ZUMA User's Manual

Alexander D. Brant, Tobias Wiersema, Arne Bockhorn,
Monica Keerthipati, and Nithin S. Sabu

30.05.2020

Contents

1 Introduction

This repository contains the ZUMA FPGA overlay architecture system that was introduced by Brant and Lemieux in 2012 [1, 2] and later extended by Wiersema, Bockhorn and Platzner [3, 4] and several students of Paderborn University. ZUMA is an open-source, cross-compatible embedded FPGA architecture that is intended as an overlay on top of an existing FPGA, in essence an "FPGA-on-an-FPGA." This approach of a virtual FPGA has a number of benefits, including bitstream compatibility between different vendors and parts, compatibility with open FPGA tool flows, and the ability to embed some programmable logic into systems on FPGAs without the need for releasing or recompiling the master netlist.

This manual provides an overview of the ZUMA system and contains the following elements:

1. Instructions on how to prepare the repository and external tools to be able to run the overlay generation flow (Section 2).
2. Instructions on how to operate the basic tool flow of ZUMA (Section 3).
3. Instructions for including a generated overlay into a new or existing FPGA design (Section 5).
4. An in-depth description of the underlying FPGA model of the virtual FPGA and how it corresponds to the build parameters of the ZUMA system, as well as an overview of the generated output files and their layout (Section 6). This should allow you to configure the virtual FPGA to your needs.

The folders included in this repository contain a number of components needed to use ZUMA, as well as examples and tests. The directory structure is as follows:

doc/	Contains this documentation.
example/	Contains a ZUMA preferences file, sample Verilog and timing SDF files, and a script to compile.
external/	Required third party tools as GIT submodules.
misc/	Contains a patch that is required to use (the very old) VPR6 with ZUMA.
source/	Scripts to generate the ZUMA Verilog components and bitstreams.
tests/	Included scripts used to test ZUMA components.
tests/integration/	Python unit tests to automatically assert the correct installation and behavior of the ZUMA scripts.
verilog/	Verilog files used for building a ZUMA system, included platform specific and simulation files.
license.txt	The license under which ZUMA can be used.
Makefile	Global Makefile to prepare a working tool flow for overlay generation.
toolpaths.py	Global path setup to tie in the third party tools – can be adapted if the provided tool submodules shall not be used.

2 Installation

ZUMA scripts are written in Python, and require a valid Python install to run. The scripts are tested with Python 2.7. To just get started, just run

```
make
```

This will fetch and build all required tools and run a unit test that asserts the correct behavior of the tool chain. If this finishes with an *OK*, then your ZUMA copy is ready to be used, and you can skip the rest of this section. If you run into build errors, please refer to the failing tool's GitHub site for help.

2.1 VTR flow

The VTR[1] tool set must also be installed in order to compile with ZUMA. ZUMA does not call the VTR flow directly, but tools thereof and requires files that are generated by them.

[1] https://verilogtorouting.org/

For convenience, a GIT submodule is located under '`external/vtr`' that points to a VTR version of the official VTR GitHub repository[2] that is known to work with this ZUMA version. This included VTR version can be build by just issuing a standard `make` in ZUMA's top directory, or in the '`external`' subdirectory.

Should you want to use an existing VTR installation with theses scripts, you can adapt the variable VTR_DIR that is used in scripts to find the VTR install. To change it globally, update the file '`toolpaths.py`' in the base directory to point to your installation location.

VPR versions prior to 7 (which are thus very old by now) do not automatically dump the routing resource graph and lack a command line switch to do so. Since this file is needed by ZUMA, you need to activate the dumping of this file via a debug switch at compile time. For modern versions of VTR and VPR, you will not need to perform the following steps and can skip to the next section.

For VPR 6 a patch file is located in the directory '`$VTR_DIR/vpr/SRC/route/`', which can be applied to the file '`rr_graph.c`' , by calling:

```
cat (ZUMA dir)/misc/patch.txt (VTR dir)/vpr/SRC/route/rr_graph.c > \
    (VTR dir)/vpr/SRC/route/rr_graph.c'
```

The patch instructs VPR to always dump its routing graph to the file '`rr_graph.echo`'. If using a different version, defining CREATE_ECHO_FILES in '`rr_graph.c`' will enable this functionality.

2.2 Yosys

To enable an automatic verification of the functional equivalence between the generated overlay configuration and the original HDL specification, you additionally need Yosys[3].

For convenience, a GIT submodule is located under '`external/yosys`' that points to a Yosys version of the official Yosys GitHub repository[4] that is known to work with this ZUMA version. This included Yosys version can be build by just issuing a standard `make` in ZUMA's top directory, or in the '`external`' subdirectory.

Should you want to use an existing Yosys installation with theses scripts, you can adapt the variable yosysDir that is used in scripts to find the Yosys install. To change it globally, update the file '`toolpaths.py`' in the base directory to point to your installation location.

3 Running the Tools

3.1 Running the Example

Calling the Python script

```
compile.sh test.v
```

will automatically build the ZUMA system Verilog '`ZUMA_custom_generated.v`', and a bitstream hex file '`output.hex`', which can be used to synthesize and configure a ZUMA system. By passing other circuit files, modifying the example ZUMA configuration file '`zuma_config.py`', or providing an alternative configuration file via the *--config* command line switch, custom architectures and bitstreams can be generated.

3.2 ZUMA Tool Flow

The detailed flow of tools to generate ZUMA overlays and configurations for them is depicted in Figure 1, adapted from [4].

[2] https://github.com/verilog-to-routing/vtr-verilog-to-routing

[3] http://www.clifford.at/yosys/

[4] https://github.com/YosysHQ/yosys

It roughly works as follows: Starting with the ZUMA parameters in the '`zuma_config.py`' the `compile.sh` uses the templates stored in '`source/templates/`' to generate the architecture description of the overlay in VTR's XML format. Leveraging this architecture description and the virtual circuit, e.g., '`test.v`', the VTR flow (or more specifically ODIN II, ABC, and VPR) can deduce the complete routing resources of the overlay, which are saved into a file (custom format in VTR ≤ 7 and XML in VTR 8), and can also synthesize, place and route the virtual circuit to the described architecture, resulting in descriptive files for the netlist, the placement and the routing. The ZUMA scripts take all of these generated files and compute the correct configuration each programmable entity of the overlay, i.e., eLUTs and programmable interconnect points, and save the complete virtual configuration as ZUMA bitstream into the files '`output.hex`' and '`output.hex.mif`'. While the former is the correct bitstream version as defined for ZUMA, the latter is an undecorated collection of only the configuration bits and nothing more, ready-to-use for inclusion by vendor tools as memory content.

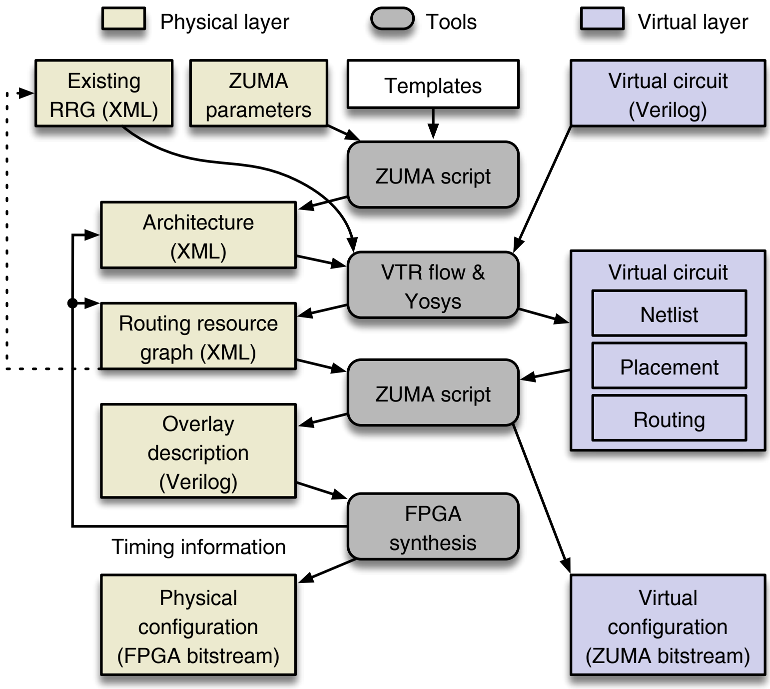


Figure 1: The tool flow to generate overlays and configurations.

For the physical side of things, the ZUMA scripts generate a description of the complete overlay fabric in Verilog, '`ZUMA_custom_generated.v`', using LUTRAM instantiation macros to define all programmable entities. This Verilog file can be included into a regular FPGA project to actually synthesize an overlay onto a physical device. For more details of this inclusion, see Section 5.

3.3 ZUMA Tool Flow Details

The following sections highlight some flow details that happen automatically when running `compile.sh` and may thus be skipped by impatient readers.

3.3.1 Generating the ZUMA Overlay Description

To generate the Verilog architecture that can be synthesized to an FPGA, configuration files are generated for the VTR tools, which are then used to generate the global routing graph used in the ZUMA architecture.

`generate_buildfiles.py` generates the VPR architecture files and build scripts used during the compilation. It takes two arguments, a template input folder and an output folder. The script reads each template file in the input folder and substitutes specific keywords with the provided ZUMA parameters. This will generate the architecture file for VPR.

VPR is then run (any BLIF file can be used), which outputs the routing resource graph, as well as the netlist, placement and routing files.

To generate the ZUMA system, `zuma_build.py` is called with the following parameters:

```
python zuma_build.py
    -graph_file 'rr_graph.echo'
    -blif_file 'abc_out.blif'
    -place_file 'place.p'
    -route_file 'route.r'
    -net_file 'netlist.net'
    -bit_file 'output.hex'
    -blif_out_file 'zuma_out.blif'
    -verilog_file 'ZUMA_custom_generated.v'
```

The filenames above are the default filenames used, and do not have to be specified if they are the same. The graph, BLIF, place, route and netlist files are all generated by earlier CAD steps, and are needed to generate the ZUMA configuration.

The 'bit' file is the ZUMA bitstream in hex format, if this parameter is not specified the bitstream generation is not performed. The Verilog file is the design file of the complete ZUMA system, that is used with other provided HDL files to run the ZUMA system. The '`blif_out`' file is a BLIF file which corresponds to the ZUMA architecture configured to the loaded design, which can be verified to be equivalent to the input BLIF using ABC.

3.3.2 Compiling to the ZUMA architecture

Generating a bitstream for the ZUMA architecture is done in the same way as generating the architecture Verilog. Call `zuma_build.py` as above with the correct input files, and if a name for the output hex file is specified, a new bitstream will be generated.

3.4 Bit to BLIF

You can also reverse ZUMA's virtual synthesis and (re-)build a BLIF file from the generated bitstream. To this end, you can call the script

```
>example/extract_logic_function.sh output.hex.mif output.blif HasClock HasReset
```

where '`output.hex.mif`' is the bitstream you want to build your BLIF from, '`output.blif`' the name of the BLIF file you want to create and *HasClock* and *HasReset* are two booleans [True/False] which indicate if the circuit uses a clock and / or a reset signal. Those two signal properties cannot be read from the bitstream so you have to specify them.

Additionally the script requires the same '`zuma_config.py`' architecture parameter configuration that was used to build the '`output.hex.mif`' initially, because the architecture details can also not be read from the bitstream.

3.5 Timing Analysis

Note: This section is currently only applicable to Xilinx devices.
First you have to generate a ZUMA overlay with a specific architecture described by your '`zuma_config.py`' configuration. For this generation the parameter *params.sdf* has to be turned off. Then you can integrate the overlay into your design (cf. Section 5) and extract the timing information from it by generating two SDF (standard delay format) files.

Assuming that your toplevel design has the name *Top*, generate the first SDF file that contains the the routing delay information by issuing:

```
>netgen -s 1  -pcf Top.pcf -sdf_anno true -sdf_path "netgen/par" \
-ne -insert_glbl true -insert_pp_buffers false -w \
-dir netgen/par -ofmt verilog -sim Top.ncd Top_no_buffer.v
```

Then generate the second SDF file that holds the flip flop delays (port delay + Tshcko):

```
>netgen -s 1  -pcf Top.pcf -sdf_anno true -sdf_path "netgen/par" \
-ne -insert_glbl true -insert_pp_buffers true -w \
-dir netgen/par -ofmt verilog -sim Top.ncd Top_with_buffer.v
```

Copy the two generated files '`Top_with_buffer.sdf`' and '`Top_no_buffer.sdf`' to your '`example/`' directory and edit the parameters *params.sdfFileName* and *params.sdfFlipflopFileName* of your configuration. Also you have to reactivate the *params.sdf* flag and edit the instance parameter.

After performing these steps, you can run the ZUMA '`compile.sh`' script with any virtual circuit to get its critical path. The path and resulting frequency f_{max} will be printed on the command line.

4 Caveats and Restrictions

A constraint for the virtual circuit file is that the head of the model must have the following signature:

```
verilog-module-name ([clock], reset, [input-name-1, input-name-2 , ... ])
```

The clock and reset signal must have the given names and positions for the scripts to recognize their special behavior. The reset is treated as the first input on the FPGA. The declaration of a clock is optional.

The LUTRAMs used in the ZUMA architecture and contained in this repository are generated using Xilinx and Altera macros. To use a different LUTRAM or memory, instantiate it in the file '`lut_custom.v`', and define a new platform (e.g., PLATFORM_STRATIXIV) in the file '`define.v`' to select this LUTRAM.

Note that the Altera macros have not been maintained and that for the current ZUMA version thus only the Xilinx side is tested and guaranteed to work with current vendor tool flows. Especially support for sequential virtual circuits has so far only been implemented for Xilinx projects.

5 Including a ZUMA Overlay in a Project

Once the Verilog architecture is created, and a hex bitstream is generated, the ZUMA system can be compiled and used. The generated Verilog file that describes the virtual fabric, along with the files in the '`verilog/generic/`' and '`verilog/platform/(platform)/`' directories should be included in a new Xilinx / Altera project, although getting it to work for Altera devices might require some (read: significant amount of) additional work. We will describe the intent and general process here, for specific details on how to include it in a Xilinx Vivado project, please refer to Section 5.1.

The generated hex file should be placed in the project directory, and specified as the initial contents of the ZUMA configuration memory. The top level file '`ZUMA_TB_wrapper`' includes a memory block which references the hex file '`output.hex`' that should be generated by the ZUMA tools and included with the project. Note that sequential circuits are so far only supported in Xilinx projects.

If you change the configuration memory size, or the configuration width, you have to create an appropriate new memory using the vendor tools. Runtime configuration of the ZUMA overlay is performed by loading each block of memory in the hex file to the port `config_data`, along with its address to config_addr, and asserting the corresponding config_en port. Configuration completes when all data are loaded.

The ports fpga_inputs and fpga_outputs provide the interface between the physical and virtual FPGA logic. As described in the original ZUMA paper [1] and Brant's master's thesis [2], the pins are located at the edges of the array, begin at the grid coordinate (0,1), and increase in the Y direction first. The pins can be fixed to correspond to those of the input Verilog by specifying a pin location file when running VPR placement.

If you want to use the timing analysis or want to build a BLIF from a hex file, see the timing and bit to BLIF readme files.

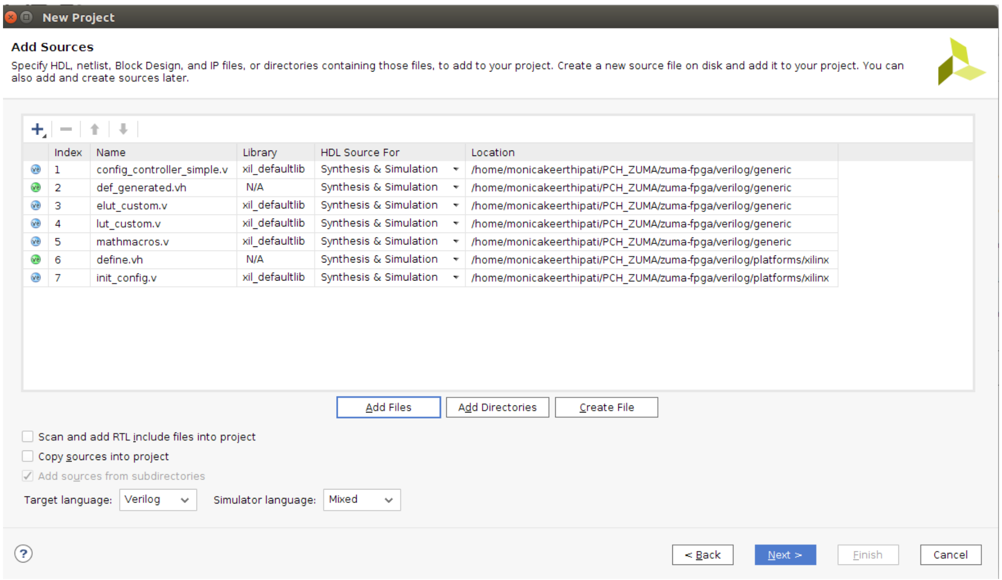
5.1 Including a ZUMA Overlay in a Xilinx Vivado Project

For the inclusion of a generated ZUMA overlay into a Xilinx Vivado project, we provide a more detailed explanation here along with screenshots. When following these steps, you should be able to synthesize a working, configurable ZUMA overlay with your project.

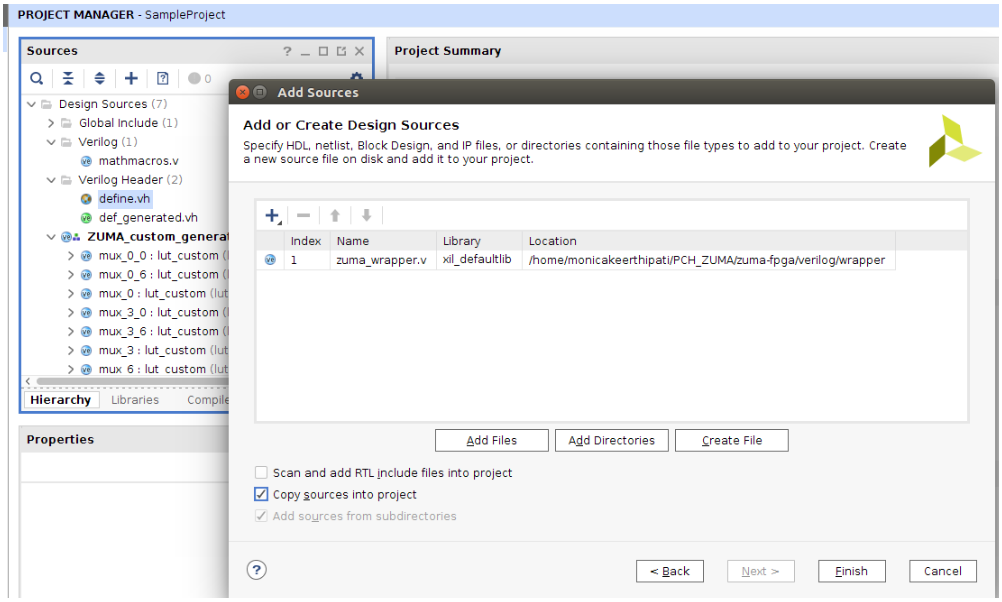
1. Create a new project in Vivado by including the source files in the following directories:

   - `verilog/generic`
   - `verilog/platforms/xilinx`

   Do not copy the files into the project, as they will be common to all projects.



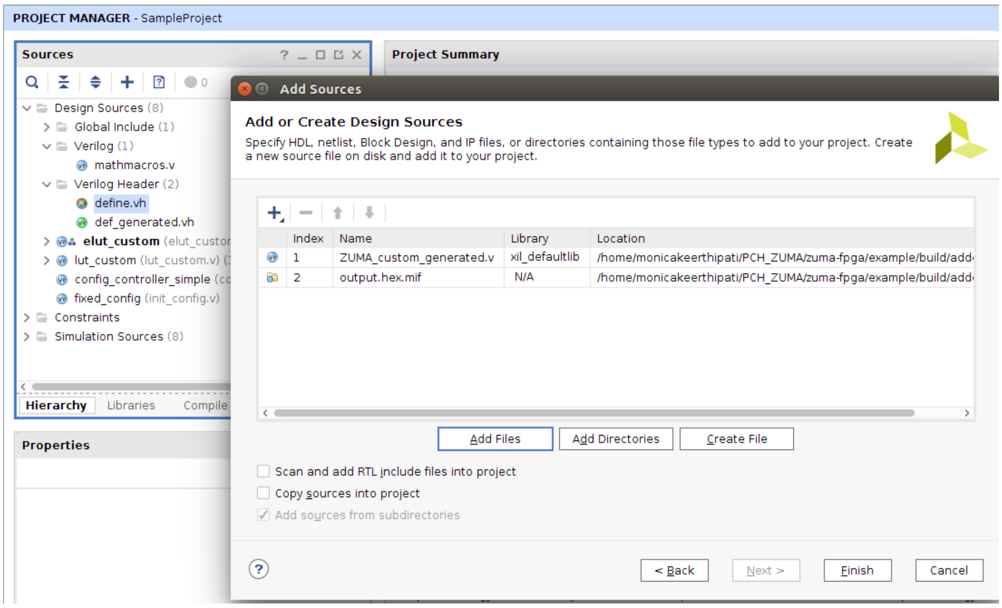
   You might want to copy the file '`verilog/generic/ZUMA_TB_wrapper.v`' to the Vivado project, however, since this will be the top test bench for the project we are building here, so you might want to adapt this.



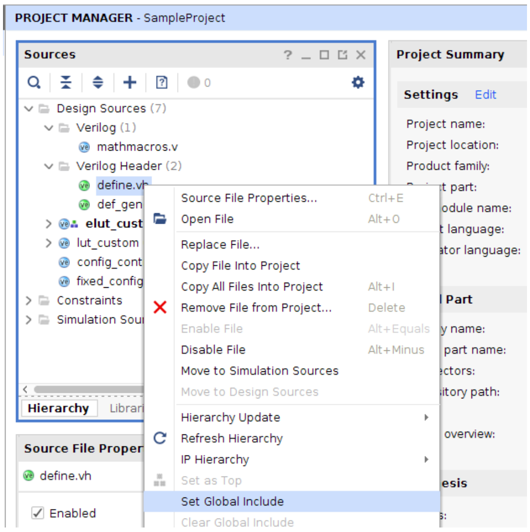
2. Click on *Add sources* and add the following source files also to the project:

    - `example/ZUMA_custom_generated.v` – overlay description
    - `example/output.hex.mif` – virtual configuration
    - `example/def_generated.vh` – generated header file

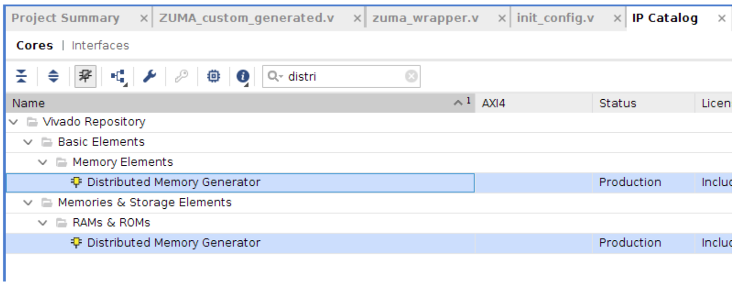
    This time, you can choose whether or not to check *Copy sources into project*. While these files change for each project, and thus could be copied, not copying them will force Vivado to read them from the ZUMA directory, so that when you regenerate them, they should be included in the new version. Should you, however, generate overlays for multiple projects in your ZUMA directory, it would be safer to copy them into the Vivado project.

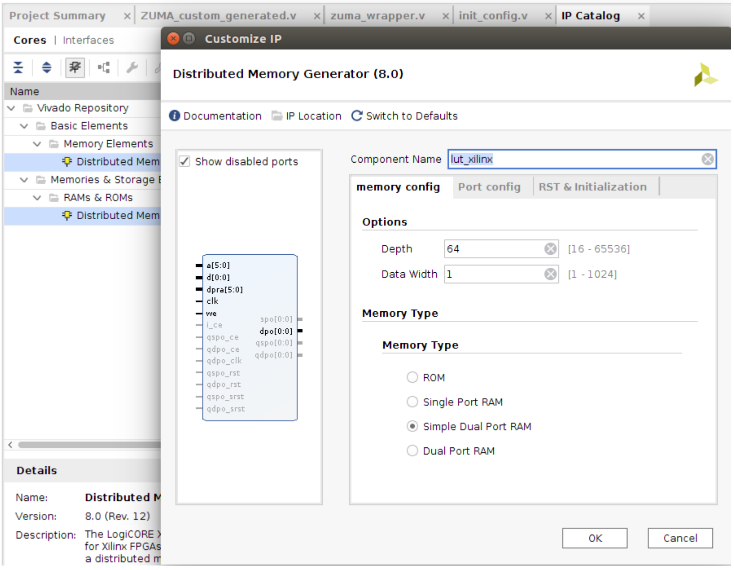


3. Right click on ‘`define.vh`’ and click *Set Global Include* as shown.
    Do the same for ‘`def_generated.vh`’.

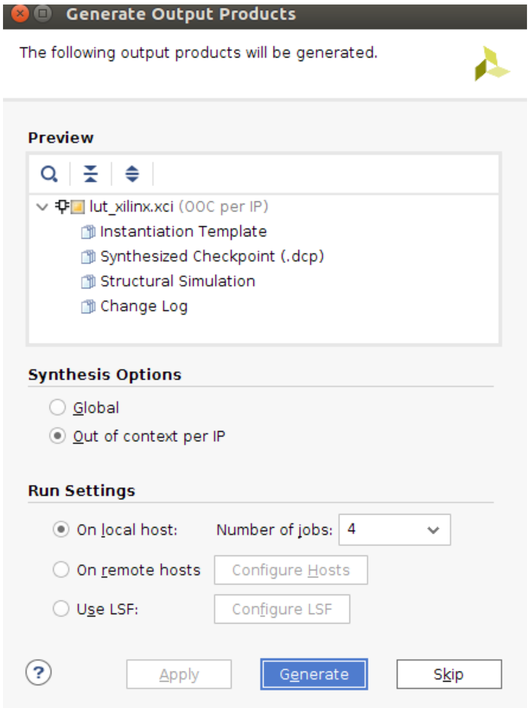


4. From the IP catalog, select the *Distributed Memory Generator* core as shown. Change the component name to `lut_xilinx`. Set *Data Width* to 1. Change *Memory Type* to *Simple Dual Port RAM* and click *OK*. This will include the basic building block of ZUMA into the project – the LUTRAMs – which the overlay will instantiate a hundredfold.

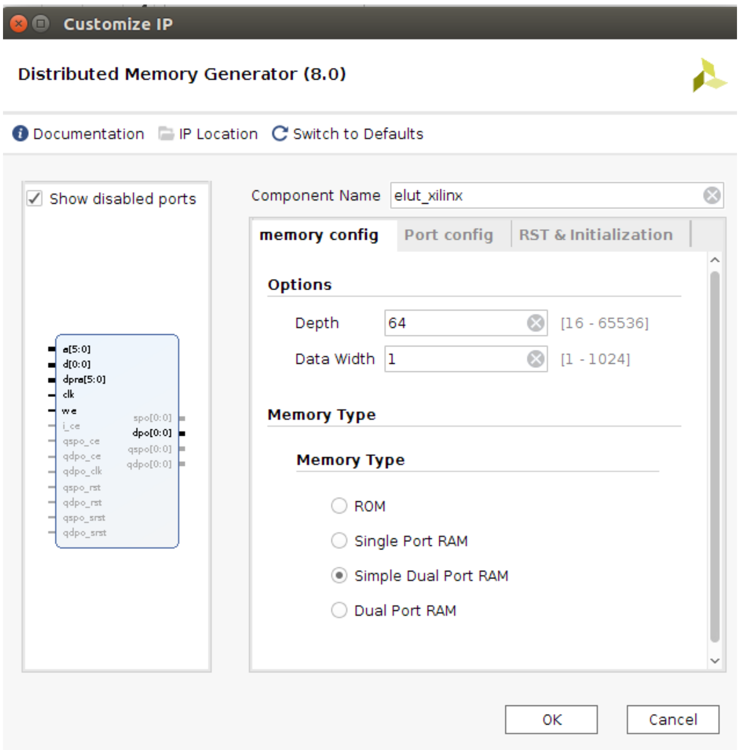




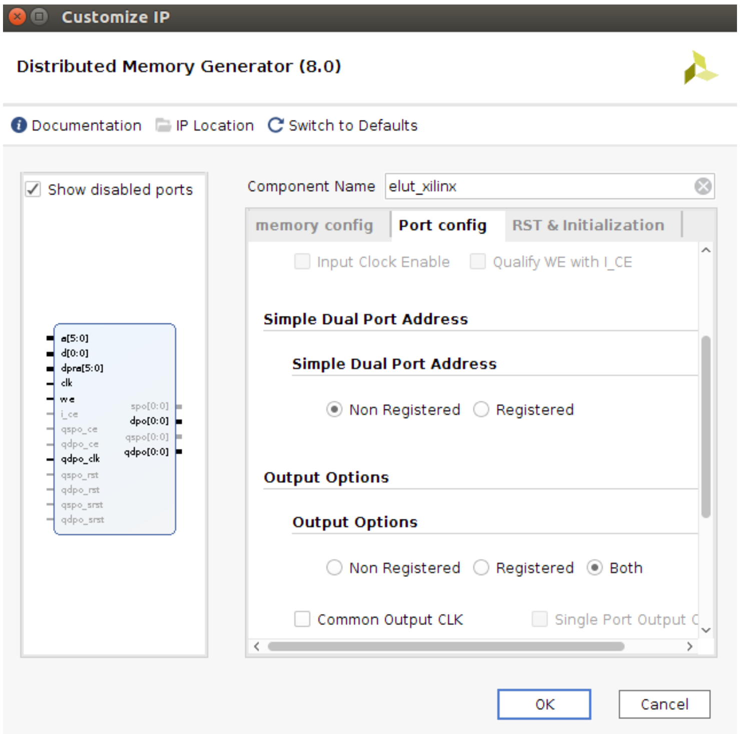
5. In the *Generate Output Products* window that appears, set Synthesis Options to *Out of context per IP* and click *Generate*.



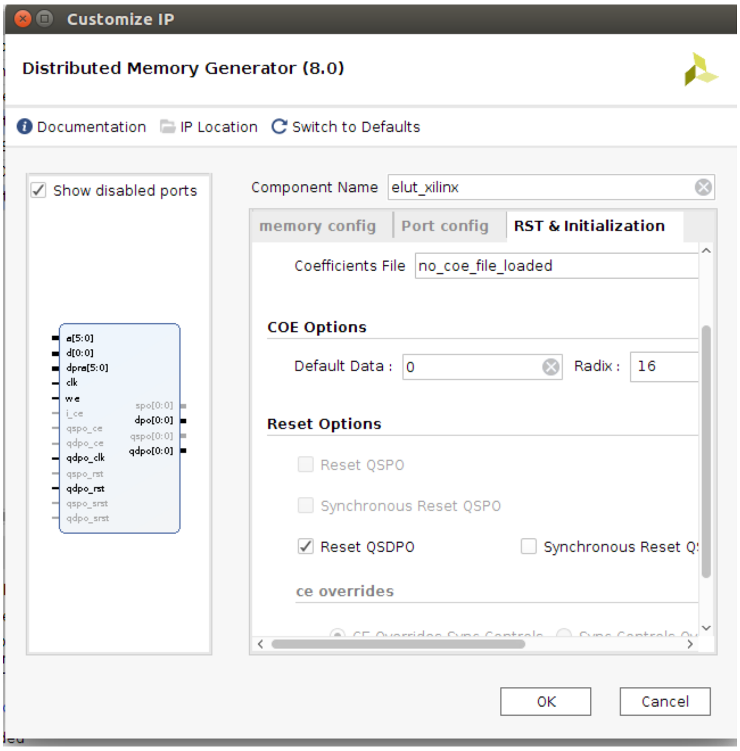
6. Since the implementation of virtual sequential circuits, ZUMA requires a special version of this building block for its eLUTs. Hence, add the *Distributed Memory Generator* core a second time now, but this time with different settings. Change the component name to `elut_xilinx`. Set *Data Width* again to 1, and change *Memory Type* also to *Simple Dual Port RAM*. Do NOT dismiss the dialog, but proceed to the next tab now.



In the *Port config* tab, set *Output Options* to *Both.* Proceed to the final tab.



In the *RST and Initialization* tab, check *Reset QSDPO* in *Reset Options.*



Now you can finally click *OK* and in the next window (*Generate Output Products*) click *Generate* again with *Out of context per IP* settings.

7. Modify the 'ZUMA_TB_wrapper.v' file based on your needs. It will be a good idea to remove the inputs and outputs from external ports and expose only part of it to the interface, since all ZUMA IO pins are general purpose IO pins and can thus be configured to be either an input or an output by the virtual configuration. The provided `fpga_inputs` and `fpga_outputs` are thus exactly twice the number of actually available IO pins, and there will not be enough pins to map all the inputs and outputs should you attempt to fully use both arrays for one configuration. Also, make sure to tie the unused inputs to ground, as otherwise, simulation will break.

5.1.1 Troubleshooting

Issue 1 *Synthesis of the complete design is nearly impossible, since Vivado finds thousands of combinational loops.*

This might happen when you have a project that is a Vivado block design containing several IP cores, where one of them acts as the wrapper and configuration controller for the ZUMA overlay, and your follow the guide in this section to instantiate the customized *Distributed Memory Generator* IP within. This will probably seem to work fine, but the synthesis of the complete design can be nearly impossible as Vivado complains about thousands of combinational loops, crashing after running out of memory. This will then happen with the block design set to global synthesis, as well as with out-of-context synthesis.

As it turns out, it is necessary to run out-of-context synthesis for each LUTRAM module. This way they are considered black boxes during final synthesis and timing loops are not reported. Unfortunately, Vivado has some serious limitations regarding nested block designs. The out-of-context synthesis products generated within the ZUMA wrapper IP are not recognized by Vivado, because nesting pre-synthesized IP cores in this manner is not supported. Some additional information can be found at https://forums.xilinx.com/t5/Design-Entry/Limitations-of-the-Block-Designs/td-p/553937

This problem can be solved by instantiating the LUTRAM modules from an .edif netlist. These can be generated by customizing the *Distributed Memory Generator* IP in a new Vivado project, generating the output products, opening the resulting .dcp file with Vivado, and using the `write_edif` tcl command. This way the LUTRAM can be instantiated by simply including this .edif as a source file. An HDL stub definition of the module is needed though, at least for Verilog. This can be generated using the `write_verilog -mode port` command. Details of this solution can be found at https://forums.xilinx.com/t5/Design-Entry/Adding-xilinx-IP-dcp-files-for-packaging-custom-IP-to-speed-up/td-p/603142 and also at https://www.xilinx.com/support/answers/54074.html.

6 Background

This section's purpose is to give you enough insight into the ZUMA overlay structure and the generated files, so that you can tweak the generated virtual FPGAs to best fit your needs.

6.1 Basic FPGA Model Used by ZUMA

Since ZUMA derives its FPGA model from architectures defined using the VTR tool flow, we will use their notions and general model division here. On the most abstract level, ZUMA thus uses an island-style FPGA layout as depicted in Figure 2, i.e., logic block islands floating on a sea of interconnect, which is also called the Toronto FPGA model. VTR usually denotes these logic blocks as as *configurable logic blocks (CLBs)*, and within the ZUMA material they are often simply called *clusters*.

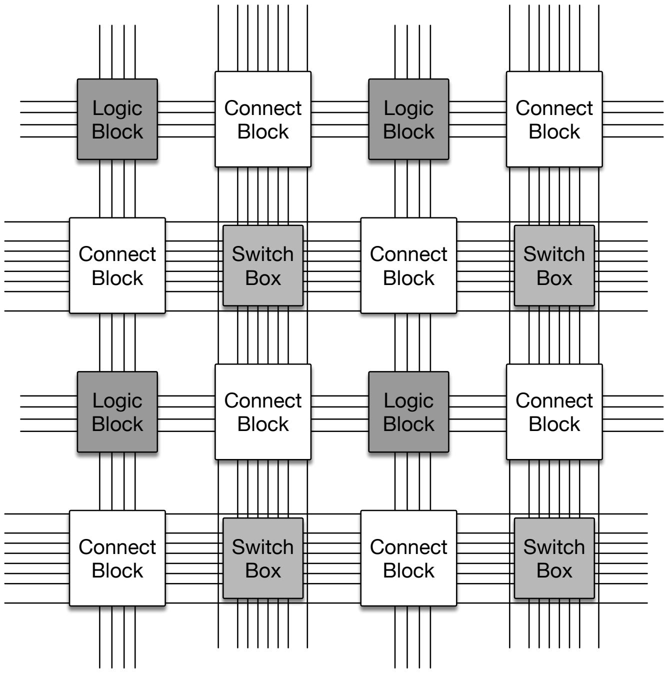


Figure 2: General island-style FPGA layout. Source: [5].

For our explanations, we consider the virtual FPGA structure on two different levels:

1. The global structure and layout with all the interconnect between the CLBs.
2. The local structure and layout within each CLB.

6.1.1 Global Structure

The outer, global structure of the generated virtual FPGAs consists of an $X \times Y$ array of CLBs as depicted in Figure 3. The inputs and outputs of the virtual device are modeled on the edges of the grid, resulting in $2 \cdot (X + Y)$ IO pads. For ZUMA, each of these IO pads comprises two general purpose IOs, i.e., IOs which can be configured to be either a global input or a global output, such that the virtual device has $\#GIOs = 4 \cdot (X + Y)$. Thus, the overlay can divide the $\#GIOs$ general purpose IOs between the global inputs and outputs as required by the current circuit.

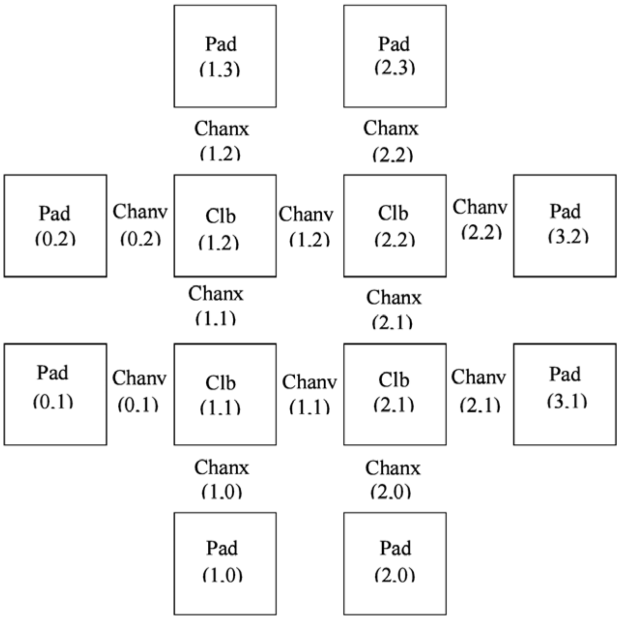


Figure 3: Global structure of a 2×2 virtual FPGA. Source: VPR manual.

The routing resources are organized as routing channels in x or y direction (*Chanx* and *Chany*), and they form a unidirectional interconnect network between the CLBs and the outer IOs. Each channel consists of a number of individual tracks that can carry one logical signal each. The channels and tracks are visualized in Figure 4.

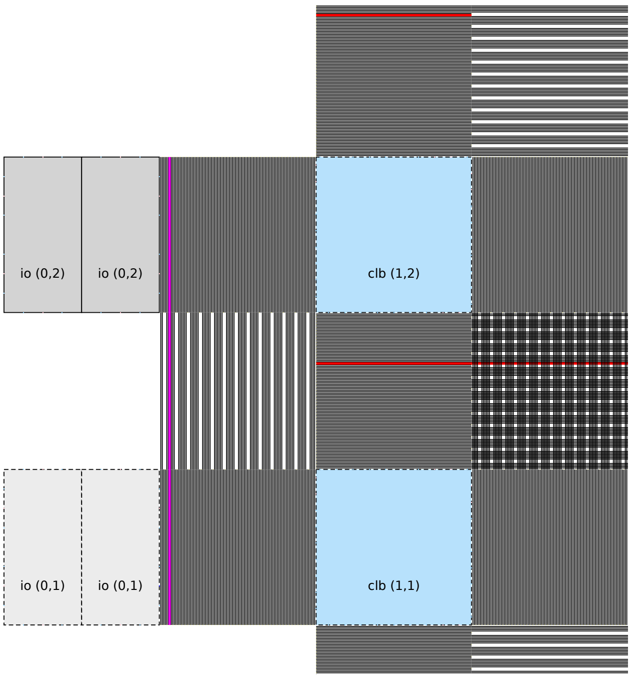


Figure 4: Tracks of different lengths and directions in channels: The purple track connects resources vertically and has a length of 2, while red ones are horizontal and the upper one has a length of 1. Source: VPR.

The channels are connected to each other via switchboxes, which are shown in Figure 5. Within these areas, a selection of tracks from each channel can be connected to a selection of other tracks of different channels. Since allowing the complete connection of any track to any other track would be too area consuming, ZUMA overlays, like many other FPGA devices, employ a crossbar pattern known as Wilton routing [6] here.

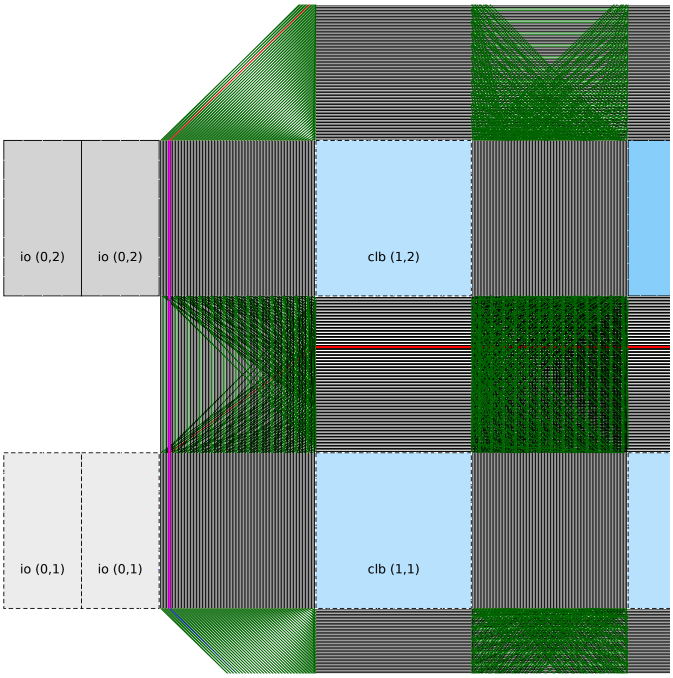


Figure 5: Tracks of channels are connected by the switchboxes (green). Source: VPR.

Each CLB connects to some of the tracks of the surrounding channels of the global routing resources – these connection locations are typically called connect blocks (cp. Figure 2). Each CLB element thus has its own connect block to connect itself to the global routing resources, and a switchbox to actually realize the global routing. Therefore the global outer structure of a ZUMA virtual FPGA consists of the global routing network (channels, connect blocks, and switchboxes), CLBs (or clusters), and IO pads.

6.1.2 Local Structure

Each CLB, or cluster, consists of an input interconnect block (IIB) for its intra-cluster input routing and N basic logic elements (BLE). These BLEs in turn comprise one lookup table (LUT) with input width K, and one flip flop (FF) that is bypassable using a configurable MUX, see image 6.

The N bit output of the whole cluster is the combined output of the N individual BLEs. The IIB is fed with the I inputs from the connect block, i.e., the connected input tracks from the global routing resources, and also with the N feedback outputs from the BLE elements. Each of these $(I + N)$ inputs must be routed to (almost) every pin of all N LUTs. Therefore the $N \cdot K$ outputs of the IIB are connected to the different input pins of the BLE elements.

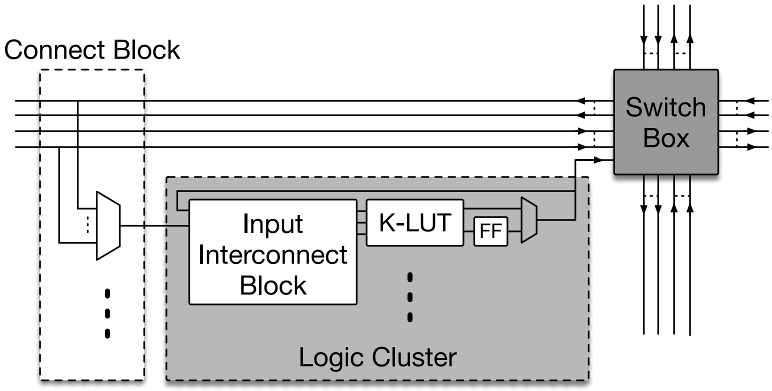


Figure 6: The structure of a ZUMA CLB, or cluster.

Currently the IIB is implemented using connected MUXes, and there are two different implementations available to choose from, each with a different MUX density:

1. The first IIB is a straightforward fully-connected crossbar between the $(I + N)$ inputs and the $N \cdot K$ outputs, which requires a considerable amount of MUXes to realize, but cannot suffer from congestion and is thus guaranteed to find a local routing for any configuration.

2. The second one is based on Clos networks [7] and uses fewer MUXes, but the local routing algorithm does not always find a valid interconnect routing, due to randomness in the current routing approach.

6.1.3 Structure Configuration Parameters

The structure-related ZUMA configuration parameters are thus as follows:

Global structure	
X	Grid size in x dimension
Y	Grid size in y dimension
L	Length of the routing channels
W	Number of tracks per routing channel
Local structure	
I	External cluster inputs (from connect block to IIB)
N	LUTs per cluster
K	LUT input width
UseClos	Whether the IIB should be Clos network-based (otherwise it is a fully connected crossbar)
Connect block	
fc_in	From how many tracks of the connect block each of the I cluster inputs can be driven
fc_in_type	Whether *fc_in* is an absolute number or relative value
fc_out	How many tracks of the connect block each of the N cluster outputs can drive
fc_out_type	Whether *fc_out* is an absolute number or relative value

References

[1] Alexander D. Brant and Guy G. F. Lemieux. ZUMA: An open FPGA overlay architecture. In *International Symposium on Field-Programmable Custom Computing Machines (FCCM)*, pages 93–96. IEEE, 2012. doi:10.1109/FCCM.2012.25.

[2] Alexander Dunlop Brant. Coarse and fine grain programmable overlay architectures for FPGAs, 2 2013. URL `http://hdl.handle.net/2429/43918`.

[3] Tobias Wiersema, Arne Bockhorn, and Marco Platzner. Embedding FPGA overlays into configurable systems-on-chip: ReconOS meets ZUMA. In *2014 International Conference on ReConFigurable Computing and FPGAs*, pages 1–6. IEEE, 12 2014. doi:10.1109/ReConFig.2014.7032514.

[4] Tobias Wiersema, Arne Bockhorn, and Marco Platzner. An architecture and design tool flow for embedding a virtual FPGA into a reconfigurable system-on-chip. *Computers and Electrical Engineering*, 55:112–122, 2016. doi:10.1016/j.compeleceng.2016.04.005.

[5] Katherine Compton and Scott Hauck. Reconfigurable computing: A survey of systems and software. *ACM Computing Surveys*, 34(2):171–210, 6 2002. ISSN 0360-0300. doi:10.1145/508352.508353.

[6] Steven J. E. Wilton. *Architectures and Algorithms for Field-Programmable Gate Arrays with Embedded Memories*. PhD thesis, 1997.

[7] Charles Clos. A study of non-blocking switching networks. *The Bell System Technical Journal*, 32(2):406–424, Mar 1953. doi:10.1002/j.1538-7305.1953.tb01433.x.