Spring 2024

EE 382N-4: Advanced Embedded Systems

Lab Assignment #1

DUE FEB 9TH, 2024

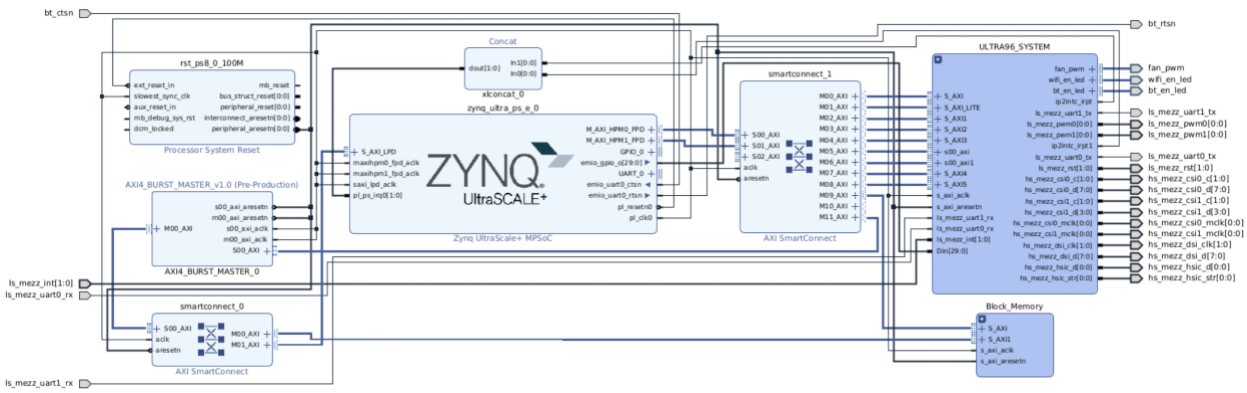
Lab Goals:

This lab introduces the student to the Xilinx Vivado Design Environment and programming in the Linux environment. The student will build an FPGA based memory tester using an AXI-4 Bus Master to test a dual-ported BRAM (located in the Programmable Logic (PL)) and the OCM (On-chip Memory) located in the processor system (PS). The memory test will be performed while dynamically changing the PS clock speed and the PL clock speed.

Setting up the LAB #1 environment:

Refer to the following document to set up your Vivado environment. This document will help the student set up a workspace on the **`/misc/scratch`** partition. This partition has enough disk space for all three labs. There will be an additional partition for the team projects.

You will notice that there are two bus masters in this baseline design. The ZYNQ_ULTRA is the Processing System (PS) and the AXI4_BURST_MASTER is in the Programmable Logic (PL).



The AXI4_BURST_MASTER will be initially used to run memory tests in the PS and PL. Ultimately it will be used in the class project to transfer data to-and-from the PS (OCM) memory. The AXI4_BURST_MASTER contains a rudimentary memory tester that uses fixed addresses and incrementing data. The AXI4_BURST_MASTER also contains an AXI slave port that will used to initialize and execute the memory test.

As mentioned above, the rudimentary memory tester will be replaced with a fully programmable memory tester. This requires removing and rewriting the Verilog code in AXI4_BURST_MASTER. In the next section we will learn how to run the built-in memory tester.

Running the built-in memory tester:

To run the tester, you will need to check out an ULTRA-96 board. It will be preconfigured with the correct DTB and FPGA BIT file. Login to the board using the info you received when you picked up the board. There are a few debugging commands that you need use to run the built-in memory tester. You need to be ROOT to run these commands.

```
/usr/bin/pm: This command writes to a memory location
     Put Memory - USAGE:  pm (Address) (write data) (optional repeat #)
/usr/bin/dm: This command displays a memory location(s)
     Display Memory - USAGE:  dm (address) (repeat #)
/usr/bin/fm: This command fills a range of memory
     Fill Memory - USAGE: fm (address) (write data) (#addresses) (data increment)
/usr/bin/frm: This command fills a range of memory with random data
     Fill Memory Random Data - USAGE:  frm (address) (# addresses)
```

Three of these commands can "brick" your system in a femto-second. Be very careful… Execute the following commands (**in green**) to verify that the built-in memory tester is working correctly. The results are shown in **purple**

```
frm 0xfffc0000 1024   // Fills the OCM (Address: 0xfffc0000) with random data
```

The last four lines of the output from the command will look like something like this:

```
0xfffc0ff0 = 0x5b6eee61
0xfffc0ff4 = 0x6c89a171
0xfffc0ff8 = 0x468696a8
0xfffc0ffc = 0x5b5353a2
root@ultra96:~#
```

Remember that the data is random so you will see different numbers every time you run the **frm** command.

```
dm 0xfffc0000 1024    //  Confirm that the last four locations are identical to four
                      //  listed above

0xfffc0ff0 = 0x5b6eee61
0xfffc0ff4 = 0x6c89a171
0xfffc0ff8 = 0x468696a8
0xfffc0ffc = 0x5b5353a2
root@ultra96:~#

dm 0xa0000000 8           // List out the 8 registers in the AXI slave port

0xa0000000 = 0x00000000   // This register starts the test
0xa0000004 = 0x00000000   // This register shows the status of the test
0xa0000008 = 0x00000000   // This register is not used
0xa000000c = 0xfffc0000   // This register is used for selecting memory address
0xa0000010 = 0xdeadfeed   // This register is a canary in the coal mine
0xa0000014 = 0x00000000   // This register is not used
0xa0000018 = 0x00000000   // This register is not used
0xa000001c = 0xfeedbeef   // This register is a canary also
root@ultra96:~#
```

To start the memory test, write a “1” to address 0xa0000000

```
pm 0xa0000000 0x1          // Start the test. Note: generates a single pulse
                           // to the state machine

dm 0xfffc0000 1024         // Display the last four words in the OCM
```

The randomized data has now been over-written by the tester. You will notice that the resulting data is simply incremented by 1 (which is not a useful test).

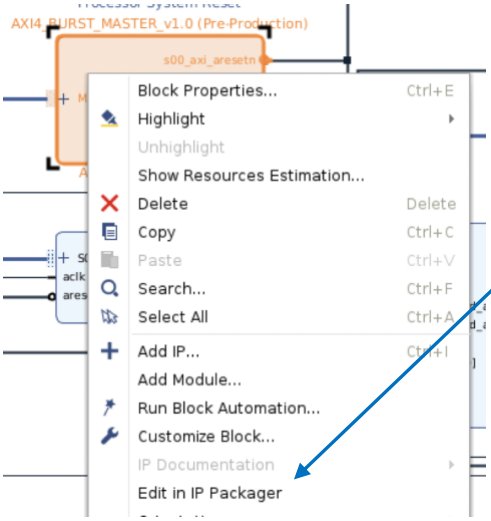
```
0xfffc0ff0 = 0x000003fd
0xfffc0ff4 = 0x000003fe
0xfffc0ff8 = 0x000003ff
0xfffc0ffc = 0x00000400
root@ultra96:~#
```

To confirmed that the test passed, read status register.

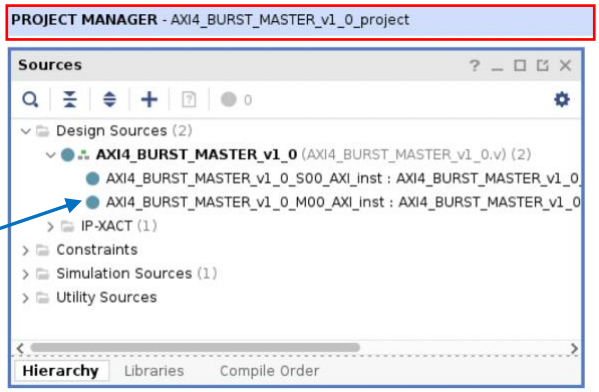
```
dm 0xa0000004              // Status register

0xa0000004 = 0x00000001    // The test passed
root@ultra96:~#
```

Verilog code used in the built-in tester:

The Verilog code for the AXI4_BURST_MASTER is edited in the IP packager.



Note that a new Project Manager window is opened, and the design sources can be edited double clicking on the file name.



Double click the AXI4_BURST_MASTER_v1_0_M00_AXI.v file

This file contains all the AXI4 Burst Master control logic.

The state machine parameters that controls the rudimentary memory test is located at line 202

```
202     // Example State machine to initialize counter, initialize write transactions,
203     // initialize read transactions and comparison of read data with the
204     // written data words.
205
206     parameter [1:0] IDLE = 2'b00; // This state initiates AXI4Lite transaction
207             // after the state machine changes state to INIT_WRITE
208             // when there is 0 to 1 transition on INIT_AXI_TXN
209
210     parameter [1:0] INIT_WRITE   = 2'b01; // This state initializes write transaction,
211             // once writes are done, the state machine
212             // changes state to INIT_READ
213
214     parameter [1:0] INIT_READ = 2'b10; // This state initializes read transaction
215             // once reads are done, the state machine
216             // changes state to INIT_COMPARE
217
218     parameter [1:0] INIT_COMPARE = 2'b11; // This state issues the status of comparison
219             // of the written data with the read data
220
221     reg [1:0] mst_exec_state;
```

The tester state machine is located at line 818:

```
818     // ------------------------------------------------------------------------------------------
819     // -----------  Implement master command interface state machine
820     // ------------------------------------------------------------------------------------------
821
822     always @ ( posedge M_AXI_ACLK)
823     begin
824       if (M_AXI_ARESETN == 1'b0 )
825         begin
826           // reset condition
827           // All the signals are assigned default values under reset condition
828           mst_exec_state      <= IDLE;
829           start_single_burst_write <= 1'b0;
830           start_single_burst_read  <= 1'b0;
831           compare_done      <= 1'b0;
832           ERROR <= 1'b0;
833         end
834       else
835         begin
836
837           // state transition
838           case (mst_exec_state)
839
840             IDLE:
841               // This state is responsible to wait for user defined C_M_START_COUNT
842               // number of clock cycles.
843               if ( init_txn_pulse == 1'b1)
844                 begin
845                   mst_exec_state  <= INIT_WRITE;
846                   ERROR <= 1'b0;
847                   compare_done <= 1'b0;
848                 end
849               else
850                 begin
851                   mst_exec_state  <= IDLE;
852                 end
```

This pulse is generated from the write to address 0xa0000000 bit[0]

```
853
854             INIT_WRITE:
855               // This state is responsible to issue start_single_write pulse to
856               // initiate a write transaction. Write transactions will be
857               // issued until burst_write_active signal is asserted.
858               // write controller
859               if (writes_done)
860                 begin
861                   mst_exec_state <= INIT_READ;//
862                 end
863               else
864                 begin
865                   mst_exec_state  <= INIT_WRITE;
866
867                   if (~axi_awvalid && ~start_single_burst_write && ~burst_write_active)
868                     begin
869                       start_single_burst_write <= 1'b1;
870                     end
871                   else
872                     begin
873                       start_single_burst_write <= 1'b0; //Negate to generate a pulse
874                     end
875                 end
876
877             INIT_READ:
878               // This state is responsible to issue start_single_read pulse to
879               // initiate a read transaction. Read transactions will be
880               // issued until burst_read_active signal is asserted.
881               // read controller
882               if (reads_done)
883                 begin
884                   mst_exec_state <= INIT_COMPARE;
885                 end
886               else
887                 begin
888                   mst_exec_state  <= INIT_READ;
889
890                   if (~axi_arvalid && ~burst_read_active && ~start_single_burst_read)
891                     begin
892                       start_single_burst_read <= 1'b1;
893                     end
894                    else
895                     begin
896                       start_single_burst_read <= 1'b0; //Negate to generate a pulse
897                     end
898                 end
900             INIT_COMPARE:
901               // This state is responsible to issue the state of comparison
902               // of written data with the read data. If no error flags are set,
903               // compare_done signal will be asseted to indicate success.
904               //if (~error_reg)
905               begin
906                 ERROR <= error_reg;
907                 mst_exec_state <= IDLE;
908                 compare_done <= 1'b1;
909               end
910             default :
911               begin
912                 mst_exec_state  <= IDLE;
913               end
914           endcase
915         end
916     end //  MASTER_EXECUTION_PROC
```

The AXI4_BURST_MASTER_v1_0_S00_AXI.v file contains the registers that provide control to the master logic and returns status to the programmer.

Lines 245-257 preset the values of the registers. In this example line 252 sets reg3 to 0xfffc000 which is the address of the OCM memory in the PS. Line 253 is a canary that is used to determine that the FPGA BIT file and DTB (Device Tree Blob) were installed correctly during bootup.

```
245     always @( posedge S_AXI_ACLK )
246     begin
247       if ( S_AXI_ARESETN == 1'b0 )
248         begin
249           slv_reg0 <= 0;                  // Tester start register
250           slv_reg1 <= 0;                  // Test status register
251           slv_reg2 <= 0;                  // Not used
252           slv_reg3 <= 32'hfffc0000;       // Preset to point to the OCM (not functional)
253           slv_reg4 <= 32'hdeadfeed;       // Canary
254           slv_reg5 <= 0;                  // Not used
255           slv_reg6 <= 0;                  // Not used
256           slv_reg7 <= 0;                  // Not used
257         end
```

NOTE: Reg3 is not functional because it is hard coded as a parameter. This will need to be fixed for Lab #1

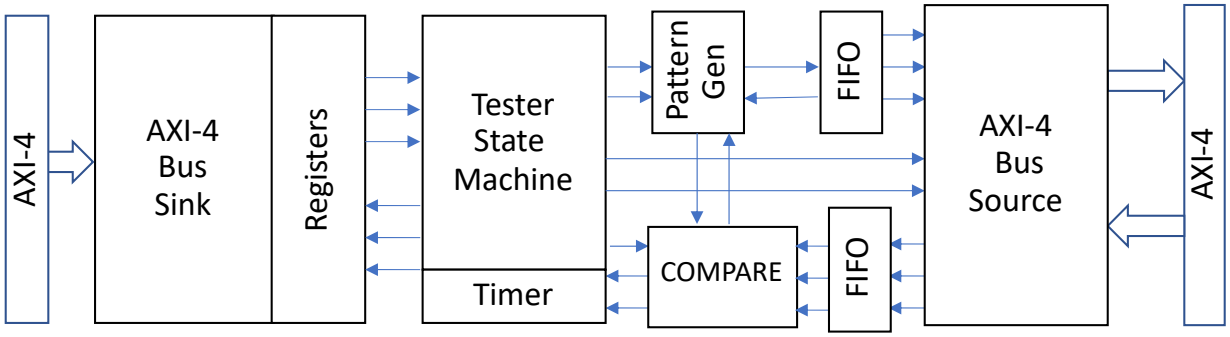
```
11         // Base address of targeted slave
12         parameter  C_M_TARGET_SLAVE_BASE_ADDR   = 32'hFFFC0000,
```

These registers can be set to anything that the user needs. This includes status, debug information, etc. In the example below, reg1 is used to indicate that the test is done, reg2 indicates if an error occurred. Reg7 is another canary to prove that the registers can be set during a read (used for debug)…

```
427     // Implement memory mapped register select and read logic generation
428     // Slave register read enable is asserted when valid address is available
429     // and the slave is ready to accept the read address.
430     assign slv_reg_rden = axi_arready & S_AXI_ARVALID & ~axi_rvalid;
431     always @(*)
432     begin
433           // Address decoding for reading registers
434           case ( axi_araddr[ADDR_LSB+OPT_MEM_ADDR_BITS:ADDR_LSB] )
435             3'h0   : reg_data_out <= slv_reg0;
436             3'h1   : reg_data_out <= {{31{1'b0}},txn_done};
437             3'h2   : reg_data_out <= {{31{1'b0}},txn_error};
438             3'h3   : reg_data_out <= slv_reg3;
439             3'h4   : reg_data_out <= slv_reg4;
440             3'h5   : reg_data_out <= slv_reg5;
441             3'h6   : reg_data_out <= slv_reg6;
442             3'h7   : reg_data_out <= {32'hfeedbeef};       // Another canary
443             default : reg_data_out <= 0;
444           endcase
445     end
```

Tester Design:

The first activity is to replace the rudimentary memory tester with one shown in the block diagram below.



Specification:

1. Develop a fully programmable tester state machine (SM) utilizing control signals from the registers. The SM must be able to utilize the AXI-4 burst mode for both reads and writes. The OCM and BRAM must be testable.

2. Develop a 32-bit PATTERN GEN (PG) unit that can feed the FIFO and the COMPARE unit. The PG must run at the same frequency as the Tester SM and be able to be stalled if the FIFO is full. The PG must be seeded from one the registers. The PG should be able to generate an LFSR pattern, a sliding 1's and a sliding 0's pattern. The pattern should be mux selected using two bits in one of programming registers.

3. Develop a FIFO that has a depth of 8 32-bit entries. The FIFO controller must connect directly to the write channel in AXI-4 Bus Source to minimize latency.

4. Develop a COMPARE Unit to confirm that the memory blocks can be written and read correctly. The compares should be done in burst mode and as fast as the AXI-4 bus can provide data.

5. Develop a TIMER that measures the number of PL clocks it takes to run a test. There will be a timer value for the Burst Writes to the memories and one for the Burst Reads. The timer values will be read when reading the registers. This implies that two registers are needed. A single timer is all that is needed.

Verilog Synthesis:

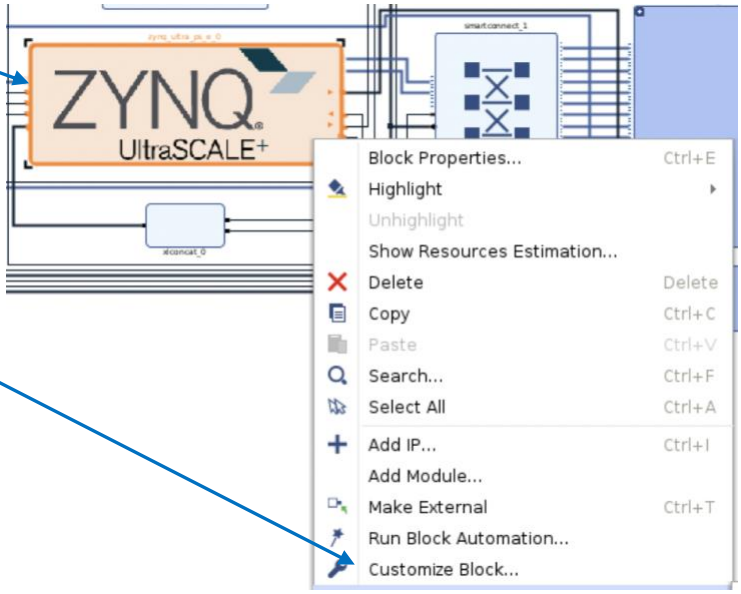
PS Clock MHz	PL Clock MHz
1499	300
1250	266
1000	187.5
858	150
416.6	100

Once the tester design is complete, you will need to run synthesis. Before running synthesis, the PS block will need to be configured for the maximum frequency. The table to the right shows the frequency combinations the tester needs to work at. There are 25 possible combinations to consider. This will be covered in the next section.

The maximum frequency for the PL will be 300 MHz. This is accomplished by setting a configuration value in the PS block as this is where the clock to the PL is generated. The procedure is shown below:

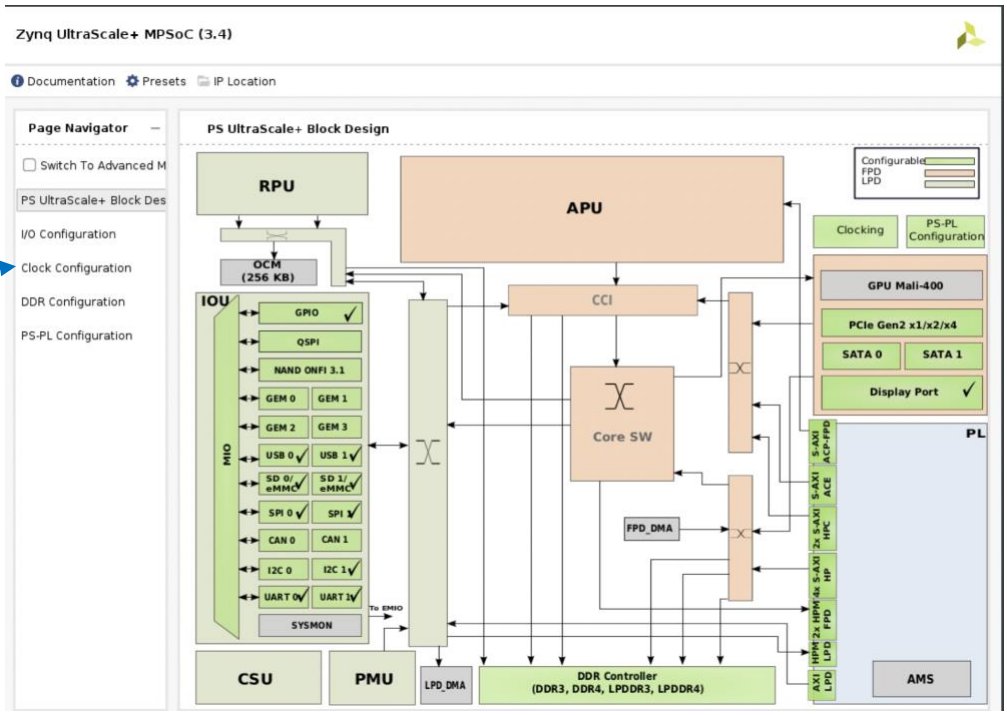
Select the Zynq block in the schematic.

Using the right mouse key to produce a pull-down menu. In the menu select Customize Block.

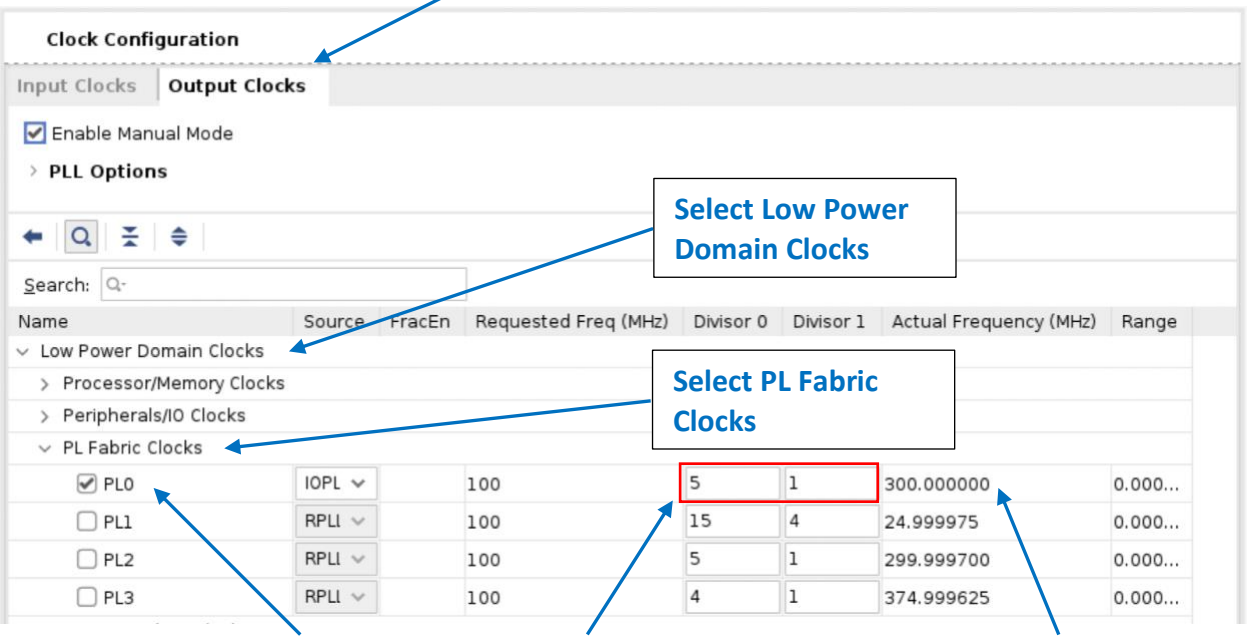


The following window will be displayed:

Select Clock Configuration



The following window will be displayed. Select **Output Clocks**



Change the setting for PL0 to Divisor_0 = 5 and Divisor_1 = 1 This will select a 300MHz clock for the PL.

All subsequent synthesis runs will be done using 300 MHz.

Test Program Development

You will need to use the Linux `srand(time(0))` and `rand()` routines to generate random seed data for the LFSR. For a code example see the `frm` command. NOTE: a new random seed for the LFSR must be generated each time a new test is invoked. In addition to randomly varying the seed data you will need to be randomly varying the clock frequencies of the CPU and PL. The frequencies that you need to use are shown to the right. There are 25 total combinations. The frequency dithering will be controlled in a C program.

PS Clock MHz	PL Clock MHz
1499	300
1250	266
1000	187.5
858	150
416.6	100

All combinations must be programmed for Test #1 and Test #2.

Test #1: Setup a BRAM memory test (@0xC000_0000, 1 page -- 4K total bytes) that uses the LFSR to vary data while randomly varying the CPU and PL clock frequencies as shown in the table aboce. The test should run continuously until interrupted with a control-c.

Test #2: Setup a OCM memory test (@0xFFFC_0000, 1 page -- 4K total bytes) that uses the LFSR to vary data while randomly varying the CPU and PL clock frequencies as shown in the table above. The test should run continuously until interrupted with a control-c.

The setup and control for the memory tests will be written in C and compiled on the Ultra96 using the GNU tool suite. The user input to the test must accept the following inputs:

1. Number of test loops -- Default: continuous
2. Number of 32-bit words to be tested -- Default: 1024 (BRAM), 1024 (OCM)
3. The memory test will display the following output upon successful completion of the test: Test passed: "xx" Loops of "yy" 32-bit words

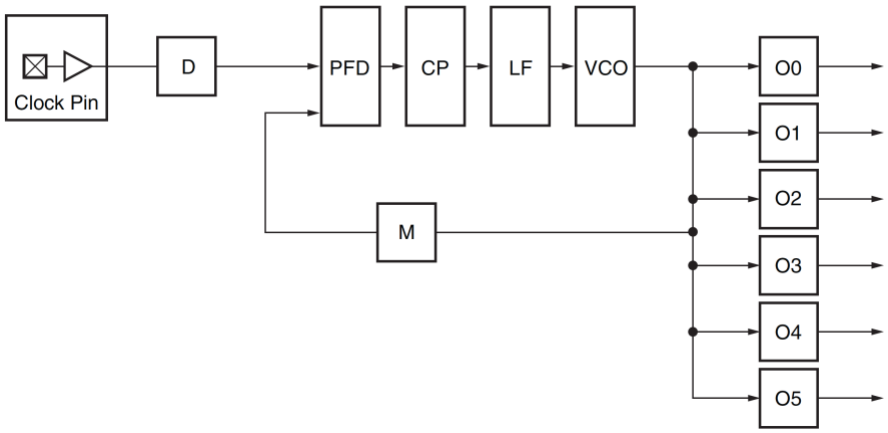
Do NOT use the Vivado SDK development tools. They are for bare-metal implementations. We will not be doing any bare-metal implementations in this class.

Deliverables

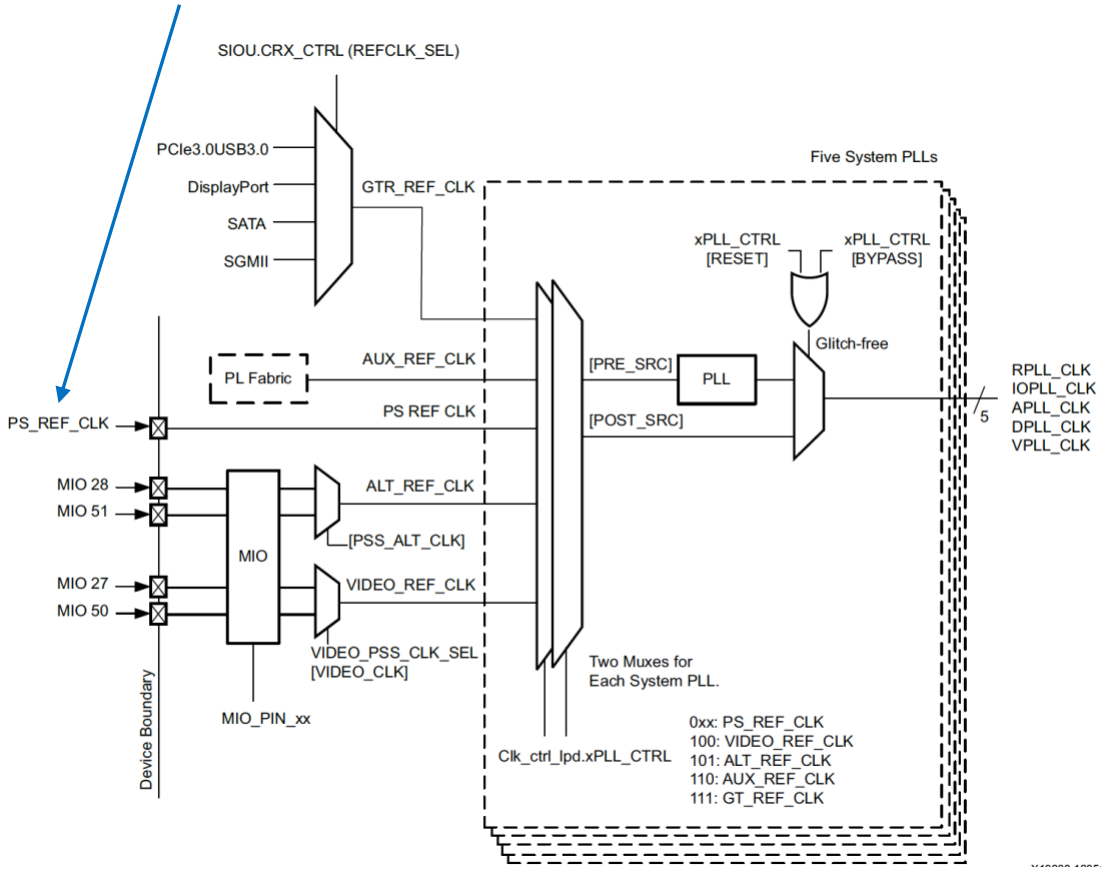
1. Run both tests for a minimum of 5 minutes while dithering the PS clock and PL clock through all 25 combinations. Report all passing and failing tests.
2. Print number of PL clock times (from the TIMER) for all burst writes and reads in CSV format. Display results using Excel.

Background Information

The figure below shows a block diagram of a typical PLL in the ZynqMP SOC. The “M” block is the feedback divider (FBDIV) which divides the output of the VCO and feeds the divided clock into the Phase-Frequency-detector (PFD) providing the frequency multiplication factor of a PLL.



The figure below shows the block diagram of the 5 system PLLs in the ZynqMP. The APLL provides clocks to the Processing System (PS). The IOPLL provides a clock to the Programmable Logic (PL). The Ultra96 PS_REF_CLK is a 33.3333MHz clock.



The following web page specifies all the accessible registers on the ZynqMP SOC:
https://www.xilinx.com/html_docs/registers/ug1087/ug1087-zynq-ultrascale-registers.html

There are three registers that affect the PS clock frequency. An example of how to set the APLL frequency is shown in Appendix A.

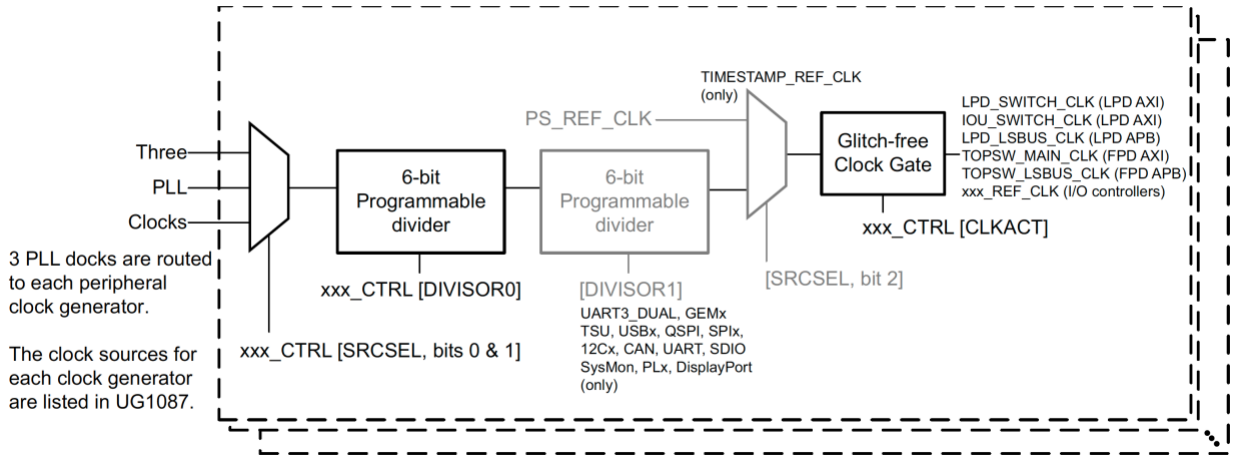
Register Name	APLL_CTRL	This register controls the FBDIV and CLKOUT values of the APLL
Absolute Address	0x00FD1A0020 (CRF_APB)	

Register Name	APLL_CFG	This register controls how quickly the PLL locks
Absolute Address	0x00FD1A0024 (CRF_APB)	

Register Name	PLL_STATUS	This register is the "LOCK" status of the APLL when changing frequencies.
Absolute Address	0x00FD1A0044 (CRF_APB)	

There is one register that affect the PL clock frequency:

Register Name	PL0_REF_CTRL	This register controls the two 6-bit clock dividers (see block diagram below)
Absolute Address	0x00FF5E00C0 (CRL_APB)	



Refer to Appendix B for an example on how to change the PL clock frequency.

Appendix A: PS Integer Multiply and Divide Programming Example

The Ultra96 FBDIV reset value is 0x48 (72) and the Output Divider is set to divide by 2. The PS clock frequency is: 33.3333MHz * 72 / 2 = 1199 MHz

Let's set the PS clock frequency to 1499 MHz. For a new frequency of 1499 MHz, the [FBDIV] value is switched to 45 (0x2D) and the output divider is set to 0x0.

NOTE: Before reprogramming the PLL clock output frequency, check that the downstream clocks are in a safe state before releasing. For example, the APU DIVISOR must be set to 2.

1. Program the new FBDIV, CLKOUT value (do NOT modify other values in the APLL_CTRL register):
   Set APLL_CTRL = 0x0000_2D00: [DIV2] = 0x0, [FBDIV] = 0X2D
   `pm 0xfd1a0020 0x00002D00`

31	30	29	28	27	26	25	24	23	22	21	20	19	18	17	16
--	--	--	--	--	POST SRC			--	PRE SRC			--	--	--	DIV2
0	0	0	0	0	0	0	0	0	0	0	0	0	0	0	0

15	14	13	12	11	10	9	8	7	6	5	4	3	2	1	0
--	FBDIV							--	--	--	--	BYPASS	--	--	RST
0	0	1	0	1	1	0	1	0	0	0	0	0	0	0	0

2. Program the control data for APLL_CFG using the data in Table 1.

FBDIV	CP	RES	LFHF	LOCK_DLY	LOCK_CNT
45	3	12	3	63	825

Set APLL_CFG[31:25] = 0x3F // LOCK_DLY
Set APLL_CFG[22:13] = 0x339 // LOCK_CNT
Set APLL_CFG[11:10] = 0x3 // LFHF
Set APLL_CFG[8:5] = 0x3 // CP
Set APLL_CFG[3:0] = 0x12 // RES

31	30	29	28	27	26	25	24	23
LOCK_DLY							--	--
0	1	1	1	1	1	1	0	0

22	21	20	19	18	17	16	15	14	13
LOCK_CNT									
1	1	0	0	1	1	1	0	0	1

12	11	10	9	8	7	6	5	4	3	2	1	0
--	LFHF		--	CP				--	RES			
0	1	1	0	0	0	1	1	0	1	1	0	0

Set APLL_CFG[31:0] = 0x7E67_2C6C
```
pm 0xfd1a0024 0x7E672C6C
```

3. Program the bypass:
   Set APLL_CTRL[31:0] = 0x0000_2D08h: [BYPASS] = 0x1
   ```
   pm 0xfd1a0020 0x00002D08
   ```

4. Assert reset. This is when the new data is captured into the PLL.
   Set APLL_CTRL[31:0] = 0x0000_2D09h: [BYPASS] = 0x1 & [RESET] = 0x1
   ```
   pm 0xfd1a0020 0x00002D09
   ```

5. Deassert reset.
   Set APLL_CTRL[31:0] = 0x0000_2D08h: [BYPASS] = 0x1 [RESET] = 0x0
   ```
   pm 0xfd1a0020 0x00002D08
   ```

6. Check for LOCK. Wait until: PLL_STATUS [APLL_LOCK] = 0x1
   ```
   while (dm 0xfd1a0044 != 0x1) do wait   // Pseudo code does NOT work in the CLI
   ```

7. Deassert bypass.
   Set APLL_CTRL[31:0] = 0x0000_2D00h: [BYPASS] = 0x00
   ```
   pm 0xfd1a0020 0x00002D00
   ```

The PLL output clock is now set to 1499 MHz.

Table 1: PLL Integer Feedback Divider Helper Data Values

FBDIV	CP	RES	LFHF	LOCK_DLY	LOCK_CNT
25	3	10	3	63	1000
26	3	10	3	63	1000
27	4	6	3	63	1000
28	4	6	3	63	1000
29	4	6	3	63	1000
30	4	6	3	63	1000
31	6	1	3	63	1000
32	6	1	3	63	1000
33	4	10	3	63	1000
34	5	6	3	63	1000
35	5	6	3	63	1000
36	5	6	3	63	1000
37	5	6	3	63	1000
38	5	6	3	63	975
39	3	12	3	63	950
40	3	12	3	63	925
41	3	12	3	63	900
42	3	12	3	63	875
43	3	12	3	63	850

44	3	12	3	63	850
45	3	12	3	63	825
46	3	12	3	63	800
47	3	12	3	63	775
48	3	12	3	63	775
49	3	12	3	63	750
50	3	12	3	63	750
51	3	2	3	63	725
52	3	2	3	63	700
53	3	2	3	63	700
54	3	2	3	63	675
55	3	2	3	63	675
56	3	2	3	63	650
57	3	2	3	63	650
58	3	2	3	63	625
59	3	2	3	63	625
60	3	2	3	63	625
61 to 82	3	2	3	63	600
83 to 102	4	2	3	63	600
103	5	2	3	63	600
104	5	2	3	63	600
105	5	2	3	63	600
106	5	2	3	63	600
107 to 125	3	4	3	63	600

Appendix B: PL Clock Control code example

```
#include "stdio.h"
#include "stdlib.h"
#include <sys/mman.h>
#include <sys/types.h>
#include <sys/stat.h>
#include <fcntl.h>
#include <unistd.h>

int main(int argc, char * argv[])
{
  int dh = open("/dev/mem", O_RDWR | O_SYNC);
  if(dh == -1) {
        printf("Must be ROOT to open /dev/mem\n");
  }

uint32_t* clk_reg = mmap(NULL,
                         0x1000,
                         PROT_READ|PROT_WRITE,
                         MAP_SHARED, dh, 0xFF5E0000);
int i = 0;
uint32_t* pl0 = clk_reg;

pl0+=0xC0;                              // PL0_REF_CTRL reg offset 0xC0

*pl0 = (1<<24)                          // bit 24 enables clock
     | (1<<16)                          // bit 23:16 is divisor 1
     | (6<<8);                          // bit 15:0 is clock divisor 0
                                        // frequency = 1.5Ghz/divisor0/divisor1
                                        //           = 1.5Ghz/6=250MHz
  munmap(clk_reg, 0x1000);
  return 0;
}
```

Field Name	Bits	Type	Reset Value	Description
Reserved	31:25	rw	0x0	reserved
CLKACT	24	rw	0x0	Clock active control. 0: disable. Clock stop. 1: enable.
Reserved	23:22	rw	0x0	reserved
DIVISOR1	21:16	rw	0x5	6-bit divider.
Reserved	15:14	rw	0x0	reserved
DIVISOR0	13:8	rw	0x20	6-bit divider.
Reserved	7:3	rw	0x0	reserved
SRCSEL	2:0	rw	0x0	Clock generator input source. 000: IOPLL 010: RPLL 011: DPLL_CLK_TO_LPD

Appendix C: PS Clock Control Registers

APLL_CTRL (CRF_APB) Register Description

Register Name	APLL_CTRL
Relative Address	0x0000000020
Absolute Address	0x00FD1A0020 (CRF_APB)
Width	32
Type	rw
Reset Value	0x00012C09
Description	APLL Clock Unit Control

APLL_CTRL (CRF_APB) Register Bit-Field Summary

Field Name	Bits	Type	Reset Value	Description
POST_SRC	26:24	rw	0x0	Select the pass-thru clock source for PLL Bypass mode. 0xx: PS_REF_CLK 100: VIDEO_REF_CLK 101: ALT_REF_CLK 110: AUX_REF_CLK 111: GT_REF_CLK
PRE_SRC	22:20	rw	0x0	Select the clock source for PLL input. 0xx: PS_REF_CLK 100: VIDEO_REF_CLK 101: ALT_REF_CLK 110: AUX_REF_CLK 111: GT_REF_CLK
DIV2	16	rw	0x1	Enable the divide by 2 function inside of the PLL. 0: no effect. 1: divide clock by 2. Note: this does not change the VCO frequency, just the output frequency.
FBDIV	14:8	rw	0x2C	Feedback divisor integer portion for the PLL.
BYPASS	3	rw	0x1	PLL Clock Bypass Mode. 0: normal PLL mode; the source clock is selected using [PRE_SRC]. 1: bypass the PLL; the source clock is selected using [POST_SRC].
RESET	0	rw	0x1	PLL reset. 0: active. 1: reset. Note: Program the PLL into bypass mode before resetting the PLL.

APLL_CFG (CRF_APB) Register Description

Register Name	APLL_CFG
Relative Address	0x0000000024
Absolute Address	0x00FD1A0024 (CRF_APB)
Width	32
Type	rw
Reset Value	0x00000000
Description	APLL Integer Helper Data Configuration.

APLL_CFG (CRF_APB) Register Bit-Field Summary

Field Name	Bits	Type	Reset Value	Description
LOCK_DLY	31:25	rw	0x0	Lock circuit configuration settings for lock windowsize
LOCK_CNT	22:13	rw	0x0	Lock circuit counter setting
LFHF	11:10	rw	0x0	PLL loop filter high frequency capacitor control
CP	8:5	rw	0x0	PLL charge pump control
RES	3:0	rw	0x0	PLL loop filter resistor control

PLL_STATUS (CRF_APB) Register Description

Register Name	PLL_STATUS
Relative Address	0x0000000044
Absolute Address	0x00FD1A0044 (CRF_APB)
Width	8
Type	mixed
Reset Value	0x00000038
Description	FPD PLL Clocking Status.

PLL_STATUS (CRF_APB) Register Bit-Field Summary

Field Name	Bits	Type	Reset Value	Description
VPLL_STABLE	5	ro	0x1	VPLL stability status. 0: not locked or bypassed. 1: locked or bypassed.
DPLL_STABLE	4	ro	0x1	DPLL stability status. 0: not locked or bypassed. 1: locked or bypassed.
APLL_STABLE	3	ro	0x1	APLL stability status. 0: not locked or bypassed. 1: locked or bypassed.
VPLL_LOCK	2	ro	0x0	VPLL lock status. 0: not locked. 1: locked.
DPLL_LOCK	1	ro	0x0	DPLL lock status. 0: not locked. 1: locked.
APLL_LOCK	0	ro	0x0	APLL lock status. 0: not locked. 1: locked.

Appendix D: PL Clock Control Registers

PL0_REF_CTRL (CRL_APB) Register Description

Register Name	PL0_REF_CTRL
Relative Address	0x00000000C0
Absolute Address	0x00FF5E00C0 (CRL_APB)
Width	32
Type	rw
Reset Value	0x00052000
Description	PL 0 Clock Generator Config.

PL0_REF_CTRL (CRL_APB) Register Bit-Field Summary

Field Name	Bits	Type	Reset Value	Description
Reserved	31:25	rw	0x0	reserved
CLKACT	24	rw	0x0	Clock active control. 0: disable. Clock stop. 1: enable.
Reserved	23:22	rw	0x0	reserved
DIVISOR1	21:16	rw	0x5	6-bit divider.
Reserved	15:14	rw	0x0	reserved
DIVISOR0	13:8	rw	0x20	6-bit divider.
Reserved	7:3	rw	0x0	reserved
SRCSEL	2:0	rw	0x0	Clock generator input source. 000: IOPLL 010: RPLL 011: DPLL_CLK_TO_LPD

Appendix E: Memory Map

Name	Interface	Slave Segment	Master Base Address	Range	Master High Address
AXI4_BURST_MASTER_0					
/AXI4_BURST_MASTER_0/M00_AXI (40 address bits : 1T)					
/Block_Memory/axi_bram_ctrl_1/S_AXI	S_AXI	Mem0	0x00_C000_0000	8K	0x00_C000_1FFF
/zynq_ultra_ps_e_0/SAXIGP6	S_AXI_LPD	LPD_LPS_OCM	0x00_FF80_0000	8M	0x00_FFFF_FFFF
/zynq_ultra_ps_e_0/SAXIGP6	S_AXI_LPD	LPD_DDR_LOW	0x00_4000_0000	1G	0x00_7FFF_FFFF
zynq_ultra_ps_e_0					
/zynq_ultra_ps_e_0/Data					
/AXI4_BURST_MASTER_0/S00_AXI	S00_AXI	S00_AXI_reg	0x00_A000_0000	4K	0x00_A000_0FFF
/AXI4_BURST_MASTER_0/S00_AXI	S00_AXI	S00_AXI_reg	0x00_A000_0000	4K	0x00_A000_0FFF
/Block_Memory/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x00_A000_2000	8K	0x00_A000_3FFF
/Block_Memory/axi_bram_ctrl_0/S_AXI	S_AXI	Mem0	0x00_A000_2000	8K	0x00_A000_3FFF
/ULTRA96_SYSTEM/BD_CTL_GPIO/axi_gpio_0/S_AXI	S_AXI	Reg	0x00_A001_0000	64K	0x00_A001_FFFF
/ULTRA96_SYSTEM/BD_CTL_GPIO/axi_gpio_0/S_AXI	S_AXI	Reg	0x00_A001_0000	64K	0x00_A001_FFFF
/ULTRA96_SYSTEM/BD_CTL_GPIO/axi_gpio_1/S_AXI	S_AXI	Reg	0x00_A002_0000	64K	0x00_A002_FFFF
/ULTRA96_SYSTEM/BD_CTL_GPIO/axi_gpio_1/S_AXI	S_AXI	Reg	0x00_A002_0000	64K	0x00_A002_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/PWM_w_Int_0/s00_axi	s00_axi	reg0	0x00_A003_0000	64K	0x00_A003_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/PWM_w_Int_0/s00_axi	s00_axi	reg0	0x00_A003_0000	64K	0x00_A003_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/PWM_w_Int_1/s00_axi	s00_axi	reg0	0x00_A004_0000	64K	0x00_A004_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/PWM_w_Int_1/s00_axi	s00_axi	reg0	0x00_A004_0000	64K	0x00_A004_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/axi_gpio_2/S_AXI	S_AXI	Reg	0x00_A005_0000	64K	0x00_A005_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/axi_gpio_2/S_AXI	S_AXI	Reg	0x00_A005_0000	64K	0x00_A005_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/axi_uart16550_0/S_AXI	S_AXI	Reg	0x00_A006_0000	64K	0x00_A006_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/axi_uart16550_0/S_AXI	S_AXI	Reg	0x00_A006_0000	64K	0x00_A006_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/axi_uart16550_1/S_AXI	S_AXI	Reg	0x00_A007_0000	64K	0x00_A007_FFFF
/ULTRA96_SYSTEM/Low_Speed_MEZZ/axi_uart16550_1/S_AXI	S_AXI	Reg	0x00_A007_0000	64K	0x00_A007_FFFF
/ULTRA96_SYSTEM/SYS_MGMT/axi_gpio_3/S_AXI	S_AXI	Reg	0x00_A008_0000	64K	0x00_A008_FFFF
/ULTRA96_SYSTEM/SYS_MGMT/axi_gpio_3/S_AXI	S_AXI	Reg	0x00_A008_0000	64K	0x00_A008_FFFF
/ULTRA96_SYSTEM/SYS_MGMT/system_management_wiz_0/S_AXI_LITE	S_AXI_LITE	Reg	0x00_A009_0000	64K	0x00_A009_FFFF
/ULTRA96_SYSTEM/SYS_MGMT/system_management_wiz_0/S_AXI_LITE	S_AXI_LITE	Reg	0x00_A009_0000	64K	0x00_A009_FFFF

You need to maintain a consistent memory map between synthesis runs. This eliminates the need to generate a new DTB. The DTB can only be loaded by rebooting the Ultra96 while the FPGA bit can be loaded into the PL using the `fpga_util` in Linux.