AXI4 - Stream interface

Luis Vega - gutierre [at] rhrk.uni-kl.de
Pedro Torruella - torrue [at] rhrk.uni-kl.de

October 11, 2012

1 AXI4-Stream short introduction

AXI4-Stream is an extension of the ARM-AMBA bus communication protocol intended to be used on streaming applications without dealing with complex control signals. Latest versions of Xilinx EDK (13 or higher) has built-in support for AXI4-Stream. Nevertheless, the documentation is very limited. That is why this short introduction could be a good starting point. Basically, AXI4-Stream consists on a Master/Slave interface that can be employed for Writing/Reading respectively. The maximum number of Master/Slave interfaces on a Microblaze-based system is 16 in total. Furthermore, AXI4-Stream is a 32-bit bus communication protocol. Tables 1, 2, 3, and 4 contain a brief description about the AXI4-Stream signals.

2 Why?

AXI4-Stream is intended to be used on a MicroBlaze-Embedded-System, in order to send/receive data over an IP core. Therefore, the following system in figure 1 will be used as reference design for this documentation. Moreover, the main goal of this documentation is to provide essential knowledge to adapt

Signal	Direction	Width(bits)	Type
s_axis_tdata	input	32	data
s_axis_tvalid	input	1	control
s_axis_tlast	input	1	control
s_axis_tready	output	1	control

Table 1: AXI-Slave signals (details)

Signal	Description
s_axis_tdata	payload data
s_axis_tvalid	receiving valid data, if s_axis_tvalid = '1'
s_axis_tlast	receiving last transaction value, if s_axis_tlast = '1'
s_axis_tready	unit is ready to receive data, if s_axis_tready = '1'

Table 2: AXI-Slave signals (description)

Signal	Direction	Width(bits)	Type
m_axis_tdata	output	32	data
m_axis_tvalid	output	1	control
m_axis_tlast	output	1	control
m_axis_tready	input	1	control

Table 3: AXI-Master signals (details)

Signal	Description
m_axis_tdata	payload data
m_axis_tvalid	sending valid data, if s_axis_tvalid = '1'
m_axis_tlast	sending last transaction value, if s_axis_tlast = '1'
m_axis_tready	unit is ready to send data, if s_axis_tready = '1'

Table 4: AXI-Master signals (description)

or create IP-cores to work with AXI4-Stream in a MicroBlaze-Embedded-System. Additionally, extra functionality to the interface is added beside the communication interface itself. For example, a reset output that can be controlled by software. Thus, users could be able to reset their system easily. All these details will be covered in later sections.

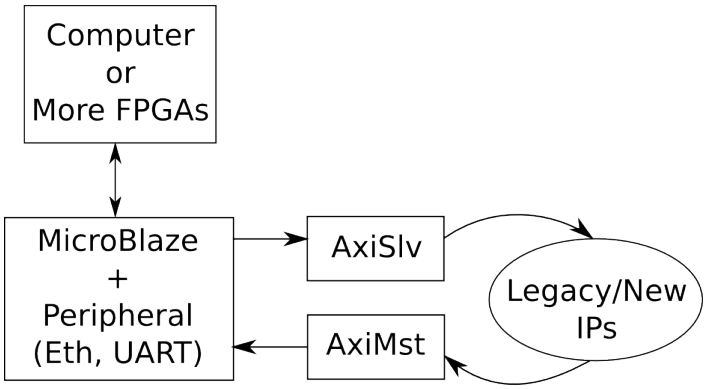

Figure 1: Microblaze Embedded System.

3 AXI-Slave interface

3.1 Brief description

The AXI-Slave interface (`AxiSlv.vhd`) is the module in charge of reading from the AXI4-Stream bus. Since the MicroBlaze can write into the AXI4-Stream bus, the AXI-Slave interface could be used to fetch data from MicroBlaze and send it to the IP of interest. The AXI-Slave block is shown in figure 2.

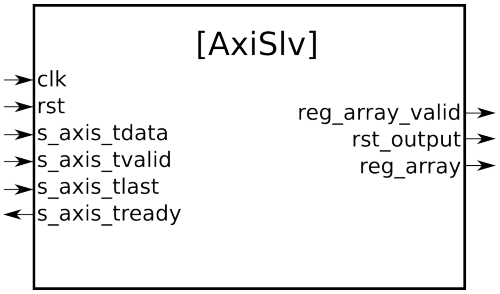

Figure 2: AXI-Slave interface block

The basic idea behind is a Serial-To-Parallel interface, data will come serially from (`s_axis_tdata`) and then place it in register's array (`reg_array`). How

the data are arranged in this register's array is shown in figure 3. The WIDTH and NUM_REG variables are generic and let users configure the interface as it is needed. The WIDTH variable is used for assigning the data-width of (`s_axis_tdata`). However, remember that AXI4-Stream work with 32 bits, so this variable should be 32. On the other hand, the NUM_REG variable states the number of registers need by your application. For example, if your application needs to feed three 32-bits-inputs, NUM_REG should be 3. These details will be covered in the following application example.
A valid signal `reg_array_valid` is provided by the interface. This signal will stall at '1' when the interface has a first configuration valid. On the other hand, a reset output `rst_output` is given that can be controlled by software. Furthermore, supposed that you would like to test an *Adder* that has two inputs (number1, number2) and you want to reset this *Adder* before sending these numbers. Then, you need to send the data sequence (1, number1, number2) over `s_axis_tdata`, where the first number in the sequence, '1' in this case, stands for activating this reset output. Otherwise, if you do not want to activate the `rst_output` the data sequence should be (0, number1, number2). Additionally to this, the number of cycles where the `rst_output` is activated can be configured by the variable NUM_RST_CYCLE in the `AxiSlv.vhd` module. Moreover, the `rst_output` is activated after the data is written in `reg_array`. Finally, the reset can be active high or low by assigning '1' or '0' to G_RESET_ACTIVE variable.

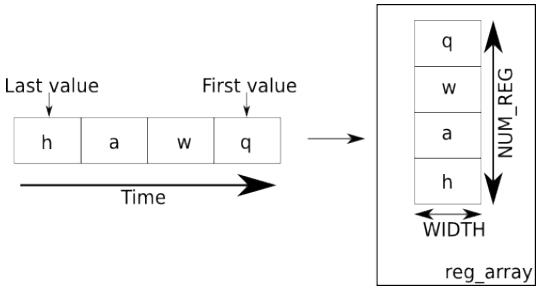

Figure 3: AXI-Slave register organization

3.2 Application example

In order to make clear the functionality of the AXI-Slave interface (`AxiSlv.vhd`), an application example is given in this section. Indeed, the application example will focus on adapting a RNG module to work with AXI4-Stream and attaching it to a MicroBlaze. Therefore, it is possible to send/receive data from our computer to the RNG module. To achieve this, we need `AxiSlv.vhd`

(Interface) and `RngUniformTausworthe88.vhd` (RNG) modules. This files are in the source repository.
First of all, a top module in VHDL is needed in order to connect the slave interface and the RNG. This top module should be as shown in the figure 4. We are going to call this module as `AxiRng.vhd`. If you do not want to write the code, then look for the *AxiRng* folder in the source repository.

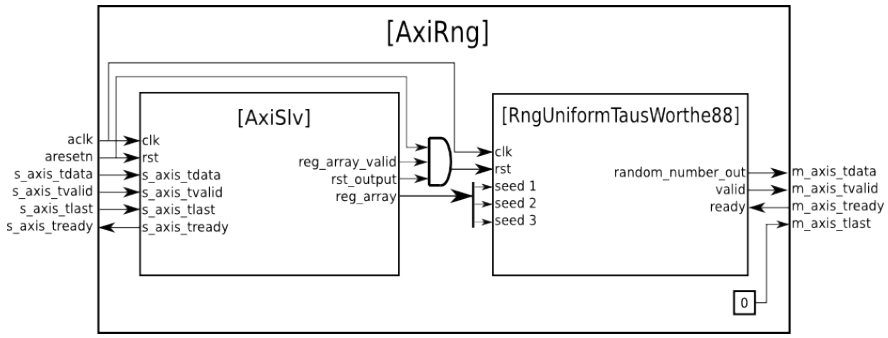

Figure 4: AxiRng.vhd

The "AND-gate" is used here between `rst_output`, `aresetn` and `reg_array_valid` to perform the reset only when the interface loads a valid configuration or the system has been restarted at the begging. The `reg_arrray` signal contains the seeds to feed the RNG module. The RNG module will load these seeds after a reset is performed, then it will start to generate random numbers. Since the `axis_tlast` signal is not used, the signal is assigned to '0'. Last but not least, the generic values for the modules are:

- G_RESET_ACTIVE = 0
- WIDTH = 32
- NUM_REG = 3
- NUM_RST_CYCLE = 3

At this point, we can go further and follow some steps to see everything working. This documentation is based on Xilinx Design Suite 14.1 and a Virtex-6 ML605. Thus, it should be worth to check if you have both. All the steps are performed on the `csiga` server, which has already installed all the necessary tools. The steps that you are going to perform on your host computer are in sections"Setting up the host" and "Download HW/SW to the FPGA". For doing so, it is enough to have Xilinx Webpack 14.1, which is available and free on Xilinx website.

Because the development steps are based on different tools, it is suggested the folder tree structure shown in figure 5. After creating this structure, you should place the tree VHDL files in the "vhdlSrc" folder:

- `AxiRng.vhd`
- `AxiSlv.vhd`
- `RngUniformTausworthe88.vhd`

The "ise, xps and sdk" folders should be empty, because the project files are going to be saved there. Finally, the following steps are given in a *Quick start* fashion, if you are interested in more details, you can check other documentation about Xilinx Design Suite 14.1 that is available in the repository.

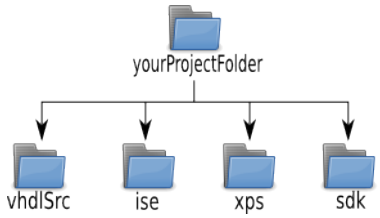

Figure 5: Project folder tree structure

3.2.1 Building the IP core - ISE

a. Open Xilinx-ISE.

b. Create a new project with the name "AxiRng" and placed it in the suggested location ("ise" folder).

c. Select the target FPGA-board (Virtex-6 ML605 revision D).

d. Add the source files to the project, this can be done by the "Add copy of source" option in ISE.

e. Select our top module "AxiRng.vhd" and synthesize it, check that there are not errors here.

f. Close ISE and follow the next section.

3.2.2 Hardware assembly - XPS

a. Open Xilinx-XPS.

b. Create a new project and placed it in the suggested location ("xps" folder).

c. Select the target FPGA-board (Virtex-6 ML605 revision D).

d. Keep the RS232_UART peripheral and remove the other peripherals. Remember to activate the interrupt in the check-box.

e. Click finish.

f. In this step, we are going to create a dummy peripheral in order to use it as template to build our MPD file, because the tool does not help us on that. Go to hardware and "Create or Import Peripheral..", then name it "template". Select AXI4-Stream bus interface and the number of input/output 32-bit words should be '1'.

g. Similarly, the AxiRng can be imported. In this case, instead on selecting create you should select "import existing peripheral". Then, go to our "ise" folder and find the "AxiRng.prj" file. After this, use the same name "AxiRng", so the tool can find the top module. The following configuration steps by the wizard should be avoided by unchecking the box.

h. Open the MPD file for the dummy peripheral, this can be done by going to the IP catalog and expanding USER and right click on "template" , then click on "View MPD". Copy everything down from "##BUS Interfaces" to the end of the file.

i. Open the MPD file for our AxiRng and paste it in the same lines from where you copied it in the other file. Then, make all the necessary modifications as shown in the figure 6.

j. Go to "Project" and click on "Rescan User Repositories".

k. Now, after the peripheral description of the AxiRng is done, we can add it to the system assembly view by double clicking on it.

l. In order to make the connection between the AxiRng and the MicroBlaze, a AXI4-Stream bus has to be instantiated. This can be achieved by double clicking on microblaze_0 in the assembly view. Then, click on

next button until you reach “PVR and Buses”, find “Select stream interfaces” and choose “AXI” instead of “FSL”, and “Number of stream links” should be ’1’. Click ok.

m. Connect the AxiRng by clicking on the respective bus in the system assembly view. M0_AXIS on microblaze_0 should be connected to S0_AXIS on AxiRng. The same for S0_AXIS on microblaze_0 should be connected to M0_AXIS on AxiRng.

n. Afterwards, the clock and reset signal of the AxiRng must be connected. Go to the port tab in the system assembly view and look for AxiRng. Then, click on ACLK and select “clock_generator_0” and select “CLK0” in the right side. The reset can be connected in a similar way, find the “ARESETN” port and click it. Then, select “proc_sys_reset_0” and “Peripheral_aresetn”.

o. Now, the complete MicroBlaze Embedded System can be generated. This can be accomplished by clicking on ‘Export Design”. Remember that you can speed up the process by going to edit → preferences and enable the “Parallel synthesis” option.

p. When the process finished, Xilinx-SDK should pop up and ask for a workspace folder. Please select the suggested folder (“sdk” folder). Xilinx-SDK will create a sub folder named “hw_platform”, this folder contain the bitstream file (*.bit). Usually, this file is called “system.bit”. Later, this “system.bit” will be used for programming the FPGA.

3.2.3 Software development - SDK

a. After selecting the workspace folder as you did in the last step, you should be able to see the eclipse environment. Check that the “hw_platform” folder created by XPS is placed in the project explorer window in Xilinx-SDK.

b. Go to “File” and create a new “Xilinx C project”.

c. Select the “hello world” template. Xilinx-SDK will create two folders, one containing the source code and the other one containing drivers and support libraries. These two folders are: “hello_world_0” and “hello_world_bsp_0”. As stated before, the “hello_world_0” folder contain the source code (*.c) in the “src” folder and the executable (*.elf)

```
##############################################################################
##
## Name      : AxiRng
## Desc      : Microprocessor Peripheral Description
##           : Automatically generated by PsfUtility
##
##############################################################################

BEGIN AxiRng

## Peripheral Options
OPTION IPTYPE = PERIPHERAL
OPTION IMP_NETLIST = TRUE
OPTION HDL = MIXED
OPTION IP_GROUP = USER


## Bus Interfaces
BUS_INTERFACE BUS=M_AXIS, BUS_STD=AXIS, BUS_TYPE=INITIATOR
BUS_INTERFACE BUS=S_AXIS, BUS_STD=AXIS, BUS_TYPE=TARGET

## Parameters
PARAMETER C_S_AXIS_PROTOCOL = GENERIC, DT = string, TYPE = NON_HDL, ASSIGNMENT =
    CONSTANT, BUS = S_AXIS
PARAMETER C_S_AXIS_TDATA_WIDTH = 32, DT = integer, TYPE = NON_HDL, ASSIGNMENT =
    CONSTANT, BUS = S_AXIS
PARAMETER C_M_AXIS_PROTOCOL = GENERIC, DT = string, TYPE = NON_HDL, ASSIGNMENT =
    CONSTANT, BUS = M_AXIS
PARAMETER C_M_AXIS_TDATA_WIDTH = 32, DT = integer, TYPE = NON_HDL, ASSIGNMENT =
    CONSTANT, BUS = M_AXIS
## Peripheral ports
PORT ACLK = "", DIR=I, SIGIS=CLK
PORT ARESETN = "", DIR=I, SIGIS=RST
PORT S_AXIS_TREADY = TREADY, DIR=O, BUS=S_AXIS
PORT S_AXIS_TDATA = TDATA, DIR=I, VEC=[31:0], BUS=S_AXIS
PORT S_AXIS_TLAST = TLAST, DIR=I, BUS=S_AXIS
PORT S_AXIS_TVALID = TVALID, DIR=I, BUS=S_AXIS
PORT M_AXIS_TVALID = TVALID, DIR=O, BUS=M_AXIS
PORT M_AXIS_TDATA = TDATA, DIR=O, VEC=[31:0], BUS=M_AXIS
PORT M_AXIS_TLAST = TLAST, DIR=O, BUS=M_AXIS
PORT M_AXIS_TREADY = TREADY, DIR=I, BUS=M_AXIS

END
```

Figure 6: AxiRng MPD file

in the "Debug" folder. The following information goes for advanced users, the build-configuration should remain in Debug-mode since the MicroBlaze Debug Mode (MDM) is going to be used for programming and test our system.

d. Then, in the "hello_world.c" file copy the code shown in the figure 7.

e. Now, after saving the file, the project should be built again. However, you could double check building it again.

```c
#include <stdio.h>
#include "platform.h"
#include "fsl.h"
#include "xuartlite.h"
#define UART_BASE_ADDR 0x40600000

char XUartLite_RecvByte();
void XUartLite_SendByte();
void print(char *str);

void UartSendInt(int number){
    int i;
    char byteToSend;
    for (i = 3; i >= 0; i--){
        byteToSend = (number >> i*8) & 0xff;
        XUartLite_SendByte(UART_BASE_ADDR,byteToSend);
    }
}

int main()
{
    int rst_cfg = 0x00000001;
    int seed1   = 0x88cb47c9;
    int seed2   = 0x8a9cdf65;
    int seed3   = 0xcaf40ed9;
    int i, RngNumber, numberValues = 10;

    init_platform();

    putfslx(rst_cfg,0,FSL_DEFAULT);
    putfslx(seed1,0,FSL_DEFAULT);
    putfslx(seed2,0,FSL_DEFAULT);
    putfslx(seed3,0,FSL_DEFAULT);

    for (i = 0; i < numberValues; i++){
        getfslx(RngNumber,0,FSL_DEFAULT);
        UartSendInt(RngNumber);
    }

    cleanup_platform();

    return 0;
}
```

Figure 7: Source code for testing AxiRng

3.2.4 Setting-up the host

a. Find a terminal application in your host computer. For example hyperterminal (Win) or cutecom (Linux).

b. Configure such terminal application to work on *baudrate* = 9600, *databits* = 8, and *stopbits* = 1.

c. Start or Open the device in order to start receiving values right after the programming is done.

3.2.5 Download HW/SW to the FPGA

a. Download the bitstream file (system.bit) and executable (hello_world.elf) from the server to your host computer. Remember that these files are in the "hw_platform" and "hello_world_0/Debug" subfolder in the "sdk" folder respectively.

b. Move the files to a handy location in your host computer, for example, your home folder.

c. Open a terminal there and type XMD and press enter. You should now be in the MicroBlaze Debug Module.

d. Type "fpga -f system.bit" and press enter, this will download the bitstream file to the FPGA.

e. Type "connect mb mdm" and press enter, this will connect the MicroBlaze to the Debug Module.

f. Type "dow hello_world.elf" and press enter, this will download the executable to our system.

g. Type "rst" and press enter, in order to reset the microblaze.

h. Type "con" and press enter, this will start the software program.

i. Then, you should receive the first ten random values as shown in figure 8.

j. You could stop the MicroBlaze by typing "stop" in the XMD environment. Also, you can find more useful commands in the MicroBlaze documentation.

k. The following information is to speed-up the programming procedure. You could copy all these commands into a file and after initializing XMD, and then you can perform "source myfile.txt".

```
C8A39675
0002C0D9
00010427
C0061519
5B65701C
CD36394D
B97524D9
5144B1BA
20B10C51
C29F1017
```

Figure 8: First ten random values - AxiRng

4 AXI-Master interface

The AXI-Master interface (`AxiMst.vhd`) is the module in charge of writing to the AXI4-Stream bus. Since the MicroBlaze can read from the AXI4-Stream bus, the AXI-Master interface could be used to send data to the MicroBlaze from the IP-core of interest. The AXI-master block is shown in figure 9.

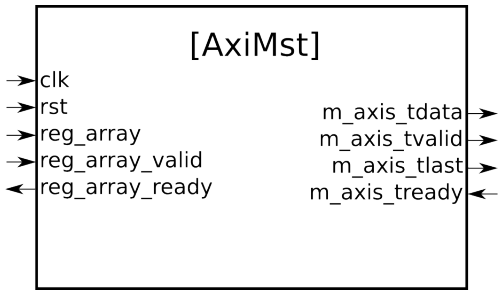

Figure 9: AXI-Master interface block

Opposite to the AXI-Slave interface, the AXI-Master interface works as a Parallel-To-Serial interface. In other words, data will come in a parallel fashion through `reg_array` and place it serially over `m_axis_tdata`. Loading the data in `reg_array` can be done by a valid signal called `reg_array_valid`, and it must be set to '1' for data loading. In addition to a valid signal, a ready signal (`reg_array_ready`) is used for stopping data loading when the interface is serializing and putting data on `m_axis_tdata`. Furthermore, when ready signal is '1', the interface is ready for receiving data. Otherwise, the ready signal will be '0'.
How the data are arranged by the AXI-Master interface is shown in figure 10. Similar to the AXI-Slave interface, the WIDTH and NUM_REG variables are generic and let users configure the interface as it is needed. Remember that

AXI4-Stream work with 32 bits, so WIDTH should be 32. More details, will be covered by the application example in the next section.

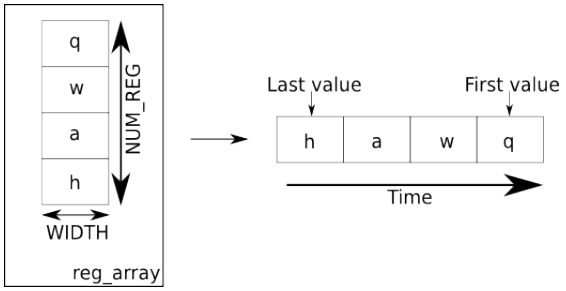

Figure 10: AXI-Master register organization

4.1 Application example

In the following application example, we will use AXI-Slave (`AxiSlv.vhd`) and AXI-Master (`AxiMst.vhd`) interfaces together with two RNG modules working in parallel. At the end of this application you should be able to adapt your own IP-core to work with the above mentioned interfaces.
First of all, a top module in VHDL is needed in order to connect all blocks. This top module should be as shown in the figure 11. We are going to call this module as `AxiTwoRng.vhd`. If you do not want to write the code, then look for the *AxiTwoRng* folder in the source repository.

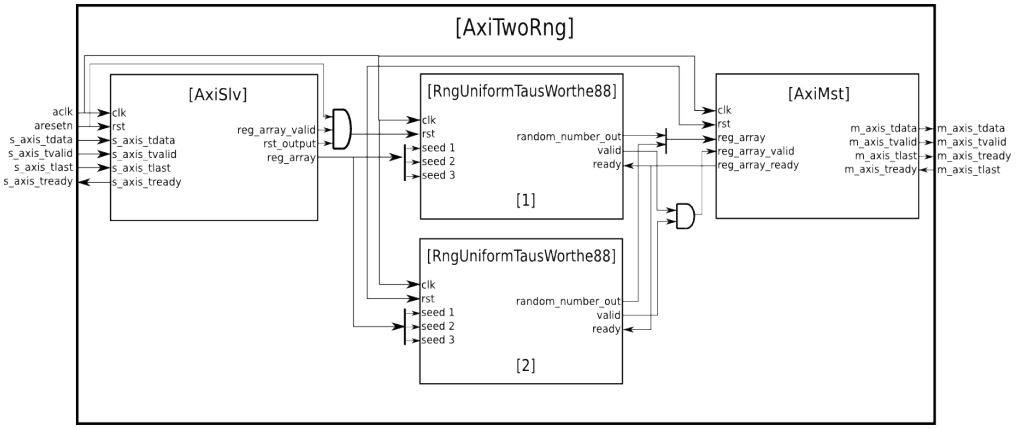

Figure 11: AxiTwoRng.vhd

Similarly to the application example of AXI-Slave, the RNG module is connected to the AXI-Slave. However, there are some additionally details. First, the `reg_array` signal will contain six seeds for the RNG modules, three seeds

for each one. Secondly, the generic parameters for the `AxiTwoRng.vhd` is used as follows:

- G_RESET_ACTIVE = 0
- WIDTH = 32
- NUM_REG_SLV = 6
- NUM_REG_MST = 2
- NUM_RST_CYCLE = 3

The number of registers (NUM_REG_SLV) is 6, because as it is stated before we need six seeds. Here, it might be good to remember that the number of registers is named as NUM_REG_SLV and not as NUM_REG. The main reason is that we need to make a difference between the number of registers for the Slave and Master interface. Therefore, the number of registers for the AXI-Master is given by NUM_REG_MST. Furthermore, since we are instantiating two RNG modules, the application will produce 64 bits. Thus, the parameter NUM_REG_MST should be 2. Finally, the number of clock cycles for the reset (NUM_RST_CYCLE) is 3.
On the other hand, the AXI-Master can control the generation of random numbers by the `reg_array_ready` signal. Meanwhile, each valid signal `valid` of the RNG modules is connected through an AND gate to the `reg_array_valid` port.
At this point, we should have the following source files:

- `AxiTwoRng.vhd`
- `AxiSlv.vhd`
- `AxiMst.vhd`
- `RngUniformTausworthe88.vhd`

From now on, it is advisable follow the same directions given by the application example for the AXI-Slave interface. However, here are some additional hints:

- The MPD file is the same used in the AxiRng application example, since the interface is the same.
- The software program (.c) for the AxiTwoRng application example is shown in figure 12.
- The output for this application example is shown in figure 13.

```c
#include <stdio.h>
#include "platform.h"
#include "fsl.h"
#include "xuartlite.h"
#define UART_BASE_ADDR 0x40600000

char XUartLite_RecvByte();
void XUartLite_SendByte();
void print(char *str);

void UartSendInt(int number){
    int i;
    char byteToSend;
    for (i = 3; i >= 0; i--){
        byteToSend = (number >> i*8) & 0xff;
        XUartLite_SendByte(UART_BASE_ADDR,byteToSend);
    }
}

int main()
{
    int rst_cfg     = 0x00000001;
    int seed1Rng1   = 0x88cb47c9;
    int seed2Rng1   = 0x8a9cdf65;
    int seed3Rng1   = 0xcaf40ed9;
    int seed1Rng2   = 0x88cb47c9;
    int seed2Rng2   = 0x88cb47c9;
    int seed3Rng2   = 0x88cb47c9;
    int i, RngNumber, numValues = 8;

    init_platform();

    //Active rst_output
    putfslx(rst_cfg,0,FSL_DEFAULT);

    //Seeds for first instance of RNG
    putfslx(seed1Rng1,0,FSL_DEFAULT);
    putfslx(seed2Rng1,0,FSL_DEFAULT);
    putfslx(seed3Rng1,0,FSL_DEFAULT);

    //Seeds for second instance of RNG
    putfslx(seed1Rng2,0,FSL_DEFAULT);
    putfslx(seed2Rng2,0,FSL_DEFAULT);
    putfslx(seed3Rng2,0,FSL_DEFAULT);

    for (i = 0; i < numValues; i++){
        getfslx(RngNumber,0,FSL_DEFAULT);
        UartSendInt(RngNumber);
    }

    cleanup_platform();

    return 0;
}
```

Figure 12: Source code for testing AxiTwoRng

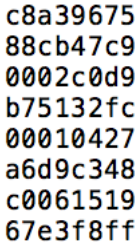

Figure 13: First eight random values - AxiTwoRng