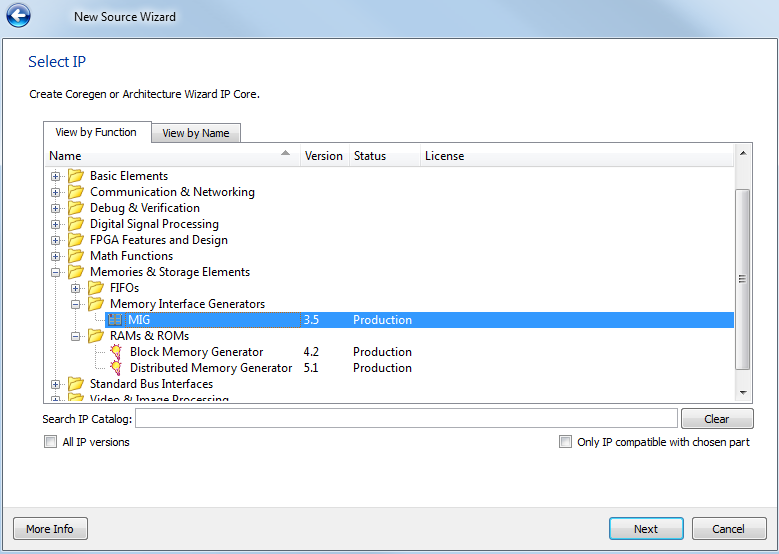
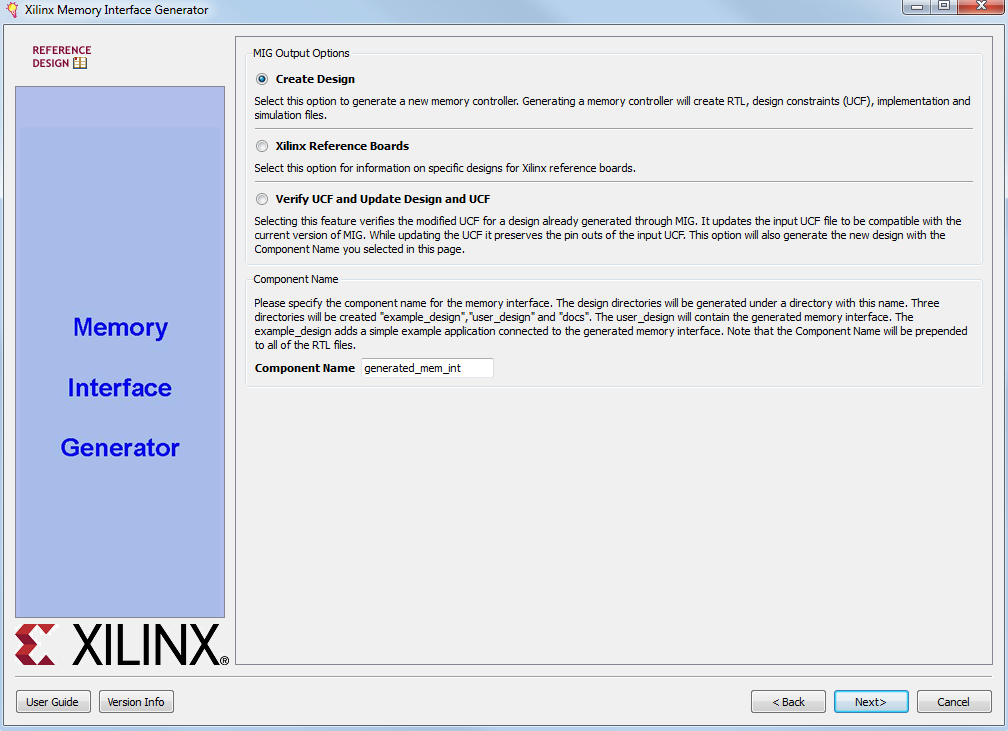
**Instantiating the Memory Interface Generator**

-Add new source

-IP Cores -> MIG



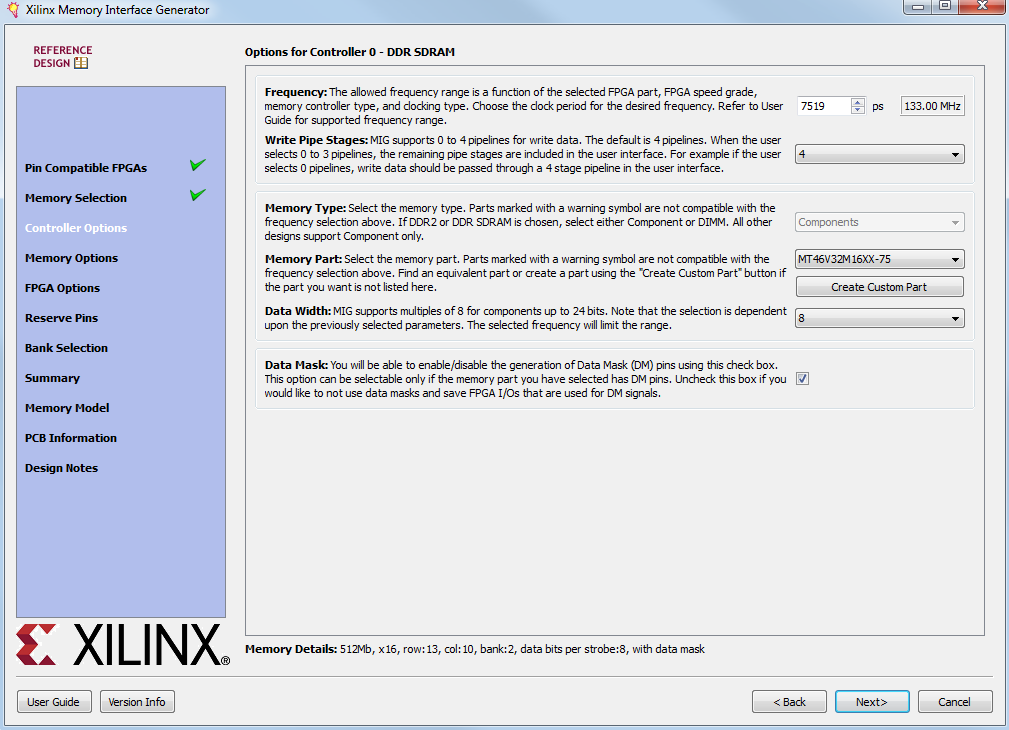
- Select Create Design



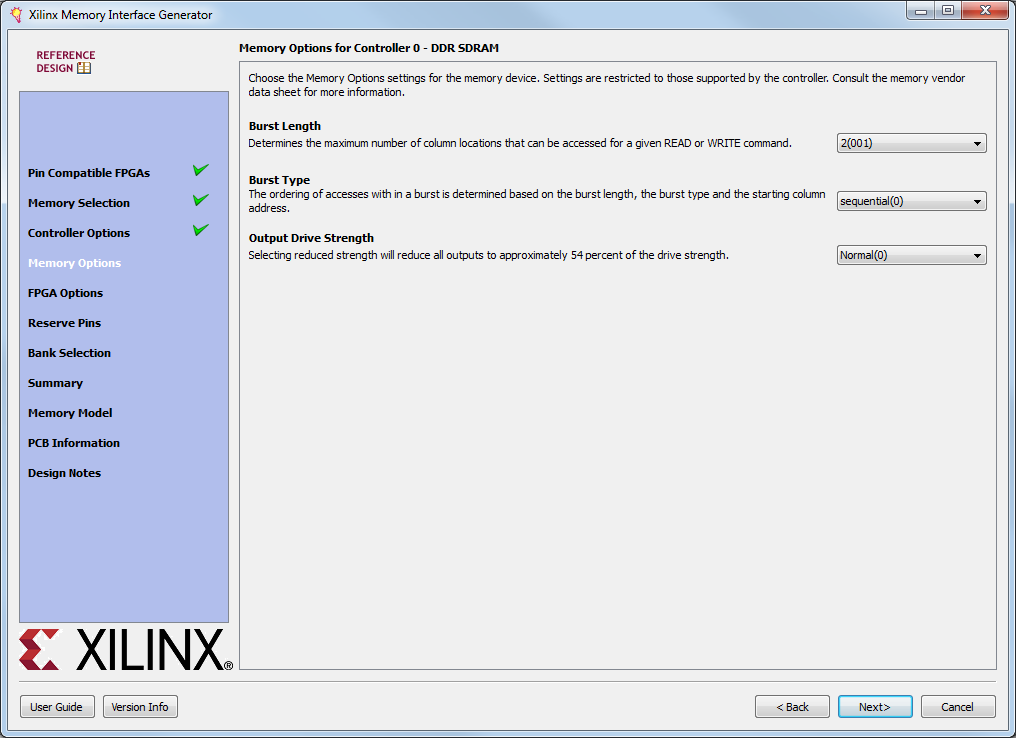
-**Pin Compatible FPGAs**: There are no other FPGA with compatible pins (you are only using the design on the starter kit anyways). Leave everything unchecked.

-**Memory Selection**: DDR SDRAM should be selected

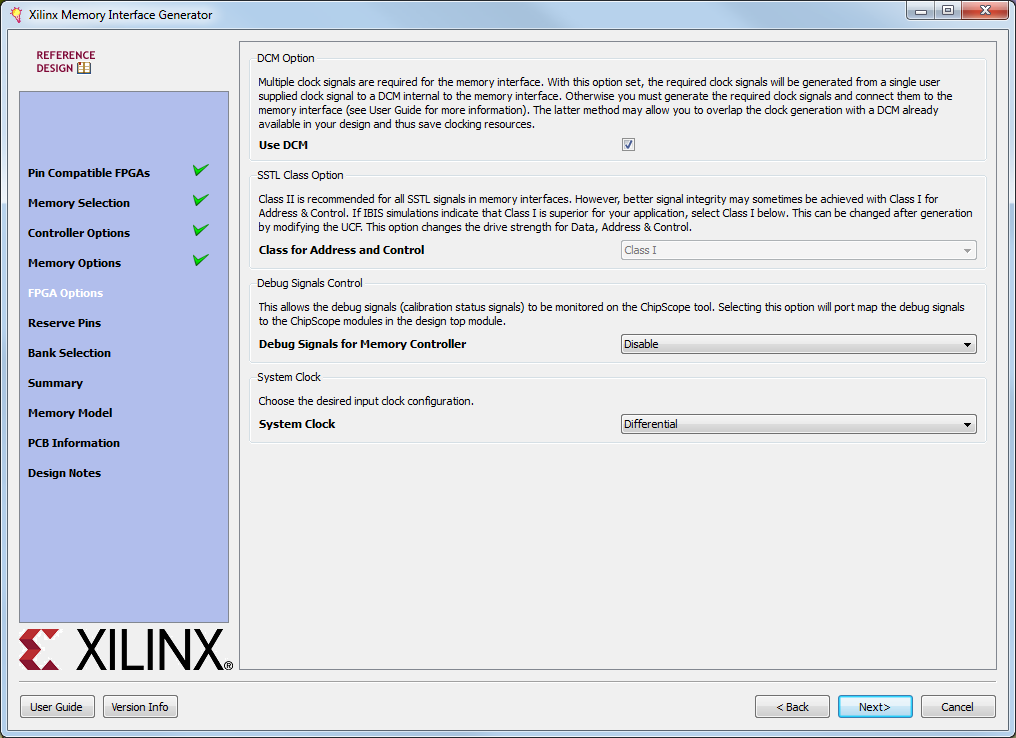
-**Controller Options**: Change Memory Part to be MT46V32M16XX-75 (memory is actually -6T but that isn't a choice, so we have to run it slower). The data width on the part is 16-bit, so select a 16 bit data width. We do not need a data mask, so you can uncheck this box.



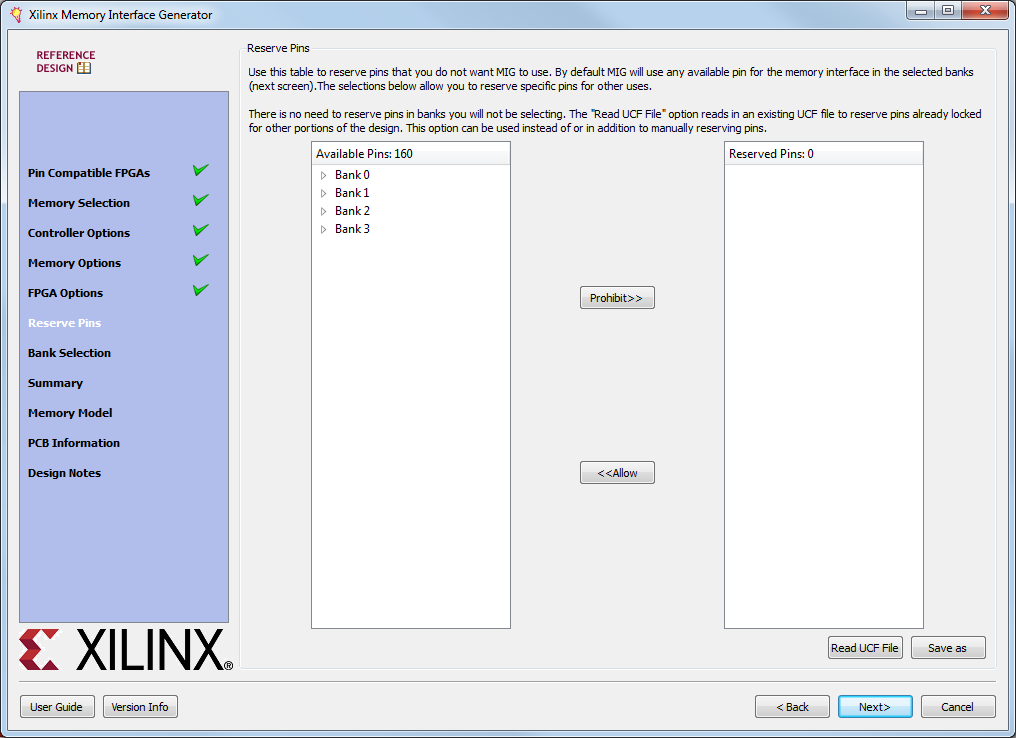
-**Memory Options**: Defaults



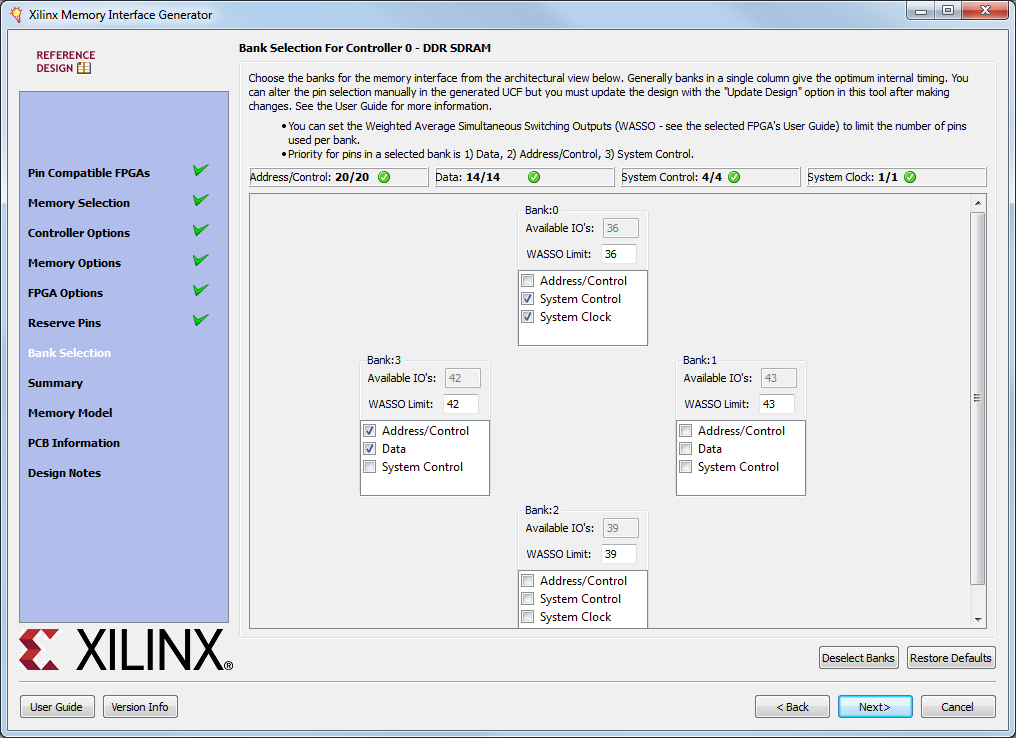
-**FPGA Options**: Make sure DCM is checked (Note, if you set clock to Single-Ended instead of Differential you only get one clock input .sys\_clk\_in rather than sys\_clk and sys\_clkb, but this tutorial used differential so you must also supply an inverted clock)



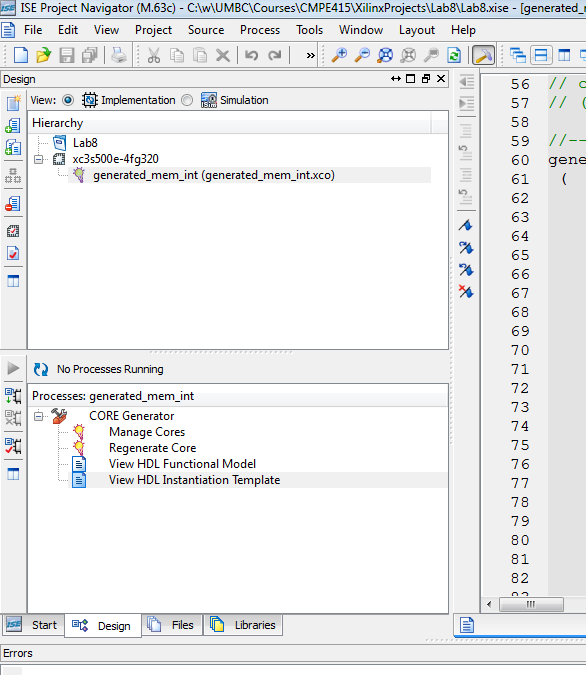
-**Reserve Pins**: Make any pin reservations you need for LEDs, buttons, etc… here. The easiest way to do this is import a ucf file. Or just, manually handle the ucf modifications later.



-**Bank Selection**: Check what banks are to be used for interfacing with the memory. This will automatically select pins and generate a ucf to the memory for you. Depending on your reserved pins you will have to select different values. If you can try to use banks 0 and 3.



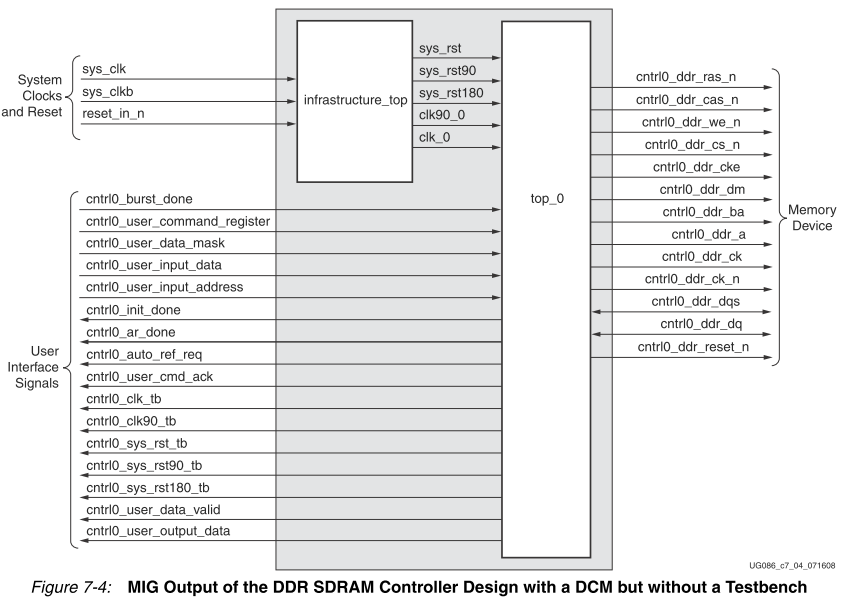
When you are finished you can view a template of how to instantiate the memory interface:



**Here is my template as a chart with some useful notes to focus your attention.**

I suggest you create another module (state-machine-based) to interface to this module so that you have something easier to work with at the top level. **I’ve underlined the only pins that really require much attention.** Detailed desc. of pins are in tables 7-7,7-8,7-9 of the user guide.

|  |  |  |
| --- | --- | --- |
| **generated\_mem\_int u\_generated\_mem\_int**  **(**  **.cntrl0\_ddr\_dq (cntrl0\_ddr\_dq),**  **.cntrl0\_ddr\_a (cntrl0\_ddr\_a),**  **.cntrl0\_ddr\_ba (cntrl0\_ddr\_ba),**  **.cntrl0\_ddr\_cke (cntrl0\_ddr\_cke),**  **.cntrl0\_ddr\_cs\_n (cntrl0\_ddr\_cs\_n),**  **.cntrl0\_ddr\_ras\_n (cntrl0\_ddr\_ras\_n),**  **.cntrl0\_ddr\_cas\_n (cntrl0\_ddr\_cas\_n),**  **.cntrl0\_ddr\_we\_n (cntrl0\_ddr\_we\_n),**  **.cntrl0\_ddr\_dm (cntrl0\_ddr\_dm),**  **.cntrl0\_rst\_dqs\_div\_in (cntrl0\_rst\_dqs\_div\_in),**  **.cntrl0\_rst\_dqs\_div\_out (cntrl0\_rst\_dqs\_div\_out),**  **.sys\_clkb (sys\_clkb), //I would have called this clk\_n**  **.sys\_clk (sys\_clk),**  **.reset\_in\_n (reset\_in\_n),**  **.cntrl0\_burst\_done (cntrl0\_burst\_done),**  **.cntrl0\_init\_val (cntrl0\_init\_val),//!!\*\*documents call this init\_done**  **.cntrl0\_ar\_done (cntrl0\_ar\_done),**  **.cntrl0\_user\_data\_valid (cntrl0\_user\_data\_valid),**  **.cntrl0\_auto\_ref\_req (cntrl0\_auto\_ref\_req),**  **.cntrl0\_user\_command\_register (cntrl0\_user\_command\_register),**  **.cntrl0\_user\_cmd\_ack (cntrl0\_user\_cmd\_ack),**  **.cntrl0\_clk\_tb (cntrl0\_clk\_tb),**  **.cntrl0\_clk90\_tb (cntrl0\_clk90\_tb),**  **.cntrl0\_sys\_rst\_tb (cntrl0\_sys\_rst\_tb),**  **.cntrl0\_sys\_rst90\_tb (cntrl0\_sys\_rst90\_tb),**  **.cntrl0\_sys\_rst180\_tb (cntrl0\_sys\_rst180\_tb),**  **.cntrl0\_user\_data\_mask (cntrl0\_user\_data\_mask),**  **.cntrl0\_user\_output\_data (cntrl0\_user\_output\_data),**  **.cntrl0\_user\_input\_data (cntrl0\_user\_input\_data),**  **.cntrl0\_user\_input\_address (cntrl0\_user\_input\_address),**  **.cntrl0\_ddr\_dqs (cntrl0\_ddr\_dqs),**  **.cntrl0\_ddr\_ck (cntrl0\_ddr\_ck),**  **.cntrl0\_ddr\_ck\_n (cntrl0\_ddr\_ck\_n)**  **);** | **I**  **I**  **I**  **I**  **O**  **I**  **O**  **O**  **O**  **O**  **I**  **O**  **O**  **O**  **O**  **O**  **O**  **I**  **O**  **I**  **I** | **TO RAM**  **TO RAM**  **TO RAM**  **TO RAM**  **TO RAM**  **TO RAM**  **TO RAM**  **TO RAM**  **TO RAM**  **TO BOARD**  **TO BOARD**  **inverted clk**  **clk**  **set low for 200+us then high**  **Signify end read/wrt. burst**  **initialization complete**  **auto-refresh done**  **Data valid for read**  **Req. pause for auto-refresh**  **3 bit command**  **Command acknowledge**  **Satus/Test Only**  **Satus/Test Only**  **Satus/Test Only**  **Satus/Test Only**  **Satus/Test Only**  **write mask,may fix to 2’b00**  **two byte read data**  **two byte data to write**  **read/write address (25bit)**  **TO RAM**  **TO RAM**  **TO RAM** |



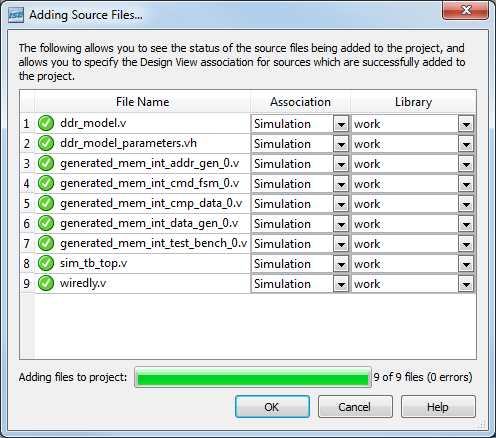
**Adding files for simulation**

Luckily, the core generator provides you a testbench to see the operation of the module, if you need it, but more importantly it provides a model of the external RAM so that you can run a simulation.

Once we have the MIG in our project, we need to add parts so that we can simulate the DDR SDRAM. This includes a model for the DDR module that is on your boards.

-Add source

-Go to /<Project Directory>/ipcore\_dir/<MIG module name>/user\_design/sim/ and add all the files in this directory. When the window pops up confirming the addition of all files **make sure to change the association from all to simulation**. The test bench and other various files will not synthesize correctly, and can only be used for simulations.



-Change your view to simulation, and run the testbench (sim\_tb\_top.v) in iSIM

If you have done everything correctly, you should see the waveforms for the connections that the MIG has. The testbench just asserts a reset signal for 200ns, just enough to reset the memory. You can modify this testbench to include the other parts of your project to test them out before you implement them in hardware.

To make sure you understand how the MIG interfaces with the DDR SDRAM, you should create a state machine that performs the following operations on the memory. You will be able to use this state machine in your project to perform reads and writes on the memory.

**To create your own logic to control the memory, remove the last module in sim\_tb\_top.v and add your own module and modify the test bench to control it. You may want to use add the system reset signal, sys\_rst\_n, to that as well.**

**////////////////////////////////////////////////////////////////////////////////////////**

**/\* REPLACE THIS MODULE WITH YOUR OWN (YOU DON’T EVEN NEED ALL THESE SIGNALS)**

**// synthesizable test bench provided for wotb designs**

**my\_memory\_interface\_test\_bench\_0 test\_bench0 (**

**.auto\_ref\_req (cntrl0\_auto\_ref\_req),  
 .fpga\_clk (cntrl0\_clk\_tb), //just use system clk  
 .fpga\_rst90 (cntrl0\_sys\_rst90\_tb),   
 .fpga\_rst180 (cntrl0\_sys\_rst180\_tb),   
 .clk90 (cntrl0\_clk90\_tb),   
 .burst\_done (cntrl0\_burst\_done),  
 .init\_done (init\_done),  
 .ar\_done (cntrl0\_ar\_done),  
 .u\_ack (cntrl0\_user\_cmd\_ack),  
 .u\_data\_val (cntrl0\_user\_data\_valid),  
 .u\_data\_o (cntrl0\_user\_output\_data),  
 .u\_addr (cntrl0\_user\_input\_address),  
 .u\_cmd (cntrl0\_user\_command\_register),  
 .u\_data\_i (cntrl0\_user\_input\_data),  
 .u\_data\_m (cntrl0\_user\_data\_mask),  
 .led\_error\_output (error), //don't need this  
 .data\_valid\_out (data\_valid\_out)**

**);**

**\*/**

**/////////////////////////////////////////////////////////////////////////////**

**Once you have the memory working with your new module, you’ll want to put separate your synthesizable module and along with the synthesizable FPGA memory controller “mem\_interface\_top0” in a separate top Verilog file as those are the only pieces that you will need for synthesis. You should modify the test bench to use that accordingly.**

**With the recommended options above, you will be presented with a 32-bit data input. You will only need to provide one address per read or write and the generated memory controller will auto-increment the column address to perform two 16-bit writes or reads.** THE READ AND WRITE TIMING DIAGRAMS DO NOT REPRESENT EVERY CONFIGURATION EXACTLY. YOU ONLY NEED TO PROVIDE ONE ADDRESS AND ONE 32-BIT DATA INPUT FOR THE CONFIGURATION WE CHOOSE.

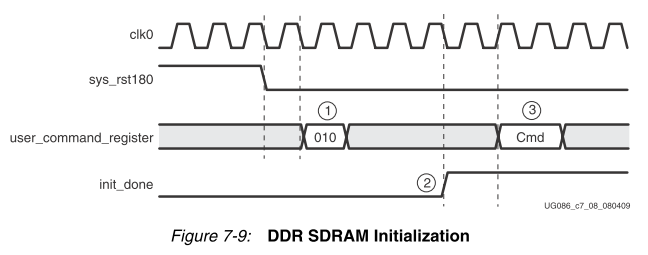
**Initialization**

**To use the SDRAM you must first reset and initialize the memory**.

-To reset the MIG, hold the reset bar low for at least 200µs

-A clock cycle after reset goes low the put init command (3’b010) on the user\_command\_register (1)

-When the init\_done line goes high (2), you can send your next command at anytime (3)



**Write**

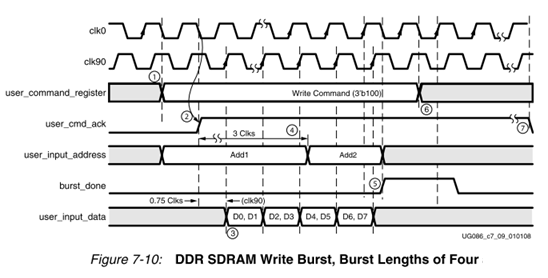
The controller automatically handles the both blocks of data being written and all 32-bits of data are located on **cntrl0\_user\_input\_data**. In addition to the handling of the data, the MIG also handles the row, column, and block selection and integrates all 3 of these into a simple 24-bit address **cntrl0\_user\_input\_address**. Since we set the burst length of 2, each burst will consist of 1 set of data.

-Put the address of where you are writing to on user\_input\_address, put the data you want to write on user\_input\_data (3), and set the write command (3’b100) on the user\_command\_register. (1)

-When user\_cmd\_ack goes high, wait for 3 clock cycles (2)(4)

-Set burst\_done to high for 2 clock cycles (5)

-Set the user\_command\_register to NOP (3’b000) (6)

-Wait until user\_cmd\_ack goes low again before sending another command (7)

**Read**

-Put the address of where you are writing to on user\_input\_address and set the write command (3’b110) on the user\_command\_register. (1)

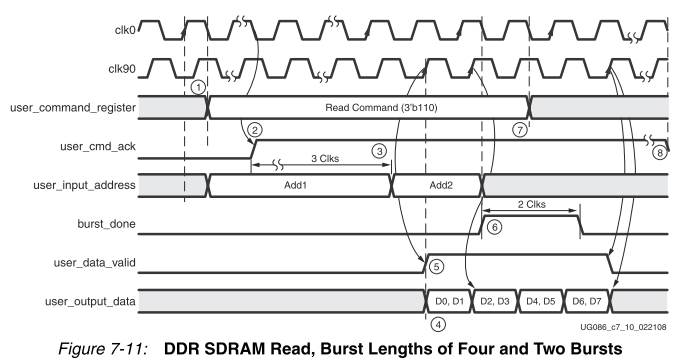
-When user\_cmd\_ack goes high, command was accepted (2)

-Once user\_data\_valid goes high the data is available on user\_output\_data (3)(4)(5)

- Set burst\_done to high for 2 clock cycles (6)

-Set the user\_command\_register to NOP (3’b000) (7)

-Wait until user\_cmd\_ack goes low again before sending another command (8)



**Auto-refresh**

If at any time the **cntrl0\_auto\_ref\_req** goes high, you have 15 clock cycles to terminate any command or the data on the DDR SDRAM will become corrupt (due to not being able to process the autorefresh). So, when **cntrl0\_auto\_ref\_req** goes high, just hold off any further operations until **cntrl0\_ar\_done** goes high. Once **cntrl0\_ar\_done** is asserted you can resume commanding the memory. Handling this issue is quite simple: just check **cntrl0\_auto\_ref\_req** before starting any new commands and if it is high wait for **cntrl0\_ar\_done** to go high**,** otherwise just execute your desired commands as usual.

A list of interface signals can be found on page 295 of the MIG ipcore.