# A dummy title

A dummy Author

October 1, 2018

**Abstract** Due to increasing size and complexity of modern hardware designs, the challenge of identifying a piece of design becomes increasingly difficult. This is especially true, if no documentation is available. This factor has a direct impact on the time that is needed to get familiar with a design. In extreme cases, the design is rendered useless for the user. A hint on what hardware category the design belongs to, would accelerate the process of familiarization. This work considers, if it is possible to categorize hardware designs, that are given as Hardware Description Language, on basis of their structure. The elaborated algorithm is able to categorize a given design in X seconds, with an accuracy of S.

**Kurzfassung** Mit steigender Größe und Komplexität von modernen Hardware Designs, wird es zusehends herausfordernder die Funktion des desselben zu identifizieren. Vor allem trifft dies zu, wenn keine Dokumentationen zum Design verfügbar sind. Dieser Umstand wirkt sich unmittelbar in einer erhöhten Einarbeitungszeit aus. In Extremfällen muss der Anwender das Design wegen Unbrauchbarkeit verwerfen. Ein Hinweis darauf welcher Hardware Kategorie das Design angehört, würde den Einarbeitungsprozess beschleunigen. Diese Arbeit untersucht, ob es möglich ist Hardware Designs, die als Hardware Description Language vorliegen, anhand ihres strukturellen Aufbaus zu klassifiziern, und in Kategorien einzuteilen. Mithilfe des erarbeiteten Algorithmus ist es möglich ein Design innerhalb von X Sekunden, mit einer Sicherheit von Y zu klassifiziern.

# Contents

# Glossary

**Hardware Description Language** dummy entry. 8, 11, 12

**OpenCores** OpenCores.com is a sharing platform for open source hardware designs.. 8, 9, 10, 11

**OpenCores** dummy entry. 11

**OpenCores** dummy entry. 12

**Unique Resource Locator.** dummy entry. 8

# 1 Introduction

As Embedded Systems find their way in an increasingly wide field of applications, with growing demands to performance and relieability, the underlying Hardware Designs also gain in diversity, complexity and size. Since it is nearly impossible, even for simple Designs, to determine the function of such, without proper documentation, a possibility to extract information directly from the Hardware Description Language representation of the Design, seems to be a welcome aid. Such a hardware design classification algorithm also allow services, which automatically handle aforementionend designs (for instance online sharing plattforms like glsoc) to assign categories without relying on user input.

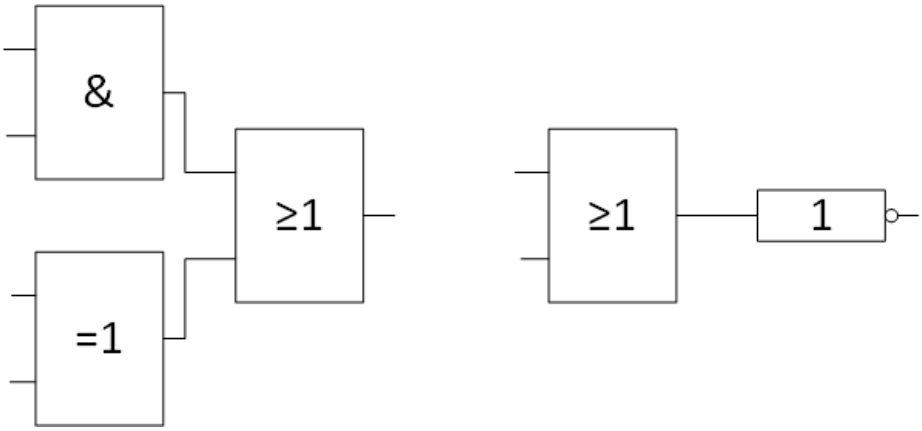


Figure 1: Example of a Two to One Connection (left) and an One to One Connection (right).

This work tries to achieve a classification of hardware designs by analyzing the connection between logical cells. The count of specified two to one and one to one connections (examples depictioned in illustration), are translated into a standardized match vector. It is expected that the match vector of similar designs gather in clusters. These clusters can then be understood as hardware categories. Using a methodolodgy like this implies, that the location and meaning of those clusters first have to be identified, by analyzing a great amount of well known hardware designs. The clustering is also in scope of this work, and is achieved by a web crawler, which gathers already categorized designs from the open source sharing plattform glsoc. These designs are then synthesized by the open source synthesis tool glsyosys and handed over to the match algorithm, which eventually determines the match vector. The so gathered set of match vectors are then grouped into clusters.

The result of this work is a ¡command line application¿ which is able to categorize large designs (XXX Cells) in X seconds with a certainty of YThe following chapters elaborate on the development, set of problems and other possible applications of this work.

# 2 State of the Art

Literature regarding this topic is very scarce, if not non-existent. At the time this work has been released, no other publications, which attempted to establish an algorithm to automatically identify hardware designs, could be found. Papers which worked on topics remotely relatable with the topic of hardware categorization, mostly presented methods to identify hardware trojans in a given design. This Hardware Trojan Identification can be seen as categorization into two groups: Non Hardware Trojan injected, and Hardware Trojan injected. Identification of those Trojans is mostly achieved via a functional analysis [put sources here], where the design is simulated and tested for a certain behaviour. Since our method aims for a structural analysis, these publications are hardly compareable to ours. Though one publication used a combination of structural- and functional analysis, to identify hardware trojan design patterns. This methodolodogy should be further looked upon.

## 2.1 Detection of Hardware Trojans

In [literature] X net types are defined, which are typical for hardware trojans. shows those proposed Hardware Trojans nets. Similar to our proposed method, the count of these nets are determined.

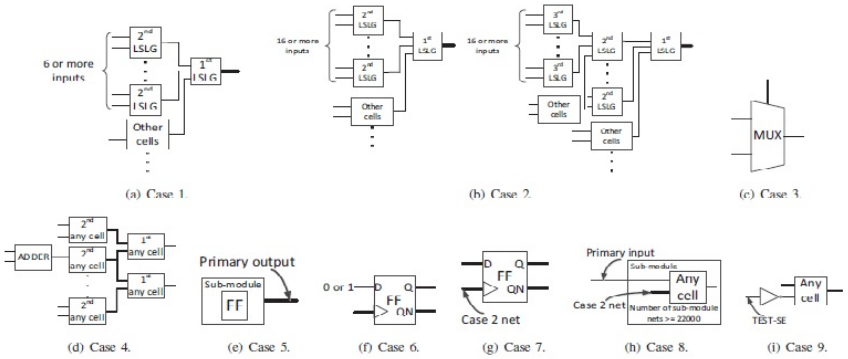


Figure 2: Register Transfer Level depiction of the proposed Hardware Trojan nets

The publication states, that trough the count of certain net types, the presence of hardware trojans can be determined, but not their absence. This insight leads to the assumption, that hardware design can be classified using the count of certain net types.

# 3 Used Technologies

blabla

## 3.1 Python

blabla

## 3.2 Scrapy

blabla

## 3.3 Yosys

blabla

## 3.4 Hardware Description Languages

blabla

# 4 Metholodgy

To achieve our goal of the categorization of hardware designs trough a structural analysis of the corresponding HDL Design, we first propose a standardized match vector, which is calculated from the Design's Two-to-One and One-to-One Gate Level Connections (see  for referencce). The vector gets standardized, so small designs remain compareable to bigger ones. This vector shall then act as categorization criteria, to sort the design into predefined design categories. Those categories are clusters in the vector space the match vector is part of. If the match vector points into such a cluster, the design is identified with the corresponding category. This section elaborates on the details of finding a match vector of a design, and how the category clusters are defined.

## 4.1 Cluster Identification

As mentionend above, before any categorization can take place, the categories itself have to be defined. In our model, categories are clusters in a n-dimensional vector space. Therefore, the task is to identify clusters in this vector space, and name them. To do this, we determine the match vector of well known designs. Since the categorization of those designs is already available, the vectors that originate from designs with identical categorizations, can be grouped together to clusters.

It is expected, that the validity of such clusters increases with the number of analyzed designs. Therefore an automatic process has to be established, to enable the analysis of a great amount of designs. To achieve this, we use Scrapy, a Python web scraping tool, to automatically obtain design files and corresponding meta-information. OpenCores serves as source for those design files, and provides following information:

1. Unique Resource Locator. to the design archive
2. The name of the design project
3. The Hardware Description Language in which the design is specified and implemented
4. The category in which a design is listed
5. The Hardware Description Language files of the design

The following chapters explain in detail, how those informations are gathered, and processed into a match vector.

### 4.1.1 Scraping OpenCores project web sites

OpenCores does not offer an interface to automatically download all available design files. Therefore it has to be resorted to the web page, which is optimized

for human interaction, but not for machine readability. Tools, which are specialized in translating human readable content into structured, machine oriented data sets, are called Scrapers. Scrapy is one of them, and provides multiple python classes which enable extensible web scraping.

Scrapy's input is the HTML representation of the web page, which it is able to retrieve by itself, by providing a URL link to the web page. The HTML code is scanned for its elements. Those are then parsed into python classes, and exposed to the user. The content of the parsed classes can be reorganized into user defined python classes, and passed to other user defined functions, which, among others, can store those informations persistently on a filesystem. Another option is to use HTML objects like Hyperlinks to further fork into the web page.

Listing 1: Scrapy Main Class

```
class IPspider_oc(scrapy.Spider):
    """The class, which is called by the scrapy framework. """
    name = 'ipspider_oc'   # the name of the spider

    def start_requests(self):
        return [scrapy.Request(
          url = "https://opencores.org",
          callback=self.login)]

    def login(self, response):
        return scrapy.FormRequest.from_response(
          response,
          formdata={'user' : 'davFreismuth',
                    'pass' : 'Qlghkeul'},
          callback=self.redirect )

    def redirect(self,  response):
        return scrapy.Request(
          url = "https://opencores.org/projects?lang=0&stage=5&
              license=0&wishbone_version=0",
          callback=self.parse )

    def parse(self, response):
        """Default callback for scrapy. Starts at start_url,
            fetches the url of the different project pages, and
            calls parse_metadata for each project page"""
        for href in response.css('td.project_a::attr(href)'):
            yield response.follow(href, self.parse_metadata)

    def parse_metadata(self, response):
        """parses metadata of an opencore.org project page and
            returns a scrapy item object."""
        def scrape_line(strArray,  removeStr):
            '''helper function which iterates trough the string
                array 'strArray', selects the first entry which
                contains 'removeStr' and returns the string without
                 removeStr and whitespaces.'''
            for str in strArray:
                if(str.find(removeStr) != -1):
                    retStr = str.replace(removeStr,  '')
```

```
34                      return retStr.strip()
35
36          def scrape_href(line):
37              """helper function, which gets an html <a> element,
                    checks if it has inner html and returns the text of
                     the inner html"""
38              start = line.find(">")
39              end = line.find("<", start)
40              if((start + 1) == end):
41                  return ""
42              else:
43                  return line[(start+1):(end)]
44
45          #fill up the fields
46          hdl_IP =  HDL_IP()
47          hdl_IP['name'] = scrape_line(response.css('h2 + p::text').
                extract(), 'Name:')
48          hdl_IP['created'] = scrape_line(response.css('h2 + p::text'
                ).extract(), 'Created:')
49          hdl_IP['updated'] = scrape_line(response.css('h2 + p::text'
                ).extract(), 'Updated:')
50          hdl_IP['file_urls'] = ['https://opencores.org' + response.
                css('p a::attr(href)')[1].extract()]
51          category = scrape_href(response.css('p a')[5].extract())
52          category = category.lower()
53          category = category.replace(' ', '_')
54          hdl_IP['category'] = category
55          hdl_IP['language'] = scrape_href(response.css('p a')[6].
                extract())
56          hdl_IP['license'] = scrape_line(response.css('h2 + p::text'
                ).extract(),  'License:')
57          hdl_IP['basePath'] = self.settings['FILES_STORE']
58          yield hdl_IP
```

**Authentication**

For our application, the first challenge is, to authenticate via the OpenCores user system, since designs are only downloadable while logged in with a cost free OpenCores account. Login information is often communicated via HTML POST Messages, which encode the message content into the message header. In the login case, the login information is encoded into the header. The web page returns an authentication token, which is stored as cookie and enables access to the web page. Scrapy is natively able to generate send such POST messages and exposes the answer to such a POST messages in Python classes, for further actions.

**Traversing of the Web Page**

As OpenCores's web page is built, like most web pages, in a tree like structure (see ¡¡LABEL¿¿ for reference), scrapy has been configured to start from the root page ¡¡URL¿¿, where the links to all projects reside. It then calls each project page, by filtering for the ¡h¿ elements of the

Like most web pages, OpenCores is built in a tree-like structure. The scraped

data is temporarily stored into Python class objects. The information in these objects are written to a .JSON file, in order to make them persistent (such that we do not have to scrape the entire OpenCores web site every time we use our classification framework). For this .JSON file, a specific format has been defined, so information can be imported from other sources then scrapy aswell.


Figure 3: .JSON File Format Description

Figure Abbildung 3 shows an example of the content of the .JSON file.

### 4.1.2 Downloading OpenCores projects

The Scrapy python module provides the functionallity to automatically download and store files that are associated with a Hardware design project, to a predefined folder. Since OpenCores requires an account to be able to download hardware designs, we use a scrapy internal function to send a POST request to the OpenCores webpage, in order to authorize ourselfs.

Because all project files from the OpenCores database are provided as tar.gz archives, we introduced an additional decompression step.

### 4.1.3 Decompressing design files

Projects from OpenCores are solely provided as *.tar.gz compressed archive. The OpenCores module tarfile enables the extraction of those archives. Additionally to the decompression, we sort the project files into folders corresponding to their associated hardware category (as stated by OpenCores). After that, the file endings of the files are analysed. If those endings indicate a Hardware Description Language file, the path to this file is added to the aforementioned .JSON file, in order to be able to address the design files of a project in later steps.

#### 4.1.4 Reading designs into synthesis tool

After the design has been decompressed, it is time to let the synthesis tool yosys read the design files and synthesise them into a text file format. In order to do this, yosys expects a yosys script file, which holds all yosys commands that should be executed on a set of files. This script file can either be provided by the user before a programm run, or a generic one can be generated automatically during a programm run, according to the files that have been provided with the .JSON file.

The synthesis tool's frontend is chosen based on the language of each design file. For OpenCores and SystemVerilog, the `verific` frontend is used. For Verilog, we use the `read_verilog` frontend. Once any combination of Hardware Description Language files has been loaded, a synthesize run attempts to generate a single HDL file from the provided files, which solely contains primitive logic blocks.

Since yosys does not support automatic dependency recognition of vhdl files, a custom solution had to be found, to determine the load order of vhdl files (in the case that the user decides that yosys scripts should be generated automatically). To accomplish this, we slightly modified the Vunit python project, which offers a function to return the OpenCores files in an ordered list. The vhdl files can then be loaded in the order dictated by this list.

#### 4.1.5 Naming designs

Each design that is read by the synthesis tool is named after the project from which the design files are downloaded. From then, the design name is the main reference for each design and serves as identification feature in all subsequent steps.

#### 4.1.6 Matching designs against standard pattern vector

#### 4.1.7 Calculating clusters of design match vectors

### 4.2 Design Identification

blabla

### 4.3 Verification

blabla

# 5 Discussion

# 6 Conclusion