Tutorial

StellarIP Interface

To

AXI Interface

4DSP LLC

Email: support@4dsp.com

This document is the property of 4DSP LLC and may not be copied nor communicated to a third party without the written permission of 4DSP LLC.

© 4DSP LLC 2014

Revision History

Date	Revision	Revision
2014-07-18	Initial release	1.0

Table of Contents

1 Introduction

StellarIP is a block-based IP integration solution similar to Xilinx Vivado IP Integrator. One important difference is that StellarIP blocks generally do not use AXI Interfaces. Since the StellarIP Catalog has fully functional IP for all 4DSP Hardware including host interfaces, memory controllers, and FMC Board Controllers, there might be a desire to reuse this code within a system using AXI Interfaces. This document is meant to be a guided description on how to convert a typical StellarIP block to have an AXI Interface.

To make this guide concrete the FMC110 IP Block (sip_fmc110) will be used, but the techniques can be applied to any StellarIP Block. The FMC110 IP Block is an Interface Controller for a 4DSP FMC110 daughter card which is a dual channel 12-bit ADC and a dual channel 16-bit DAC. The FMC110 controller is able to configure the clock tree, ADCs and DACs as well as capture data from the ADC and send data to the DAC.

1.1 Overview

To convert the FMC110 IP Block to AXI we first obtain the source code and create an AXI wrapper around it. At this point the interface conversion is complete but we continue with a simulation to gain confidence and finally build a design targeting a Xilinx VC707 development board to prove functionality.

1.2 Requirements

The following tools are required to completely follow this tutorial:

- 4DSP FMC110 Daughter Card
- Xilinx VC707 Carrier Board
- 4DSP Board Support Package
- Xilinx Vivado 2014.1 or later
- Python Programming Language Interpreter
- Analog Devices VisualAnalog

Please refer to the Xilinx user guides, 4FM Getting Started Guide, and other 4DSP documents to make sure all of these are properly installed.

1.3 Terms and definitions

Throughout the document the following terms are used with the indicated definitions

Entity	A VHDL keyword to define an interface. More abstractly it can represent a functional block or an actual file with VHDL code.
StellarIP Block or Star	All source code that makes up a StellarIP Block. A top level entity and all sub-entities.
Interface or Channel	A collection of signals.

2 Obtaining 4DSP IP Source Code

All 4DSP Hardware comes with a Board Support Package that includes an FPGA reference design that that exercises the functionality of the hardware. Since in this tutorial we want to take the FMC110 StellarIP Block and convert it to have an AXI-Interface we first need to obtain all the source code we are going to reuse.

2.1 Locate the VHDL source code.

It is typically located in the star lib folder "/star_lib/sip_fmc110".

2.2 Copy the required IP source files.

The FMC110 IP is designed to work with many different carrier boards having different FPGAs so there might be different version of files that are used with different FPGAs. To know which files are needed a list is provided depending on what FPGA is being used.

Table 1: Source List

File Ending	Description
_v7.lst	Design files used with a Virtex-7 FPGA
_k7.lst	Design files used with a Kintex-7 FPGA
_v6.lst	Design files used with a Virtex-6 FPGA

Since we are targeting a VC707 Carrier Board which has a Virtex-7 FPGA we look in the *_v7.lst file and copy each of the listed files in "/star_lib/sip_fmc110/sip_files/sip_fmc110_v7.lst" to a local directory called "Src".

```
Src/
└── sip_fmc110
    ├── ad9517_init_mem.ngc
    ├── ad9517_init_mem.vhd
    ├── ads5400_fifo.vhd
    ├── ads5400_init_mem.ngc
    ├── ads5400_init_mem.vhd
    ├── ads5400_phy_fifo.ngc
    ├── ads5400_phy_fifo.vhd
    ├── ads5400_phy_sp_v7.vhd
    ├── ads5400_storage_fifo.ngc
    ├── ads5400_storage_fifo.vhd
```

```
├── bit_align_machine.vhd
├── dac5681z_init_mem.ngc
├── dac5681z_init_mem.vhd
├── dac5681z_phy_fifo.ngc
├── dac5681z_phy_fifo.vhd
├── dac5681z_phy_v7.vhd
├── dac5681z_wfm_ctrl.vhd
├── dac5681z_wfm_dpram.ngc
├── dac5681z_wfm_dpram.vhd
├── dac5681z_wfm_input_fifo.ngc
├── dac5681z_wfm_input_fifo.vhd
├── dac5681z_wfm_output_fifo.ngc
├── dac5681z_wfm_output_fifo.vhd
├── dac5681z_wfm.vhd
├── dac_mmcm.vhd
├── dac_mmcm.xco
├── fmc110_ad9517_ctrl.vhd
├── fmc110_ads5400_ctrl.vhd
├── fmc110_cpld_ctrl.vhd
├── fmc110_ctrl.vhd
├── fmc110_dac5681z_ctrl.vhd
├── fmc110_if_v7.vhd
├── fmc110_stellar_cmd.vhd
├── pack_16to12.vhd
├── pulse2pulse.vhd
├── serdes_v7.vhd
├── sip_fmc110.vhd
└── sip_freq_cnt16.vhd
```

3 Conversion entities

In the majority of cases StellarIP Blocks have just three types of interfaces: Data Wormhole Inputs (wh_in), Data Wormhole Outputs (wh_out) and StellarIP Commands (cmd_in and cmd_out). They will be converted into AXI4-Stream Slave, AXI4-Stream Master, and AXI4-Lite Slave interfaces respectively. Example entities to perform the interface conversion have been provided.

```
Src/
└── conversion
    ├── axistream_to_whin.vhd
    ├── delay_bit.vhd
    ├── stellarcmd_to_axilite.vhd
    ├── axis2wh_fifo.ngc
    ├── axis2wh_fifo.vhd
    └── whout_to_axistream.vhd
```

Table 2: Conversion Entities

Conversion Entity	From Interface	To Interface	Used For
stellarcmd_to_axilite.vhd	StellarIP Command	AXI4-Lite Slave	FMC1110 Commands
axistream_to_whin.vhd	Wormhole Input	AXI4-Stream Slave	FMC110 DAC0/DAC1
whout_to_axistream.vhd	Wormhole Output	AXI4-Stream Master	FMC110 ADC0/ADC1

3.1 Wormhole Output to AXI-Stream Master

A StellarIP output channel is typically made up of three signals: DATA, VALID and STOP. This works as expected: DATA contains the output data, VALID is HIGH when DATA holds valid data and STOP indicates that the receiver can no longer accept data and we should stop sending.

The conversion between a StellarIP output channel and an AXI-Stream Master is straight forward connection because the channels are mostly compatible. The only difficulty is that when a STOP is asserted in a StellarIP Interface data will not immediately stop flowing, it might take up-to 8 cycles, unlike with TREADY. To accommodate this we use a FWFT FIFO.
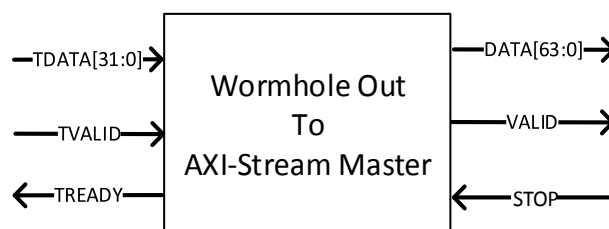

Figure 1: whout_to_axistream.vhd

All the signals in an AXI-Stream channel, except for ACLK, ARESETn and TVALID are optional. The other signals are set to their default values, see the AXI-Stream Standard for details.

3.2 Wormhole Input to AXI-Stream Slave

A StellarIP input channel is typically made up of three signals: DATA, VALID and STOP. This works as expected: the signal DATA contains the input data, the signal VALID is HIGH when the signal DATA holds valid data and the signal STOP provides back pressure when valid data can no longer be accepted.

The conversion between a StellarIP input channel and an AXI-Stream Slave is straight forward because the interfaces are compatible. In the AXI-Stream standard all signals expect for ACLK, ARESETn and TVALID are optional giving a straight forward conversion.
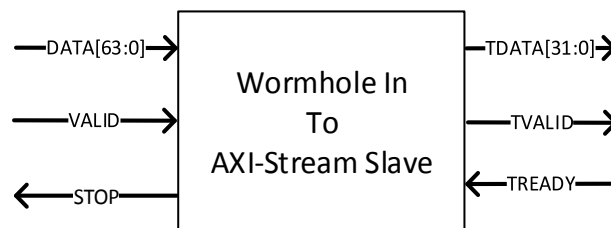

Figure 2: axistream_to_whin.vhd

3.3 StellarIP Command to AXI-Lite

A StellarIP command interface is generally made of two signals: CMD and CMD_VALID. The command signal is 64-bits wide.

Table 3: StellarIP Command Format

Range	63...60	59..32	31..0
Field	CMD	Address	Data
Description	The command defines what is in the other fields, it can be one of the following: CMD_WR = 0b0001 CMD_RD = 0b0010 CMD_RD_ACK = 0b0100	Contains the address which we are trying to write to or read to.	Contains the data we are going to write when sent with CMD_WR. Ignored when sent with CMD_RD. Contains the read data when received with CMD_RD_ACK.

The interface is not bidirectional and require two pairs for writing and reading; CMD_IN, CMD_IN_VALID, CMD_OUT and CMD_OUT_VALID. The command interface can be thought as a distributed RAM with an address range which we read and write from. The conversion entity which has been provided, **stellarcmd_to_axilite.vhd,** is loosely based on the Xilinx AXI GPIO IP.
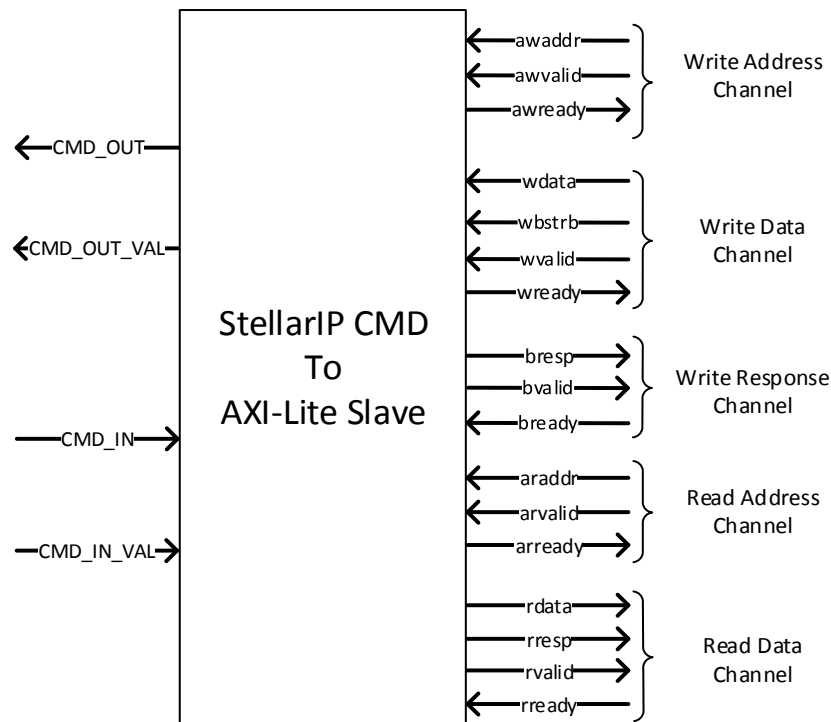
Figure 3: stellarcmd_to_axilite.vhd

The conversion is done by accepting a CMD_IN StellarIP command and issuing an AXI-Lite command and also by accepting an AXI-Lite command and issuing a CMD_OUT StellarIP command. The conversion can be best understood by becoming familiar with the AXI-Lite Standard. As an example, image a write of 0xAAAAAAAA to address 0x7. The address would come from the Write Address Channel and the data would come from the Write Data Channel. These would be combined to form the following StellarIP command:

| 0x1 | 0x0000007 | 0xAAAAAAAA |

The command would be sent out CMD_OUT and a reply would be sent through the Write Response Channel.

4 Wrapping into AXI

At this point we have the source code for the StellarIP Block we are going to reuse and the conversion entities that help speed up the process. Looking at top level entity, **sip_fmc110.vhd**, for the FMC110 IP we can see that we have two wh_in interfaces for each of the DACs, two wh_out interfaces for each of the ADCs, and the command interface; cmd_in and cmd_out.

```
63 entity sip_fmc110 is
64 generic (
65    GLOBAL_START_ADDR_GEN                 : std_logic_vector(27 downto 0);
66    GLOBAL_STOP_ADDR_GEN                  : std_logic_vector(27 downto 0);
67    PRIVATE_START_ADDR_GEN                : std_logic_vector(27 downto 0);
68    PRIVATE_STOP_ADDR_GEN                 : std_logic_vector(27 downto 0)
69 );
70 port (
71 --Wormhole 'clk' of type 'clkin':
72    clk_clkin                             : in    std_logic_vector(31 downto 0);
73
74 --Wormhole 'rst' of type 'rst_in':
75    rst_rstin                             : in    std_logic_vector(31 downto 0);
76
77 --Wormhole 'cmdclk_in' of type 'cmdclk_in':
78    cmdclk_in_cmdclk                      : in    std_logic;
79
80 --Wormhole 'cmd_in' of type 'cmd_in':
81    cmd_in_cmdin                          : in    std_logic_vector(63 downto 0);
82    cmd_in_cmdin_val                      : in    std_logic;
83
84 --Wormhole 'cmd_out' of type 'cmd_out':
85    cmd_out_cmdout                        : out   std_logic_vector(63 downto 0);
86    cmd_out_cmdout_val                    : out   std_logic;
87
88 --Wormhole 'adc0' of type 'wh_out':
89    adc0_out_stop                         : in    std_logic;
90    adc0_out_dval                         : out   std_logic;
91    adc0_out_data                         : out   std_logic_vector(63 downto 0);
92
93 --Wormhole 'adc1' of type 'wh_out':
94    adc1_out_stop                         : in    std_logic;
95    adc1_out_dval                         : out   std_logic;
96    adc1_out_data                         : out   std_logic_vector(63 downto 0);
97
98 --Wormhole 'dac0' of type 'wh_in':
99    dac0_in_stop                          : out   std_logic;
00    dac0_in_dval                          : in    std_logic;
01    dac0_in_data                          : in    std_logic_vector(63 downto 0);
02
03 --Wormhole 'dac1' of type 'wh_in':
04    dac1_in_stop                          : out   std_logic;
05    dac1_in_dval                          : in    std_logic;
06    dac1_in_data                          : in    std_logic_vector(63 downto 0);
07
08 --Wormhole 'ext_fmc110' of type 'ext_fmc110':
09    fmc_to_cpld                           : inout std_logic_vector(3 downto 0);
10    front_io_fmc                          : inout std_logic_vector(3 downto 0);
11
```

Each of these channels will be connected to the appropriate conversion entity. We create a new file, **axi_fmc110.vhd**, that instantiates the StellarIP block we are reusing and one appropriate conversion entity for each of the interfaces. A block diagram of the final result is shown below.
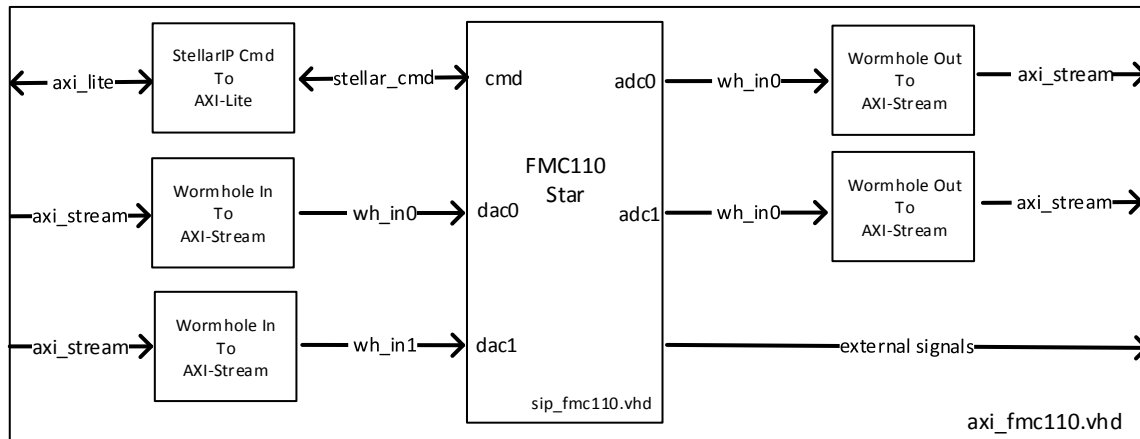
Figure 4: Block Diagram of a StellarIP Block with AXI Interfaces

The external signals are simply propagated through the wrapper.

5 Simulate

It is a good idea to run a simulation to gain confidence that the design works.

5.1 File descriptions

For the simulation to work we need the following files.

```
./Src/
├── axi_fmc110.vhd
├── conversion/* (See Section 3)
├── data_gen/
│   ├── axi_stream_send.vhd
│   └── rom.vhd
├── simulation/
│   ├── axi_fmc110_tb.vhd
│   ├── addr.coe
│   ├── data.coe
│   ├── fmc110_model/
│   │   ├── ads5400_init_mem.mif
│   │   ├── ad9517_init_mem.mif
│   │   ├── dac5681z_init_mem.mif
│   │   ├── fmc110_cpld.vhd
│   │   ├── fmc110_model.vhd
│   │   └── i2c_slave_model.vhd
│   ├── sip_cmd.sip
│   └── XilinxCoreLib/
│       ├── BLK_MEM_GEN_V6_1.vhd
│       ├── BLK_MEM_GEN_V6_2.vhd
│       ├── fifo_generator_v8_1.vhd
│       └── fifo_generator_v9_3.vhd
└── sip_fmc110/* (See Section 2)
```

Table 4: Simulation File Descriptions

File or Folder	Description
axi_fmc110.vhd	This is the top level entity of the converted IP. We took an IP Block from StellarIP and wrapped around conversion entities to have an AXI Interface. This is explained in **Section 4** of this document.
conversion/*	This folder contains conversion entities provided as examples by 4DSP. These take typical StellarIP type of interfaces and

	convert them to AXI Interfaces. This was explained in **Section 3** of this document.
sip_fmc110/*	This is the StellarIP Block source code we are reusing. This was explained in **Section 2** of this document.
simulation/axi_fmc110_tb.vhd	This is the top level test bench, see the section below for creating this test bench.
data_gen/*	This entity was created for this application note to function as a simple AXI-Stream Master used to generate data that will go into the DAC interface of the FMC110. When enabled it sequentially reads a ROM and sends out the contents. It produces a waveform that will be seen out of the DACs of the FMC110 daughter card.
simulation/addr.coe **simulation/data.coe**	These files are used by the Xilinx Traffic Generator to generate the write commands. StellarIP blocks work with commands to configure the burst length, read status information, enable capturing etc. Typically it is required to read the documentation for the StellarIP block you are using to figure what needs to be configured. Since an example is provided, sip_cmd.sip, we can simply take the commands from that file.
simulation/sip_cmd.sip	Many StellarIP blocks come with simulation files including a file called sip_cmd.sip which contain a list of commands used by a StellarIP host interface model.
simulation/ XilinxCoreLib/*	In Vivado 2014.1 the XilinxCoreLib library has been removed. This library contained simulation models for ISE IP cores such as FIFOs and Block Memories. Simulation models of Xilinx Vivado IP cores are delivered as an output product when the IP is generated but because many StellarIP blocks have been designed with ISE they depend on files from XilinxCoreLib for simulation to work. We must copy the simulation files that are being used from a previous version of ISE. A typical location is "C:\Xilinx\14.7\ISE_DS\ISE\vhdl\src\XilinxCoreLib".
simulation/ fmc110_model/*	This is a simulation model for the FMC110 daughter card. It sends data through the external pins so we are able to simulate the capturing of the ADC data.

5.2 Creating the project

We will now step through creating the simulation project and running it.
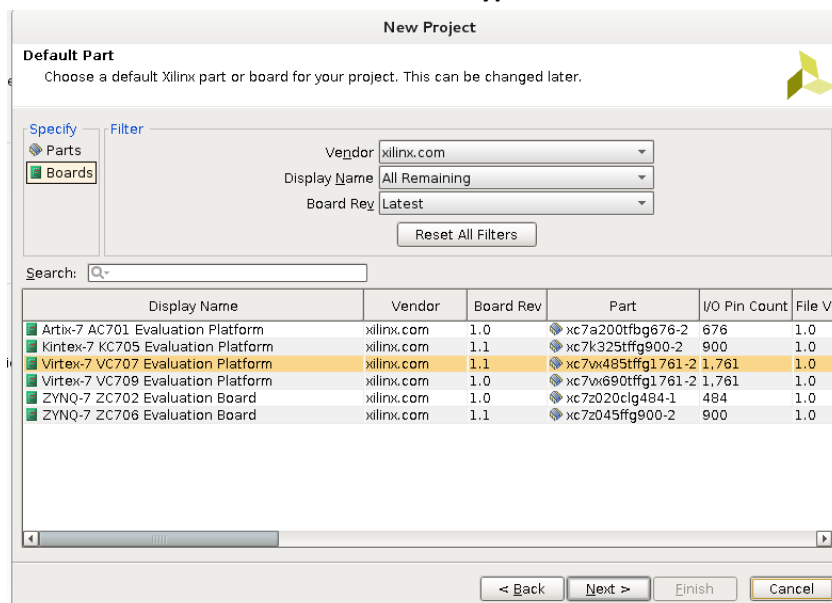
1. Select **Create New Project**

2. Enter a project name and the location where to save it for **Project Name**.
3. Select RTL Project for **Project Type**
4. Click Next for **Add Sources/ Add Constraints/ Add Existing IP**
5. Select VC707 under Boards for **Device Type**

6. Select **Finish**

At this point we have a Vivado project created and we now need to add all the design files. It is important to add files that are used for both synthesis and simulation by selecting **Add or Create Design Sources**. If the file is used only for simulation select **Add or Create Simulation Sources**.

7. Add the files from the **conversion** folder by selecting **Add or Create Design Sources**.

8. Add the files from the **sip_fmc110** folder by selecting **Add or Create Design Sources**.
9. Add the file from the **fmc110_model** folder by selecting **Add or Create Simulation Sources**.
10. Add the files from the **data_gen** folder by selecting **Add or Create Design Sources.**
11. Add the files from the **XilinxCoreLib** folder by selecting **Add or Create Simulation Sources**. Make sure to change the library to **xilinxcorelib**.
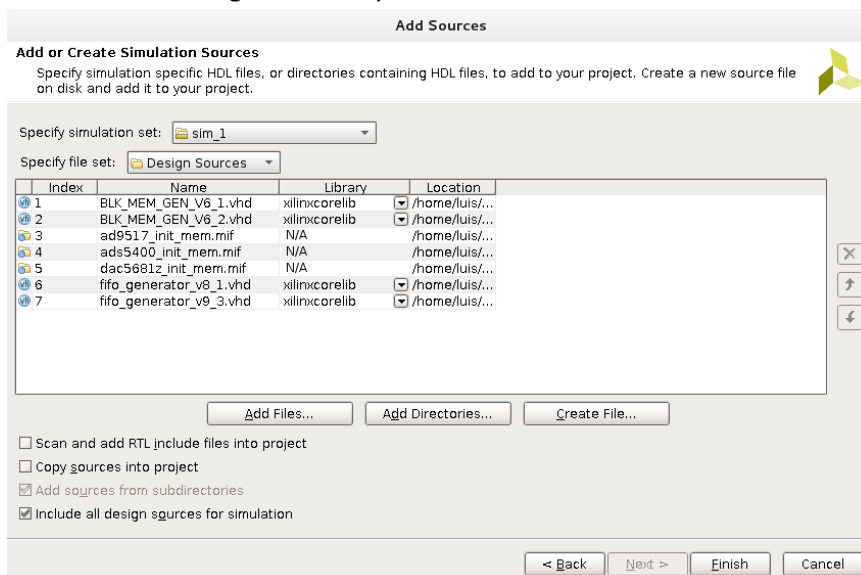
Next we will add the AXI Traffic Generator IP from the Xilinx IP Catalog. This is used to send configuration commands to our IP Block.

12. Open **IP Catalog** from Flow Navigator
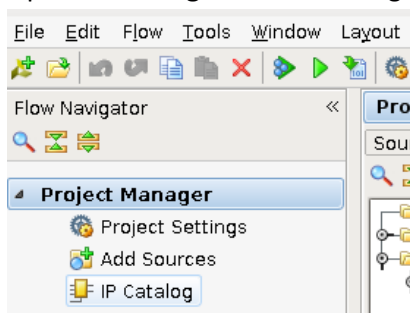
13. Select **AXI Traffic Generator**

At this point we must configure the AXI Traffic Generator IP with the required settings. It is important we select "AXI4-Lite" as the protocol since this is the command interface we created for our IP. Mode

needs to be "System Init", this is the simplest mode where only sending commands is allowed. The "Transaction Depth" must be equal to the number of addresses and data in our COE files. The Address COE file and Data COE file were created by looking at sip_cmd.sip; it is important these two files are selected here in the **COE File Paths** section.

14. Select the appropriate settings
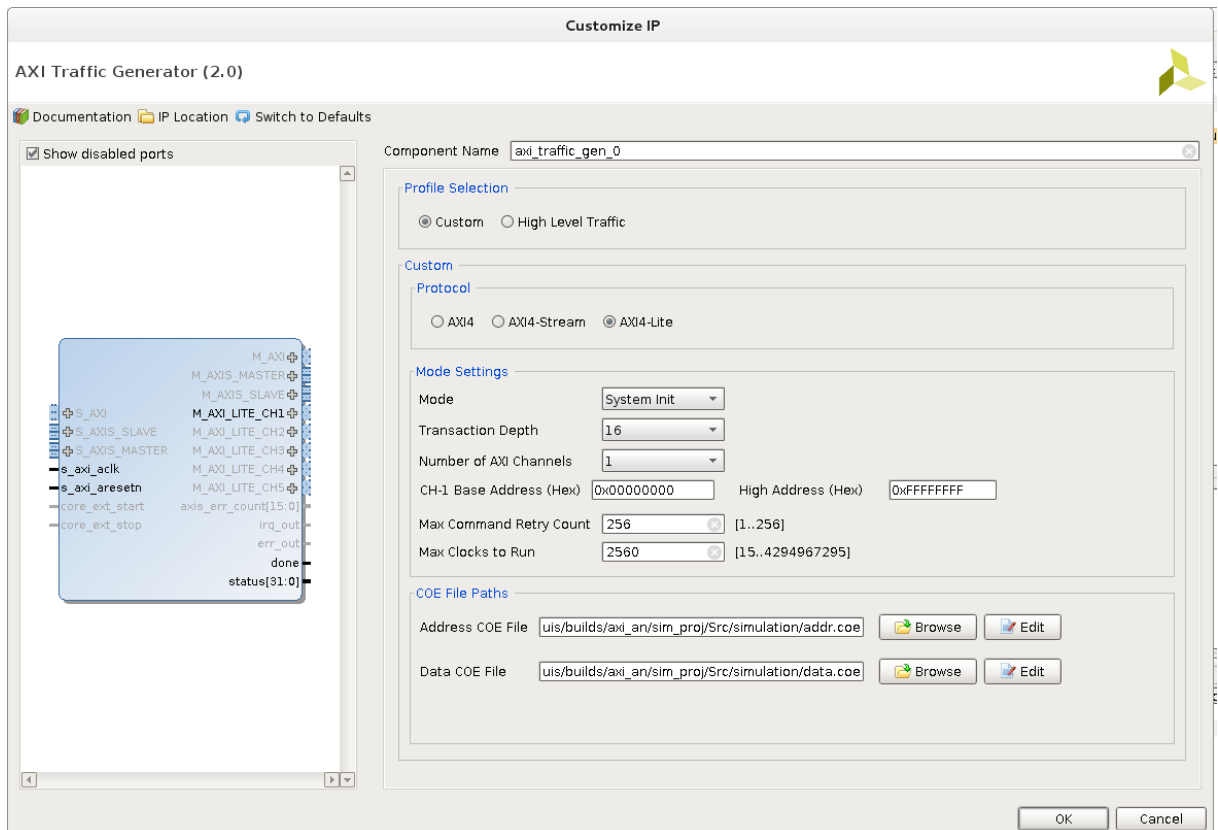
15. Select **OK**. This should synthesize the Traffic Generator IP.

5.3 Creating the test bench

We now create a new file, **axi_fmc110_tb.vhd**, by instantiating the blocks as shown below. An example on how to do this has been provided.
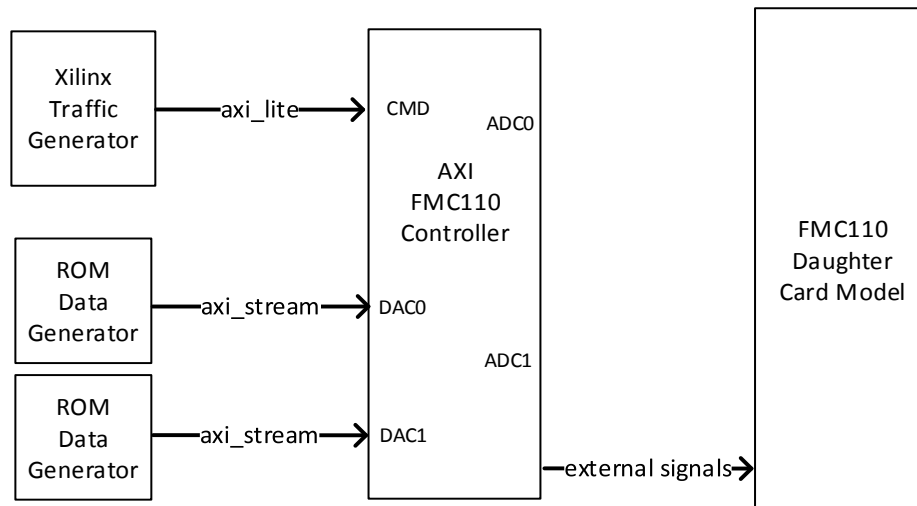

Figure 5: Block diagram of simulation top-level

5.4 Running Simulation

Once the top-level test bench is ready we can **Run Simulation.** Vivado will process the design files and a waveform viewer will appear.

5.4.1 Command Interface

We first inspect that the appropriate commands are being sent from the Xilinx Traffic Generator Block through its AXI-Lite Interface. Add all the top level entity signals from the Xilinx Traffic Generator entity to the waveform viewer.
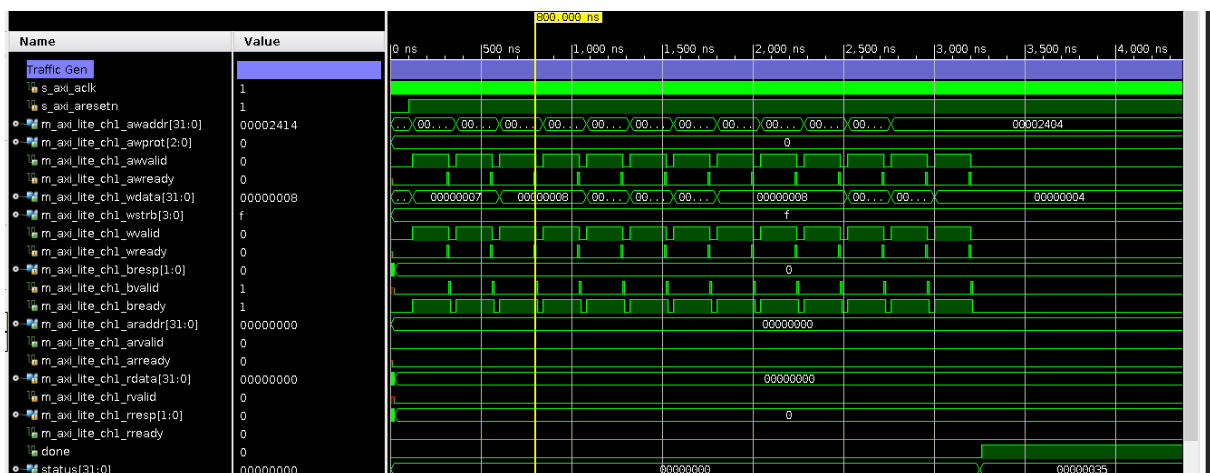

Figure 6: Waveform view of the Traffic Generator

We can also confirm that commands are arriving in **fmc110_ctrl.vhd,** for example the register NB_BURSTS_REG should get the value '1'. This demonstrates that conversion from AXI-Lite to StellarIP commands is working.

Once the Traffic Generator sends all its commands we should see DONE='1' and STATUS[1:0] = "01" indicated success.

5.4.2   ADC and DAC Interfaces

For the ADC we should see ADC0_TDATA with data when ADC0_TVALID='1'. This data is generated in the FMC110 Model and captured by the FMC110 Controller.

For the DAC we should see data coming out on DAC0_TDATA when DAC0_VALID='1'. We can trace this data going to a memory entity inside the FMC110 IP Block. Once a trigger is sent, through the Traffic Generator, data is read out of the memory and seen on the external pins.

Figure 7: Waveform view of DAC0 and ADC0

We have successfully simulated the design. We were able to send commands to the StellarIP Block, we were able to capture data from the ADCs and we were able to play a waveform in the DACs.

6 Synthesize

To have a synthesizable design we must make slight modifications to the simulation design. We switch the Xilinx Traffic Generator to Xilinx JTAG to AXI-Master. This gives us more control; it allows us to read as well as write and allows us to repeat a sequence of commands, for example, to capture ADC results multiple times. Also, we must capture the ADC data where we can view it; in simulation it is possible to add any signal to the waveform viewer. For synthesis we use the Xilinx Integrated Logic Analyser (ILA).

6.1 File Descriptions

The following files are used to synthesize the design and described below.

```
Src/
├── axi_fmc110.vhd
├── conversion/* (See Section 3)
├── data_gen/
│   ├── axi_stream_send.vhd
│   └── rom.vhd
├── python/
│   ├── FMC110.vac
│   ├── ila_raw.py
│   ├── samples.txt
│   └── waveform.csv
├── sip_fmc110/* (See Section 2)
├── synthesis/
│   ├── axi_fmc110_synth.vhd
│   ├── axi_fmc110_synth.xdc
│   ├── brd_clocks.vhd
│   └── pll0.vhd
└── tcl/
    ├── fmc110_capture.tcl
    ├── fmc110_generate.tcl
    └── fmc110_init.tcl
```

Table 5: Synthesis file descriptions

| File or Folder	Description
synthesis/axi_fmc110_synth.vhd	Top level entity
synthesis/brd_clocks.vhd	Clock generation entity. Accepts a differential external clock and generate a 200 MHz and a 125 MHz clock using a PLL (pll0.vhd)
synthesis/axi_fmc110_synth.xdc	Constraint file for VC707 + FMC110.

	Note: When creating a constraint file it is important to know that everything is case sensitive, unlike VHDL. The signal names must match the case of the top level signal names. The Vivado commands need to be lower case, i.e., it is important to have set_property and not SET_PROPERTY.
python/ila_raw.py	A script written in the Python programming language used to read a CSV file, extract the data of interest, and convert the data to the format required for processing. This is used on python/waveform.csv which was generated with the Vivado command **write_hw_ila_data.** The output of the Python script is python/samples.txt which can be plotted and analysed with VisualAnalog using the template python/ FMC110.vac.
python/FMC110.vac	Design file for Analog Device's VisualAnalog Software, used to visualize and process data samples.
tcl/ fmc110_init.tcl	Vivado TCL script to send commands through the Xilinx JTAG to AXI Master IP. Used to initialize the FMC110 Card. More information in the **Running Design** section.
c/sipif.cpp	Modified read and write functions in the reference application for VC707+FMC110. Used to log what is being written and read.

6.2 Modifying the project for synthesis

1. In **Flow Navigator** select **IP Catalog**
2. Select **JTAG to AXI Master**
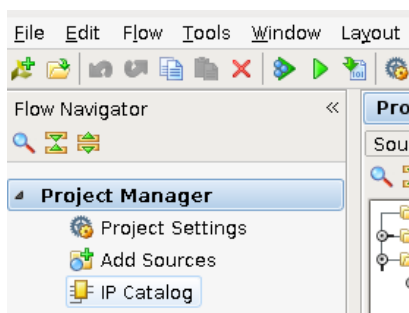
3. Configure the IP Block with the following settings
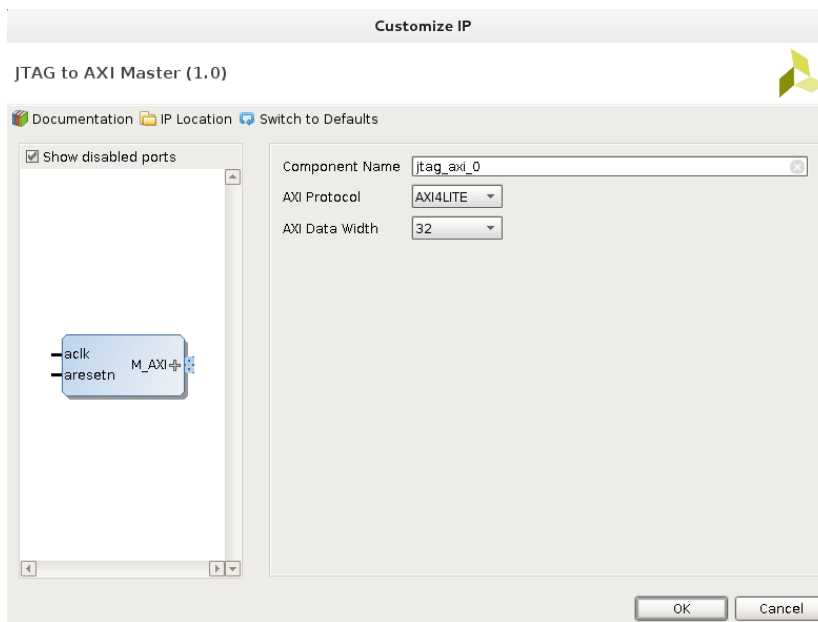
4. Select **OK.** JTAG to AXI Master IP should now synthesize**.**

In simulation the ADC output could be added to the waveform viewer without being connected to anything. For synthesis, to view the ADC output, we use the Xilinx Integrated Logic Analyzer (ILA) IP Block.

5. In **Flow Navigator** select **IP Catalog**
6. Select **ILA (Integrated Logic Analyzer).**
7. Configure the IP Block using: Interface = Native, Number of Probes = 1, Probe Width = 132.
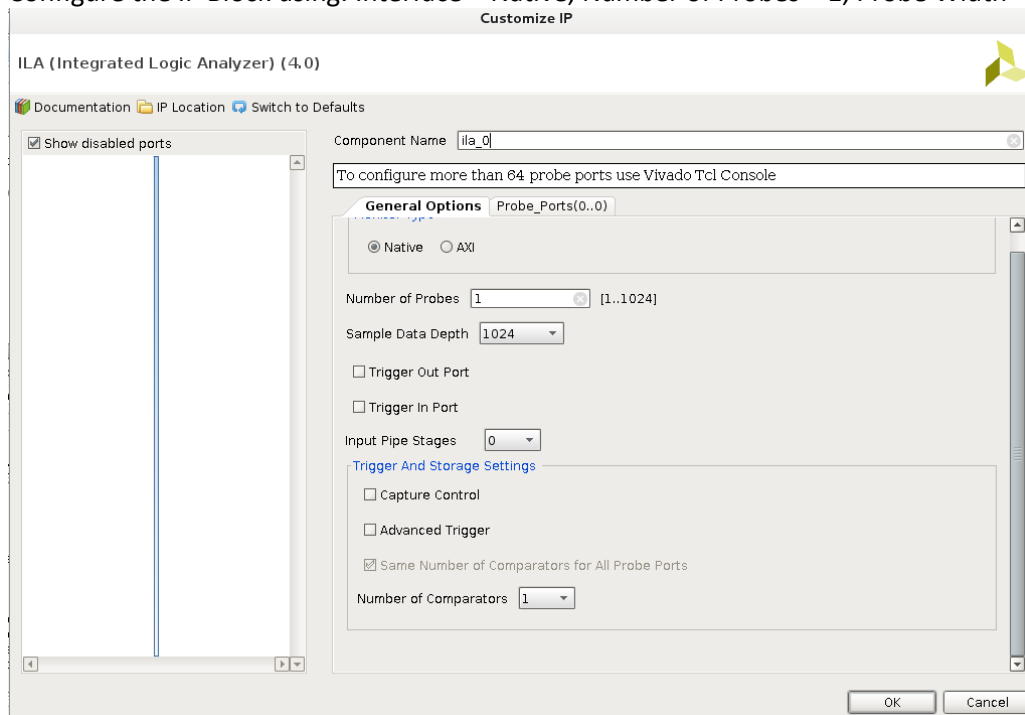
8. Select **OK**. ILA should now synthesize.

In simulation a reset is created by using the VHDL after keyword that won't synthesize. For synthesis we use the Xilinx VIO (Virtual Input/Output) IP Block.
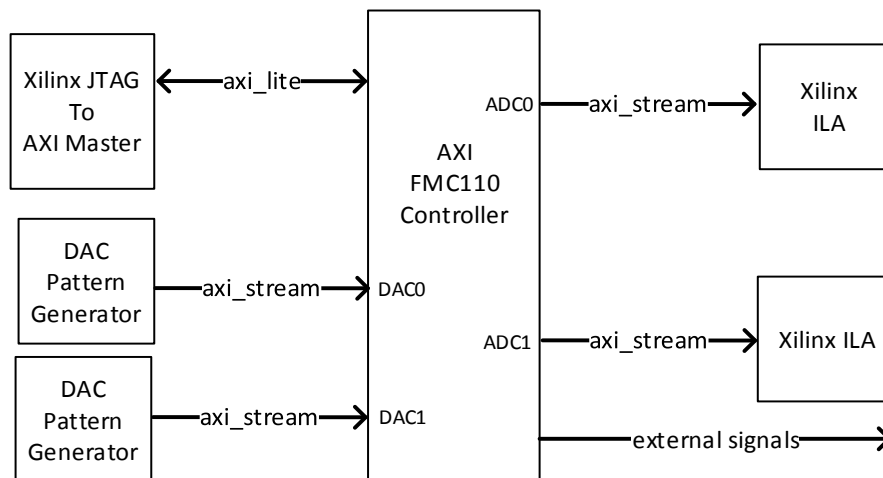
9. In **Flow Navigator** select **IP Catalog**.
10. Select **VIO** (Virtual Input/Output).
11. Configure the IP Block using, Input Probe Count = 1, Output Probe Count = 1,
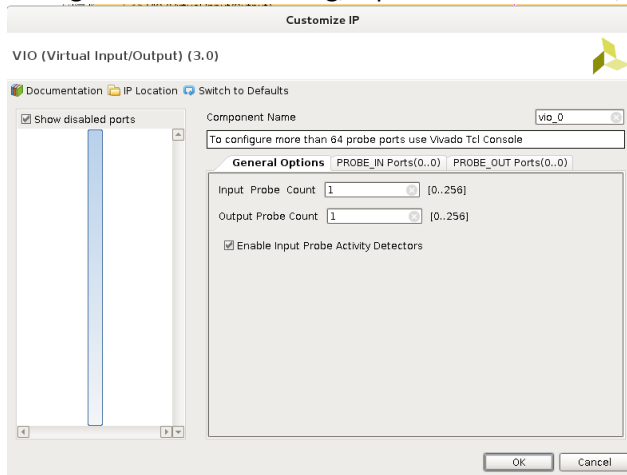
12. Select **OK**. VIO should now synthesize.
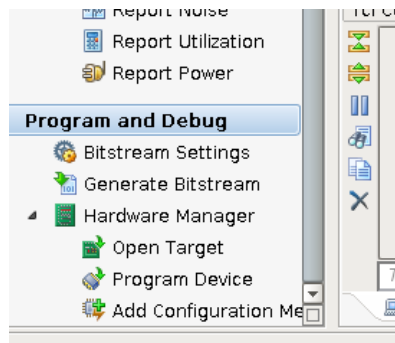
6.3   Creating the top level

The final block diagram should look as following:

Finally we can select **Run Implementation**.

6.4 Running and Initialization

Once we have successfully generated a bit file we open **Hardware Manager** and select **Program Device** to program the bit file to the FPGA.

6.4.1 Initialization Script

With the FPGA programmed and with the Hardware Manager opened we must execute the initialization script, **fmc110_init.tcl,** in the Vivado TCL console to configure the FMC110 daughter card.

```
source fmc_init.tcl
```

The command sequence is a lot longer than it was for simulation; we are running on hardware and we must correctly configure all devices. Generally this requires a careful reading of the datasheet for each device (ADCs/DACs/Clock Synthesizers/etc...) and figuring out what registers, if any, need to be configured. 4DSP has done this and provides a reference design that performs the complete configuration in software.

By modifying the write and read wrapper functions used in the 4DSP reference design application for VC707+FMC110 (see **sipif.cpp)** we can obtain a complete list of all commands sent.

Read and writes logged from the VC707+FMC110 reference application by modifying sipif.cpp.	The same read and writes done through Xilinx JTAG to Master through the fmc110_init.tcl script
[RD] 00002408 00000003	axi_lite_read 2408 ;#00000003
[RD] 00002d24 00000000	axi_lite_read 2d24 ;#00000000
[WR] 00002d24 00000020	axi_lite_write 2d24 00000020
[WR] 00002d24 00000000	axi_lite_write 2d24 00000000
[RD] 00002d24 00000000	axi_lite_read 2d24 ;#00000000
[WR] 00002d24 00000040	axi_lite_write 2d24 00000040
[WR] 00002d24 00000000	axi_lite_write 2d24 00000000
[WR] 00002d24 0000001e	axi_lite_write 2d24 0000001e
[WR] 00002d25 00000000	axi_lite_write 2d25 00000000
[WR] 00002d26 00000000	axi_lite_write 2d26 00000000
[RD] 00002d24 0000001e	axi_lite_read 2d24 ;#0000001e
[RD] 00002707 00000053	axi_lite_read 2707 ;#00000053
[WR] 00002714 0000007c	axi_lite_write 2714 0000007c
…	…

Once we run the initialization script we have access to the two commands **axi_lite_write** and **axi_lite_read**, see the file **fmc_init.tcl** or Xilinx UG908 Programming and Debugging for implementation details.

6.5 Capturing ADC Data

Once the synthesized design is loaded and the FMC110 has been initialized we are ready to capture data. For this test a 30 MHz sine wave from a function generator is being fed to ADC0.
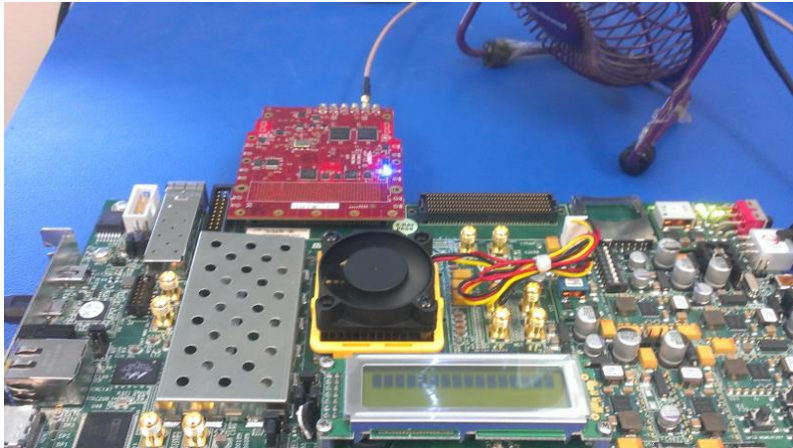

Figure 8: FMC110 Data Capture

We now configure the ILA for capturing. The important signals to capture are ADC0_TDATA, and ADC0_TVALID, trigger the ILA on **adc0_valid = '1'** to see the ADC samples.
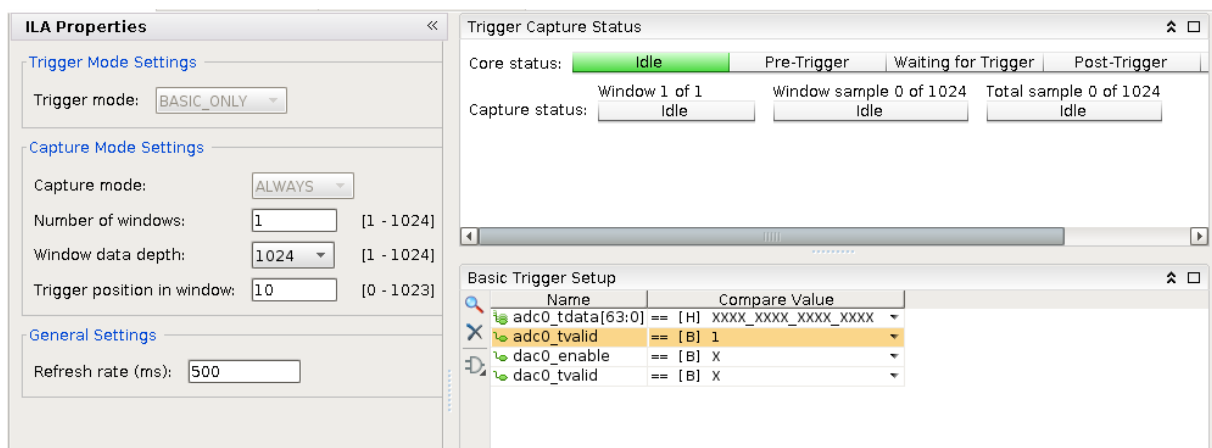

Figure 9: Trigger Configuration

Issue the ADC capture command sequence through the Vivado TCL console.

```
axi_lite_write 2405   0000050d ;# Enable ADC0 & DAC0&DAC1
axi_lite_write 2404   00000001 ;# ARM
axi_lite_write 2404   00000004 ;# Trigger
```

An ILA trigger occurs when ADC0_TVALID='1' and data appears on the waveform viewer. We can see data arriving on ADC0_TDATA. Virtual buses were created to see individual samples and we get the following waveform:
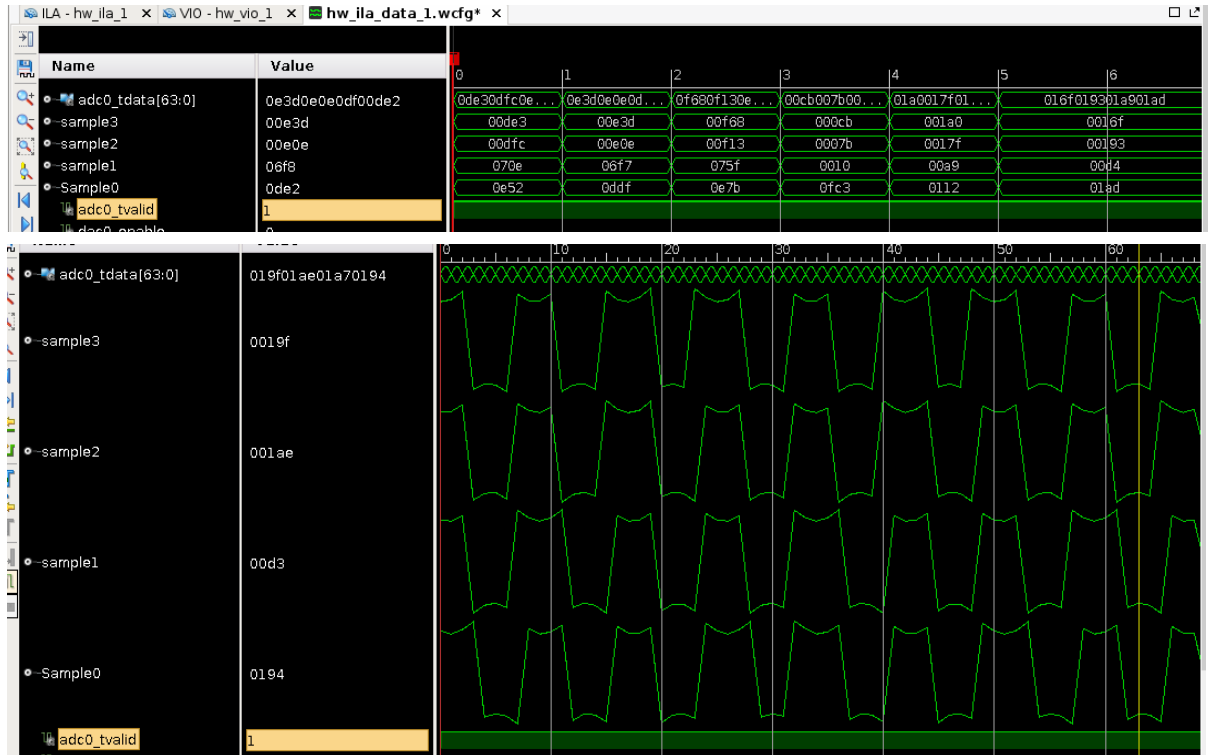
The signal doesn't look like a sine wave because the ADC outputs data in two's complement form. Also, it would be best to see the samples sequentially, not four in parallel, unfortunately there is nothing in the Vivado waveform viewer to accommodate this. We can, however, download the samples by issuing the following command in Vivado:

```
write_hw_ila_data ila_data_file.zip [upload_hw_ila_data hw_ila_1]
```

This saves the waveform into an archive with a CSV file which has the raw captured data. Using the provided python script **(ila_raw.py)** we can convert from binary strings to integers, which can be read by Analog Device's VisualAnalog Application.
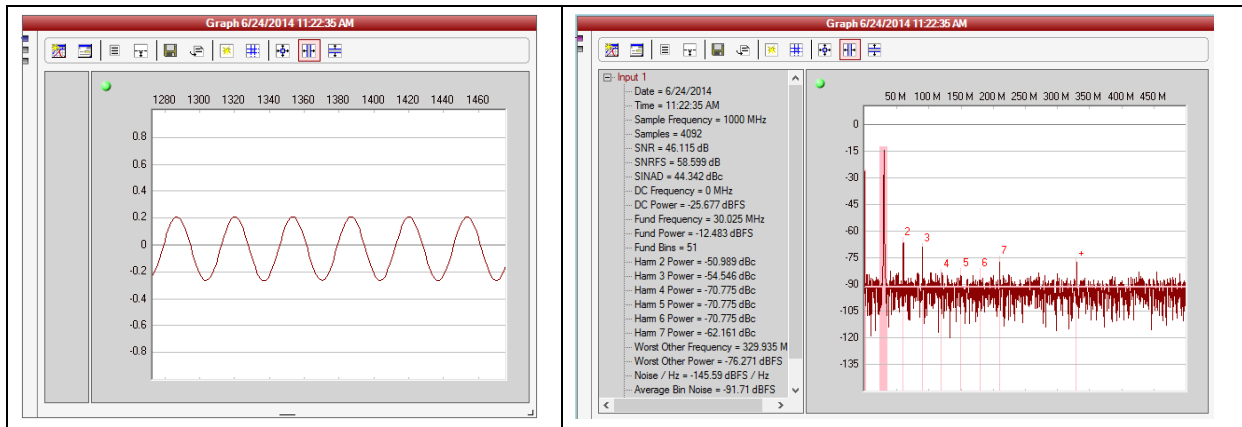
Figure 10: VisualAnalog display of the captured data.

From the graph and FFT we see the captured data correctly represents a 30MHz signal.

6.6 DAC Results

The DAC is enabled when the following command sequence is executed:

```
axi_lite_write 2406    00000001 ;# Number of bursts
axi_lite_write 2407    00000100 ;# Burst size
axi_lite_write 2405    00000004 ;# Enable DAC0
axi_lite_write 2404    00000008 ;# Load WFM (Firmware will monitor for this command)
axi_lite_write 2404    00000004 ;# Trigger
```

There is a process in **axi_fmc110_synth.vhd** (and **axi_fmc110_tb.vhd** when simulating) that monitors for a DAC load waveform command and enables the data generator (**axi_stream_send.vhd)** any time it is seen. The generated data come into the FMC110 Controller through one of the AXI Slave interfaces that was created in Section 4 for the DAC. The data gets stored inside a BRAM where it will continuously be sent to the DAC when a trigger command is asserted.

```
process(clk)
begin
    if rising_edge(clk) then
        if axi_wdata = x"00000008" and axi_awaddr = x"00002404"  and axi_awvalid = '1' and axi_wvalid = '1' then
            dac0_enable <= '1';
        else
            dac0_enable <= '0';
        end if;
    end if;
end process;
```

Figure 11: DAC Data Generation Enable

The following 16 samples make up the sine wave that is generated.

```
0000,30fb,5a81,7640,7fff,7640,5a81,30fb,0000,CF05,A57F,89C0,8001,89C0,A57F,CF05
```

The DAC is operating at 1.0 GSPS which means that the DAC consumes a sample every 1 ns so the pattern is repeating every 16 ns. 16 ns translates to 62.5 MHz, so we would expect to see a 62.5 MHz signal on an oscilloscope.
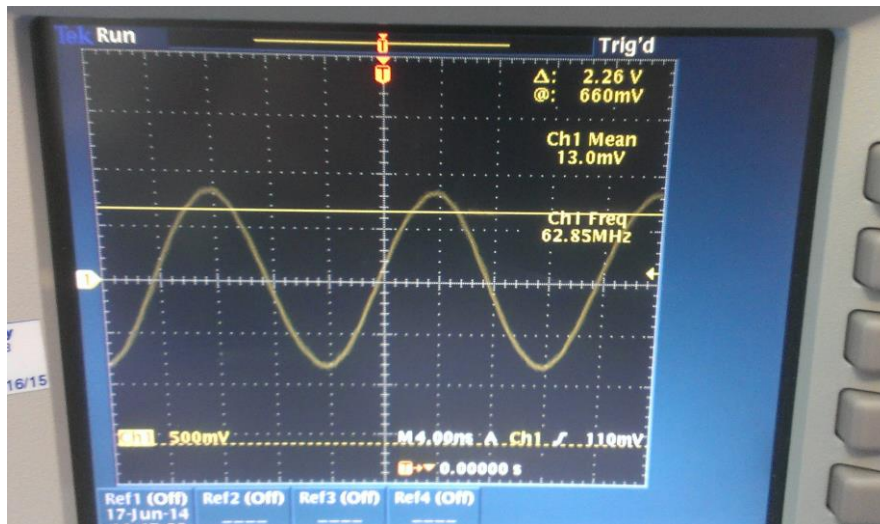
Figure 12: Oscilloscope view of DAC0

Connecting DAC0 from the FMC110+VC707 to an oscilloscope shows a waveform of 62.85 MHz, this confirms that the AXI FMC110 Controller is working.

7 References

- **Vivado Design Suite User Guide Programming and Debugging**
- **4FM Programmer's guide**
- **FMC110 User Manual**
- **FMC110 Star Documentation**
- **Xilinx AXI Traffic Generator IP Documentation**
- **Xilinx JTAG to Master IP Documentation**
- **Xilinx AXI GPIO IP Documentation**