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 get started with 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. Details on advanced usage scenarios, 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 generate advanced, custom-built virtual FPGAs that exactly fit 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.

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 Getting started

3.1 Running the Example

Calling the Python script

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

[1] https://verilogtorouting.org/
[2] https://github.com/verilog-to-routing/vtr-verilog-to-routing
[3] http://www.clifford.at/yosys/
[4] https://github.com/YosysHQ/yosys

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

An overview of the flow of tools required to generate ZUMA overlays, and configurations for them, is depicted in Figure 1, adapted from [4].

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 $\leq$ 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 for 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. For a more in-depth, step-by-step discussion of the flow, and for using advanced features such as static timing analysis or timing-driven placement and routing of virtual circuits, see Section 6.2.

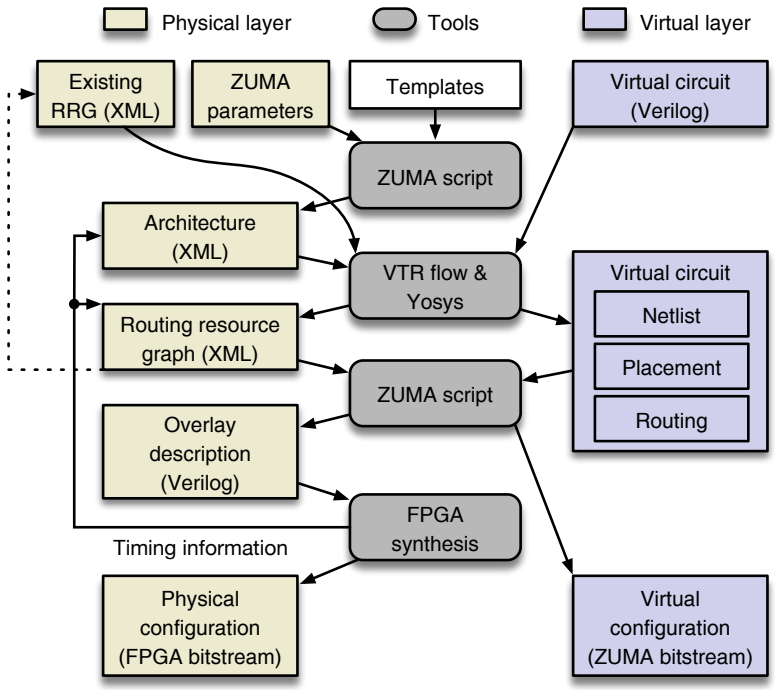


Figure 1: The tool flow to generate overlays and configurations.

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 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.

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.

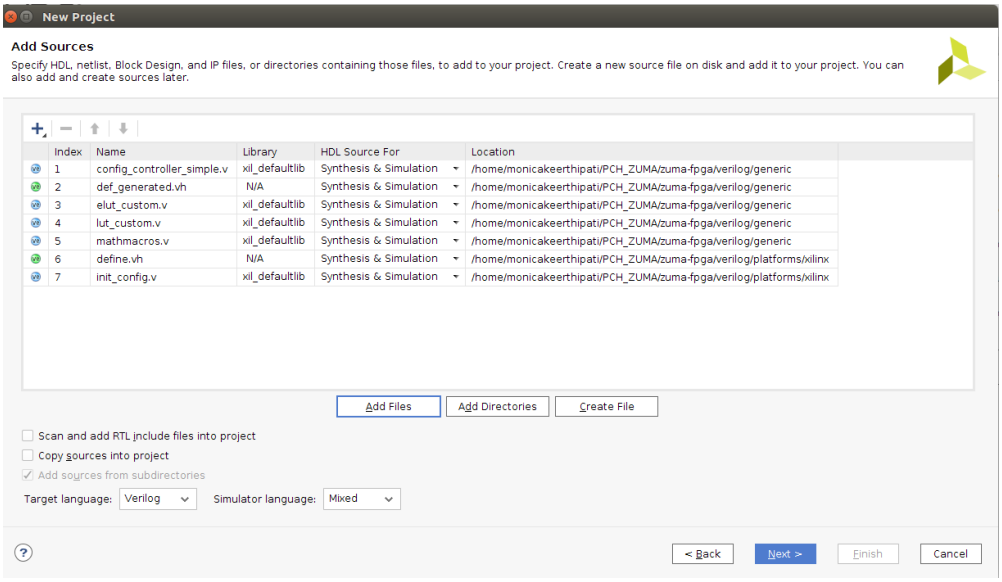
5.1 Including a ZUMA Overlay in a Xilinx Vivado Project

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.

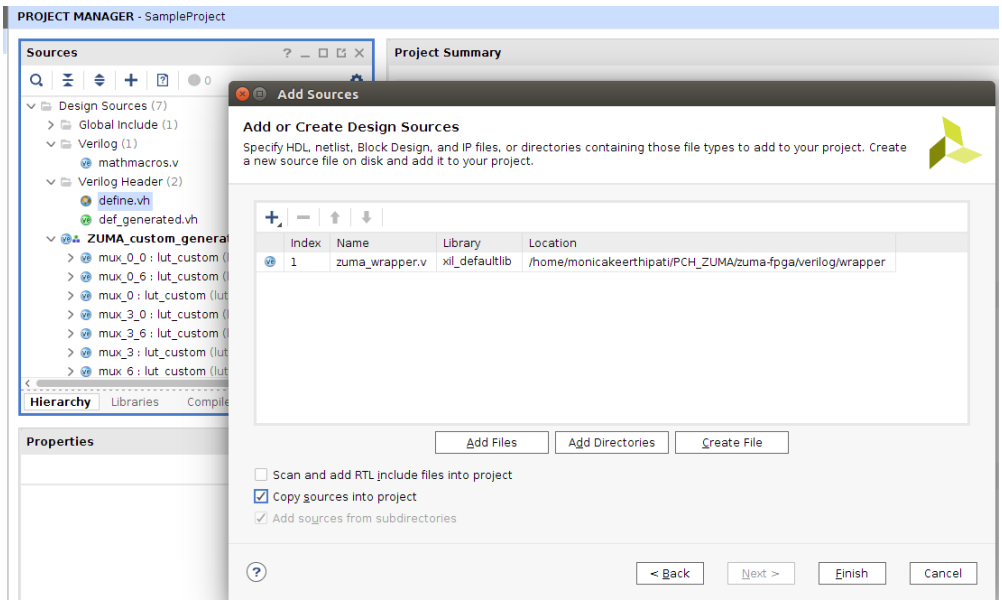
1. Create a new project in Vivado by including the source files in the following directories:

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

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



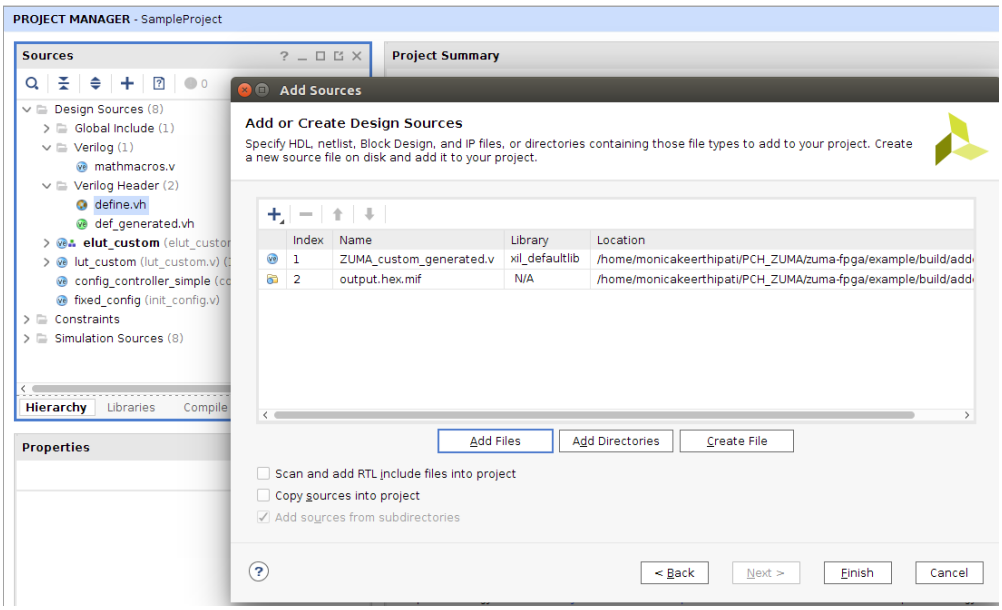
You might want to copy the file '`verilog/generic/ZUMA_TB_wrapper.v`' to the Vivado project, however, since this will be the top test bench for the project we are building here, so you might want to adapt this.



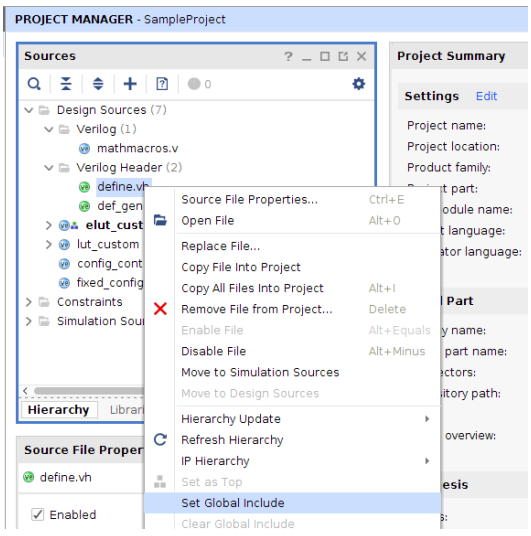
2. Click on *Add sources* and add the following source files also to the project:

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

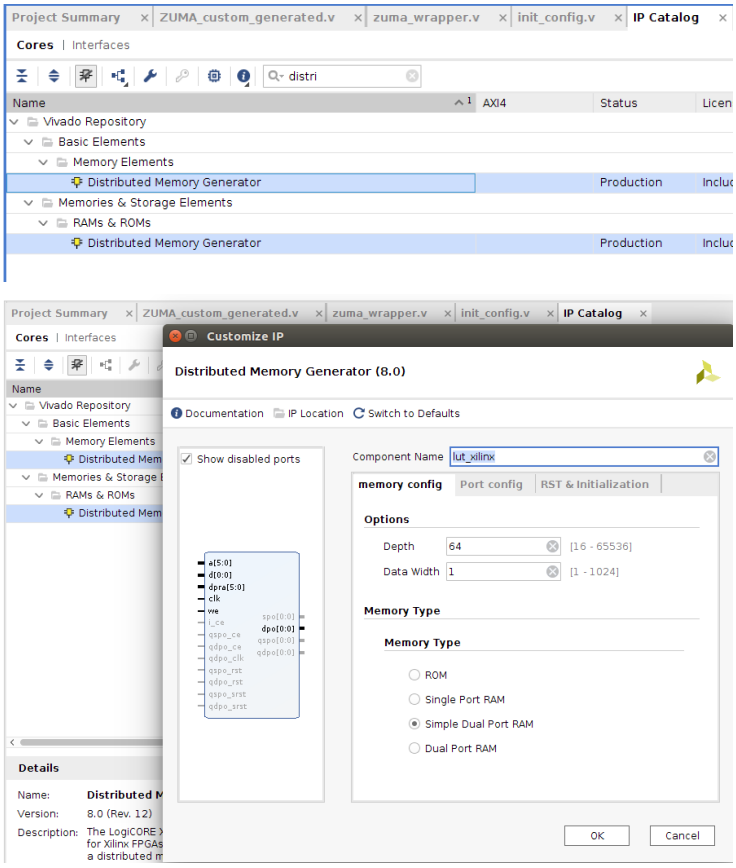
   This time, you can choose whether or not to check *Copy sources into project*. While these files change for each project, and thus could be copied, not copying them will force Vivado to read them from the ZUMA directory, so that when you regenerate them, they should be included in the new version. Should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



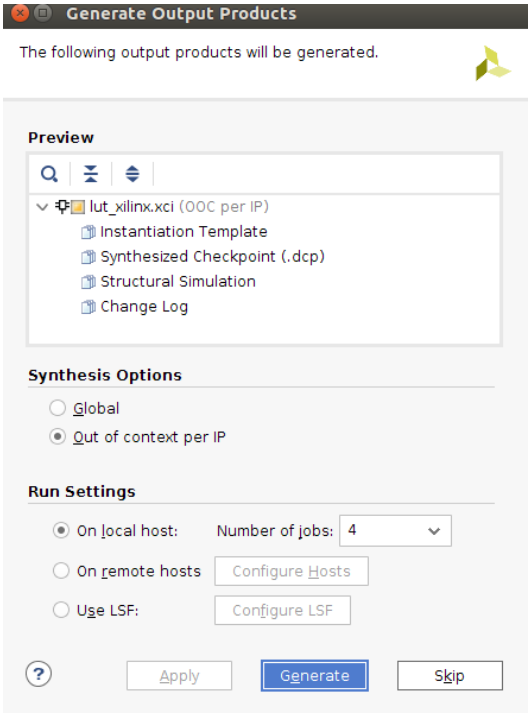
3. Right click on '`define.vh`' and click *Set Global Include* as shown.
   Do the same for '`def_generated.vh`'.



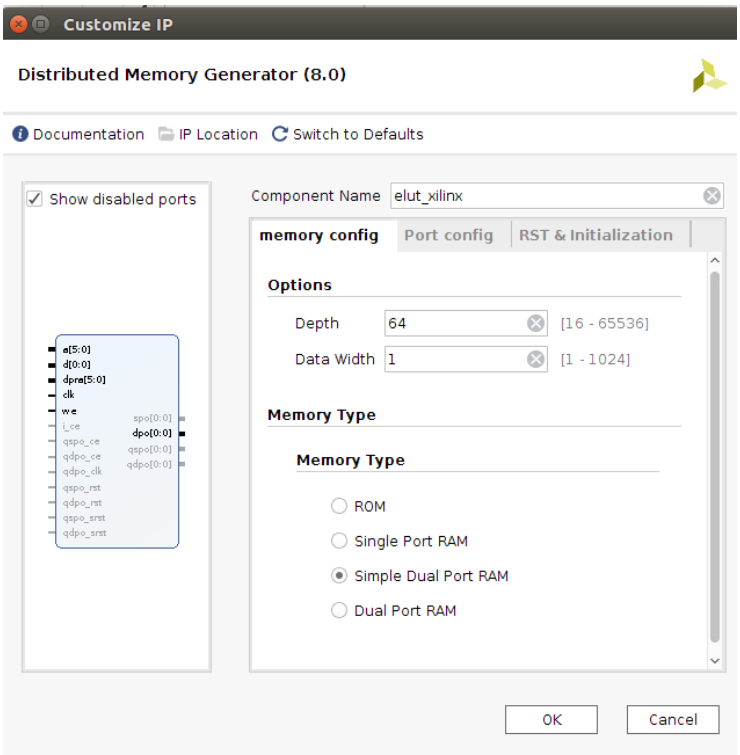
4. From the IP catalog, select the *Distributed Memory Generator* core as shown. Change the component name to `lut_xilinx`. Set *Data Width* to 1. Change *Memory Type* to *Simple Dual Port RAM* and click *OK*. This will include the basic building block of ZUMA into the project – the LUTRAMs – which the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



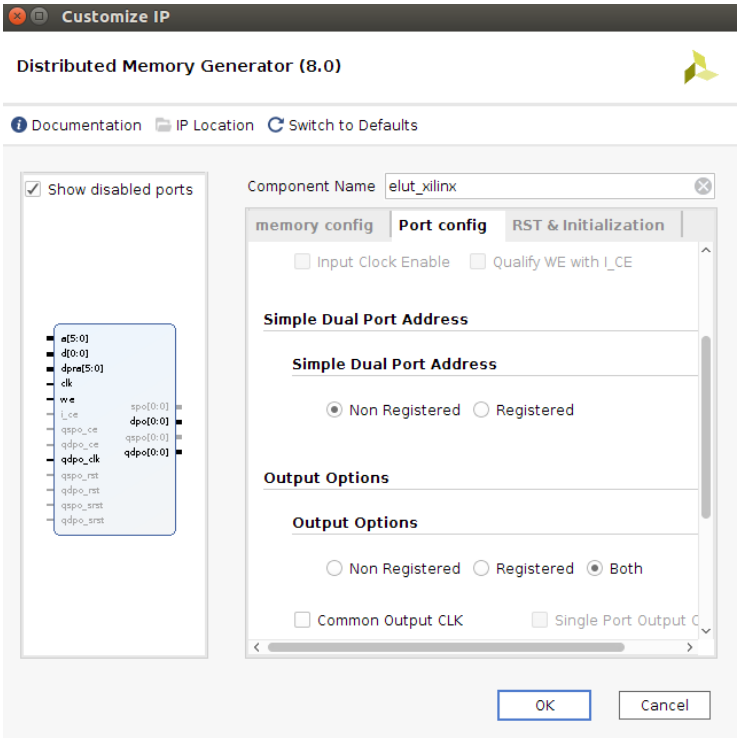
5. In the *Generate Output Products* window that appears, set Synthesis Options to *Out of context per IP* and click *Generate*.



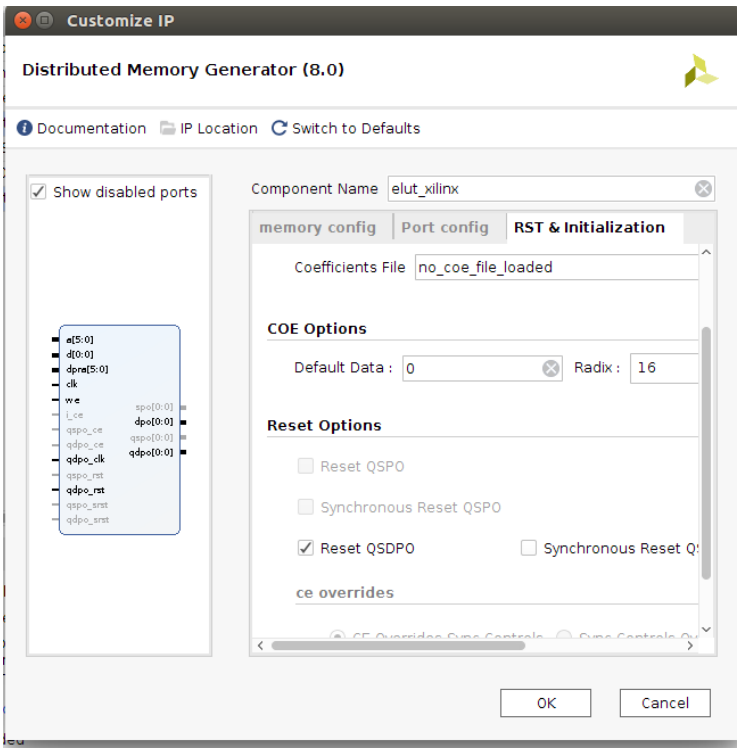
6. Since the implementation of virtual sequential circuits, ZUMA requires a special version of this building block for its eLUTs. Hence, add the *Distributed Memory Generator* core a second time now, but this time with different settings. Change the component name to `elut_xilinx`. Set *Data Width* again 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 Advanced Usage

This section's purpose is to give you enough insight into the ZUMA overlay structure and the file generation, 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*.

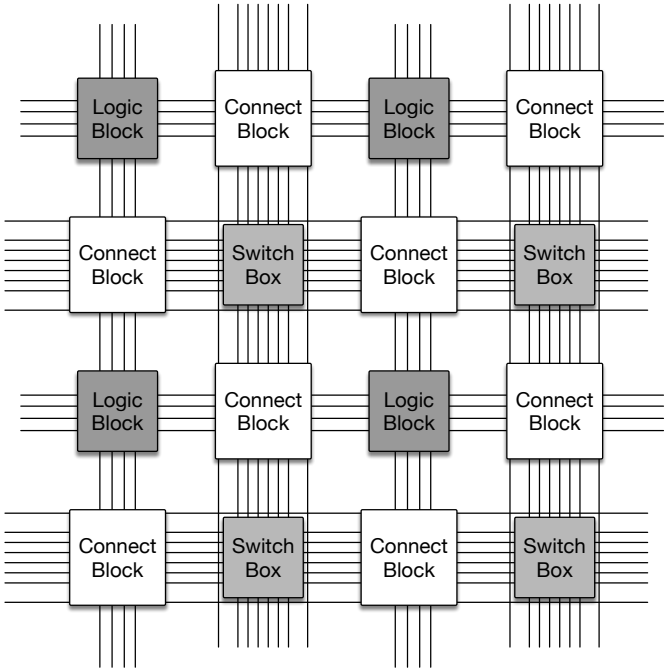


Figure 2: General island-style FPGA layout. Source: [5].

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.

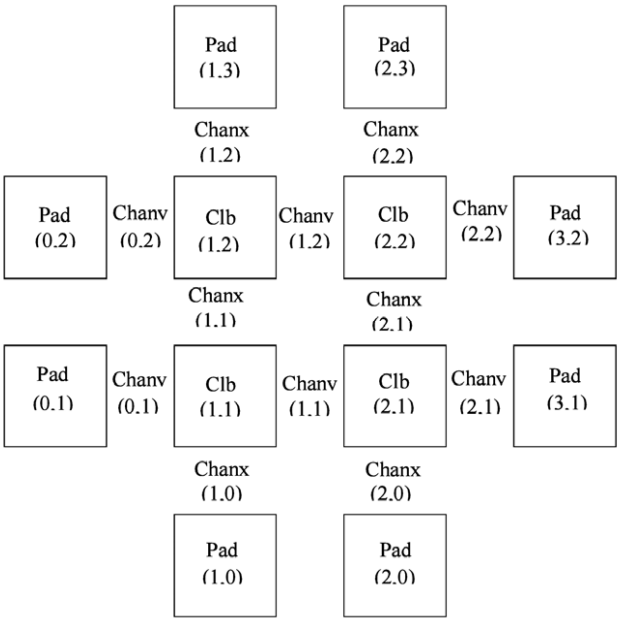


Figure 3: Global structure of a 2×2 virtual FPGA. Source: VPR manual.

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.

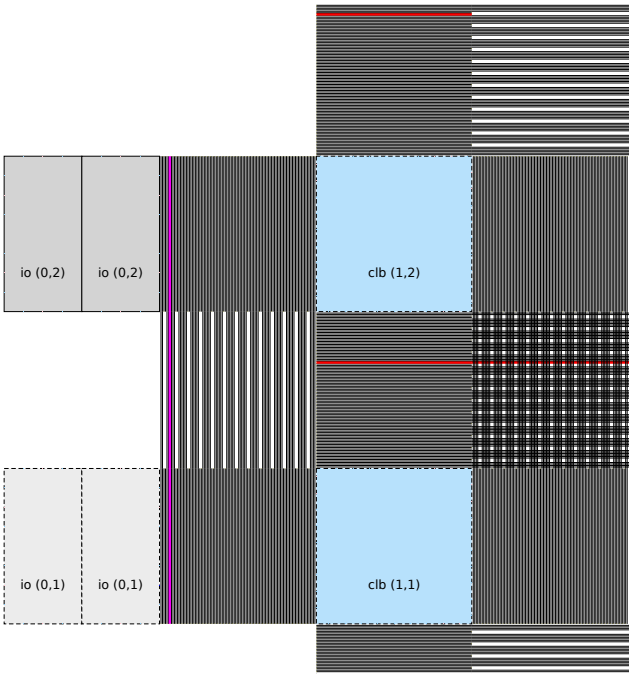


Figure 4: Tracks of different lengths and directions in channels: The purple track connects resources vertically and has a length of 2, while red ones are horizontal and the upper one has a length of 1. Source: VPR.

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.

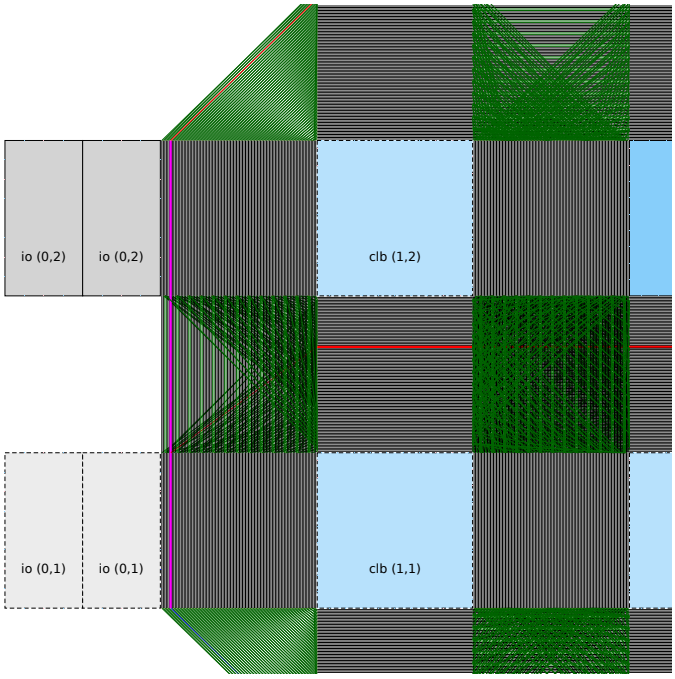


Figure 5: Tracks of channels are connected by the switchboxes (green). Source: VPR.

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.

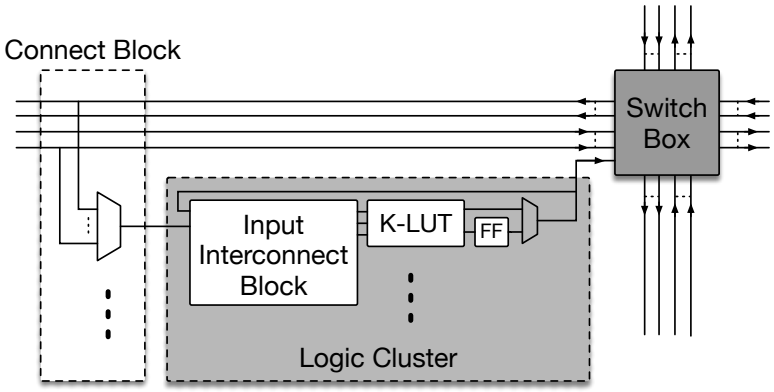


Figure 6: The structure of a ZUMA CLB, or cluster.

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

6.1.4 Data Structures

The whole virtual FPGA structure is saved in different data structures, whose scope and purpose we will briefly discuss here.

Within the VTR flow, the overall layout and the local structure are stored in an XML architecture description file, e.g., '`ARCH.xml`'. The expanded global structure of the FPGA is stored in the routing resource graph (RRG), which is also an XML file in VTR8, e.g., '`rr_graph.xml`'.

Within ZUMA, the whole structure is built into one complete graph, let us call it overlay description graph (ODG) here for easier reference. This ODG includes all virtual resources and structures, i.e., IO pads, routing MUXes, eLUTs, and FFs. To be able to use Xilinx' LUTRAM macros of a specific input width l (or the Altera alternative) to implement the overlay later, the ODG has to be made l-feasible, i.e., each node must have a fan-in of at most l and a fan-out of exactly 1. This process is one of the internal processes within the overlay generation, and the resulting l-feasible ODG is used for most processing steps within the flow. Within the rest of this document, we will call the former graph the ODG, and the latter one the technology-mapped ODG.

6.2 ZUMA Tool Flow Details

To fully understand all generated files, we have to take a closer look into the inner workings of the ZUMA overlay generation, which we will do in this section. These are thus the flow details that happen automatically when running `compile.sh`.

6.2.1 Basic Flow

The detailed steps to generate an overlay using the flow depicted in Figure 1 are as follows for the basic version without timing analysis or optimization (user interaction is only required for top-level items):

1. Set all values in '`zuma_config.py`' to fit your needs.
2. Run the ZUMA flow ('`compile.sh`' with your virtual circuit).
   (a) '`generate_buildfiles.py`'
   - Generates the VPR architecture files and build scripts from templates, i.e., currently '`ARCH_vpr8.xml`', '`abccommands.vpr8`', '`vpr8.sh`', and '`vpr8_timing.sh`' in the '`build`' directory, or their VPR 7 alternatives.
   - Applies the parameters from '`zuma_config.py`' to all generated files.

   (b) `VTR`
   - Synthesizes the virtual circuit into a technology-mapped BLIF file.
   - Generates all local and global routing resources in VPR.
   - Saves the global ones as routing resource graph (RRG).
   - Saves the result of its packing, placement and routing steps as a netlist, placement and routing file.

   (c) '`zuma_build.py`'
   - Parses the global routing resources from the RRG.
   - Builds up an internal overlay description graph (ODG, cp. Section 6.1.4) representing all global and local resources from the parsed global information and statically for the local ones.
   - Builds the technology-mapped ODG from the ODG by expanding nodes which use too many inputs to fit into the targeted LUTRAM macros.
   - Writes out the Verilog description for the complete overlay by going through all nodes and edges of the technology-mapped ODG and generating a correspondingly connected LUTRAM instantiation in Verilog.

- Parses the information from the BLIF, netlist, placement, and routing files and correlates it with the technology-mapped ODG, i.e., which LUT node implements which functionality, which signal name is used where, how do the routing nodes need to be configured to pass along the signals the correct way, etc.
- Builds the ZUMA bitstream for the virtual circuit by going through all nodes of the technology-mapped ODG and saving their configuration as the corresponding LUTRAM configuration.

(d) Verifies whether the generated circuit is functionally equivalent to the original specification.

3. Include the generated overlay in an FPGA design (cf. Section 5).
4. Run the device vendor's EDA tools.
5. Configure the ZUMA overlay with the bitstream.

6.2.2 Timing-augmented Flow

Note: This section is currently only applicable to Xilinx devices.
The detailed steps to generate an overlay using the flow depicted in Figure 1 are as follows for the version with all bells and whistles, which allows for static timing analysis of the virtual circuit mapped to the physical device, as well as timing-driven virtual placement and routing. For the steps that are already part of the details presented in Section 6.2.1, we only give reduced details here, to focus on the differences (user interaction is again only required for top-level items, except for the SDF extraction):

1. Set all values in '`zuma_config.py`' to fit your needs.
2. Set params.*vprAnnotation* = True and params.*sdf* = False in '`zuma_config.py`'.
3. Run the ZUMA flow ('`compile.sh`' with dummy / any virtual circuit).

   (a) '`generate_buildfiles.py`'
   - Generates '`ARCH_vpr8.xml`', '`abccommands.vpr8`', '`vpr8.sh`', and '`vpr8_timing.sh`' in the '`build`' directory.
   - Within '`ARCH_vpr8.xml`', every CLB and BLE are their own individual instance with zeroed timing.

   (b) `VTR` (with timing analysis off)

   (c) '`zuma_build.py`'
   - Writes out the Verilog description for the complete overlay.
4. Include the generated overlay in an FPGA design (cf. Section 5).
5. Run the device vendor's EDA tools.
6. Extract and copy the standard delay format (SDF) files describing the timing properties of the overlay.
   - Assuming that your top-level 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.
- Edit the parameters params.*sdfFileName* and params.*sdfFlipflopFileName* to these names and set params.*sdf* = True in '`zuma_config.py`'.

7. Run the ZUMA flow for a second time ('`compile.sh`' with dummy / any virtual circuit).
   (a) '`generate_buildfiles.py`'
   (b) `VTR` (with timing analysis off)
   (c) '`zuma_build.py`'
      i. '`ReadSDF.py`'
         - Parses the SDF file, identifies mappings of SDF cells and technology-mapped ODG (cp. Section 6.1.4) nodes and augments the ODG with the timing information for routing MUXes and LUTs (`addLutCellDelayToMappedNode`), as well as FFs (`addFlipflopCellDelayToMappedNode`).
         - Parses iopath and read port delays, as well as setup and hold times.
         - Parses only worst-case times, as VPR can only handle one delay entry per entity, and the worst case is then usually the most interesting one.
      ii. '`TimingAnalysisSDF.py`' `performTimingAnalysis()` calculates the critical path from the delay-augmented technology-mapped ODG.
      iii. '`NodeGraphTiming.py`'
         - Computes the delay information of the ODG nodes from the imported information in the technology-mapped ODG nodes, i.e., congregates the delay of expanded nodes into a delay for their supernode.
         - This step is necessary, as only the nodes of the original ODG correspond to the entities known to VPR.
      iv. '`TimingAnnotation.py`' `annotateClusterTiming()`
         - Computes delays of all paths within the clusters which connect entities to each other that are known to VPR:
           – annotateBleOutToMuxOut
           – annotateMuxOutToBleIn
           – annotateBle
           – annotateClbInToBleIn
      v. '`TimingAnnotation.py`' `annotateBack()` writes all computed delays from the augmented (original) ODG into the files '`ARCH_vpr8_timing.xml`' and '`rr_graph_timing.xml`'.

8. Run the ZUMA flow for a third time ('`compile.sh`' with your actual virtual circuit) – or multiple times thereafter for new virtual circuits.
   (a) '`generate_buildfiles.py`'
   (b) `VTR`
      - Now using '`vpr8_timing.sh`' and thus with timing analysis on.
      - Uses the delay-augmented '`ARCH_vpr8_timing.xml`' and '`rr_graph_timing.xml`'.
      - Can thus perform (meaningful) timing-driven placement and routing.
   (c) '`zuma_build.py`'
      i. Now uses '`rr_graph_timing.xml`' and RRG.

ii. 'ReadSDF.py'

iii. 'TimingAnalysisSDF.py' performTimingAnalysis() can now do a proper timing analysis for the current virtual circuit, giving a delay estimate with a corresponding maximum frequency f_{max}.

iv. Builds the ZUMA bitstream for the virtual circuit.

9. Set the clock for the virtual device to a frequency $\leq f_{max}$.

10. Configure the ZUMA overlay with the bitstream.

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. To create a new timing-augmented overlay, repeat steps 1 through 7, to just map a new virtual circuit to an existing overlay, repeat steps 8 through 10.

6.3 Generated Data Structures and Files

In this section we will now give an overview of the intermediate and final generated files and data structures.

For thorough descriptions of the architecture[5] and routing resource graph[6] files, see the documentation of the VTR flow.

6.3.1 Overlay Description Graph

The first own data structure of the ZUMA flow is the ODG, the internal representation of all resources of the virtual FPGA. It is generated in two steps:

1. The global, outer structure of the virtual FPGA is generated from the RRG, consisting of the global routing network (channels and switchboxes) and clusters, but not the inner structure within such a cluster.

2. The inner structure of the different clusters (BLE elements and interconnect routing) is created from scratch analogously to the architecture file.

For the first step, Figure 7 shows an example of such an outer structure. To get such a structure the script parses the RRG file generated by VPR, which contains the outer structure visualized in Figure 7 as a graph. We load this RRG in the function load_graph(filename) in the file 'InitFpga.py'. Because we can only obtain the outer, global structure from it, we have to add the nodes for the inner, local structure ourselves. In our example this only affects the cluster location 1_1. Figure 8 shows the original cluster, while the updated cluster is shown in Figure 9.
The graph was altered by the following steps of the function build_inner_structure() in the file 'InitFpga.py':

1. For each BLE add an eLUT (node 62) and a MUX (63), which decides whether the flip flop of that BLE is bypassed or not. The FF does not have to be added as a separate node as the eLUT and FF are grouped together into one entity for the generated Verilog file. Connect the new nodes appropriately.

2. Connect the MUX output with the cluster output pin (28) for routing the signal out of the cluster. To this end, we dismount the OPIN from the source node (25) because this node is not useful anymore and will be removed in the build of the Verilog file.

3. Add the nodes for the IIB in the function buildSimpleNetwork(): Add enough MUXes to route the cluster inputs and the BLE outputs to every pin of every LUT. Therefore each pin gets its own MUX (64 - 69) and the inputs of these MUXes are the cluster input node (27) and the output of the one LUT MUX (63).

[5] Architecture: https://docs.verilogtorouting.org/en/latest/arch/
[6] RRG: https://docs.verilogtorouting.org/en/latest/vpr/file_formats/#routing-resource-graph-file-format-xml

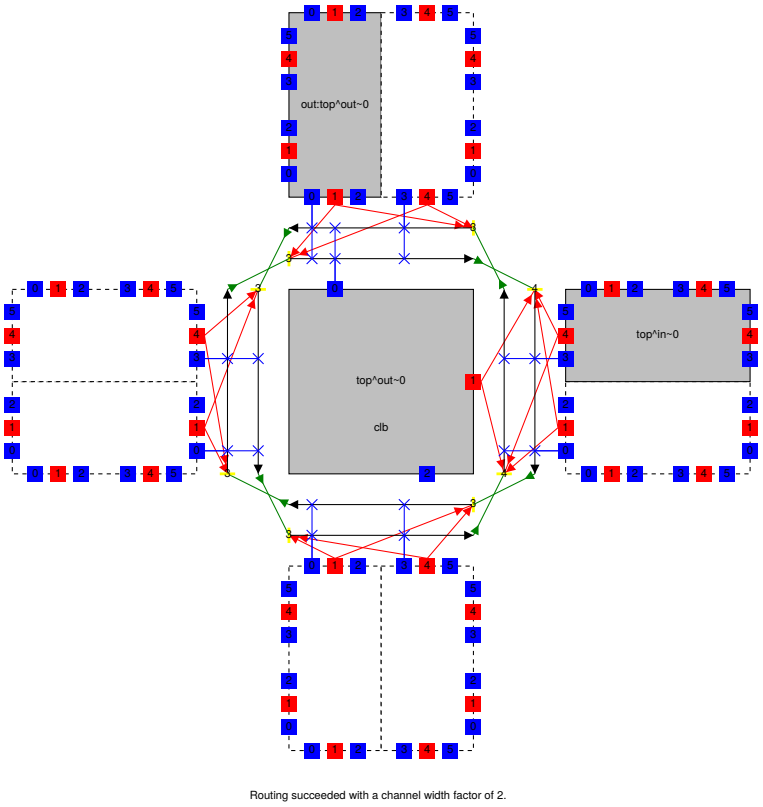


Figure 7: A small outer structure: We have 8 channels of length one (black arrows), 8 output pins (OPIN, red numbered blocks) on the IO pad which route the input of an fpga. We have 8 input pins(IPIN, blue numbered blocks) on the IO pads, which route to the output of the fpga. We have one used cluster input (IPIN, blue) and one output pin (OPIN,red). Also we have switchbox connections (green arrows).

Furthermore we have to generate a k_{host}-feasible version of the ODG, i.e., we have to replace each node with more than k_{host} input edges (more than k_{host} pins) with several nodes, because the real FPGA hardware we want to synthesize our virtual FPGA on, has only k_{host} pins per LUT. k_{host} is implemented as global constant $globs.host\_size = 6$.

The generation of the possible LUT types is realized in:

- writeSimpleLut()
- writeTightlyLut()
- writeComplexLut()

In this document, we call this new, expanded overlay description graph the technology-mapped ODG.

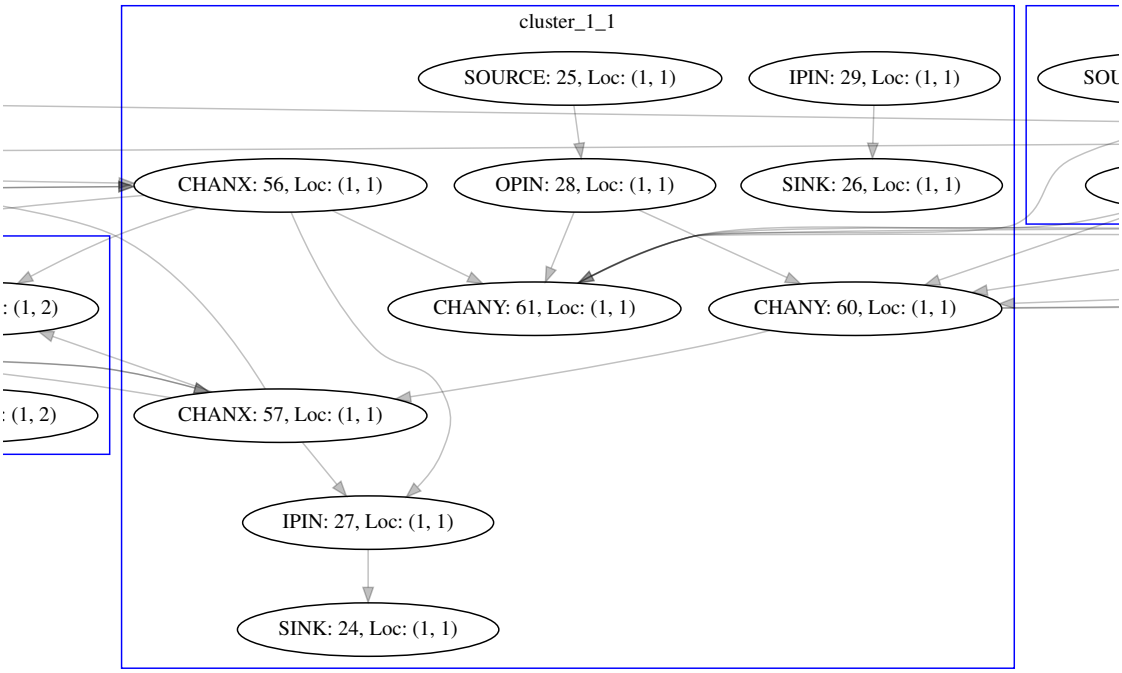


Figure 8: An overlay description graph, after the RRG parsing.

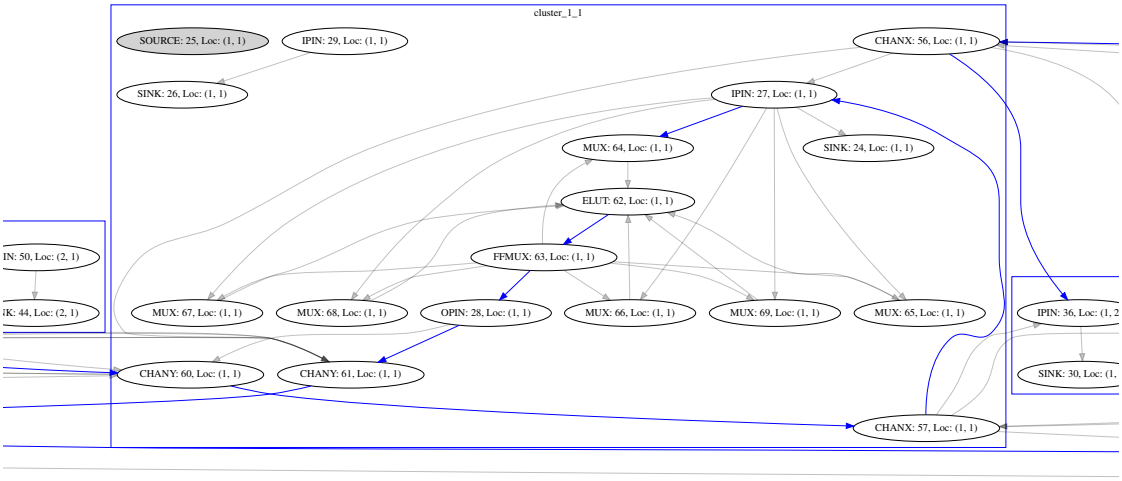


Figure 9: An ODG with added nodes for the local structure.

6.3.2 Verilog file

The central Verilog file '`ZUMA_custom_generated.v`', which describes the complete configurable overlay, is generated in the function `build_global_routing_Verilog` in the file '`BuildVerilog.py`'. As explained before, the basic approach is to traverse the ODG and write its structure into a file, translated to Verilog primitives and Xilinx macros. We also perform several optimizations to omit the generation of useless nodes, which:

- Are sources/sink nodes,
- have no input, or
- have only one input, i.e., passthrough nodes. For these, we use Verilog assigns instead of generating routing MUXes. An exception are FF MUXes, because they will have two inputs on the real FPGA hardware, but not in our graph.

Listing 1 shows an example excerpt of a generated Verilog file, with some of the interesting elements:

- The outward Verilog module interface (which we will discuss in a later section).
- ODG nodes that are translated to wires.
- Passthrough nodes from the ODG that are realized using Verilog assigns.
- A LUTRAM macro instantiation for a routing MUX (`mux_337`).
- A LUTRAM macro instantiation for an eLUT (`c_mux_1316`) and its FF MUX (`c_mux_1317`).
- The IO pad endpoints `fpga_outputs[i]` and `fpga_inputs[i]`.
- The instantiation of the simple configuration controller.

Listing 1: Example Verilog excerpt from ZUMA_custom_generated.v.

```
`include "define.v"
//ZUMA global routing Entity
//automatically generated by script
module ZUMA_custom_generated
#(
    parameter N_NUMLUTS = 8,
    parameter I_CLINPUTS = 28,
    parameter K_LUTSIZE  = 6,
    parameter CONFIG_WIDTH  = 32
 )
(
clk,
fpga_inputs,
fpga_outputs,
config_data,
config_en,
progress,
config_addr,
clk2,
ffrst
);
    // [..]
    wire node_0;
    wire node_1;
    wire node_2;
```

```
    wire node_3;
    wire node_4;
    // [..]
    //sbox driver x at  (1, 2, 2, 2)
    assign node_836 = node_986;
    //sbox driver x at  (1, 2, 2, 2)
    assign node_837 = node_1298;
    // [..]


    //cluster input at (2, 1)
    //size: 6
    //inputs: [536, 537, 604, 605, 648, 649]
    lut_custom mux_337 (
        .a(wr_addr), // input [5 : 0] a
        .d(wr_data[5]), // input [0 : 0]
        .dpra({node_536,node_537,node_604,node_605,node_648,node_649})
            , // input [5 : 0] dpra
        .clk(clk), // input clk
        .we(wren[4]), // input we
        .dpo(node_337));

    // [..]

    //internal cluster node (eLUT) at  (1, 2)
    //size: 6
    //inputs: [1332, 1333, 1334, 1335, 1336, 1337]
    elut_custom c_mux_1316 (
        .a(wr_addr), // input [5 : 0] a
        .d(wr_data[11]), // input [0 : 0]
        .dpra({node_1332,node_1333,node_1334,node_1335,node_1336,
            node_1337}), // input [5 : 0] dpra
        .clk(clk), // input clk
        .we(wren[27]), // input we
        .dpo(node_1316_unreg), // unregistered output
        .qdpo_clk(clk2), // run clk
        .qdpo_rst(ffrst), // input flip flop reset
        .qdpo(node_1316_reg)); // registered output

    //internal cluster node (ffmux) at  (1, 2)
    //size: 1
    //inputs: [1316]
    lut_custom c_mux_1317 (
        .a(wr_addr), // input [5 : 0] a
        .d(wr_data[12]), // input [0 : 0]
        .dpra({node_1316_reg,node_1316_unreg,1'b0,1'b0,1'b0,1'b0}), //
            input [5 : 0] dpra
        .clk(clk), // input clk
        .we(wren[27]), // input we
        .dpo(node_1317));

    // [..]

    assign fpga_outputs[0] = node_0;
    assign fpga_outputs[1] = node_3;
    // More of these
    assign node_1 = fpga_inputs[0];
    assign node_4 = fpga_inputs[1];
    // More of these
```

```
    parameter NUM_CONFIG_STAGES = 78;
    config_controller_simple
    #(
        .WIDTH(CONFIG_WIDTH),
        .STAGES(NUM_CONFIG_STAGES),
        .LUTSIZE(K_LUTSIZE)
    )
    configuration_ctrl
    (
        .clk(clk),
        .reset(1'b0),
        .wren_out(wren),
        .progress(progress),
        .wren_in(config_en),
        .addr_in(config_addr),
        .addr_out(wr_addr)
    );
endmodule
```

6.3.3 Configuration Bitstream

Using the files generated by VPR, the ZUMA scripts compute a routing for every MUX node in the ODG (which input to which output) and a LUT configuration for each LUT node.

TODO: Describe bitstream format.

6.4 ZUMA Overlay Configuration Process

In this section we will briefly explain which parts of the generated overlay is involved how in the (re-)configuration of a ZUMA overlay.

6.4.1 Module Description

The port interface of the module in the generated hardware overlay '`ZUMA_custom_generated.v`' are as follows:

- `clk`: The clock which is only used to configure the virtual FPGA.
- `clk2`: The clock which is used for running the virtual circuit, i.e., which is connected to the FFs of the virtual FPGA.
- `fpga_inputs`, `fpga_outputs`: Global IOs of the virtual FPGA.
- `config_en`: A signal to put the overlay into reconfiguration mode.
- `config_data` Input for configuration lines from the '`output.hex.mif`' file. This is used to configure the LUTs in the virtual FPGA. Will be described later in detail.
- `config_addr` The address at which to write the current `config_data` item. Starts at 0 and ends after processing the entire '`output.hex.mif`'. Will be described later in detail.
- `progress` signals with a rising edge to high that the configuration is finished.

6.4.2 Configuration Details

To configure a ZUMA overlay, you have to connect the *ZUMA_custom_generated* of '`ZUMA_custom_generated.v`' with a configurator module. We provide the file '`verilog/generic/ZUMA_TB_wrapper.v`', which can be used as a template to perform such a configuration. In it we connect the ZUMA module with the configurator *fixed_config* from the file '`verilog/platforms/xilinx/init_config.v`', as shown in Listing 2.

Listing 2: Excerpt from ZUMA_TB_wrapper.v.

```
// Reverse the retrieved config data, as the
// overlay requires it in reverse direction as stored
generate
    genvar i;
    for (i = 0; i < CONFIG_WIDTH; i = i + 1)
    begin: reverse
       assign cfg_in[CONFIG_WIDTH-1-i] = cfg[i];
    end
endgenerate

// Fetch config data for next address
fixed_config #(.LUT_SIZE(LUT_SIZE), .NUM_STAGES(NUM_STAGES))
    config_data (
    .address_in(next_address),
    .clock(clk),
    .q(cfg)
);

// Include the actual overlay
ZUMA_custom_generated XUM (
    .clk(clk),
    .fpga_inputs(inputs),
    .fpga_outputs(outputs),
    .config_data(cfg_in),
    .config_en(write),
    .progress(),
    .config_addr(address),
    .clk2(clk),
    .ffrst(virtual_reset)
);
```

6.4.3 The Configuration Module fixed_config

To load the configuration, *fixed_config* contains a RAM block with the complete content of the file '`output.hex.mif`'. At runtime, the module feeds each line of this file (one per clock cycle) to the overlay, which has been put into reconfiguration mode. The line address is given by the input `address_in` starting from 0 to the length of '`output.hex.mif`'−1, and the addressed line is provided by the output `q`. This output is then directed to the ZUMA input `config_data` by the wrapper.

6.4.4 Wrapper Module

To start a configuration we therefore just have to raise the ZUMA input `config_en` to high and start increasing the address input connected to both the *ZUMA* and *fixed_config* module starting by 0.

6.4.5 ZUMA_custom_generated module

To understand the configuration process, we must remember how LUTs are configured. Figure 10 depicts the configured output for each input of a 6-input LUT. For every input combination there is exactly one output bit assigned. To configure a LUT we thus only have to store this output vector.

To speed up the reconfiguration process, ZUMA configures blocks of 32 LUTs in parallel. Each such block is internally called a stage. For each stage ZUMA configures each line of the 32 LUTs step-by-step as shown in Figure 11. Therefore, every bit of the `config_data` input is used

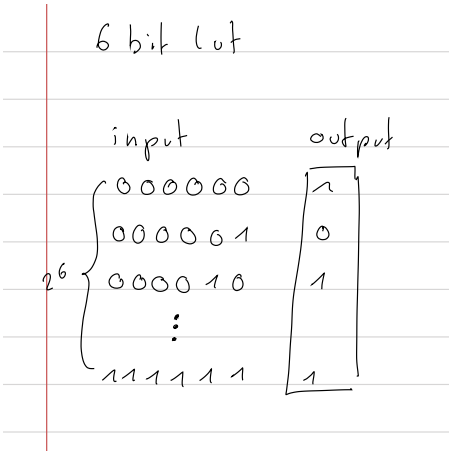


Figure 10: LUT config vector

for one LUT. ZUMA thus renames that input to `wr_data` and assigns the proper bit `wr_data[i]` as input to each LUT of a stage. To address only the LUTs of a single stage at a time during this

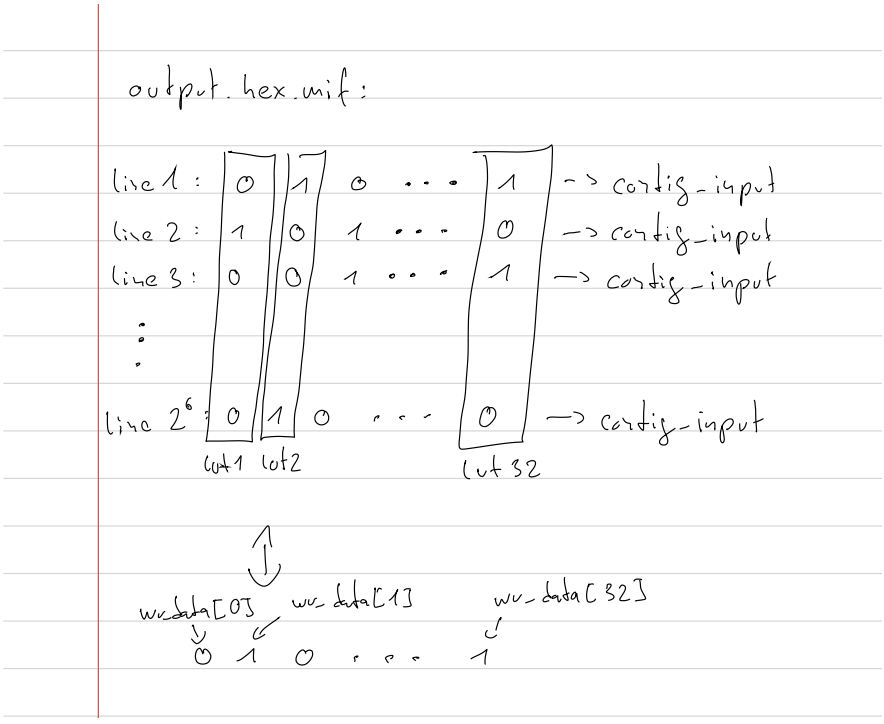


Figure 11: configuration line per line

configuration writing, ZUMA uses two address signals: `wr_addr` and `wren`. These are assigned by the configuration controller module *config_controller_simple* as shown in Listing 3. `addr_out` is directly the last `LUTSIZE`-1 bits of the incoming `addr_in` signal of ZUMA, however, `wren` is a bit more special: For each stage number i, only the i-th bit `wren[i]` is 1 while all the others are 0.

Listing 3: Excerpt from config_controller_simple.v.

```
assign addr_out = addr_in[LUTSIZE-1:0];
```

```
generate
    genvar stage;
    for(stage = 0; stage < STAGES; stage = stage+1) begin: stage_m
        assign wren[stage] = ((addr_in >> LUTSIZE) == stage) &&
            wren_global;
    end
endgenerate
```

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.