Loopy. Quick start guide. Tutorial 2

Vladimir Rybalkin - rybalkin [at] rhrk.uni-kl.de

MICROELECTRONIC SYSTEMS DESIGN RESEARCH GROUP

The purpose of this tutorial is to explain how to configure the driver generator and develop the client application for the specifics of your design. For profound understanding of the material of the current tutorial we highly encourage you to use “HOPP Driver Generator Documentation” [1], which was used as a base for the current tutorial. Before you start with this tutorial it is implied that you are familiar with “Loopy. Tutorial 1”.

1) **Short Introduction**. The board-side driver running on FPGA and host-side driver together with client application running on the server are intended to be used for communication purposes between computer and board, in order to send/receive data over medium. Therefore, the following system in the Figure 1 will be used as a reference for this documentation.

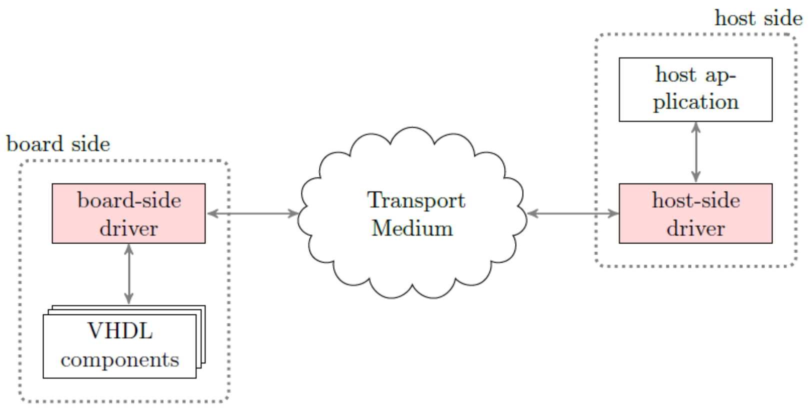

Figure 1: A high-level view of the data flow from a host application to hardware components on the board through the generated driver

In order to make clear the process of driver and client application development, an application example is given in this section. Indeed, the application example won’t focus on the hardware design of the modules. Therefore, it is recommended to use sample hardware module Figure 2. from the source repository *<root dir>/examples/AxiFIFO* (check out “Loopy. Tutorial 1” to know how to find the repository). *“AxiFifo.vhd”* is a top-level module with AXI4-Stream interface ports; *“Fifo.ngc”* is a netlist file of the core (FIFO queue) generated using Xilinx CORE Generator, which is instantiated in the top-level module mentioned above. When a generated core is used, only a netlist file has to be provided to the driver generator. This point will be elaborated in the following chapter.

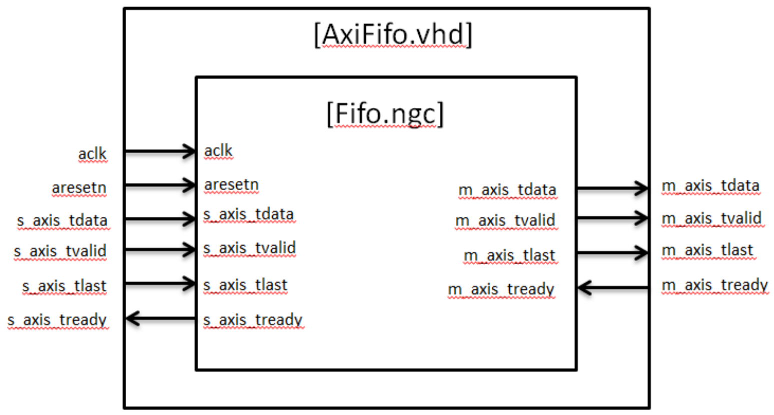

Figure 2: AxiFifo.vhd – top-level module of the hardware design

2) **Configuring driver generator.** The behavior of the driver itself is only influenced by the provided *.bdf* file. The *“system.bdf”* file from the root directory describes behavior of the driver in the reference design.

The board description language is used to specify a board, for which a driver and project files should be generated. A board specification file consists of several declarations, which may appear in arbitrary order. In these declarations blocks and code blocks are used.

The ***medium*** declaration describes over which medium board and host should communicate with each other. The Ethernet medium is selected using the keyword ***ethernet***. The block has to specify the mac address, ip address, subnet mask, standard gateway and port number in the common case. However, in the case of financial server, when DHCP settings are used only mac address, key word *“dhcp”* and port has to be specified. You can find a mac address of the given board on a sticker attached to the board.

```
medium ethernet
{
        mac "00:0a:35:02:31:95"
        dhcp
        port 8844
}
```

You can specify several global properties, at this time, this includes global hardware queue with the size equal to zero, means that hardware queue is disabled.

```
hwqueue 0
```

A core is a template for components on the board. It requires behavior specification in form of VHDL, Verilog HDL or netlist files and description of the interface. A core is declared by the keyword ***core*** followed by the name of the core (*"axififo"* in this case) and a version string (*"1.00.a"* in this case). These two values are used as unique identifier of the core. If Xilinx Toolsuit version 14.6 is used for a driver generation, which is the case for *finance.eit.uni-kl.de* server, core name cannot have upper case charectors.

```
core axififo 1.00.a
{
        . . . . .
}
```

Properties of the core are described within a following declaration block. This block consists of two parts. First is a list of sources that describe the implementation of the core. As it was already mentioned before, when a generated core is used, only a netlist file has to be provided to the driver generator. If both netlist file and .vhd files of the generated core are mentioned as a source, the error message during the process of driver generation will be generated. It is further recommended to use the same name for the source file as for a core name. As a result the core, source file and entity in HDL share the same name.

```
source "vhdl_sources/AxiFifo.vhd"

source "vhdl_sources/Fifo.ngc"
```

The second part of the core declaration block describes the interface of the core. An interface description consists of several port declarations. The keyword ***port*** is followed by a direction specifier (*"in"* or *"out"*) and an identifier (*"s_axis"* and *"m_axis"* in this case). Identifiers have to be locally unique and may therefore occur only once within a core. It is possible, to provide several, comma separated identifiers to declare several ports sharing the same properties.

```
port in s_axis

port out m_axis
```

The ports described using this syntax have to be AXI4-Stream compliant and the ports of the HDL component have to follow a strict naming pattern. The four ports that make up an AXI4-stream have to be suffixed _tdata, _tready, _tvalid and _tlast accordingly. For example, an in-going AXI4-Stream port *s_axis_tvalid, s_axis_tdata, s_axis_tlast, s_axis_tready* (see the reference design in HDL).

Finally, the core requires bindings for the clock and reset ports. The keywords ***clk*** and ***rst*** begin these bindings. Following the keyword is the identifier (*"aclk"* and *"aresetn"* in this case) of the port in the HDL component. For the clock port, the frequency (in MHz) can be specified afterwards. The reset port requires a polarity flag, indicating on which signal the component is reset.

```
clk aclk 100

rst aresetn 0
```

The instance declaration instantiates a core on the board. An instance declaration begins with the keyword ***instance***, followed by a reference to the used core (has to have the same name as during declaration) and the identifier of the instance itself (*"axififo_a"* in this case). Avoid using the same instance name for the core as for any other instance in the design.

```
instance axififo 1.00.a axififo_a
{
        . . . . .
}
```

The keyword ***cpu*** connects the specified port to the board-side driver. The host-side driver will provide methods for direct communication with these ports of the component. Again, a block is used for properties of these driver attached ports. Specifiable properties include queue sizes.

```
cpu s_axis
{
        swqueue 8
}

cpu m_axis
{
        poll 64
}
```

A board-side queue or software queue (*"swqueue 8" in the code listing above*) is introduced for each in-going port on the board in order to decrease communication overhead.

To increase overall throughput, a queue is implemented in hardware and placed in between the component and the stream interface of the processor. These hardware queues provide the component with a new value as soon as required. The current piece of code doesn't include hardware queues.

In the case of polling ports (*"poll 64" in the listing*) the host-side driver explicitly requests values, as soon as a read operation is performed on such a port. So, values are only sent if required. However, polling results in poll messages being sent by the host-side driver, increasing traffic. Therefore, switching of such a port into polling mode should be considered carefully.

3) **Developing client application.** A board is described using multiple components. A component is a designed hardware unit, which can have several ports, over which data can be sent to or from the component. Usually components receive data, process it and send back some results, though on another port.

The host driver contains an abstract generic component, describing components in general. For each user-defined core, a new subclass is created, which contains the ports specified in the core description. For each instance of a core (*"axififo_a"* in the following) on the board, an object of the cores subclass together with all its ports (*"s_axis"* and *"m_axis"* in the following) is instantiated in the driver respectively. Communication with the board's components happens through these port objects.

```
axififo_a.s_axis.write(vector_of_values_to_board);

axififo_a.m_axis.read(vector_of_values_from_board);
```

A port marks an AXI-Stream interface used to send data to or receive data from components. A port is always assigned to a single component, but a component can have multiple ports. Ports can be sending (in-going), receiving (out-going) or bi-directional (dual). These designations in parentheses do not describe the ports on the host-side, but the ports of the driver itself. The user should work with the driver, as if it were the actual board. Consequently, data is sent to an in-going port (*"s_axis"*) and received from an out-going port (*"m_axis"*).

The program has to be started with a special method *startup()* before using the API and should be shut down afterwards with another method *shutdown().* If DHCP settings are used, provide *startup()* function with IP address assigned to the board.

```
int main()
{
        startup("131.246.74.3");

        YourFunction();

        shutdown();
}
```

References:

[1] Thomas Fischer, "HOPP Driver Generator Documentation", Software Technology Group of the University of Kaiserslautern in cooperation with the Microelectronic Design Research Group of the University of Kaiserslautern, July 4, 2013