for dynamic memory allocation, so if some program you are writing is running out of working memory, moving the heap to a larger memory should alleviate this problem.

Generate a *Run Configuration*, connect to the *UART Terminal*, and then run the program on the Nexsys board, based on the instructions from tutorial 2. Verify that we see the expected output. Next, create a *Debug Configuration* and run the application in debug mode, again based on the instruction from tutorial 2. Select **Yes** when it asks you to change to the *Debug perspective*. recall from tutorial 2 that this debug console implements the debug features of the gdb debugger.

```c
#include <stdio.h>
#include "platform.h"
#include "xil_printf.h"
#include "xparameters.h"

#define MEMSIZE 16
volatile unsigned int* membase = (unsigned int*) XPAR_AXI_BRAM_CTRL_0_S_AXI_BASEADDR;

int main()
{
    int i, val, err = 0;

    init_platform();

    print("Writing to Memory\n\r");
    for(i = 0; i < MEMSIZE; i++)
    {
        *(membase+i) = i+1;
    }

    print("Reading from Memory\n\r");
    for(i = 0; i < MEMSIZE; i++)
    {
        val = *(membase+i);

        if(val != i+1)
        {
            xil_printf("Error: at addr \%x, read \%d but expected \%d\n\r", membase+i, val, i+1);
            err = 1;
        }
    }

    if(!err)
    {
        print("No errors encountered\n\r");
    }

    cleanup_platform();
    return 0;
}
```

Figure 10: Modified Hello World application using BRAM memory

In regards to memory, one useful debug feature is the ability to add a memory monitor. Click on the memory tab in the lower right corner of the *Debug perspective*. If the memory tab isn't visible, go to **Window** → **Show View** → **Memory** to open the view. In the memory pane, click the + sign to add a *Memory Monitor*. In the dialog box that comes up, enter **brambase** and press

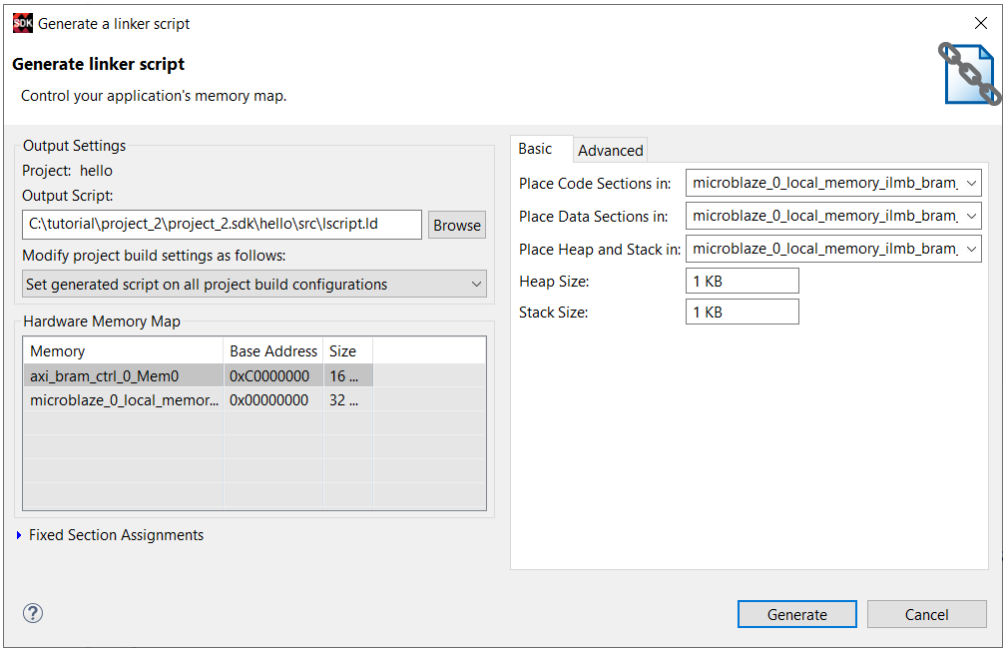

Figure 11: Generate Linker Script dialog

OK (see Figure 12 for reference). Now add some breakpoints to the code and run the program to watch the memory change as the writes to *brambase* are encountered in the code. This is a useful way to examine the contents of memory to debug your code.

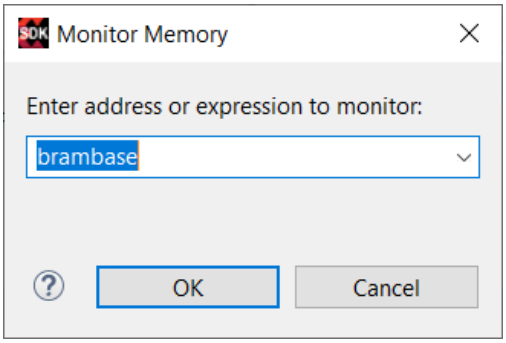

Figure 12: Adding a memory monitor in the GDB debug view

5 Adding an AXI DDR Controller

We want to replace the AXI BRAM controller we added to the project with an AXI DDR controller, to allow us to access the DDR memory on the Nexsys board. **Close** the Vivado SDK and reopen the block diagram containing your MicroBlaze project. Delete the *AXI BRAM Controller* and the *Block Memory Generator* from the design. **Right click** anywhere in the diagram and select **Add IP**. Search for the *Memory Interface Generator (MIG) peripheral* and add that core to your project. Press **Run Block Automation** and select to run *Block Automation* on the *mig_7series_0* core that was just added to your project. Error [BD 41-1273] may appear during automation, but can be

safely ignored. The block automation used the data in the board file to set the configuration details for the memory controller. If you open the configuration window for the *MIG* (by double-clicking) and step through the panes of the configuration, you'll note that there are a lot of configuration options for the memory controller. In fact, the *MIG* even requires all the pin locations to be entered by hand. Using the board file saves us this hassle. Make sure not to change any of the settings while you look at the configuration panes of the *MIG*.

To connect the *MIG* to the MicroBlaze, click **Run Connection Automation**. Select all of the check-boxes for the *MIG* signals except for the *sys_clk_i* signal; the connection automation tries to connect this signal to the output of the *Clocking Wizard*, which generates errors later in the project flow. Once the connection automation has finished, manually connect the *sys_clk_i* signal to the *sys_clock* siganl used as the input to the *Clocking Wizard*. Open the *Address Editor* and examine the newly added entry for the *MIG*; the *MIG* has been allocated a memory space of size 128M, which corresponds to the size of the DDR chip connected to the FPGA on the Nexsys board. Select the **DDR2** bus on the memory controller, right click and select **Make External**, to propagate these signals to the pins of the FPGA.

As a final step, we'll modify the MicroBlaze such that the instruction port of the processor can access the AXI interface as well. Note, our program from before was small enough to fit in the BRAM, but we include these steps here as they may be useful for future projects in which the application cannot fit in the BRAM. **Double Click** the MicroBlaze to bring up its configuration window. Advance to the fourth page and select **Enable Peripheral AXI Instruction Interface** (see Figure 13 for reference). Press **OK** to enact the change. You'll notice that a second AXI port appeared on the MicroBlaze processor. To connect this port, run **Connection Automation** once again. The *AXI Interconnect* will be automatically modified to include 2 AXI master ports, with both the data and instruction port from the MicroBlaze connected.

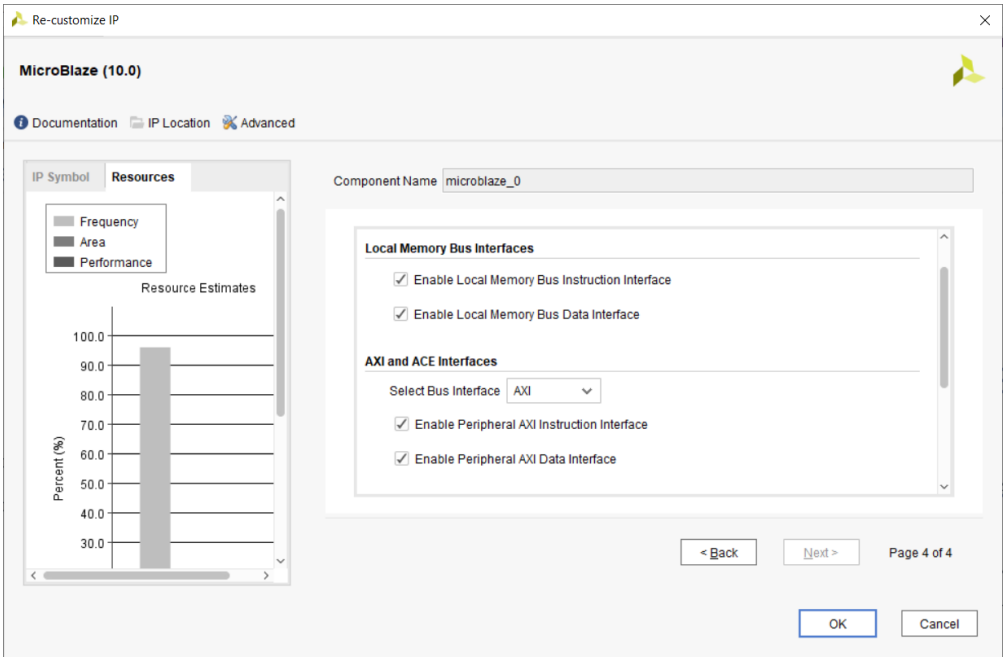

Figure 13: Reconfiguring the MicroBlaze to support AXI for instructions

Now that we've added the MIG and reconfigured the MicroBlaze, validate the deign and then synthesize, implement, and generate the bitstream for the project. Once the project is done building,

export the hardware and launch the **Vivado SDK**. See tutorial 2 for details. Once the *Vivado SDK* window comes up, open the helloworld example that we created earlier and change the contents of the *helloworld.c* file to reflect the changes in Figure 14.

```
1  #include <stdio.h>
2  #include "platform.h"
3  #include "xil_printf.h"
4  #include "xparameters.h"
5
6  #define MEMSIZE 16
7  volatile unsigned int* membase = (unsigned int*) XPAR_MIG_7SERIES_0_BASEADDR;
8
9  int main()
10 {
11     int i, val, err = 0;
12
13     init_platform();
14
15     print("Writing to Memory\n\r");
16     for(i = 0; i < MEMSIZE; i++)
17     {
18         *(membase +i) = i+1;
19     }
20
21     print("Reading from Memory\n\r");
22     for(i = 0; i < MEMSIZE; i++)
23     {
24         val = *(membase+i);
25
26         if(val != i+1)
27         {
28             xil_printf("Error: at addr \%x, read \%d but expected \%d\n\r", membase+i, val, i+1);
29             err = 1;
30         }
31     }
32
33     if(!err)
34     {
35         print("No errors encountered\n\r");
36     }
37
38     cleanup_platform();
39     return 0;
40 }
```

Figure 14: Modified Hello World application using DDR memory

The primary change made is the updating of line 7 with the base address of the *MIG*:

```
XPAR_MIG_7SERIES_0_BASEADDR
```

This macro is now defined in the *xparameter.h* file of the *board support package*, which was updated when we reexported the hardware for our newly updated bitstream. **Run** the *Run Configuration* that we already created earlier in the tutorial and observe the output printed to the console (make sure to connect to the UART device from the SDK Terminal tab first). We should see the output *No errors encountered*. Also, rerun the *Debug Configuration* and verify that you can add a *Memory Monitor* for ddr based memories as well. **Add** a *Memory Monitor* for address *membase*, **add** a breakpoint right before the write to *membase*, and then step through the program to see the memory change one entry at a time.

6 Summary

MicroBlaze systems often need to connect to memories other than the primary data and instruction memory created by the *Block Automation* tool. The AXI protocol is generally used to facilitate this connection, and Vivado has AXI-based controllers for both the BRAM memory within the FPGA and the DDR memories often connected to the pins of the FPGA. The MicroBlaze project was augmented with both of these types of memories, and an example program was developed that could write and read from these two types of memories. The use of *board files* greatly eased this process, as it included both the pin constraints for all external pins we needed to use, removing the need for an *xdc* file, and the configuration details for the DDR2 memory.