David FREISMUTH

Bachelor Thesis

384.088, Winter Term 2018

Supervisor:

DI Christian Krieg

November 2, 2018

Development of a testing environment for an automatic Hardware Design Classification

Contact us:

christian.krieg@alumni.tuwien.ac.at

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

1 Introduction **5**

2 State of the Art **6**
- 2.1 Detection of Hardware Trojans 6

3 Methodology **7**
- 3.1 Cluster Identification 7
  - 3.1.1 Fetch 8
  - 3.1.2 Download 10
  - 3.1.3 Decompress 11
  - 3.1.4 Synthesis 11
  - 3.1.5 Find Pass 11
  - 3.1.6 Clustering 12
- 3.2 Design Identification 12
- 3.3 Verification 12

4 Conclusion **13**

5 Discussion **14**

Glossary

cookie Give me a description!. 10

CSS A language to define the Layout of a webpage. 10

HDL Hardware description language; A language to describe the function of Digital Hardware textually.. 4, 7, 11

HTML Hyper Text Markup Language; A language to describe the layout of text and media.. 8, 10

hyperlink Points to another location/ressource on the internet. 8

OpenCores A wegpage (opencores.org), which provides open source Hardware Design Language Intellectual Properties . 7, 8, 10, 11, 15

post Give me a description!. 10

Python A interpretive programming language. 8, 10, 11

Scraper A piece of software, that transforms information optimised for human readability into machine oriented data sets.. 4, 7

Scrapy A Web Scraper based on Python. See also Scraper. 7, 8, 10, 11, 15

URL A name used to address a ressource located on a computer or in a computer network . 7, 8, 15

VHDL Very-high-speed integrated circuits hardware description language. See also hardware description language (HDL). 11

Yosys An open source synthesis tool. . 11

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.

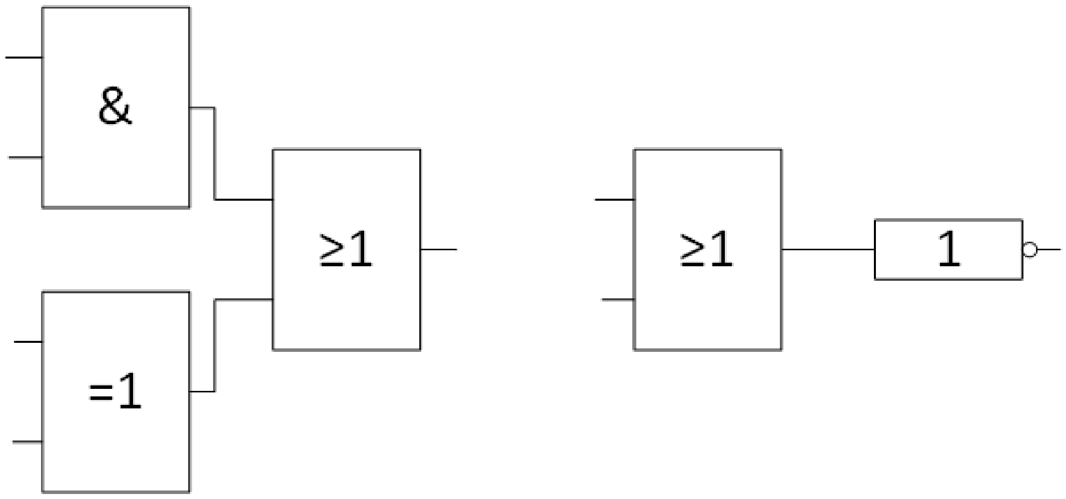


Figure 1: Example of a Two to One Connection (left) and an One to One Connection (right).

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 timethis 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 [1] 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.

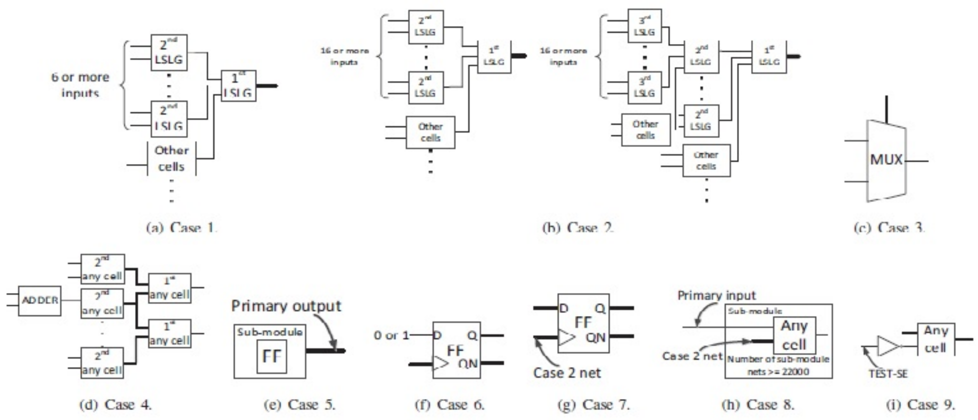


Figure 2: Register Transfer Level depiction of the proposed Hardware Trojan nets [1]

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 Methodology

To achieve our goal of the categorization of hardware designs trough a structural analysis of the corresponding HDL Design, we firstly propose a non-standardized match vector, which contains the count of the Design's Two-to-One and One-to-One Gate Level Connections (see referencce) as each of its entries, and secondly a standardized match vector, which reduces the information hold by the non-standardized match vector down to a two dimensional vector. This standardized match vector shall then act as categorization criteria, to sort the design into predefined design categories. Those categories are clusters in the vector space the standardized 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.

3.1 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 two dimensional vector space. Therefore, the task is to identify clusters in this vector space, and name them. To do this, we determine the match vectors 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. Conveniently, the website OpenCores offers a great collection of Hardware Designs that already have been categorised manually. OpenCores though 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. Luckily there are Tools, which are spezialized in translating human readable content into structured, machine oriented data sets. These are called Scrapers. Scrapy [2] is a python framework that exposes the functionallities of such a Scraper, and provides multiple python classes which enable extensible web scraping. Because of its ease of use and extensibility, Scrapy has been chosen, to solve the majority of problems that come with the challenge of retrieving a great amount of Hardware Designs.

Among the set of information that is specifically provided for each design by OpenCores, we fetch and process following information.

1. Uniform Ressource Locator (URL) to the design archive

   So the Cluster Identification can be validated at a later point in time (asuming the files still exist under the same link).

2. The name of the design project

   Used to address single designs.

3. The HDL in which the design is specified and implemented

   Necesarry for the synthesis process, later in the cluster identification process

4. The category in which a design is listed

   Mandatory for the Cluster identification process.

5. The HDL files of the design

   Trivially necesarry to analyse the hardware design.

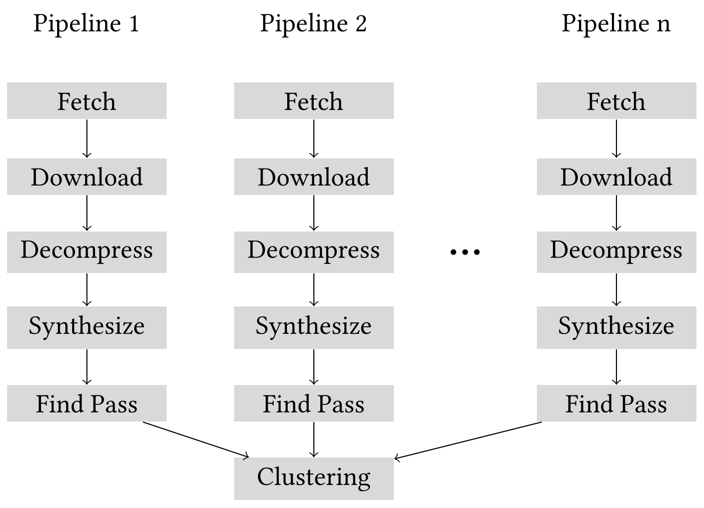


Figure 3: Pipeline stages of the Scrapy implementaion. As seen in the figure, Pipelines can be executed parallelly. The final stage "Clustering" is not implemented as pipeline stage, but rather waits on the completion of all pipelines to process all the gathered data.

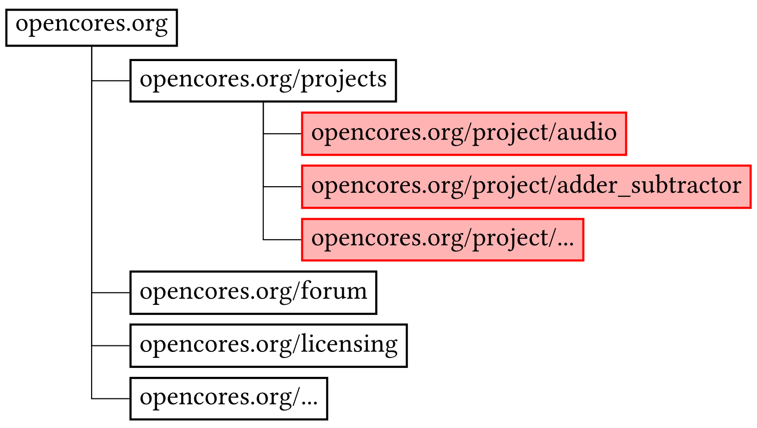


Figure 4: The structure of the OpenCores website. The URLs where the projects are located, are marked red.

Scrapy is able to process data within a pipeline structure, which, in our case, enables us to push user defined data sets trough those stages, an perform specific actions on them. The advantage herein lies in the parallelity, and therefore speed increase, that can be achieved by using this strucutre, since multiple pipelines can be active at a given time. section 3.1 shows the pipeline stages of the scrapy implementation that has been developed during the run of this thesis. The following chapters explain in detail, how those Pipeline stages manipulate the data stated in the enumeration above.

3.1.1 Fetch

In this stage, the OpenCores site is called to find the URLs where the hardware designs and corresponding meta informaiton resides. As OpenCores's web page is built, like most web pages, in a tree like structure (see fig. 4 for reference), Scrapy has been configured to start from the root page www.opencores.org, where the links to all projects reside (corresponds to the function call in listing 1 line »XXXX«). "Start" in this context means, that the Hyper Text Markup Language (HTML) representation of the site is retrieved by Scrapy and exposed via a Python class to the User. Scrapy then offers functions to extract informations from the HTML code, like filtering for all <h> elements (which represent Hyperlinks that fork deeper into the website.).

As seen in listing 1 line »XXXXX« for each »element«< that has been found, a new function

»>function«< is invoked, which further calls the webpage the »elements« points to.

```
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' : 'Q1ghkeul'},
         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,  '')
                   return retStr.strip()

       def scrape_href(line):
           """helper function, which gets an html <a> element, checks if it has
   inner html and returns the text of the inner html"""
           start = line.find(">")
           end = line.find("<", start)
           if((start + 1) == end):
               return ""
           else:
               return line[(start+1):(end)]

       #fill up the fields
       hdl_IP =  HDL_IP()
       hdl_IP['name'] = scrape_line(response.css('h2 + p::text').extract(), 'Name
   :')
       hdl_IP['created'] = scrape_line(response.css('h2 + p::text').extract(), '
   Created:')
       hdl_IP['updated'] = scrape_line(response.css('h2 + p::text').extract(), '
   Updated:')
       hdl_IP['file_urls'] = ['https://opencores.org' + response.css('p a::attr(
   href)')[1].extract()]
```

```python
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
```

Listing 1: Scrapy Main Class

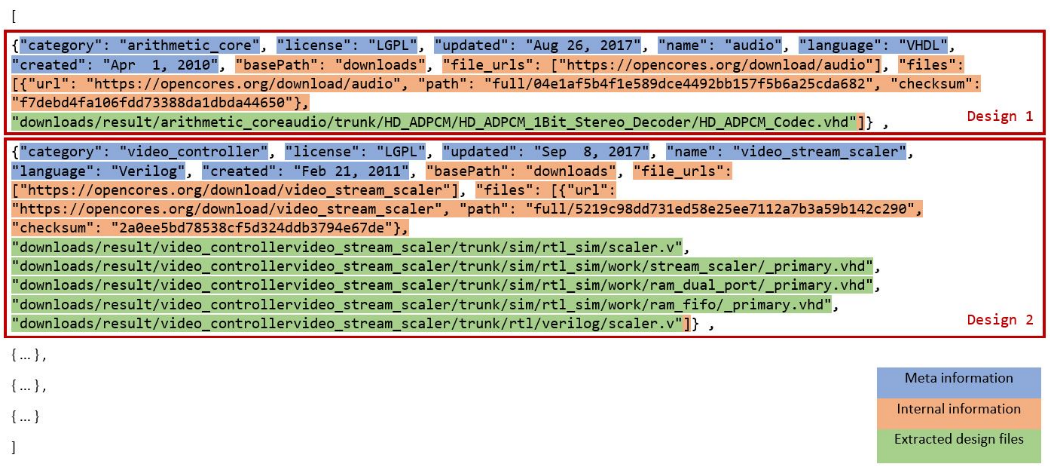
Authentication

Besides navigating to and trough the webpage, another task for the Fetch stage is the authentication with 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 message header. After the login information has been accepted by the web page, it returns an authentication token, which is stored as Cookie and enables access to the web page. Scrapy is natively able to generate and send such POST messages and exposes the answer to such POST messages in Python classes, for further actions. The function which sends the POST message can be seen in listing 1 line »XXXXX«.

Extraction

Once Scrapy navigated to the project page, the above mentionend information is extracted by using Cascading Style Sheet (CSS) selectors, as seen in listing 1 line >XXXXXX< to »YYYYYYYY«. The so gathered data sets are stored in the user defined Python HDL_IP object (derived from the Scrapy class item), which serves as container that gets pushed trough the Pipeline stages. The Fetch stage ends with returning the HDL_IP object to the Scrapy framework, where it gets handed over to the next Pipeline stage.



```
[
{"category": "arithmetic_core", "license": "LGPL", "updated": "Aug 26, 2017", "name": "audio", "language": "VHDL",
"created": "Apr  1, 2010", "basePath": "downloads", "file_urls": ["https://opencores.org/download/audio"], "files":
[{"url": "https://opencores.org/download/audio", "path": "full/04e1af5b4f1e589dce4492bb157f5b6a25cda682", "checksum":
"f7debd4fa106fdd73388da1dbda44650"},
"downloads/result/arithmetic_coreaudio/trunk/HD_ADPCM/HD_ADPCM_1Bit_Stereo_Decoder/HD_ADPCM_Codec.vhd"]} ,
{"category": "video_controller", "license": "LGPL", "updated": "Sep  8, 2017", "name": "video_stream_scaler",
"language": "Verilog", "created": "Feb 21, 2011", "basePath": "downloads", "file_urls":
["https://opencores.org/download/video_stream_scaler"], "files": [{"url":
"https://opencores.org/download/video_stream_scaler", "path": "full/5219c98dd731ed58e25ee7112a7b3a59b142c290",
"checksum": "2a0ee5bd78538cf5d324ddb3794e67de"},
"downloads/result/video_controllervideo_stream_scaler/trunk/sim/rtl_sim/scaler.v",
"downloads/result/video_controllervideo_stream_scaler/trunk/sim/rtl_sim/work/stream_scaler/_primary.vhd",
"downloads/result/video_controllervideo_stream_scaler/trunk/sim/rtl_sim/work/ram_dual_port/_primary.vhd",
"downloads/result/video_controllervideo_stream_scaler/trunk/sim/rtl_sim/work/ram_fifo/_primary.vhd",
"downloads/result/video_controllervideo_stream_scaler/trunk/rtl/verilog/scaler.v"]} ,
{…},
{…},
{…}
]
```

Figure 5: .JSON File Format Description

Figure Figure 5 shows an example of the content of the .JSON file.

3.1.2 Download

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. The download is started by pushing an item object with an URL atrribute into a pipeline stage declared as a donwload stage. The Scrapy framework then begins the download of the file specified in the URL atrribute to the path defined in the atrribute spider . settings ['FILE_STORE']. If the file has been downloaded successfully, its name gets added to the HDL_IP object under the attribute item. file .

3.1.3 Decompress

Projects from OpenCores are solely provided eiter as *.tar.gz or *.zip compressed archive. The stage 'Decompress' is therefore dedicated to decompress the previously downloaded archive and to order the contained files into an predefined structure. The Python modules tarfile and zipfile enable the extraction of such archives. After the decompression, the project files are sorted 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 HDL file, the path to this file is added to the aforementioned HDL_IP object, in order to be able to address the design files of a project in later stages. If no such files could be found, the HDL_IP object is discarded, and will not be further processed.

3.1.4 Synthesis

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 in an automated manner, for each discovered design, a yosys script file is generated on the basis of the information, that has been gathered in previous steps. This Script instructs Yosys about the path to the discovered files and with which frontend to load them. A sample script file can be seen in listing 2. An alternative for this automatic script generation is to supply a custom made script, which will then be loaded instead of the generic one. Bigger design may need to be synthesised with such a custom script, since the dependecies may be much to complex, to be computed automatically.

```
 asdfsdafasdfasdfasdfsdaf
\end{listing}
\caption{An example Yosys Script file.}
\label{lst:yosysScriptFile}

```

Listing 2: An example Yosys Script file.

The synthesis tool's frontend is chosen based on the language of each design file. For very-high-speed integrated circuits hardware description language (VHDL) and SystemVerilog, the verific frontend is used. For Verilog, we use the read_verilog frontend. Once any combination of HDL 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 VHDL files in an ordered list. The vhdl files can then be loaded in the order dictated by this list.

3.1.5 Find Pass

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.

3.1.6 Clustering

blabla

3.2 Design Identification

blabla

3.3 Verification

blabla

4 Conclusion

5 Discussion

References

[1] M. Oya, Y. Shi, M. Yanagisawa, and N. Togawa. “A score-based classification method for identifying Hardware-Trojans at gate-level netlists”. In: (Mar. 2015), pp. 465–470. ISSN: 1530-1591. DOI: 10.7873/DATE.2015.0352.

[2] *Source of Scrapy*. URL: www.scrapy.org.

List of Figures

1 Example of a Two to One Connection and an One to One Connection 5

2 Register Transfer Level depiction of the proposed Hardware Trojan nets [1] 6

3 Pipeline stages of the Scrapy implementaion. As seen in the figure, Pipelines can be executed parallelly. The final stage "Clustering" is not implemented as pipeline stage, but rather waits on the completion of all pipelines to process all the gathered data. 8

4 The structure of the OpenCores website. The URLs where the projects are located, are marked red. 8

5 .JSON File Format Description 10