RIFFA Builder User Guide

Qiongzhi WU

Dec 2015

Rev 0.1

# Concept

RIFFA Builder is a toolset to help people rapidly developing FPGA acceleration applications on PCI-Express based system. Once people give the Verilog algorithm module, RIFFA Builder will generate the FPGA top-level project and host C/C++ program base on the RIFFA PCI-Express core automatically. It means that a FPGA acceleration developers, especially HLS users, do not have to pay much attention on hardware integration and CPU/FPGA communication.



Fig 1 The RIFFA Builder concept

The main input of RIFFA Builder is the user’s algorithm module. It should be Verilog module source file. RIFFA Builder can analyze this v file and get the module’s port information. The source file can be hand-written or output of HLS tool. The Verilog module’s ports definition should obey some basic principles which will be described following. Beside the Verilog file, RIFFA Builder also need some option information inputs, including hardware type (FPGA board), host operation system type and some data type and length information.

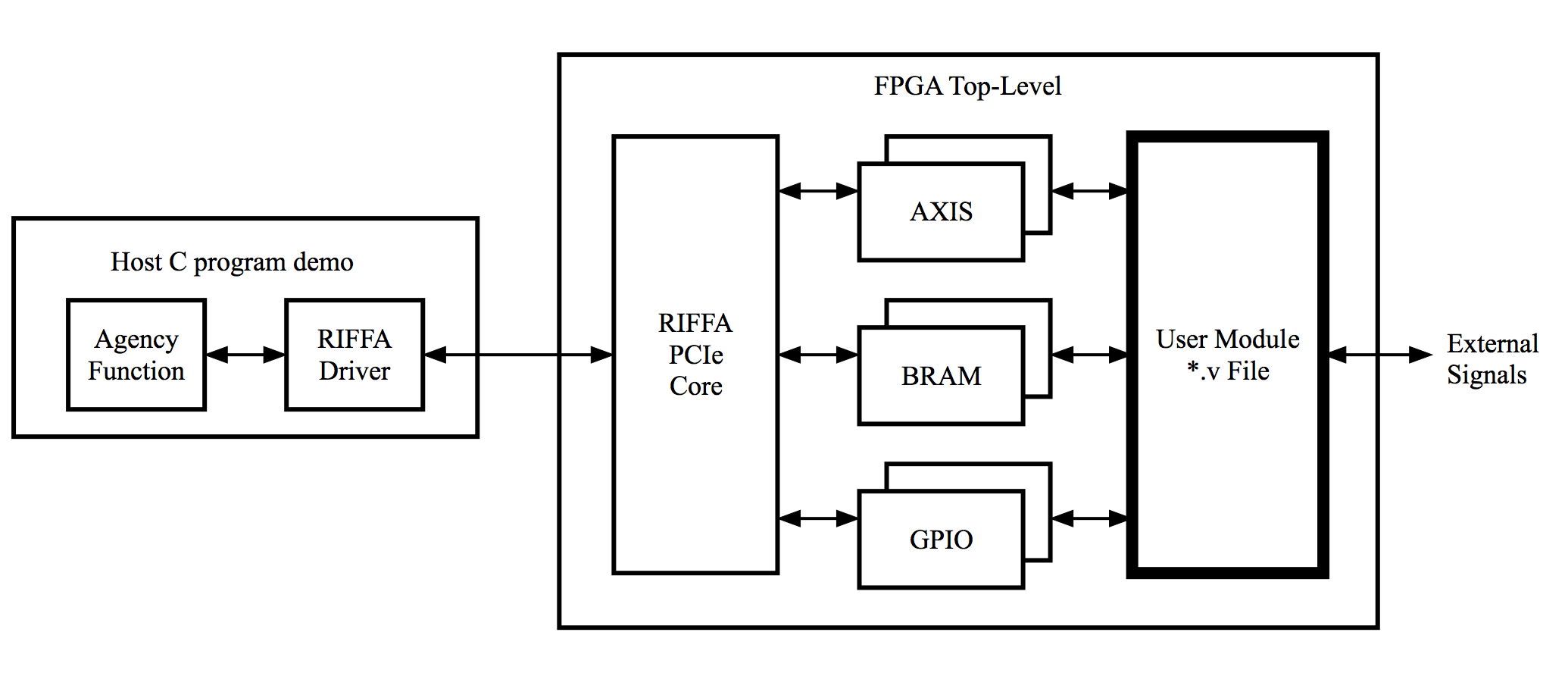


Fig 2 System Architecture

With these inputs, RIFFA Builder will generate a complete system as shown in Fig 2. It includes two main outputs, the final FPGA design and the host program.

The FPGA design is a complete, ready-to building, Vivado project. RIFFA Builder can prepare all the necessary source files into project folder. It will generate the glue logic between the user’s module and the RIFFA PCIe core, and wrap them into a top-level source file. It will also generate the project configuration file. With this auto-generated FPGA project, the only thing that user need to do is to generate the bit file and download.

The host program has to two parts, the Agency Function and the demo main program. The Agency Function is a C function whose parameters correspond with user’s Verilog module. This looks a little bit like an inverse procedure of HLS. Inside the function, RIFFA Builder generate a multi-thread flow to manage all the data transferring and control operations. With this function, user get a direct connection for the C program with the FPGA algorithm module. RIFFA Builder will also generate a demo main() function showing how to call the Agency Function. Users just need to do some small changing on the main() function, for example adjust the input data value.

# Install

* File Location

RIFFA Builder is a group of some Python script files and some design code source files. All these files reside in the folder ‘riffa\_builder’. Put this folder into same location of the RIFFA’s root folder (riffa2.2.0), as shown in Figure 1. User’s target design can be in any other location.

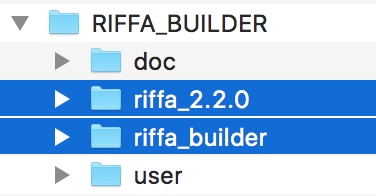


Fig Install Location

* Python

Please install a Python 3.5 environment, for example the IDLE.

# Operation

The file ‘riffa\_builder\_top\_script.py’ is the top script. Edit and execute this script will start the RIFFA Builder’s working. In this file, we can see some user options as shown in Fig 3. Please make sure giving these options correct values before running.



Fig User Options

Table User Options

|  |  |
| --- | --- |
| Option | Description |
| output\_dir | Full path of the user’s target design. RIFFA Builder will create two sub-folders, FPGA\_PRJ and HOST\_PRJ, in this folder. |
| verilog\_src\_file | Full path of the user’s FPGA Verilog module file.  If user’s design has hierarchy, then only set the most top level file. And make sure all the other source files are in the same location. Otherwise you will have to add missing files to the auto-generated FPGA project manually. |
| prj\_start | V0.1 only support ‘Verilog’. Don’t edit it. |
| create\_fpga | Whether you want to create the FPGA design. |
| fpga\_board\_name | The FPGA board type you will use. If not sure, leave this option as an empty string (‘’). Then the program will give a list of supported board types and let you make a choice.  V0.1 was tested on the ‘VC709’ |
| fpga\_template\_name | The RIFFA’s example design name which will be template of your target design. If not sure, leave this option as an empty string (‘’). Then the program will give a list and let you make a choice.  V0.1 was tested on the ‘VC709\_Gen3x4If128’ |
| copy\_fpga\_template | Whether you want to copy the whole FPGA template project to the FPGA\_PRJ.  For the first time, please set it to True.  If you already have the FPGA project and only want RIFFA Builder to update the Verilog top-level source file, set it to False. |
| clk\_div | Clock divide ratio from the RIFFA PCIe core’s user\_clk to user module’s clock. Can be 1 to 255  For example, the VC709\_Gen3x4If128 template‘s user\_clk is 250 MHz. If clk\_div = 5, then user’s module will running in 50 MHz clock. |
| external\_prefix | The name prefix of user module external signals. For example, if external\_prefix = [‘ext\_’, ‘uio\_’], then all the signals starting with ‘ext\_’ or ‘uio\_’ will be considered as external signals and be connected to top-level ports instead of internal wires. |
| create\_host | Whether you want to create the host c program. |
| host\_os | V0.1 only support ‘windows’. Will support ‘Linux’ in the future. |
| host\_platform | ‘x64’ or ‘x86’. It depends on the C/C++ compiler you are using.  V0.1 was tested on the x86 GNU C/C++ compiler |
| host\_template\_type | V0.1 only support ‘Common’. Don’t edit it. |
| copy\_fpga\_template | Whether you want to copy the whole host template project to the HOST\_PRJ.  For the first time, please set it to True.  If you only want RIFFA Builder to update the c source files, set it to False. |
| performance\_counters | Whether insert performance counter in FPGA logic. |
| performance\_counters\_tick\_div | Performance counter clock divider ratio to the user’s module clock. 0=no divide; 1=divide by 2; 2==divide by 4; 3=divide by 8; and so on.  Performance counter’s width is 32-bit by default. If the user module has long running time, you can set this option to let tick running slowly. |
| port\_atrbs | Attributes of user’s Verilog module ports. This is a string array. Every string describes one port. The string format is  ‘*Module Port Type Length Debug*’ (separated by a space)   * Module: The Verilog module name * Port: The port’s prefix name. * Type: The c/c++ type mapping to this port. * Length: The number of data elements. * Debug: ‘debug’ means setup debug strobs in Vivado project. Leave it blank means no debug.   The port attributes will be introduced in detail in the ‘FPGA Programming Model’ section. |
| host\_debug\_level | Set how many debugging information are print out. 0 or 1 means no printing. 2 means printing some. 3 means printing all. |
| hardware\_timeout | How long time, in millisecond, the host program should wait for the FPGA running. |

# The FPGA Project

The RIFFA Builder 0.1 now only support Xilinx Vivado. The project will reside in the subfolder ‘FPGA\_PRJ’ of target path (the user option *output\_dir*). Open this project we can see that the user Verilog module has been a sub module of the top-level, e.g. the func.v in Fig 5.

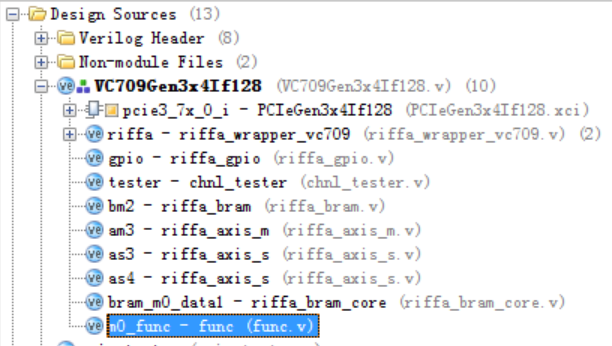


Fig Vivado Project

After open the XPR file with Vivado, following operation may need:

* Upgrade Vivadio Version

Vivado may prompt converting to new version format. That because you are using a newer verison than the RIFFA’s templates.

* Upgrade IP verison

Vivado may prompt upgrade the IP cores, especially the PCIe IP core, to newer version. Just follow it.

* Add additional design files

If user had put all the necessary source \*.v file into the same folder with the top module v file (the user option *verilog\_src\_file*), RIFFA Builder can find and add them to the project. But if there are source file in other location or they are not \*.v file (e.g. a IP core), then we have to add them manually.

* Setup debug strobes

Debug strobes can be insert after synthesis. The user option ‘port\_atrbs’ can let RIFFA Builder makes port signals as ‘debug’. This will make it easy for user to find them when setup the strobes.

Please be careful that running RIFFA Builder may overwrite the whole project or source files. If you only want to regenerate the top-level Verilog file (e.g. VC709Gen3x4If128.v in Fig 5) and the project setting file (xpr file), please set the user option ‘copy\_fpga\_template’ to False.

# FPGA Programming Model

User’s Verilog module should follow some rules to let RIFFA Builder could analyze it. It support the user’s Verilog module with these port types:

* System controlling signals.
* Scalar value input or output ports.
* AXI-Streaming input or output ports.
* BRAM ports.

RIFFA Builder recognizes these ports by signal’s name. For example, if a signal name begins with ‘*ap\_*’, then it is a system control port. If there are three signals with same prefix name and end with \_TDATA, \_TREADY, \_TVALID, then they belong to a AXIS port. Actually, HLS users can find that these naming rules are very similar with Vivado HLS’s RTL output. This will make it easy to integrate HLS and RIFFA.

Table 2 User’s Verilog module port types

|  |  |  |  |
| --- | --- | --- | --- |
| Port Type | Signals | Direction | Describe |
| System ports | ap\_clk | In | The clock for module running.  The user option ‘clk\_div’ can set how fast this clock. |
| ap\_rst\_n / ap\_rst | In | The reset for module. The *\_n* means low value, otherwise high valid. |
| ap\_start | In | Start signal. Once assert 1, the module should begin running.  Optional |
| ap\_done | Out | Done signal. Once the module finish all the calculation, it should assert ap\_done to 1.  Optional |
| ap\_ready | Out | Optional |
| ap\_idle | Out | Optional |
| Scalar input port | *portname*[N-1:0] | In | A Scalar input means a single value input to the module and will be stable for the whole procedure.  Host program will present the scalar value at port before triggering the module running.  Width N could be any value no more than 128. |
| Scalar output port | *portname*[N-1:0] | Out | A scalar output means a single result value output after the module finish the procedure.  Host program will read this value after the module assert the ap\_done.  Width N could be any value no more than 128. |
| AXIS input port | *portname*\_TDATA[N-1:0] | In | Standard AXIS slave port. Width N can be any value no more than 64. But RIFFA will always round it to nearest 8/16/32/64 width. |
| *portname*\_TVALID | In |
| *portname*\_TREADY | Out |
| AXIS output port | *portname*\_TDATA[N-1:0] | Out | Standard AXIS master port. Width N can be any value no more than 64. But RIFFA will always round it to nearest 8/16/32/64 width. |
| *portname*\_TVALID | Out |
| *portname*\_TREADY | In |
| BRAM port | *portname*\_Clk\_A (\*) | Out | RAM port. RIFFA Builder will insert a dual-port RAM between the user module and the RIFFA PCIe core. The dual-port RAM’s A port connects to user’s module and B port connects to RIFFA.  *portname*\_Clk\_A is the clock to drive the RAM. If absents, RIFFA Builder will use the ap\_clk as RAM’s clock.  *portname*\_Rst\_A is reset to RAM. It’s no really usage, only for compatibility with HLS.  *portname*\_Addr\_A is byte address to RAM. Width K could be a large value, e.g. 32. The Builder will use the port attribute ‘length’ of calculate the real address width.  *portname*\_EN\_A is clock enable of RAM.  *portname*\_Dout\_A is data input of RAM.  *portname*\_Din\_A is data output of RAM.  *portname*\_WEN\_A is write enable of RAM. WEN\_A can be multiple bits to make byte write selection.  Signals with (\*) are optional. |
| *portname*\_Rst\_A (\*) | Out |
| *portname*\_Addr\_A[K-1:0] | Out |
| *portname*\_EN\_A (\*) | Out |
| *portname*\_Dout\_A[N-1:0] (\*) | In |
| *portname*\_Din\_A[N-1:0] (\*) | Out |
| *portname*\_WEN\_A[K-1:0] (\*) | Out |

RIFFA Builder automatically generate the FPGA top-level file. All the scalar ports will be mapped to the RIFFA DMA channel 0. Multiple RAM ports can be mapped to one RIFFA channel. Every AXI-Stream port will be mapped to a unique RIFFA DMA channel. As we know, the RIFFA 2.2.0 core has maximally 12 channels. Then, logically, the RIFFA Builder can support up to 11 AXIS input ports, 11 AXIS output ports and almost infinite scalar or RAM ports.

Beside all the information in the user’s Verilog file, RIFFA Builder still need some other attributes to create a correct system. They are:

* Data type of a port  
  64bit data port can be \_\_int64, unsigned \_\_int64 or double. 32bit data can be int, unsigned int or float. 16bit data can be short or unsigned short. 8bit data can be char or unsigned char.
* Data length of port

For AXIS output port and RAM port, RIFFA Builder need to know how long is the data.

These attributes can be set in the user option ‘port\_atrbs’, shown in Table 1. If absent, RIFFA Builder will prompt user to make a choice at runtime.

# The Host Program

RIFFA Builder 0.1 can generate a host c/c++ program as shown in Fig 6, which reside in the HOST\_PRJ subfolder. Execute the compile.bat can compile the program with GNU compiler. There are two outputs: main.exe and rst\_fpga.exe. The main.exe is main program for running the design. The rst\_fpga.exe is only for reset the FPGA, when you think it’s not work well.

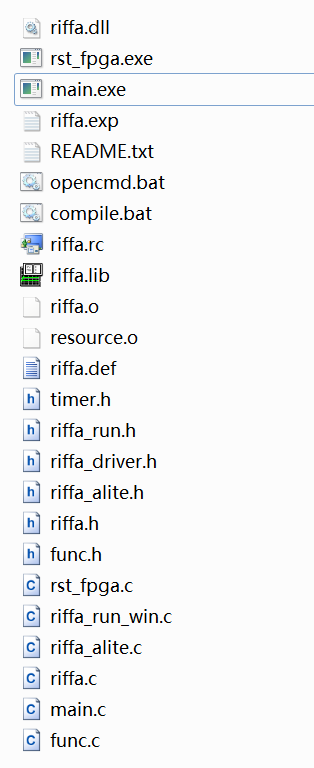


Fig Host Program Files

There are three important source file which worth for look into:

* The main program source file: main.c

This is an auto-generated source file. It shows how to open the hardware, call the agency function and close the hardware.

* The agency function source file: func.c (only for this example)

This is an auto-generated source file. The file name is same as the user’s Verilog module name. There is an agency function in this file, which will be described later.

* The RIFFA mid-level source file: riffa\_run.c

This is a static source file, not generated. The function riffa\_run() shows the control and data flow of the HOST-FPGA co-working.

# Agency Function

For the host program, RIFFA Builder wrap all the PCIe read/write operations into the auto generated C function, called the Agency Function. Agency Function has same name with the user’s Verilog module, and has the parameters almost one-by-one map with Verilog module ports. All the parameter’s data type is selectable by user corresponding to the data width.

For HLS users, the Agency Function will be very friendly. Because their Verilog module is generated from a C algorithm function. With the RIFFA Builder, they will find that the C function appear again with almost same name and parameters. And all the algorithm will run on FPGA by call this new function.

Table Verilog port to C parameter mapping:

|  |  |
| --- | --- |
| Verilog Port | C Function parameter |
| System | None.  All the system signal operation are wrap inside the Agency Function |
| Scalar input | Single data |
| Scalar output | A pointer to a single data |
| AXI-Stream input | A pointer to a buffer |
| AXI-Stream output | A pointer to a buffer |
| RAM | A pointer to a buffer |

All the control operations are packaged in the Agency Function. This makes user don’t have to take care of any control operation. But, understanding the control flow of code generated by RIFFA Builder will very helpful for users to write they algorithm module. The module has following control flow step by step:

1. Function is called.
2. Write all scalar input value.
3. Fill all RAM input data.
4. Assert the ap\_start to let user module begin to run.
5. Startup all the AXIS read/write transactions in same time.
6. Wait for all the AXIS transactions finish.
7. Wait for the ap\_done assert.
8. Read out all RAM output data.
9. Read out all scalar output data.
10. Function return.