Implementing DDR2 Cache Memory using Xilinx Memory Interface Gen in Pipeline MIPS (May 2022)

Joshua A. Rothe, *Student, Johns Hopkins University, Whiting School of Engineering*

Table of Contents

I. Abstract

The pipeline MIPS processor from Li utilizes a TLB (Translation Look-Aside Buffer) to access memory coded directly onto the FPGA fabric as internal memory. This implementation is simple (relatively) to code in Verilog but utilizes a lot of FPGA slices in doing so, which hurts resource utilization upon synthesis. In most FPGA designs, space is at a premium, and a constant trade-off between timing requirements, space, reliability, and power usage occurs. A typical pipeline MIPS processor utilizes several cache memory instances to allow pipelining to work as needed – this is space that can be utilized for other purposes, if the cache memory can instead take advantage of the provided memory on the FPGA device being synthesized to.

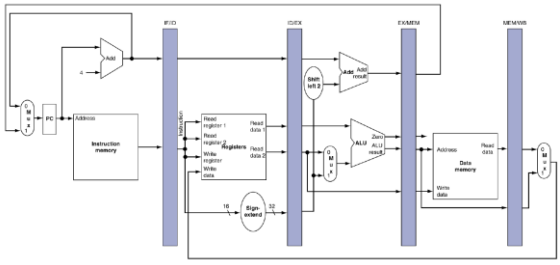

Fig. 1 – Pipelined MIPS Block Diagram

The Nexys A7 board from Xilinx, the development board provided for this lab course, has dedicated DDR2 memory that can be tapped into and utilized to free up space on the FPGA fabric. In keeping best practices for FPGA design, a better cache memory implementation would be to utilize this dedicated DDR2 memory. To tap into this resource, Xilinx's Memory Interface Generator (MIG) can be used, which configures the memory on the board for use with a Vivado FPGA design and can be instantiated into a project like any other IP module once configured.

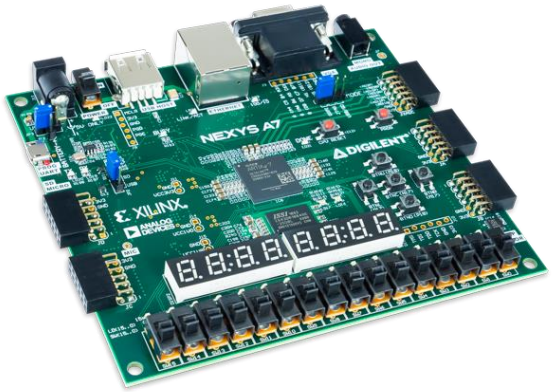

Fig. 2 – Nexys A7 FPGA Board

The result of a successful MIG implementation is the freeing up of slices on the fabric, allowing for lower power usage and more optimized resource utilization.

This project did not reach its final goal of successfully implementing the Memory Interface Generator, as Vivado 2021.2 seems to have differing implementation than previous versions for the memory on the Nexys A7 board. Despite this setback, the MIG has been instantiated and the Verilog code within the MIPS pipeline processor has been set up to accept the full implementation of the MIG once the issue is resolved.

Ideas for overcoming these setbacks are also stated at the end of this report.

II. Introduction

The textbook Computer Principles and Design in Verilog HDL, referred to as the Li textbook (after its author, Yamin Li), provides several example Verilog projects that illustrate different types of processors implemented in HDL. [1] One of them, the Pipeline-With-Cache-FPU-VM, implements cache memory to complement its instruction pipelining, but the cache memory implemented is implemented directly on the FPGA fabric and are more akin to registers than true memory. This is of course resource intensive, and the Instruction and Data caches are good candidates for being moved to the dedicated DDR2 memory on the Nexys A7 boards. These boards have 128MB of DDR2 memory on hardware and can be implemented if we call the synthesizer to utilize them [2].

III. Background

The following gives an overview of several computer architecture topics which are referenced in this paper, and of course directly relevant to this project's execution.

A. Pipelined Processors

A pipelined processor is a type of processor that, unlike standard single-cycle processors, can queue up several instructions and have one instruction begin executing while the previous instruction has not yet finished. This is accomplished by splitting a processor's actions into several stages, which are typically Instruction Fetch (IF), Instruction Decode (ID), Execute (EX), Memory Access (MEM), and Writeback (WB).

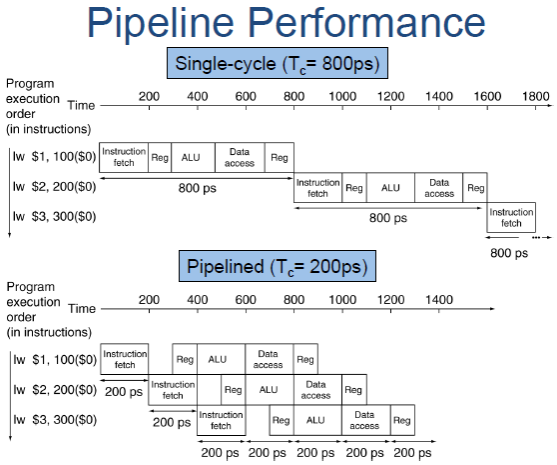

Fig. 3 – Example of Speed-Up Obtained with Pipelining [3]

These stages can each execute independent of each other, provided one stage does not require the results of a previous stage to execute. When such cases occur, the processor will stall – these are known as hazards, and pipelined processors are often designed with these in mind to limit stalls as much as possible. There are typically registers in between stages to hold data between cycles and enable the pipelining to function as needed.

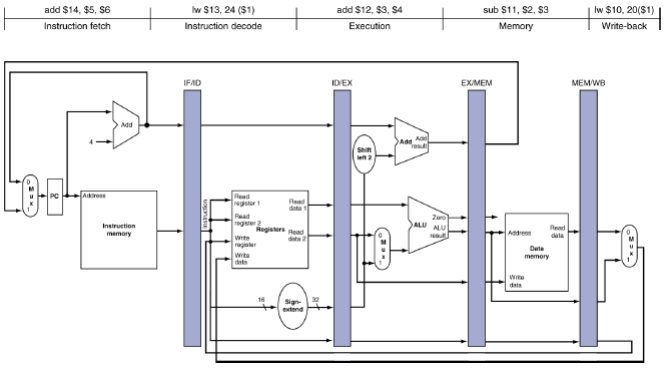

Fig. 4 – Single-Cycle Pipeline Diagram [3]

Regarding the pipeline stages, Execute and Memory Access typically take the longest amount of time, since the execution stage typically involves the ALU (Arithmetic Logic Unit) and memory access by itself is a slower operation. In attempts to speed up modern processors, these two slowest stages are often tackled, and in the latter's case cache memory and TLBs are implemented to try and reduce the number of times a processor must access the full memory.

B. Cache Memory

Cache memory is a faster-access memory system that stores frequently used instruction and data values for faster access. Since a pipelined processor is only as fast as its slowest stage, and memory access tends to be one the slowest, cache memory is an essential part of any processor. Because cache memory is typically smaller and closer to the processor, access speed is by nature faster. It is also typically a faster memory type such as SRAM, as opposed to DRAM or disk memory. Since these faster memory types are usually expensive in both size and power requirements, it is prudent to organize the frequently used data onto the faster cache memory when possible and store the rest of the memory on slower and more efficient memory systems.

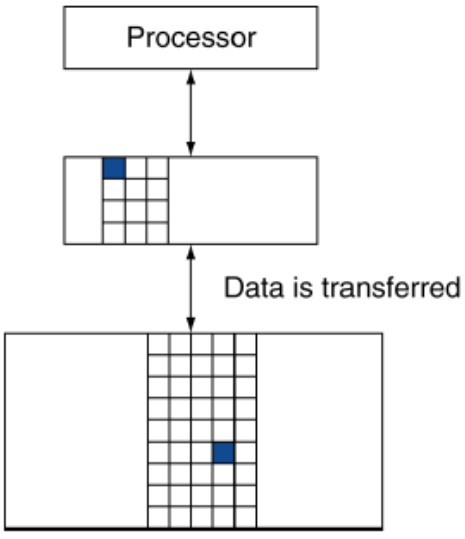

Fig. 5 – Memory Hierarchy Levels [4]

Typically, a processor only needs to access a small amount of memory at a given time, so the typical processor workflow lends itself well to cache implementation. There are several ways a cache can decide which data to save and which data to overwrite – it can choose direct mapping (which simply maps the data directly to a cache address), to replace the last data word used, the oldest word taken into the cache, a random value, or something else entirely.

Related to the memory replacement function is the address mapping – how does the processor find the required data value it is looking for? Typically, that refers to how the cache is mapped – direct mapping (as mentioned above) simply maps to the address value of the cache based on a subset of the address value of the data accessed.

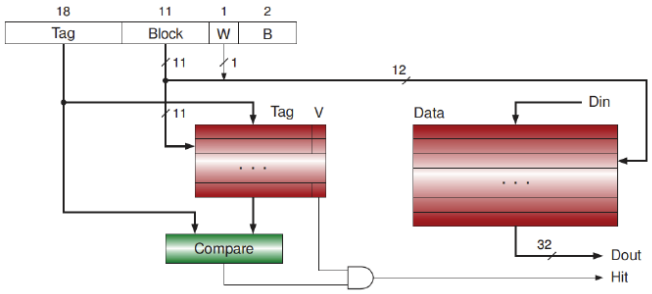

Fig. 6 – Direct Mapping Block Diagram [4]

Another type of mapping is associative mapping, which assigns a tag value (bits) as a pseudo-address to help the processor locate the data value. The cache will not have a full-sized address like the full memory will, and so we map to the smaller address size in various ways. There are different types (fully associative versus n-way set associative), with the former allowing a block to go into any cache entry but the n-way sets use block number to determine which set to view. The n-way set associative mapping requires less tag bits to be read and can be easier on space during implementation, but of course fully associative will be fastest. The tradeoff decision is the responsibility of the design engineer.

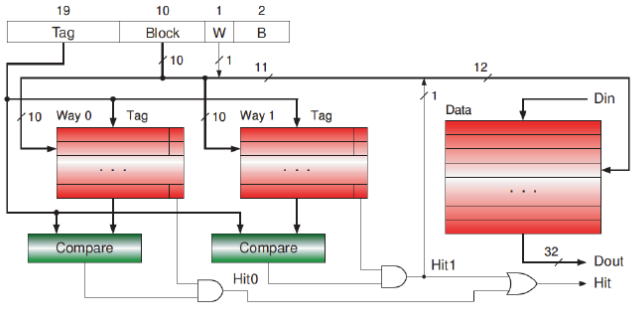

Fig. 7 – Set-Associative Mapping Cache Block Diagram [4]

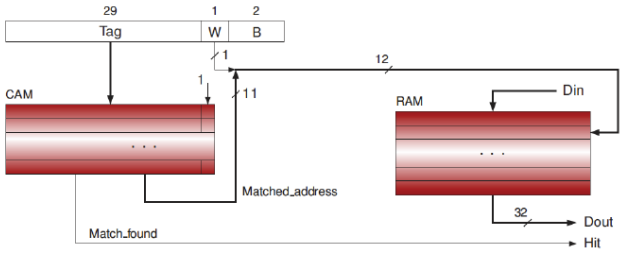

Fig. 8 – Fully-Associative Mapping Cache Block Diagram [4]

C. *Translation Look-Aside Buffers (TLB)*

Translation Look-Aside Buffers can be considered a type of cache, but they are more of a complementary feature since they typically reside between the processor and the cache, or the cache and the primary storage memory. They use page-table entries to match virtual addresses to physical addresses (or intermediary), improving performance by speeding up access time. They can eliminate the slowdowns of a slower cache mapping scheme by speeding up the overall addressing action.

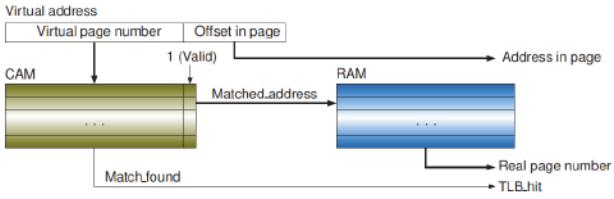

Fig. 9 – Block Diagram of TLB Using CAM [5]

Like cache memory systems, they have a hit/miss ratio to design for, with optimal designs minimizing the miss rates to allow for fast performance of the TLB.

D. *Cache and TLB Structure*

As mentioned previously, cache memory and TLBs are complementary. In the Li pipelined processor this project works with, the processor uses fully associative mapping, with eight entries in each of the TLBs. The two caches for instruction and data are direct mapped, and the cache tag uses the most significant bits of the translated physical address to locate the data.

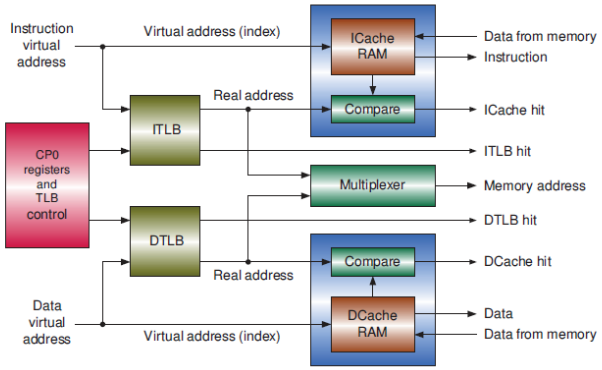

Fig. 10 – Block Diagram of TLB and Cache Layout in Li Processor [1]

E. *The Li Processor: Pipeline-With-Cache-FPU-VM*

In addition to the previously mentioned TLB and cache implementation, the Li processor is set up to allow the processor to fetch instruction data and access memory data simultaneously, but currently only one memory module exists for storing both instruction and data cache values. A demultiplexer with prioritization, stalls, and interrupts is the answer, and is already implemented in the processor. This architecture lends it well to implementing two blocks of DDR2 memory for instruction and data caches.

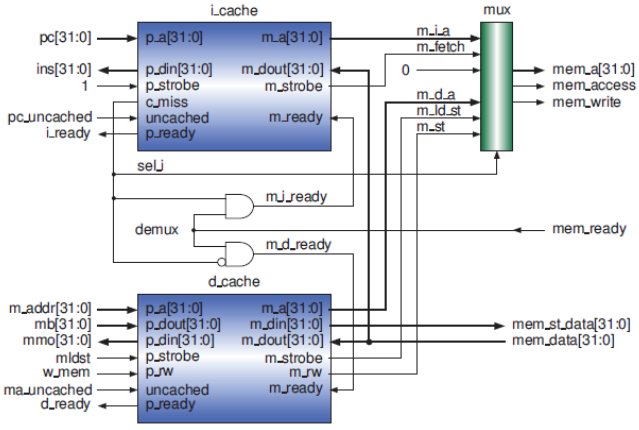

Fig. 11 – Block Diagram of i_cache and d_cache [1]

Finally, the processor already has framework for misses and stalls as necessary with the TLB framework. It is an ideal skeleton for the implementation of a true cache memory using the MIG.

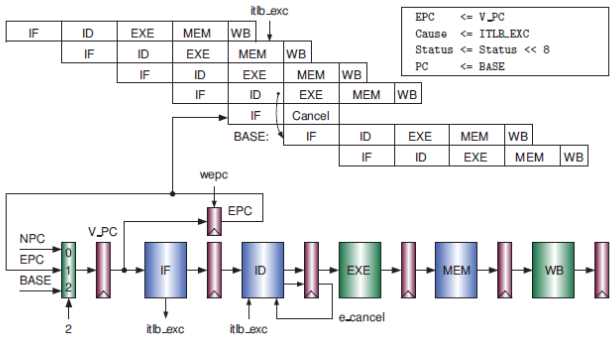

Fig. 12 – Block Diagram of TLB Hit/Miss [1]

F. *Current Memory Implementation on the Li Processor*

For the Li processor, the memory needed is coded in a physical_memory.v file, which instantiates four uram.v files. Each ram implementation is simply partitioning out a set of FPGA registers to access like you would a traditional RAM, and this synthesizes onto the board onto the fabric rather than the true memory.

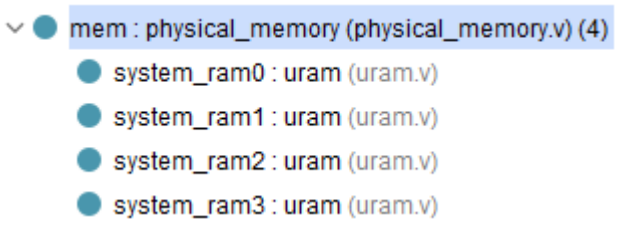

Fig. 13 – Memory Hierarchy in Vivado

```
//Universal RAM
module uram(input clk,
            input we,
            input cs,
            input[31:0] addr,
            input[31:0] data_in,
            output reg[31:0] data_out);

parameter A_WIDTH = 0; //Address Bit width (gives 2^A_WIDTH 32 bit words)
parameter INIT_FILE = "";
parameter READ_DELAY = 0;

reg[31:0] ram[0:(1<<A_WIDTH)-1]; //Nx32 Instruction RAM
wire [31:0] addrmasked;
assign addrmasked = addr & 2**(A_WIDTH+2)-1;
always @(posedge clk) begin
    if(we & cs) begin
        //Ram accesses in this system are word-oriented
        ram[addrmasked>>2] <= data_in;
    end
end

generate
    if(READ_DELAY == 0) begin
        //Asynchronous read (in the same clock cycle)
        always @(*) data_out = cs ? ram[addrmasked>>2] : 0;
    end else if(READ_DELAY == 1) begin
        //Registered read (ready by next clock cycle)
        always @(posedge clk) begin
            if(cs) data_out <= ram[addrmasked>>2];
            else data_out <= 0;
        end
    end
endgenerate
```

Fig. 14 – uram.v

IV. Approach

The approach is fairly straightforward – initialize an instantiation of the MIG for DDR2 memory and implement it into the physical_memory.v code, while providing the control and data/address signal connections necessary for it to work.

A. Clocking Wizard Configuration

The MIG requires a faster reference clock, provided by the already-instantiated clock generator, which is routed to the MIG's instantiation. Any other coding modifications needed to get the data into a form the MIG can use, such as masking off unused bits, is done in the code as needed.

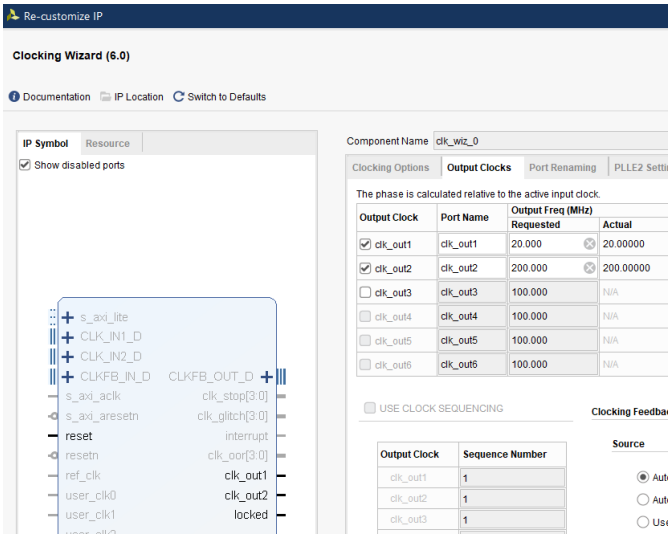

Fig. 15 – Clocking Wizard IP Modification

```
clk_wiz_0 clk_wiz_0(.clk_in1(CLK100MHZ), .clk_out1(clk_out), .clk_out2(clk_out2));
```

Fig. 16 – Adding clk_out2 port to clk_wiz_0

```
cpu_cache_tlb_memory compsys (.SI_ClkIn(clk_out),
                              .SI_mem_clk(clk_out2),
                              .SI_Reset_N(CPU_RESETN),
```

Fig. 17 – Adding clk_out2 to compsys Instance

B. Memory Interface Generator (MIG) Setup

Next, the MIG itself was initialized from the IP catalog and configured. This section walks through the configuration of the MIG for this project.

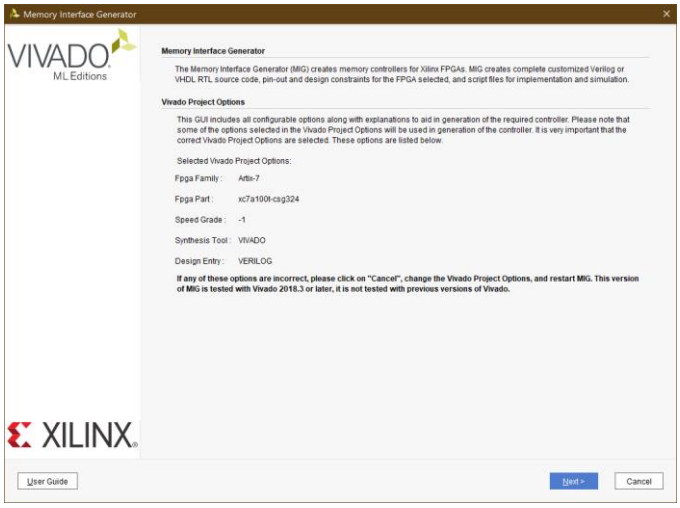

Fig. 18 – MIG Page 1

The initialization window displays an opening summary statement as well as basic information about the device it is generating the IP for.

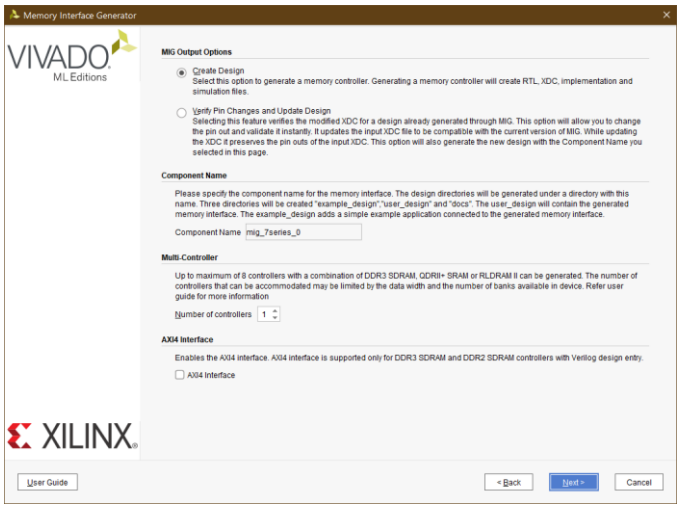

Fig. 19 – MIG Page 2

The next page, Figure 19, shows the creation a new design with a unique name for instantiation, with one controller (which we attach to the MIPS control and data signals as needed).

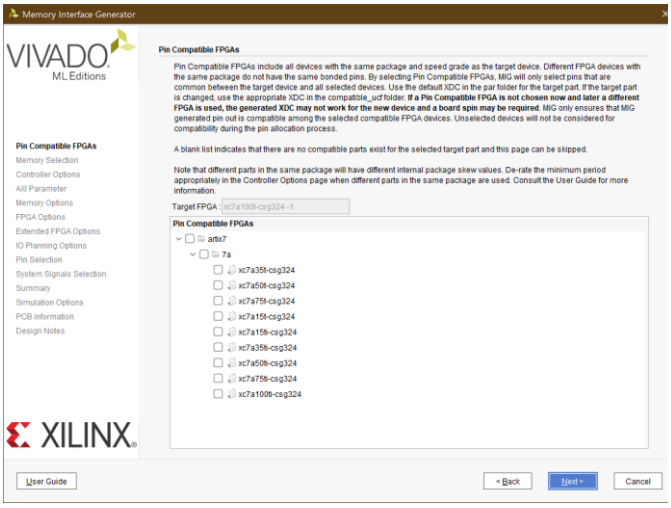

Fig. 20 – MIG Page 3

Additional processor types are left de-selected, as this is only being implemented on the Nexys A7 board.

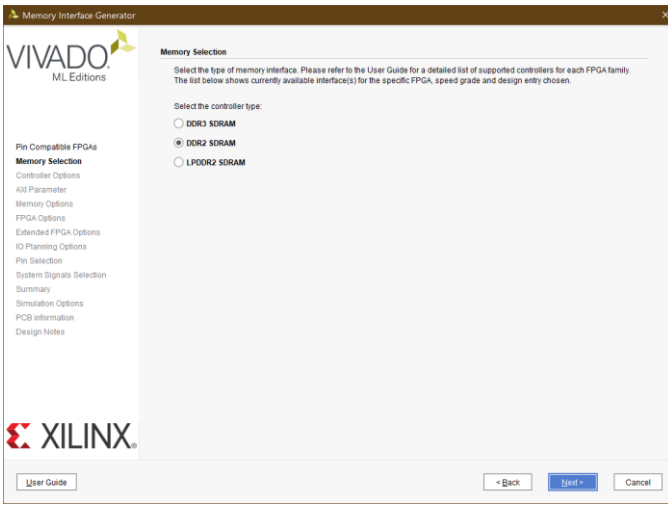

Fig. 21 – MIG Page 4

DDR2 is selected, as this is the memory that is in the Nexys A7 hardware available to use.

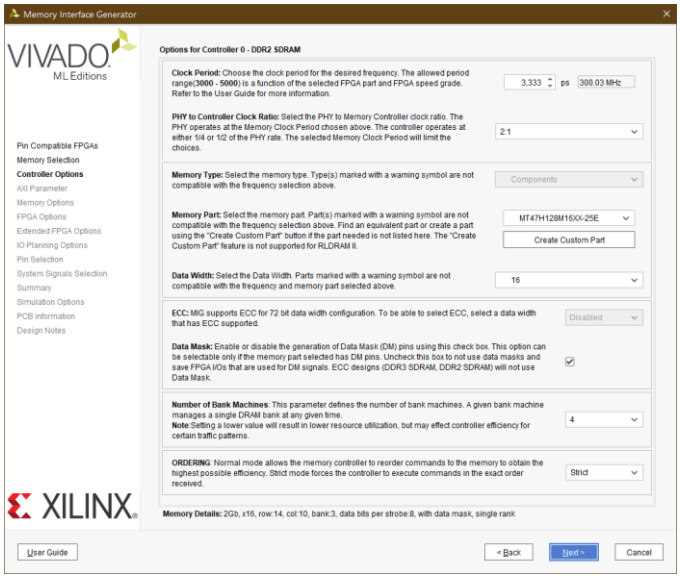

Fig. 22 – MIG Page 5

A clocking period of 3,333 ps is selected – this is the value chosen to allow for the 200MHz clock reference to be selectable for the MIG. A 2:1 PHY rate is also desirable for this implementation, as is the data width of 16 bits. This is not to be confused with the actual width of data going into the MIG (which can be confusing in this interface), which is 64 bits, and we mask off the unused bits as needed. Strict ordering is to ensure the MIG instantiation does not try to re-order instructions, as all ordering will be the sole responsibility of the MIPS pipeline processor.

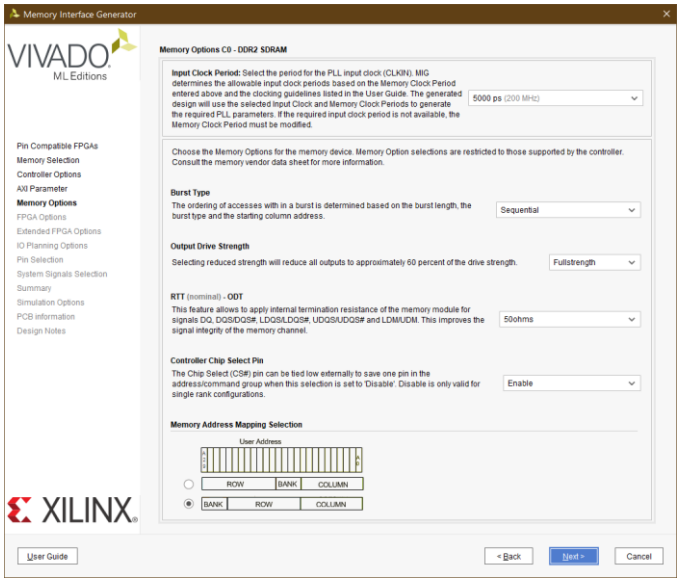

Fig. 23 – MIG Page 6

Since the proper internal clocking period was selected, an input clock period of 200 MHz is selectable. The 50-ohm RTT is a standard value choice, as is keeping the bank to the MSBs.

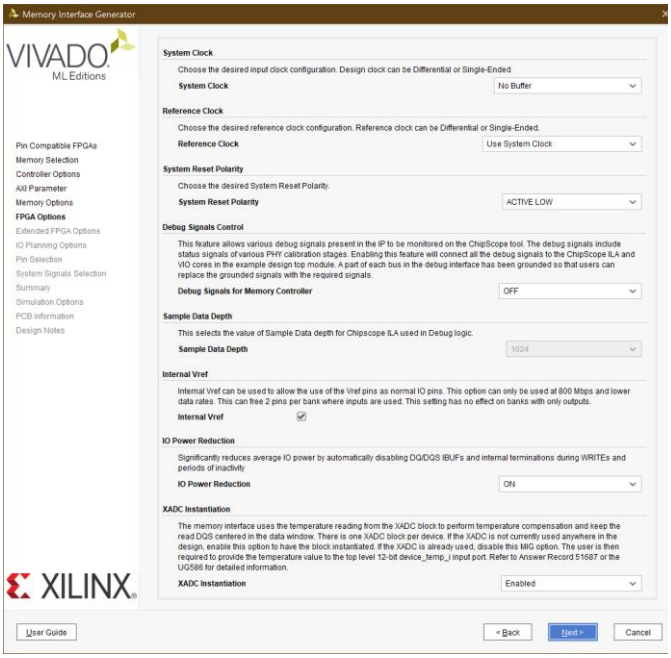

Fig. 24 – MIG Page 7

The system clock is directly pulled from the input clock, as it is all timed to 200 MHz, so No Buffer is selected. For the reference clock, the system clock will suffice and is preferred. Active low reset is a standard used throughout the MIPS, and the internal Vref being selected allows for the Vref pins to be used as normal I/O pins. This is necessary for I/O pin selection.

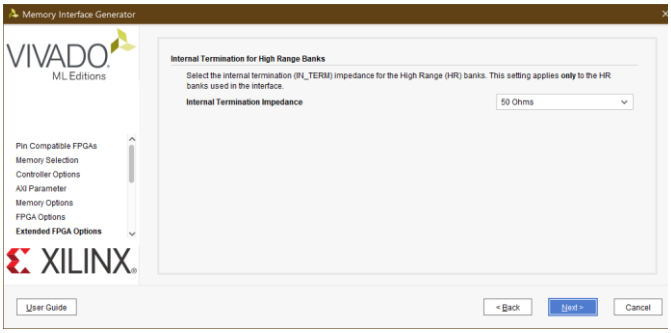

Fig. 25 – MIG Page 8

A standard 50-ohm value is the default and is selected.

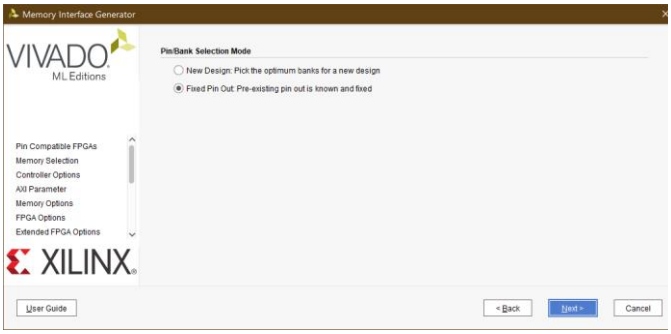

Fig. 26 – MIG Page 9

Selecting fixed pin-out allows us to pull in an existing pin configuration – this is in the form of a UCF file given by the Xilinx website under the Nexys A7 file system.

Fig. 27 – MIG Page 10

Here the .ucf file is loaded in, and the pins are validated. The first snag appears here – the IP demands that ddr2_addr[13] is set, which is not standard, so the UCF file had to be modified to accommodate. Once all required pins are assigned appropriately, the IP will validate, and configuration can continue.

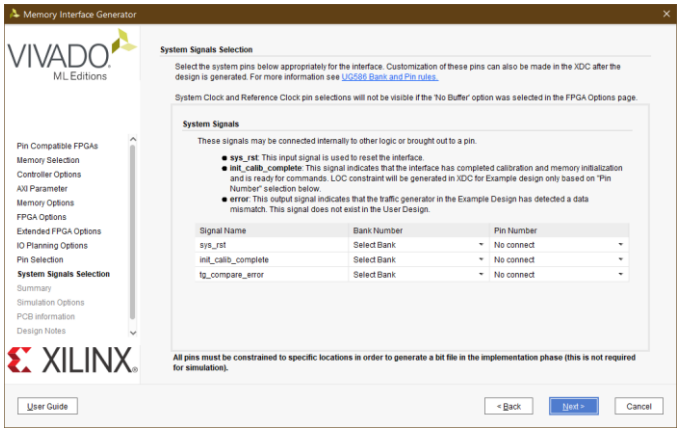

Fig. 28 – MIG Page 11

The signals on this page are left disconnected, as the MIPS processor does not use them.

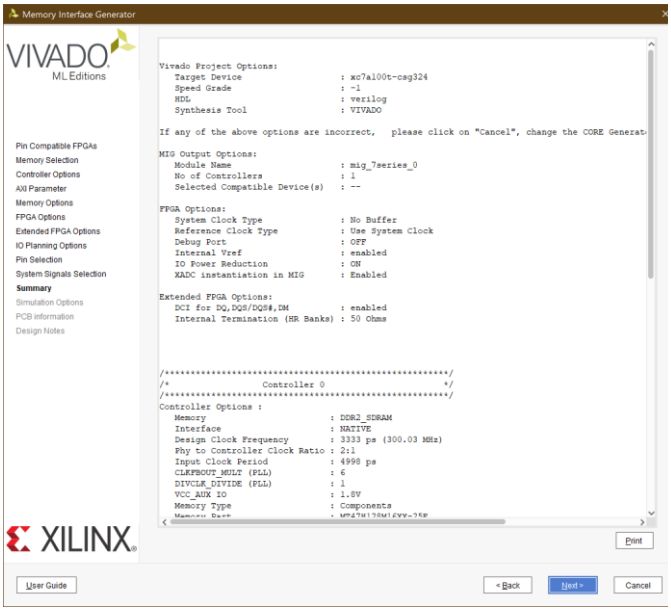

Fig. 29 – MIG Page 12

Finally, after configuring everything satisfactorily, the IP interface will generate a summary. A couple more windows to click through and the IP is generated for use in the MIPS processor code.

C. *Verilog Code Modification*

As mentioned previously, several Verilog code modifications are necessary to properly instantiate the MIG and have it working within the MIPS processor.

```
// 31 30 29 28 ... 15 14 13 12 ...  3  2  1  0
//  0  0  0  0      0  0  0  0      0  0  0  0   (0) 0x0000_0000
//  0  0  0  1      0  0  0  0      0  0  0  0   (1) 0x1000_0000
//  0  0  1  0      0  0  0  0      0  0  0  0   (2) 0x2000_0000 - instruction cache
//  0  0  1  0      0  0  1  0      0  0  0  0   (3) 0x2000_2000 - data cache

wire   [31:0] m_out32 = a[13] ? mem_data_out3 : mem_data_out2;  // instruction cache is 2, data cache is 3
wire   [31:0] m_out10 = a[28] ? mem_data_out1 : mem_data_out0;
wire   [31:0] mem_out = a[29] ? m_out32       : m_out10;
wire   [31:0] dout    = ready ? mem_out       : 32'hzzzz_zzzz;
```

Fig. 30 – Addressing Scheme Prior to MIG Implementation

The Li processor uses bits 13, 28, and 29 from the address to choose which memory cache to access, all of which are uram.v instantiations. For this project, the instruction and data cache are being replaced with the MIG instantiation, so bit 29 will be deciding whether i_cache or d_cache are read.

```
// (0) 0x0000_0000- (virtual address 0x8000_0000-)
 wire          write_enable0 = ~a[29] & ~a[28] & rw;
uram #(.A_WIDTH(12), .INIT_FILE(RAM_FILE0), .READ_DELAY(0)) system_ram0
     (.clk(memclk), .we(write_enable0), .cs(strobe), .addr(a), .data_in(din), .data_out(mem_data_out0));
 wire          write_enable1 = ~a[29] &  a[28] & rw;
uram #(.A_WIDTH(12), .INIT_FILE(RAM_FILE1), .READ_DELAY(0)) system_ram1
     (.clk(memclk), .we(write_enable1), .cs(strobe), .addr(a), .data_in(din), .data_out(mem_data_out1));
 wire          write_enable2 = a[29] & ~a[13] & rw;
uram #(.A_WIDTH(12), .INIT_FILE(RAM_FILE2), .READ_DELAY(0)) system_ram2
     (.clk(memclk), .we(write_enable2), .cs(strobe), .addr(a), .data_in(din), .data_out(mem_data_out2));
 wire          write_enable3 = a[29] & a[13] & rw;
uram #(.A_WIDTH(12), .INIT_FILE(RAM_FILE3), .READ_DELAY(0)) system_ram3
     (.clk(memclk), .we(write_enable3), .cs(strobe), .addr(a), .data_in(din), .data_out(mem_data_out3));
```

Fig. 31 – uram.v instantiation

The bottom two uram.v instantiations are i_cache and d_cache, which are commented out, and the MIG is instantiated instead.

```
59    wire   [31:0] mem_out = a[29] ? m_out32       : m_out10;       // m_out32 (from i and d cache)
60    wire   [31:0] dout    = ready ? mem_out       : 32'hzzzz_zzzz;
61
62    wire   write_enable_ddr2 = a[29] & rw;  // dont care about a[13] anymore for ddr2
63
64    wire   [27:0]ddr_addr;
65    assign ddr_addr = a[27:0];             // taking 27 LSBs from addr input
66
67    wire   [31:0]m_out32;
68    wire   [63:0]app_rd_data;
69    assign m_out32 = app_rd_data[31:0];    // masking data from MIG DDR2 so only LSB 32 bits are taken
70
71    wire [63:0]ddr_in;
72    assign ddr_in = {32'b0, din};          // takes data into LSB 32 bits
73
74    // DDR2 memory for instruction and data cache
75    // ** todo *** initialize memory files
76    mig_7series_0 mig_7series_0(.sys_clk_i(memclk),
77                                .sys_rst(clrn),
78                                .app_rd_data(app_rd_data),          // data width is 64
79                                .app_wdf_data(ddr_in),
80                                .app_addr(ddr_addr),                // addr width 28
81                                .app_en(strobe),                    // cs
82                                .app_wdf_wren(write_enable_ddr2));  // we
83
84    // (0) 0x0000_0000- (virtual address 0x8000_0000-)
85     wire          write_enable0 = ~a[29] & ~a[28] & rw;
86     uram #(.A_WIDTH(12), .INIT_FILE(RAM_FILE0), .READ_DELAY(0)) system_ram0
87          (.clk(memclk), .we(write_enable0), .cs(strobe), .addr(a), .data_in(din), .data_out(mem_data_out0));
88     wire          write_enable1 = ~a[29] &  a[28] & rw;
89     uram #(.A_WIDTH(12), .INIT_FILE(RAM_FILE1), .READ_DELAY(0)) system_ram1
90          (.clk(memclk), .we(write_enable1), .cs(strobe), .addr(a), .data_in(din), .data_out(mem_data_out1));
91     //wire          write_enable2 = a[29] & ~a[13] & rw;
92     //uram #(.A_WIDTH(12), .INIT_FILE(RAM_FILE2), .READ_DELAY(0)) system_ram2
93     //     (.clk(memclk), .we(write_enable2), .cs(strobe), .addr(a), .data_in(din), .data_out(mem_data_out2));
```

Fig. 32 – Modified Verilog Code for MIPS Instantiation

The MIG is instantiated here, with the ddr_in and app_rd_data set up to mask the unused bits and only utilize the 32 bit words that the processor requires. The wire MUX logic is rewritten to accommodate the MIG as opposed to the commented-out uram.v instantiations. All code modifications and their comments are shown in Figure 32 above.

V. Results

The results of this should he a functioning cache memory, or at least a cache memory that can be debugged and modified as needed, but the project refused to synthesize because of the pin selections for ddr2_addr[13]. Trying any available pin leads to various errors, either in the IP generation itself, or when synthesis is attempted. Cursory searches on the Xilinx website turn up several complaints that version updates in Vivado have broken the MIG for certain applications, specifically the DDR4 (which is essentially the A7, just an older label). This would not be the first time that key functionality was broken by a version update, as Vivado is a very complicated IDE.

VI. Future Work

The MIG instantiation issue is something that could very much be fixed with a path, workaround, or future update. Due to the convenient architecture of the Li processor, it should work with very little configuration from this point, provided that Vivado allows it to synthesize.

Other possible future projects could be using different hardware – the A7 could be limited by availability of pins, something a larger board may not have issue with. The available of other types of memory, such as DDR3, open up additional possibilities for implementation of cache memory on this processor. Any solution that is developed would of course be hardware specific.

VII. Conclusion

The project was an excellent learning tool, and the framework for interfacing between the Xilinx MIG and the Li processor was laid out in a complete and satisfactory fashion. In addition to being a solid learning tool for computer architecture, it is my hope that I will be able to find a

workaround and finalize this effort into a working MIPS processor as was originally intended. More troubleshooting will be necessary to determine why the MIG does not accept any available pin configurations for ddr2_addr[13], but should a workaround be found, the MIG implementation should not require much more effort to implement on the back of this project.

VIII. Bibliography

[1]Y. Li, *Computer Principles and Design in Verilog HDL*. New York: John Wiley & Sons, Inc., 2015.

[2]A. Brown, "Nexys A7 Reference Manual - Digilent Reference", *Digilent.com*, 2022. [Online]. Available: https://digilent.com/reference/programmable-logic/nexys-a7/reference-manual. [Accessed: 09- May- 2022].

[3]N. Beser, "The Processor (CPU and Control Unit) Pipelined Processor", Johns Hopkins Whiting School of Engineering, 2022.

[4]N. Beser, "Memory Subsystems - Cache", Johns Hopkins Whiting School of Engineering, 2022.

[5]N. Beser, "Memory Subsystems – Virtual Memory", Johns Hopkins Whiting School of Engineering, 2022.