FP2P - FPGA Port To Pin

Michał Kruszewski

Sep 16, 2021

CONTENTS:

1 Overview **1**
- 1.1 Status . . . 2
- 1.2 License . . . 2
- 1.3 Citation . . . 2

2 Glossary **3**

3 Installation and usage **5**
- 3.1 Installation . . . 5
- 3.2 Usage . . . 5
- 3.3 Windows . . . 6

4 File types **7**
- 4.1 Mapping file . . . 7
- 4.2 Tree file . . . 8
- 4.3 Assignment file . . . 8

5 Special attributes **11**
- 5.1 _default_ . . . 11
- 5.2 terminal . . . 12
- 5.3 regex . . . 12
- 5.4 prefix . . . 12
- 5.5 suffix . . . 13
- 5.6 end_prefix . . . 13
- 5.7 end_suffix . . . 13
- 5.8 set_property . . . 13
- 5.9 set_instance_assignment . . . 13

6 Subcommands **15**
- 6.1 resolve . . . 15
- 6.2 graph . . . 15
- 6.3 assign . . . 17

7 Mistakes detection **19**

8 Tips and quirks **21**
- 8.1 TRUE and FALSE in design constraint properties . . . 21
- 8.2 Net names on schematics . . . 21

9 Indices and tables **23**

CHAPTER

ONE

OVERVIEW

FP2P (FPGA Port To Pin) is a tool for robust port to pin assignment in multi-board FPGA designs. Fig. 1.1 shows an example of a multi-board FPGA design.

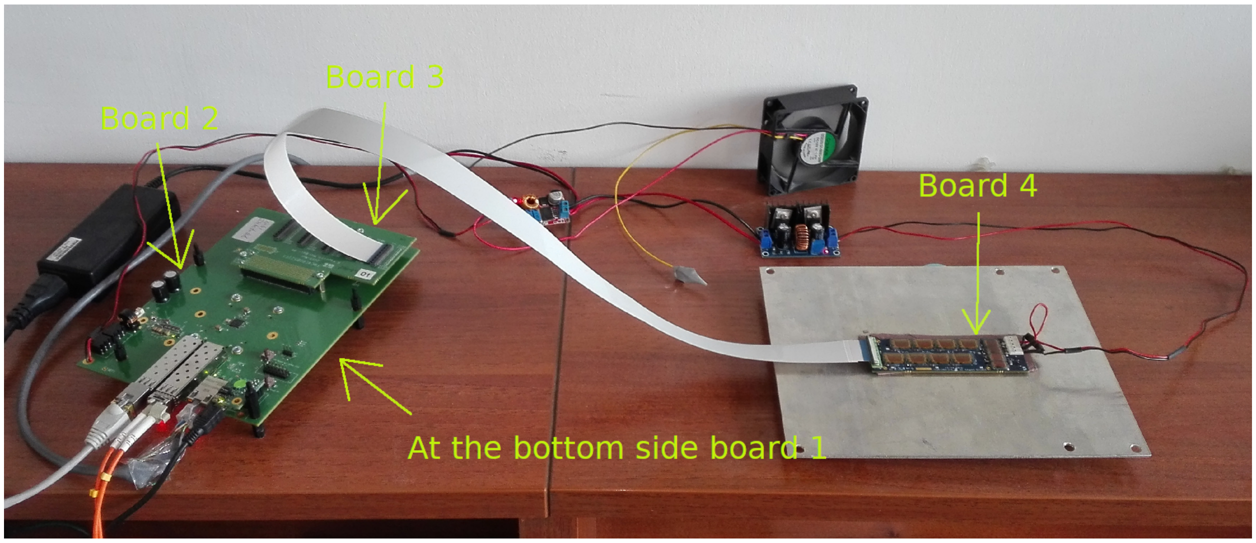

Fig. 1.1: Example of a multi-board FPGA design. This setup is a development setup and needs assignment for 42 differential signals. The final design needs assignment for 210 differential signals.

The problem has been known for a long time and is especially annoying in data acquisition systems, where hundreds of signals are routed via multiple boards. However, there is still no generic user-friendly open-source solution (or at least none has been found). The implementation has two main goals, safety (check as many potential human mistakes as possible) and reusability (reuse connections mappings, defined in files, in multiple designs). It is fully declarative and programming language-agnostic from the users perspective. Currently 2 target EDA tools are supported Vivado and Quartus. Adding support for another target EDA tool is very easy, as the analysis and resolving are completely decoupled from the constraint file generation.

If you want to jump straight to the examples check example and tests.

1.1 Status

The project is considered as finished. Only bug fixes and minor improvements not breaking backward compatibility will be accepted.

1.2 License

The project is licensed under the GPL-2.0 License.

1.3 Citation

If you find fp2p useful, and write any academic publication on a project utilizing fp2p please consider citing.

```
@ARTICLE{mkru_fp2p_ieee,
  author={Kruszewski, Michał and Zabołotny, Wojciech Marek},
  journal={IEEE Transactions on Nuclear Science},
  title={Safe and Reusable Approach for Pin-to-Port Assignment in Multiboard FPGA Data␣
↪Acquisition and Control Designs},
  year={2021},
  volume={68},
  number={6},
  pages={1186-1193},
  doi={10.1109/TNS.2021.3074530}
}
```

CHAPTER TWO

GLOSSARY

assignment file

See *Assignment file*.

logical mapping

Mapping done in software or firmware (within FPGA logic).

mapping file

See *Mapping file* .

mistake

Fault introduced by a human (*mistake != error*).

setup

Set of physical, connected PCBs.

tree file

See *Tree file*.

CHAPTER

THREE

INSTALLATION AND USAGE

3.1 Installation

fp2p is not available on the PyPI website. Why? fp2p is pure program, contained within single file, there is no library part, it can't be imported. This will *never* change. What is more, I didn't want to litter PyPI with a program that is relatively rarely used (although being very useful). It is not like new boards are designed or manufactured once per week or month.

User has full freedom in terms of installation. For example, you can do one of the followings:

- Copy `fp2p.py` file to the repository with the FPGA project design.
- Add fp2p repository as a submodule to the repository with the FPGA project design.
- Clone the repository *somewhere* in the filesystem and call the program providing path `python ~/your/path/fp2p/fp2p.py ....`
- Copy `fp2p.py` file to some directory included in the `PATH`. You may want to add shebang to the file, for example `#!/bin/python` and `chmod +x fp2p.py`.

3.2 Usage

fp2p supports 3 subcommands:

1. `assign` - assings ports to pins,
2. `graph` - resolves a mapping tree and prints graph,
3. `resolve` - resolves a mapping tree and prints the result.

After preparing proper mapping tree files or assignment files user only needs to call `fp2p` with one of the subcommands and proper order of arguments. This is further explained in the section *Subcommands*. You can also always run `python fp2p.py -h` or `python fp2p.py {subcommand} -h`.

3.3 Windows

The program has not been tested on Windows OS, and no one has so far reported that it works corretly. However, it *should* work, as no OS specific mechanisms or system calls are used.

CHAPTER

FOUR

FILE TYPES

The whole fp2p concept is based on 3 files types:

1. *mapping file* - one or more per setup,
2. *tree file* - one per setup,
3. *assignment file* - one per setup.

These three types share the same format and syntax, however their semantics is different. YAML has been chosen as the file format. Another user-friendly alternative was JSON, which is provided in the Python Standard Library. However, YAML has been chosen as it is less verbose and allows placing comments in the files, what might be useful in case of documenting any peculiarities in the mappings.

4.1 Mapping file

The mapping file is used for defining physical connections between pins within the board, cables or connectors.

Listing 4.1: Example mapping file.

```
A[1-3]:
  end: s[1-3]
  regex:
A4:
  end: end_pin_4
  terminal:
B1[2-3]:
  end: s_diff_pin_[pn]
  regex:
```

The mapping file is a dictionary, where the keys are the names of an input ends and values are dictionaries. So, mapping file is a dictionary of dictionaries (single outer dictionary, one or more inner dictionaries). The inner dictionary has one mandatory key, namely `end`, that denotes the name of the output end. Within the inner dictionary, some metadata about particular mapping can be placed. There are two optional special keys `regex` and `terminal`, that do not need any value to be correctly interpreted (can be solely the key).

4.2 Tree file

The tree file is used for defining the structure of the setup (how boards, connectors and cables are connected).

Listing 4.2: Example tree file.

```
name: board_1
files:
  - b_1.yaml
nodes:
  - name: board_2_connector_1
    files:
      - b_2_c_1.yaml
  - name: board_2_connector_2
    files:
      - b_2_c_2_f_1.yaml
      - b_2_c_2_f_2.yaml
```

Tree file consists of nodes, each node is a dictionary. Single node has 2 mandatory keys `name`, `files` and one optional key `nodes`.

The `name` of the node must be unique and serves as a scope for pin names within this node. It is necessary in order to avoid pin name collisions. Otherwise, name collisions could potentially occur in two scenarios:

1. same pin names in two different mapping files,
2. same mapping file used more than once in the tree file.

The `files` is a list of mapping files that constitute a given node. Each tree node can consist of multiple mapping files. Thanks to this, it is possible to define mappings for different connectors located on the same board in separate files or split connector mappings into multiple files to group them by functionality.

The `nodes` is a list of children nodes connected with a given node.

4.3 Assignment file

The assignment file is used for defining port assignments to the terminal pins.

Listing 4.3: Example assignment file.

```
_default_:
  set_property:
    IOSTANDARD: LVDS_33
    DIFF_TERM: "TRUE"

board_2_connector_1:
  port[1]:
    end: end_pin_1
    set_property:
      IOSTANDARD: LVDS_25

board_2_connector_2:
  port[2]:
    end: end_pin_2
```

(continues on next page)

(continued from previous page)

```
    set_property:
      DIFF_TERM: "FALSE"

  port[3]:
    end: end_pin_3

  # Differential pair example
  diff_[pn]:
    node: board_2_connector_2
    end: end_diff_pin_[pn]
    regex:

board_1:
  port[4]:
    end: end_pin_4
    end: s3
```

The assignment file can also be seen as a dictionary of dictionaries. Within the outer dictionary, single item is a dictionary defining assignments within the particular node (except the `_default_` key, see *_default_*). The *key* is the name of the node, and the *value* is a dictionary.

Within the inner dictionaries, single item is an assignment. The *key* is the name of the port. The destination pin name is placed under the `end` key.

CHAPTER FIVE

SPECIAL ATTRIBUTES

Special attributes are optional keys, that ease the work and reduce verbosity.

5.1 _default_

Both mapping and assignment file types have one reserved key `_default_`. All information, set within value of the default key, is automatically applied to all remaining items in the file. Setting the same option within given entry overwrites the default option. In other words, everything defined within given entry has higher precedence than what is copied from the default. Anything can be placed in the default section. If it has no meaning, it will be simply ignored.

In case of the assignment file, there can be single *_default_* section per file and single *_default_* section per node. Valuse set in the per node section take precedence. Values set within the *_default_* sections can always be overwritten within the particular items.

Listing 5.1: `_default_` attribute example.

```
_default_:
  set_property:
    IOSTANDARD: LVDS_33
    DIFF_TERM: "TRUE"
  prefix: foo_
  suffix: _suffix

board_1:
  _default_:
    prefix: prefix_

  A:
    end: pin_A
  B:
    end: pin_B
    set_property:
      DIFF_TERM: "FALSE"
    prefix: ""
  C:
    end: pin_C
    set_property:
      IOSTANDARD: LVDS_25
    suffix: ""
```

5.2 terminal

The `terminal` key functions like a type annotation. By default end pins are not intended to be connected to the FPGA ports. To indicate, that the end pin must be connected to the port, a special property called terminal must be added to this end pin. What is more, nothing more can be mapped to the pin marked as terminal. To some extent, this mechanism is similar in its nature to the mutability aspect of the Rust programming language and provides the basics for extensive mistakes detection. Terminal pins can be defined in each node of the tree, they are *not* limited only to the leaf nodes.

5.3 regex

The `regex` key is used to reduce verbosity. It enables regex expanding (sre_yield) for the given mapping. After expansion, natural (human) sorting (natsort) is applied to both names so that they can be mapped in a deterministic way. In case of any mismatch of the lengths of generated lists the error is immediately reported.

Listing 5.2: `regex` attribute example.

```
A[1-3]:
  end: s[1-3]
  regex:
# Above is equivalent to the following:
A1:
  end: s1
A2:
  end: s2
A3:
  end: s3
```

5.4 prefix

If `prefix` is set, its value is prepended to the name of each *key* denoting port name or pin name.

Listing 5.3: `prefix` attribute example in a mapping file.

```
_default_:
  prefix: foo_
A1:
  end: s1
A2:
  end: s2
# Above is equivalent to the following:
foo_A1:
  end: s1
foo_A2:
  end: s2
```

5.5 suffix

If `suffix` is set, its value is appended to the name of each *key* denoting port name or pin name.

5.6 end_prefix

Same as `prefix`, but applied to the value under the `end` key. Not yet implemented as there was no need for it so far.

5.7 end_suffix

Same as `suffix`, but applied to the value under the `end` key. Not yet implemented as there was no need for it so far.

5.8 set_property

The `set_property` section is used for setting FPGA pin properties. This attribute makes sense only for the assignment file and Vivado EDA tool. When setting property, *key* is the name of the property and *value* is the value of the property. See also *TRUE and FALSE in design constraint properties*, as it may save you some time.

Properties are not checked by the fp2p tool in any way. They are simply forwarded to the auto generated constraint file, and later checked by the EDA tool.

5.9 set_instance_assignment

The `set_instance_assignment` section is used for setting FPGA pin properties. This attribute makes sense only for the assignment file and Quartus EDA tool.

CHAPTER

SIX

SUBCOMMANDS

6.1 resolve

The `resolve` subcommand resolves mapping tree and prints it to the standard output. It takes only single positional argument, namely the tree file to resolve. The `resolve` subcommand is useful for debugging and in initial design phases to quickly check what is connected with what.

Listing 6.1: Example `resolve` subcommand call.

```
python fp2p.py resolve path/to/your/tree_file.yml
```

Listing 6.2: Example `resolve` subcommand output.

```
{'board_1': {'end_pin_4': {'pin': 'A4', 'terminal': None}},
'board_2_connector_1': {'end_pin_1': {'pin': 'A1', 'terminal': None},
                        'led_0': {'pin': 'C2'}},
'board_2_connector_2': {'end_diff_pin_n': {'pin': 'B12', 'terminal': None},
                        'end_diff_pin_p': {'pin': 'B13', 'terminal': None},
                        'end_pin_2': {'pin': 'A2', 'terminal': None},
                        'end_pin_3': {'pin': 'A3', 'terminal': None}}}
```

6.2 graph

The `graph` subcommand renders the graph for tree file. It has been added to ease debugging of relatively long tree files. After exceeding a certain number of nodes in the tree file it becomes a bit hard to see how nodes relate with each other and what files constitute particular nodes. With graph it becomes trivial.

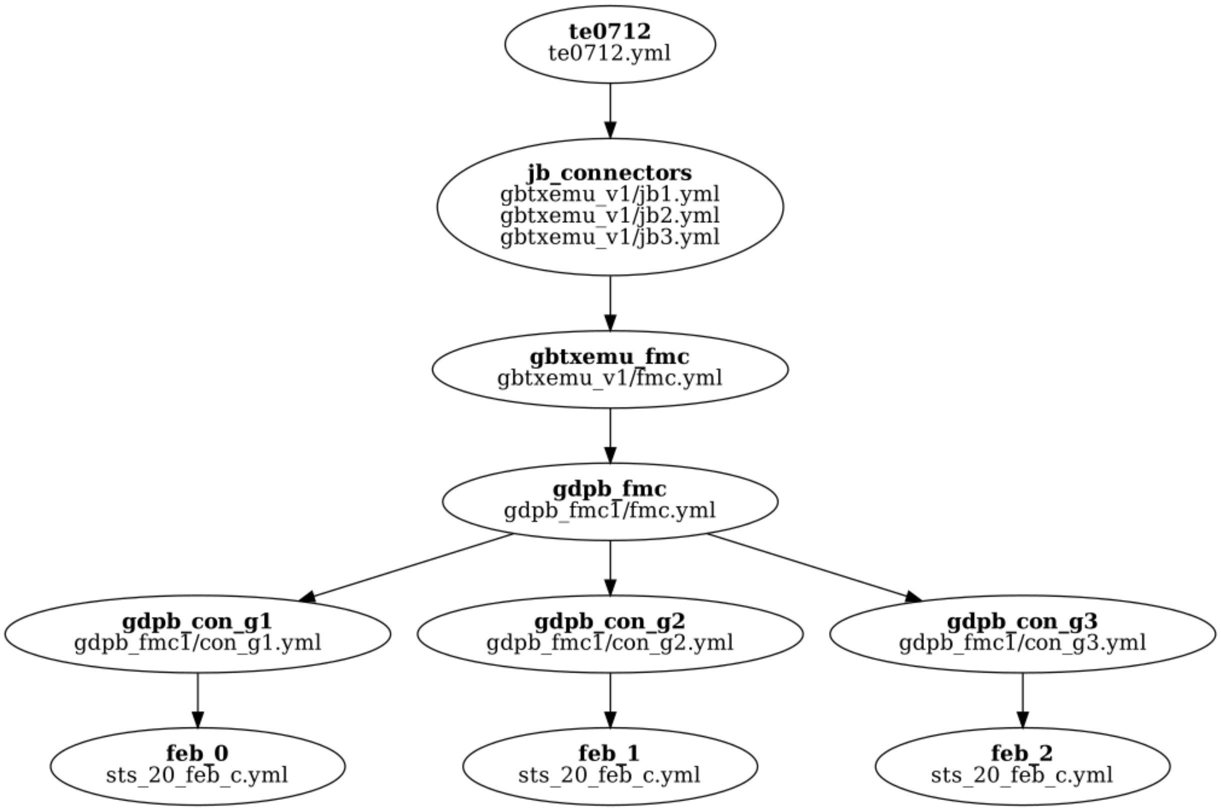

Fig. 6.1: Example output graph.

Listing 6.3: Example `graph` subcommand call.

```
python fp2p.py graph path/to/your/tree_file.yml
```

Fig. 6.1 shows an example output graph. Bold text within the node is the node name. Below the node name there is a list of files constituting given node.

6.3 assign

The `assign` subcommand is used for generating design constraint file. This is the core reason why the fp2p tool has been implemented.

Listing 6.4: Example `assign` subcommand call.

```
python fp2p.py assign tree.yaml assignment.yaml vivado
```

Listing 6.5: Example auto generated design constraint output.

```
# This file has been auto generated by the fp2p tool.
# Do not modify it by hand!
# Files used for generation:
#   Tree file: tree.yaml
#   Assignment file: assignment.yaml
# More information on the website https://github.com/m-kru/fp2p.

set_property PACKAGE_PIN A1 [get_ports {port[1]}]
set_property IOSTANDARD LVDS_25 [get_ports {port[1]}]
set_property DIFF_TERM TRUE [get_ports {port[1]}]

set_property PACKAGE_PIN A2 [get_ports {port[2]}]
set_property IOSTANDARD LVDS_33 [get_ports {port[2]}]
set_property DIFF_TERM FALSE [get_ports {port[2]}]

set_property PACKAGE_PIN A3 [get_ports {port[3]}]
set_property IOSTANDARD LVDS_33 [get_ports {port[3]}]
set_property DIFF_TERM TRUE [get_ports {port[3]}]

set_property PACKAGE_PIN B12 [get_ports {diff_n}]
set_property IOSTANDARD LVDS_33 [get_ports {diff_n}]
set_property DIFF_TERM TRUE [get_ports {diff_n}]

set_property PACKAGE_PIN B13 [get_ports {diff_p}]
set_property IOSTANDARD LVDS_33 [get_ports {diff_p}]
set_property DIFF_TERM TRUE [get_ports {diff_p}]

set_property PACKAGE_PIN A4 [get_ports {port[4]}]
set_property IOSTANDARD LVDS_33 [get_ports {port[4]}]
set_property DIFF_TERM TRUE [get_ports {port[4]}]
```

CHAPTER

SEVEN

MISTAKES DETECTION

Safety and support in finding mistakes were among the core goals of the implementation. The idea of the program is to report mistakes as soon as possible and exit with verbose mistake description. It does not gather mistakes to report them all at the end, but reports them immediately at a given check point. The application is capable of detecting and reporting the following mistakes:

1. Unassigned terminal pins - terminal pins not assigned to any ports.
2. Dangling terminal pins - terminal pins not mapped to any FPGA pins.
3. Assigning to non-terminal pins.
4. Conflicting mappings in each tree node. It detects such conflicts even if a given tree node is defined in multiple files.
5. Conflicting assignments - the same port assigned twice.
6. Assigning to missing pins.
7. Mapping to terminal pins.
8. Duplicated node names - at least 2 nodes with the same name.

CHAPTER

EIGHT

TIPS AND QUIRKS

8.1 TRUE and FALSE in design constraint properties

By default 'TRUE', 'true', 'True' are interpreted by YAML as boolean and then printed to the file as 'True' string. In .xdc file we want 'TRUE' string so double qoutes " are needed. Not sure which EDA tools can handle correctly 'True' or 'true' strings, but Xilinx Vivado can't. The same applies to the "FALSE".

```
_default_:
  set_property:
    IOSTANDARD: LVDS_33
    # Use "TRUE" or "FALSE" strings for boolean properties.
    DIFF_TERM: "TRUE"
```

8.2 Net names on schematics

It is always preferred to put numbers at the end of net names in the board schematics files. This is because in case of HDL, the array indexes are always at the end. For example, following declaration:

```
_default_:
  regex:

foo_[pn]_\[[0-4]\]:
  end: bar_[0-4]_[pn]
```

leads to improper assignment as order in natural sorting is different than what user expects. This is because [0-4] is before [pn] in bar_[0-4]_[pn].

Instead user needs to write it in a much more verbose way:

```
_default_:
  regex:

foo_[pn]_\[0\]:
  end: bar_0_[pn]

foo_[pn]_\[1\]:
  end: bar_1_[pn]

foo_[pn]_\[2\]:
```

(continues on next page)

(continued from previous page)

```
  end: bar_2_[pn]

foo_[pn]_\[3\]:
  end: bar_3_[pn]

foo_[pn]_\[4\]:
  end: bar_4_[pn]
```

Such verbosity could be avoided if the nets on the schematic were named in the following way `bar_[pn]_[0-4]` instead of `bar_[0-4]_[pn]`. Then following declaration

```
_default_:
  regex:

foo_[pn]_\[[0-4]\]:
  end: bar_[pn]_[0-4]
```

would work as user expects.

CHAPTER NINE

INDICES AND TABLES

- genindex
- modindex
- search