CHAPTER

1 The TLUT tool flow: An introduction

The TLUT tool flow is a tool flow that generates FPGA configurations. Its biggest difference with the traditional FPGA flow is the use of a new technology mapper, the TLUT mapper. The TLUT tool flow was developed to implement Dynamic Circuit Specialization (DCS) on FPGAs. A DCS implementation of an application specializes its circuit for the current values of a number of specific inputs, called parameters. This specialized circuit is smaller, and in some cases faster, than the original circuit. However, it is only correct for one set of parameter values. Each time the parameters change value, a new specialized circuit is generated. This new specialized circuit is then loaded into the FPGA using partial run-time reconfiguration.

The TLUT tool flow offers an efficient implementation of DCS because it uses the concept of parameterized configurations. A parameterized configuration is a configuration in which some bits are expressed as Boolean functions of the parameters. Before the FPGA can be configured, the parameter values are used to evaluate the Boolean functions. This generates the specialized configuration. Several papers on the academic underpinnings of the TLUT tool flow are listed in the wiki of this project or are contained in the documentation directory. There you can also find more information on our current research on extending and improving the TLUT tool flow. The TLUT tool flow itself and how to use it, is described in much more detail in this user guide.

1.1 Roadmap

We are releasing the TLUT tool flow in two phases. First, we have made the adapted TLUT technology mapper public. This will allow you to compare its results with a conventional technology mapper. We have included a framework to make this comparison very easy. The following pages provide clear examples and show how to adapt the framework for you own uses.

In the second phase, which will be released soon, we will provide you with the scripts and information necessary to integrate the TLUT technology mapper with the Xilinx FPGA tool flow. This will allow you to implement your DCS implementations on commercial Xilinx FPGAs. Our first target FPGA is the Virtex II Pro. We are currently working on extending this tool flow to more modern FPGAs, such as the Virtex 5 or 6.

1.2 Contact information

The TLUT tool flow is released by Ghent University, ELIS department, Hardware and Embedded Systems (HES) group (*hes.elis.ugent.be*).

If you encounter bugs, want to use the TLUT tool flow but need support or want to tell us about your results, feel free to contact us. We can be reached at *hes@elis.ugent.be*.

1.3 Referencing the TLUT tool flow

If you use the TLUT tool flow in your work, please reference in your publications the following paper:

Karel Bruneel, Wim Heirman, and Dirk Stroobandt. 2011. "Dynamic Data Folding with Parameterizable FPGA Configurations." ACM Transactions on Design Automation of Electronic Systems 16 (4).

You may also refer to one of our others papers if you think it is more related.

1.4 Contents of this document

Chapter **??**: *Setup*: A setup manual
Chapter **??**: *Your design*: A manual for running your own experiments
Chapter **??**: *Troubleshooting*: Problems you may encounter and their solutions
Chapter **??**: *Contents*: Description of the contents of this package

CHAPTER

2 Setup

```
> make
```

This builds a number of dependencies and the TLUT technology mapper and creates a ‘source’ file which sets environment variables. Set your environment variables using:

```
> . source
```

Your installation can be tested using the script in directory ‘tests’.

```
> cd tests
> ./examples_test.sh
```

Examples can be run using the included Python scripts. E.g.

```
> cd examples/treeMult4b
> ./treeMult4b.py
```

2.1 Dependencies

- Dependencies to be provided by the user:
  Quartus II (tested with Web Edition version 11, Web Edition v12 is unsupported), Altera Corporation, `http://www.altera.com`
  Java (tested with version 1.6.0), Oracle, `http://www.java.com`
  Python 2.7, `http://www.python.org`

- Dependencies automatically downloaded and installed:
  Aiger (tested with version 1.9.4), JKU Institute for Formal Models and Verification, `http://fmv.jku.at/aiger`
  ABC (tested with version 810ba683c042, 5 October 2012), Berkeley Logic Synthesis and Verification Group, `http://www.eecs.berkeley.edu/~alanmi/abc`

CHAPTER

3 Your own design

3.1 The run function

To test the TLUT mapper on your own design (described in VHDL or Verilog), import the *run* function from *fast_tlutmap* in your own Python script and call it with the following arguments:

```
run(module, submodules, K, performCheck, verboseFlag)
```

- **module** - String
  The location of the top level of your design. If your design consists of only one VHDL/Verilog module, then you only have to pass the location of this module as the first argument and you can ignore the second argument.

  E.g. run('yourDesign.vhd')

- **submodules** - list of Strings - *optional*
  If your design consists of a top level module and several submodules, you can add a list of submodules here, for example if you have one top level module and two submodules.

  E.g. run('yourTopLevelModule.vhd', ['yourFirstSubModule.vhd', 'yourSecondSubModule.vhd'])

- **K** - integer - *default 4*
  You can choose the number of inputs a LookUp Table has on your target FPGA

  E.g. run('yourDesign.vhd', 6) or run('yourDesign.vhd', K=6)

- **performCheck** - boolean - *default True*
  When checks are turned on, the resulting mapping is verified using a miter and satisfiability solver. This ensures that the mapped circuit implements the same functionality as the input circuit.

  E.g. run('yourDesign.vhd', 6, True)
  or run('yourDesign.vhd', performCheck=False)

- **verboseFlag** - boolean - *default False*
  This activates the verbose mode in which more information gets printed regarding the execution of the flow.

  E.g. run('yourDesign.vhd', 6, True, True)
  or run('yourDesign.vhd', verboseFlag=True)

3.2 A Step-by-step Approach

Below, we have provided a more step-by-step approach to set up your own Python script starting from the *treeMult4b* example:

1. Make a folder for your design and copy the treeMult4b folder from the examples directory. You need at least the files *abc.rc* and *treeMult4b.py*

   ```
   > mkdir yourDesign
   > cp -r examples/treeMult4b/* yourDesign/
   ```

2. Replace treeMult4.vhd by copying your VHDL/Verilog files, describing your design, into the *yourDesign*-folder

   ```
   > rm treeMult4b.vhd
   >
   ```

3. Annotate the parameters in your top level VHDL/Verilog file.

   Any input signal, or combination of input signals, of your top level module can be chosen as parameter. Preferably the designer will choose the slowly changing input signals as parameters, because a change in the value of the parameters results in a reconfiguration of the FPGA. Different combinations can be tested easily by changing the annotations. In the following example a simple multiplexer is described in VHDL and the 'sel' input signal is annotated as a parameter.

   ```
   entity multiplexer is
   port(
     --PARAM
     sel : in  std_logic_vector(1 downto 0);
     --PARAM
     in  : in  std_logic_vector(3 downto 0);
     out : out std_logic
   );
   end multiplexer;

   architecture behavior of multiplexer is
   begin
     out <= in(conv_integer(sel));
   end behavior;
   ```

   The Verilog annotations are quite similar. The following Verilog example, a simple multiplier, can be found in the examples folder;

   ```
   module mult(x,y,z);
   parameter N = 16;
   input [N-1:0] x;
   //PARAM
   input [N-1:0] y;
   //PARAM
   output [2*N-1:0]z;
   assign z = x * y;
   endmodule
   ```

4. Modify the Python script

   ```
   > mv treeMult4b.py yourDesign.py
   > nano yourDesign.py
   ```

   Edit the 6th line, in which the *run* function is called, according to the documentation in section **??**.

5. Run the modified Python script

   ```
   > ./yourDesign.py
   ```

6. Check the '.par' file in the work directory, to ensure the parameters were correctly recognised. It should enlist all the signal names of the parameters.
   The tool for extracting parameters from VHDL and Verilog is limited. It cannot parse all VHDL/Verilog constructs, so it may not recognise all your parameters in all cases. You should avoid using signal types other than std_logic, std_logic_vector and arrays thereof. In addition, only one definition is allowed per line. Similarly, only one Generic (or Verilog parameters) used in the definition of the parameter signals should be defined per line.

7. Analyze the results. As an example, the output of the Python script of the treeMult4b project is given;

   ```
   Stage: TLUT mapper
   Luts (TLUTs)      depth      check
   12.0 (12.0)       1.0        PASSED
   Stage: SimpleMAP
   Luts              depth      check
   67.0              10.0       PASSED
   Stage: ABC fpga
   Luts              depth      check
   67.0              10.0       PASSED
   ```

   The first column shows the number of LUTs in each mapping solution. These can be compared directly and represents the number of K-LUTs needed to implement the design. In this case, the TLUT mapper needs 12 LUTs, and SimpleMAP and ABC's fpga need 67.

   SimpleMAP and ABC fpga both perform mapping for a conventional, not parameterised, configuration. The same results could be expected for both tools because they implement basically the same algorithm. However, ABC fpga contains some optimisations, such as 'area recovery', which haven't been (completely) implemented in SimpleMAP. While the number of LUTs may differ between the two tools, you can expect the depth to be always equal.

   The number between brackets '()' is only relevant for the TLUT mapper. It shows the number of LUTs that will actually be reconfigured at run-time. We call these LUTs TLUTs. In this case all 12 of the LUTs are TLUTs and will be reconfigured at run-time.

   The third column shows the depth of each mapping solution. This is the number of LUTs in the longest path, a measure for the speed of the circuit.

   Finally, column 4 shows if the mapping solution has passed the equivalence test. This check is performed to see if the resulting circuit still has the same functionality as the original input.

3.3 Advanced

It is also possible to start from an '.aag' or '.blif' file. As a reference for this we provide examples *AES* and *tripleDES*. Advanced experiments can be set up by copying *fast_tlutmap* and using this file as a template for your experiment.

CHAPTER

4 Troubleshooting

Try running one of the examples (in the examples folder) first. If this works as expected your environment is set up correctly.

These are a few common errors and their solution:

- *Python ImportError: No module named fast_tlutmap (or others):*
  The environment variables probably aren't set up correctly. Try running '. source' in the main directory. If you have moved the main directory, the paths in the 'source' file are not longer correct. Remove this file and recreate it using 'make source'.

- *java.lang.OutOfMemoryError:*
  The Java technology mappers don't have enough memory available. Try to use the *setMaxMemory* function. It can be imported from the *fast_tlutmap* Python script and should be called before the *run* function. This function sets the maximum memory usage for the Java tools in megabytes.

  ```
  from fast_tlutmap import run, setMaxMemory

  setMaxMemory(4096)
  run('treeMult4b.vhd', K=4, performCheck=True, verboseFlag=False)
  ```

- *The number of TLUTs is 0, or the reduction in number of LUTs is lower than expected:*
  Check if all parameters were correctly extracted from your VHDL/Verilog. See section **??**, step **??**.

CHAPTER

5 Contents

The ‘tlut_flow’ folder contains a number of folders and three files: ‘Makefile’, ‘README.md’ and ‘LICENSE’. The ‘Makefile’ is used to initially set up the tool flow. It will download some necessary third party tools and compile the used programs. More information about the license under which the TLUT tool flow is released can be found in ‘LICENSE’.

The ‘tlut_flow’ folder contains the following folders: ‘examples’, ‘tests’, ‘documentation’, ‘java’, ‘python’ and ‘third_party’.

- In the ‘examples’ folder you can find a number of designs that use the TLUT tool flow. You can run each of them by executing the Python script contained in the respective subfolder. You can run all examples at once (except AES) by executing the ‘run_all.sh’ script.
- In the ‘tests’ folder you can currently find one test, namely ‘examples_test.sh’. Executing this script will run the ‘examples/run_all.sh’ script and compare the output to the expected output.
- The ‘documentation’ folder contains this document, and its Latex source files, and the Ph.D. thesis by Karel Bruneel which contains more information about the academic underpinnings of the TLUT tool flow.

The remaining folders contain binaries, source code and wrapper scripts. You may want to edit these when doing advanced experiments.

- The ‘java’ folder contains the Java source and binary files of a simple technology mapper and an adapted parameterized version of this technology mapper, the TLUT technology mapper.
- After setup, the ‘third_party’ folder will contain ABC, the logic synthesis and technology mapping tool of the university of Berkeley, and Aiger, a tool used to handle textual (.aag) and binary representations of and-inverter-graphs (.aig).
- The ‘python’ folder contains the high level Python scripts that are used to interface with the tools in the other folders. It contains three files: ‘fast_tlutmap.py’, ‘genParameters.py’ and ‘mapping.py’. The ‘genParameters.py’ script will extract the parameters from the annotated VHDL file. The ‘mapping.py’ script contains high level wrappers that call the different mappers and a number of utility programs. These 2 last scripts are best used as given.

  The ‘fast_tlutmap.py’ script contains the main code in the ‘run’ function. This function can easily be adjusted to meet the user’s needs. The run function mainly consists of 2 steps: synthesis and technology mapping. The synthesis step converts the VHDL file into a logic circuit in blif format using Quartus II. The technology mapping step will map the logic circuit to a circuit with LUTs. Three different technology mapping tools are included to allow an easy comparison: the simple mapper, the TLUT technology mapper and the ABC mapper. The first two are not as optimized as ABC. For example ‘area recovery’ is not yet completely implemented. The ABC tool flow is an academic framework that does include a large number of such optimizations. Several commercial tools are based on this technology mapper.

  The ‘run’ function also contains some intermediate steps:

  – the conversion of the blif format into the aag format, needed for the Java mappers
  – extraction of the parameters out of the annotated VHDL file