NTNU – Trondheim
Norwegian University of
Science and Technology

Analysis and Visualisation of Clock Tree Power in a full-chip-design

Hans Jørgen Myrvang Ro

Electronics System Design and Innovation

Submission date:	June 2014
Supervisor:	Snorre Aunet, IET
Co-supervisor:	Jan Egil Øye, Nordic Semiconductor

Norwegian University of Science and Technology
Department of Electronics and Telecommunications

Analysis and Visualisation of Clock Tree Power in a full-chip-design

Hans Jørgen Ro

June 2014

MASTER THESIS
Department of Electronics and Telecommunication
Norwegian University of Science and Technology

Supervisor 1: Snorre Aunet
Supervisor 2: Jan Egil Øye

Preface

This thesis is the final work of my masters degree in digital circuit design at NTNU. The project was suggested by Jan Egil Øye at Nordic Semiconductor. I have worked together with him and other engineers in the digital design department at Nordic in the design of this tool. My advicor Snorre Aunet has guided me in writing this report.

Trondheim, 2014-06-07

Hans Jørgen Ro

Acknowledgment

I would like to thank the following persons for their great help during my thesis:

- Jan Egil Øye, my adviser at Nordic Semiconductor for suggesting the project and giving feedback with great enthusiasm. It really helped to achieve a useful tool. Your insight on the problems and your constructive feedback has helped me immensely in writing an interesting and useful thesis.
- Snorre Aunet, my adviser at NTNU for giving me feedback on the writing of this thesis. Keeping a fun project academically relevant is hard without a guide.
- Nordic Semiconductor for giving me a good work environment to focus on me thesis in, and everyone at the digital design group for being interested in what i have been doing.

H.J.R

Summary

In this thesis a tool to graph power density in a chip by combining placement data with power estimation results has been made for Nordic Semiconductor. The goal was to utilize the data generation power of the power estimation tool, preserving the detail of the data instead of looking at averages and totals. Extra focus was put on displaying power from clock trees, as they are generated in a part of the design process where power estimation data based on placement and routing is available. This thesis presents results from a tool able to graph power consumption over the chip area, a feature not seen in commercial tools. A thorough description of the features, how they were obtained and why they were needed is also included. The tool was developed with continuous feedback from the engineers that will be using it. It was implemented in the design phase for Nordic Semiconductor chip currently in progress to gain insight in the clock trees. The tool showed that several test benches used for estimating clock power were in fact printing the exactly the same output, meaning they were redundant.

The tool transform power estimation reports to frames optimized to be joined together for animations. It can process reports with an average run time of 1.5 seconds per report when each report contains 40 000 cells. Rendering each frame takes an average of 2.1 seconds. The tool helps the designer to gather information on ineffective layout in reports that were previously deemed too large to be useful. Visualizing data over the chip surface shows rich information in a intuitive way, which can be useful for other domains.

Contents

Chapter 1

Introduction

This thesis describes the design and test process of a graphing tool made on a request from Nordic Semiconductor. The tool visualizes when and where on a chip power is consumed by combining reports from the Primetime PX' power estimation tool with location information from the IC compiler. The aim is to use it to improve the understanding of the generated clock trees so they can be made more efficient.

1.1 Background

Nordic Semiconductor is a Norwegian company that focuses on ULP(Ultra Low Power) chips for short range wireless communication. These chips can be found in battery driven wireless products produced around the world ranging from computer mice to heart rate monitors. In the ULP wireless chip industry reducing power is always important. This thesis was suggested by Jan Egil Øye to gain new tools for reducing power consumed by the clock tree. The clock tree is a significant source of power consumption, often taking up as much as 40% of the total power[1]. Clock tree power consumption will only be harder to limit as circuits scale down, and it is along with leakage power one of the main hurdles of chip scaling[3]. Many different power reduction techniques for clock trees have been proposed[1]. Understanding the power consumption is key to using the right methods of reducing it, and there is currently no commercial tools to visualize power density over a circuit surface over time. The idea of using a power consumption density plot to understand the circuit builds on a graph of IR drop made by Martin Olsson. It was made in relation to his work with Jhonny Phil who now works at Nordic and Per Larsson-Edefors, a professor at Chalmers[8]. The graph was a 2 minute animation that showed IR drop over 2 clock periods, and sparked interest in future

1.2 The goal of the thesis

The goal of this thesis is to give the designer a tool that can enrich data from estimation so that he or she can better evaluate the changes done to the clock tree. This should be achieved by a visualization tool that can combine location data from the placement and routing tool with detailed power data from the estimation into a power density plots. The plots should clearly show power density over the circuit surface, which is much more informative than the total power sum currently used. The location and detailed power data are not used in the current design process, but are easily produced by the tools already in use. The graphing tool should not add significantly to the data generation time, and be optimized for finding interesting data in the reports. The project caught my eye, as I had some experience with parsing reports for graphing. During my last summer internship I gained relevant experience by making a tool for running simulations on the newest build and graphing the history of the results to a web page. The opportunity to write for a well known company was also very tempting.

The basic request was to create a program able to parse reports containing power consumed per cell, resulting from from power estimations on the circuit. This was to be combined with placement data gathered from IC Compiler, a layout tool from synopsis, to create a new data set showing power density. The power density data was to be plotted in a three dimensional graph over the circuit layout and should also be able to animate variations over time. The specification of this graph was quite loose, and I worked together with the engineers who will use the tool in order to define the visual representation. By doing this I got very useful feedback, and was able to make sure the tool would have the features the designers would want. This way the program would fulfill its purpose and will be able to help engineers reduce power consumption in the clock tree by evaluating more detailed data.

1.2.1 Meaningful visual representation

Finding out what information is interesting and how to display it is a big part of this thesis. The system designers can easily generate vasts amount of data, but it is difficult to get access to rich data that is also simple to read. The graphs should help the engineers find the vital data points, and separate them clearly from the rest of the data. To achieve this you need to test the program with the engineers to figure out what questions are being raised, and how to find the answers in the data.

1.3 Outline

1.3.1 Theory

Gives the reasons to put focus on power consumption of clock trees. Covers how they are generated in the design process, and how the power usage is analyzed. Shows work flow of the tool is supposed to fit into, and how that will happen. Covers the programs used to generate the data needed to create the graphs.

1.3.2 Method

Describes the design process done in cooperation with the engineers at Nordic Semiconductor. Program features are explained with motivation of why they are needed. The program flow, and the explanation of each feature should ease the design process for someone wishing to make a similar tool tailored to other domains. Also describes how the program was included in the design process for final testing.

1.3.3 Results

What the graphs contain is presented in a clear manner. Presents graphs from the tool from several different user cases where points of interest are pointed out.

1.3.4 Discussion

The discussion goes through the results point by point and compares the result with the goal of the thesis. Special attention is put on how the features of the program highlights important data. The reader can also see how to further improve the work flow around the program as well as suggestion for other work.

Chapter 2

Theory

To understand how to build a tool that can help designers create better clock trees, you have to look at how they create the current clock trees. This chapter will present the environment for clock generation to help show the motivation. The tool itself is a programming issue, and high performance has not been in focus, as it is not necessary in this particular domain. The theory part is focused on the application of the tool for Nordic Semiconductor, and why it is needed. A lot of the insight in this chapter is based of meetings with Jan Egil Øye, where he explained the designing process.

2.1 Ultra Low Power challenges

The Internet of things is a term many use to predict the next step in technological advancement. I the current state a Internet of computers and smart phones exists, connecting people together for easy communication. The Internet of things describes object communicating with each other automatically to improve information flow. For objects to communicate they need wireless components to send messages. Ultra low power technology is vital to this idea, as many objects are not normally connected to power, so they need to be powered by batteries. There are many ways to extend battery life. The easiest way is to lower performance and removing features. Wireless chips are focused communication though certan wireless standards, so the performance and features are set by requirements of the standard used. The other way to reduce power use is to reduce wasteful switching and unnecessary activity. This means optimized placement and routing, and cutting off the power of inactive modules on the circuit. By gating the clock tree, you can also stop the clock from being propagated to the inactive modules. When the circuit is idle nearly the entire circuit will be cut from power. 7

2.1.1 Problems with scaling

As technology advances, the systems get more advanced, and the circuit elements become smaller. Scaling down circuits is great for the power used per computational power, but since the systems are becoming larger and more complex, this effect is not enough. New challenges and unexpected power consumers appear when the scale becomes smaller and circuits become larger. The paper "Impact of technology scaling in the clock system power"[3] concludes that clock tree power consumption will increase as circuits scale down. Together with leakage power, it will be one of the main challenges in keeping power consumption from scaling badly with size.

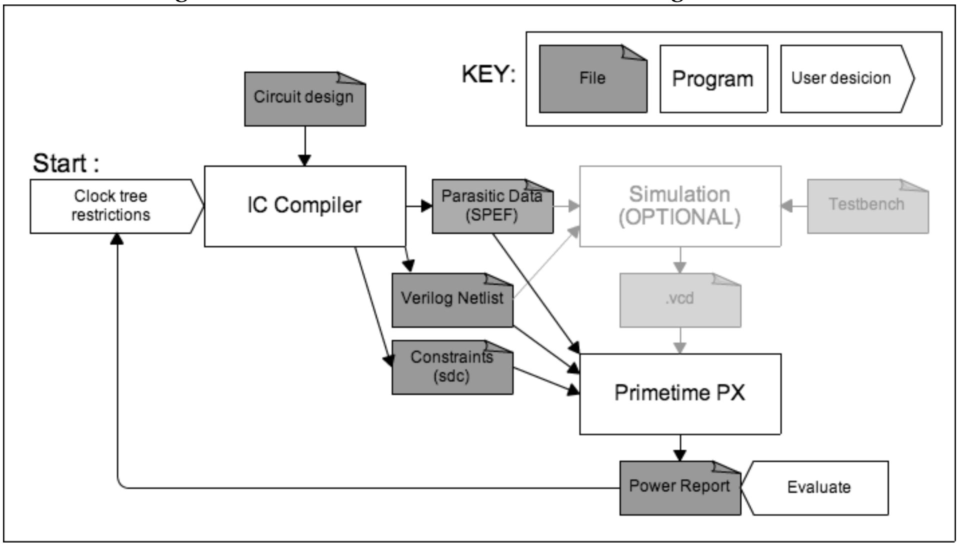
2.1.2 Clock skew and power aware clock generation

One of the main contributors to clock trees consuming power is that as the systems become larger, the clock trees have to be more complex. This has its roots in that the distance the clock signal has to travel increases relatively to its clock frequency. The distance the clock signal travels will result in a delay. Since clock signals are the time keeping mechanism of the circuit, a delay in the clock signal can result in signals arriving at incorrect times, ending in a circuit that does not work. The faster the clock frequency, the smaller delay is tolerable. To counter this, clock trees have to be balanced properly, making sure every part of the clock tree has traveled approximately the same distance. Clock tree generation is often handled by place and route tools like IC Compiler[12](more in section 2.3.1). Since the 80s the scientific community have been aware that the clock tree power consumption would need attention when planning clocks[10]. Since then, progress has been made on techniques that generate clock trees with a goal to minimize power in addition to keeping requirements to clock skew[1].

2.2 Clock tree design process

Through meetings with Jan Egil, I have gained insight in the part of the circuit design process where clock trees are generated. I will present the current design process, and where the tool in this paper fits in.

Figure 2.1: Current workf low for clock tree generation.



2.2.1 Current workflow

A flowchart showing the work flow used in Nordic Semiconductor can be found in Figure 2.1. After the Verilog description of the circuit is finalized, the Verilog net list is sent to Synopsis IC Compiler. IC Compiler is a place and route tool that will create a layout of the circuit that includes a balanced clock tree connected to all the circuit elements. There are restrictions on the clock tree that the designer can modify. From IC compiler to Primetime PX there are two paths. They both use the layout of the chip and the parasitic data. The parasitic data includes the resistance of every cell and the impedance of the following wires. The quick path to Primetime PX uses constraints from IC Compiler to generate static reports on the clock tree. These constraints include definitions for each clock pulse, which can be used as switching data. The other path needs a test bench to create actual switching activity dumps(value change dump: vcd), which is used for more accurate analysis. The reports from the power analysis are evaluated, and the designer will then modify restrictions for the clock tree. This cycle continues until the designer is happy with the resulting clock tree. This whole work flow happens in the time intensive sign off phase right before the chip is sent to tape out. This means that the designer has a limited time window to optimize the clock tree, as a delay in prototyping is costly.

Understanding the prototype When the prototyped chip returns from production, it will be tested and measured. At this time circuit test benches are ready. Then the optional path through simulation seen in figure 2.1 is used. This is a less time critical phase, and the increased accuracy in the power analysis is valuable. When the prototype is ready, small adjustments can still be made to the clock tree but major changes are avoided. It is however still possible to reduce clock tree power consumption though minor tweaks.

2.2.2 Adittional file generation

For the current work flow, the designers use a single total power number to judge the merits of the clock tree. To create a power density graph, additional information is needed. Extracting this is easily implemented in the work flow already in use. Figure 2.2 shows the target work flow planned after the tool has been constructed. The following files need to be generated:

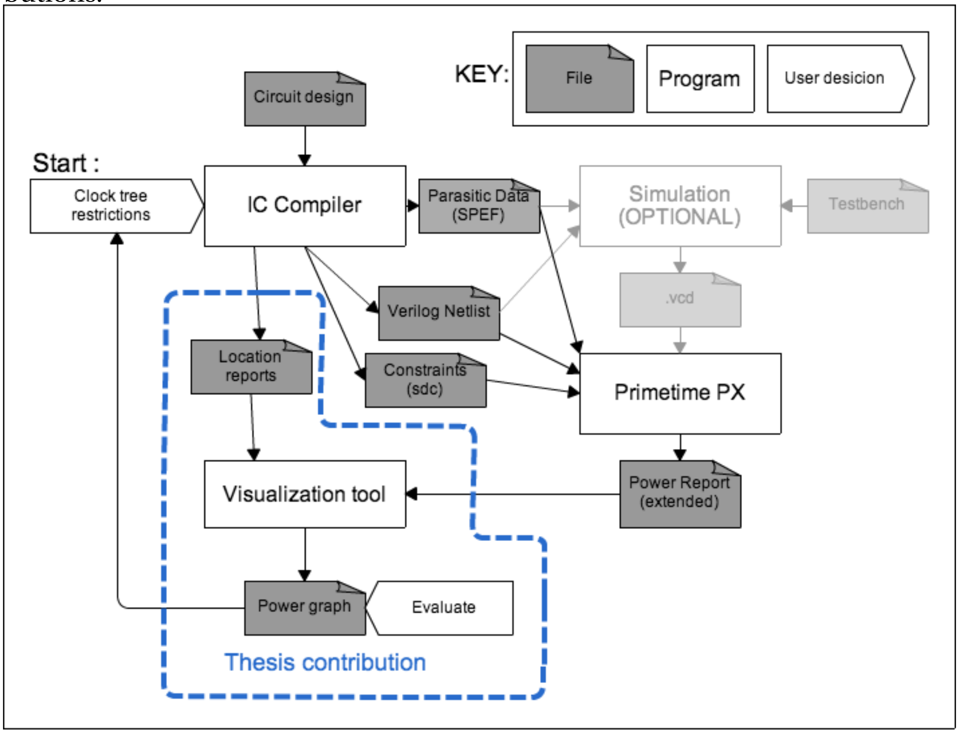
Location files The location of every cell can be printed out from the IC Compiler once the layout is done.

Power per cell The reports giving power per cell can be generated by the Primetime PX power simulation.

2.2.3 Issues with static power analasys

Since most of the clock tree generation happens in a time critical phase power estimates has to be done quickly. The estimation is based of a simple toggle mode. Every clock source is set to toggle constantly and the signal is propagated

Figure 2.2: Desired work flow for clock tree generation including thesis contributions.



through the buffers, gates and registers. Static power analysis is mostly used for estimating leakage power[2], as you don't need activity data as you would for active power. Since a clocks main trait is that it switches constantly, static analysis can be conducted with a pure toggle function on the clock(contained in the sdc file in figure 2.1). For a circuit with a single clock and no possibility to shut down parts of the circuit, this would be quite accurate. With more advanced circuits, there are certain features which will make this inaccurate. Jan Egil Øye mentioned these as main error sources through our meetings:

Clock gating Not all parts of the circuit are used in every mode. When starting up or loading a program, the wireless transmitter for example will not need any power. Some modules might have high down time. With clock gating, the branch of the clock tree connected to the module will not consume any power during downtime. In the static power estimation this will not be visible.

Changing clock frequencies Some modules in the circuit might be able to run on different clock frequencies. The correct frequency will be gated in, and the other one will be blocked. With static power estimation, both frequencies will be active in that area at the same time. This will inflate the power spent in that area.

To simulate clock gating, additional power estimations can be run where a gate will reduce all power spent behind it with a certain percent. By doing several estimations with different percentages, you can get an idea of how much the gates are helping. This does not include the actual down time behind every gate, so it can be misleading.

2.3 Relevant tools used in the design process

2.3.1 Synopsis IC-compiler

According to the Synopsis website[12]:"IC Compiler is a single, convergent, chip-level physical implementation tool that enable designers to implement today's high-performance, complex designs on schedule". The IC-compiler has several tools for optimizing the circuit layout, and tools for generating clock trees. More information can be found on the synopsis web page.

Clock Tree Synthesis, CTS, can be performed in IC-compiler by adding specific clock tree constraints in different scenarios. By adding these constraints, the clock tree implementation can be optimized for skew, insertion-delay and power-dissipation. Some of the more important constraints used by Nordic include:

Transition-time requirement Sets the maximum transition time allowed within each clock tree domain (typically rise time). This means that the transition time on the output of each individual clock tree buffer inserted during CTS will meet the target transition time requirement, typically in the order of 0.5-1 ns.

Insertion-delay Maximum time-delay from the clock source pin through the clock tree and to the target clock sink pin. Typically, this parameter defaults to “0” meaning the CTS-implementation will target minimum possible insertion delay. Sometimes a target insertion delay is necessary to constraint because two or more on-chip clocks must be balanced to each other to obtain better timing in clock-domain-crossing scenarios.

Exclusion points Certain parts of the circuit may only have internal connections and can be excluded from the global clock tree constraints. This means that clock sink pins behind these exclude-points will not be balanced with respect to skew and insertion-delay as part of the CTS. Adding such exclusion points can simplify the clock tree implementation and thus save power dissipation but it must be used with care.

Transition time beyond exclusion points When part of the clock tree has been excluded from the CTS-implementation by introducing exclusion-points, a local requirement can be set to ensure that transition time is still maintained behind the exclusion point although there will not be any balancing with respect to clock skew.

Target skew Allows the designer to slack the requirements of low skew. A less strict constraint on the target skew can simplify the clock tree implementation by reducing the number of buffers required and thus reduce power. The penalty may be that it will be more difficult to meet overall timing-closure but this will be a trade-off individual for each chip.

S ome of these parameters may be set globally for the whole design like global transition-time requirements and zero insertion-delay and skew. However, clock-tree-specific transition-time and exclusion points are examples of important user-defined constraints that gives the designer the opportunity to use her or his detailed knowledge of the circuit to improve the clock tree implementation and thus improve power dissipation.

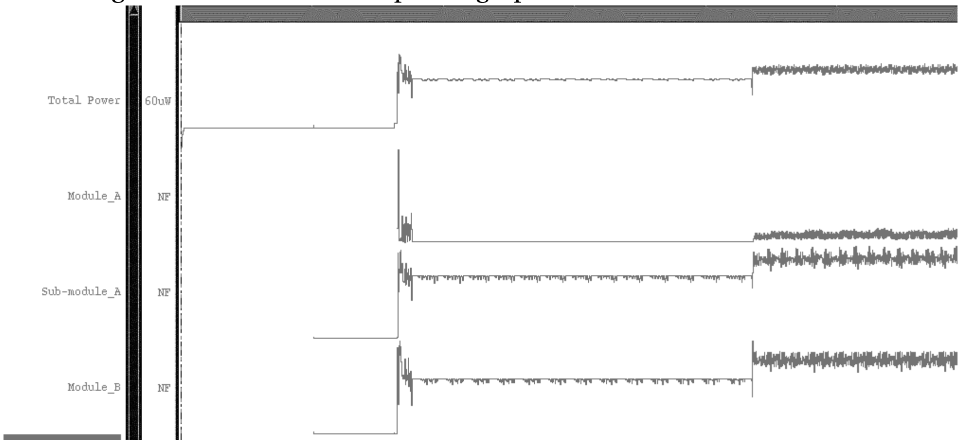
2.3.2 Synopsis Primtetime PX

Synopsis Primetime[13] is a tool made for use during the sign-off phase; the final phase before delivering the plans to production(called tape-out). It aims to help find faults in the circuit before it is too late to fix them. Primetime has a subprogram called Primetime PX which can do power analysis on the circuit. You can get power analysis based of simulation dumps, or choose a static estimation as described in 2.2.3. Primtime PX uses switching activity, parasitic data and the layout to do accurate power analysis.

Static power estimation Static power estimation fetches its activity data from the constraints file from IC compiler. These constraints are much quicker to analyze as you only need a single power number as a result. This method only works for the clock tree as the switching activity for the clock tree is static. This estimation of a never changing clock is deemed good enough for the time sensitive sign off phase, but the problems are mentioned in 2.2.3.

Time based power analysis Time based power analysis is based on simulation. The circuit layout, the parasitic data and a test bench driving all the in-signals is required for the simulaton. The resulting activity dump(vcd file) can be used instead of the constraints (sdc) file used in the static analysis, as seen in figure 2.1. The power analysis based on these files will result in a time based view of how much power is used in the entire circuit. You can also choose to highlight separate modules seen in figure 2.3. These types of graphs where the modules are separated are used later in the design process when the circuit power consumption is measured in the lab. The power measurements only show the total power consumption over time, so the estimations help the designers understand what is using the power in the chip. With the time based estimations a user can choose to generate a series of reports each containing the average power for each cell within consecutive time frames. This will make each frame a sampling point, and you can see what cells consume power in each time frame.

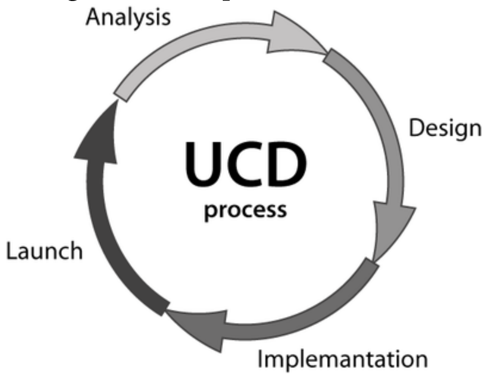
Figure 2.3: Primetime PX power graphs for time based estimation



2.4 User centered design

When developing a tool it is helpful to have a clear design methodology in mind. In this project the initial requirements for the tool were quite few, and there were a lot of design decisions to make. User Centered Design (UCD) is a process which put heavy emphasis on the needs the user of the final product[4]. The user can be involved in every state of the design process from identifying the problem to validating the solution. Reaching out to the user is often part of the problem, but when working alongside the user the process can be split into four repeating phases:

Figure 2.4: Simplistic UCD model



Analyze Involves finding the problem, identifying exactly what needs to be fixed. This can be done through interviews, questioneers or discussion with the user.

Design Finding a solution to the problem. This involves brainstorming, setting up different strategies, and choosing a plan of action. Users can be contacted for ideas in brainstorming or help decide upon the best strategy to solve the problem.

Implementation Implementation phase means carrying out the plan of action. Users are less part of this process, but if the implementation is not straight forward and changes has to be made to the plan of action, they can help modify the strategy.

Launch Testing is important to see if the problem identified is actually fixed. Launch is showing the customer what the next iteration of the product looks like. Users can best identify if the solution is successful. Once one problem has been fixed, other problems might arise.

T his feedback loop continues until the product is ready for release, and will continue for further patch updates in sowftware production.

2.5 Regular expressions

Regular expessions(Regex) is a standard for searching in strings[11]. The standard allows a string made by the user to produce matches when used on strings or texts. By following the rules, you can make advanced searches with just a single input string. There are boundless regular expression resources online, and the technique is widespread in programming after it got known in the linux terminal based program grep. Now many languages support i in their native libraries, like for example python[9].

Chapter 3

Method

The program for this project was developed in close collaboration with my adviser at Nordic Semiconductor and other engineers working in the digital design group. The idea of the program was quite general and the features were not set so they needed to be decided as part of the thesis. This ended in a user centric design process where the engineers had direct influence over how the tool was to display the data, and how it would fit into the design work flow. This chapter describes the path from a loose set of requirements to a finished product. It also encompasses the testing scheme.

3.1 Request from Nordic

The programs are based of the initial requests from Nordic Semiconductor. In order to improve their basis for decision on generated clock trees, they wanted a quick analyze tool, and a graphing tool. In this section the basic requirements and intentional use will be presented.

3.1.1 Quick analyze tool

The quick analyze tool is intended to run on extended reports in the first phases of setting constraints on the clock tree algorithm. It is intended to get a quick overview over what types of elements in the clock tree the power is spent. This can give an easy to use, but rough pointer at what to adjust in the constraints to get a more efficient clock.

Design requirements

- Low run time
- Give an overview of the report
- Allow user set categories and search terms

3.1.2 Graphing power density

The main goal of the thesis is to make a tool able to combine power data with location data to create a power density plot. The goal is to used it for optimizing clock trees and understanding the circuits power consumption in general.

Design requirements

- 3-Dimensional graph displaying power density
- Automatically construct data from two types of reports:
  - Power used per cell
  - (x,y) location for each cell
- Visual representation of circuit to improve readability
- Allow for animated graphs based on several consecutive reports, to graph density over time
- Keep the program as generic as possible to allow for other uses

3.2 Choosing a language

To start programming, I had to choose a programming language. I chose Python as it is a language I am familiar with, it also has many attributed which fits the project well.

Widespread - Python is a popular and established language so it is easy to find someone who can follow up on the code when I am done with the thesis.

Open Source - With no licence fee distributing it around the company is painless. Compared to plotting in Matlab.

Good for prototyping - Python has many libraries that help the programmer get something up and running fast. Therefore it is well regarded for its strength in prototyping. This is a strong argument when using User Centered Design. Python has good tools for all the tasks needed in this project, allowing the whole work flow to be gathered in a single language:

Graphing library - Matplotlib is a long running graphing library for Python[6]. It allows gnuplot style graphing directly from Python.

Easy file handling - Reading and writing to files is trivial in Python.

P ython and other scripting languages are regarded as slower than compiled languages like C. This is the strongest argument against using Python for this thesis. The programming challenge is not very complex, so I decided that the extra run time would be less significant to the project than the ease of use. If run time should be a problem, alternative interpreters like pypy[7] can greatly improve run time.

3.3 Input data and quick analysis tool

In this section I will first present the form factor of the input data tool is supposed to graph. I will then move through the solution created for the quick analysis tool.

3.3.1 Input data

The input data consists of two files. The report from the power simulation and a location report, telling where on the circuit each cell is situated. The size of these files can vary a lot. Circuits can have several hundred thousand cells, but in this section we will only show a few example lines, as the length of the file does not change the complexity of reading it.

Power reports

The power reports are the results of the power simulation. The quality of the simulation can vary greatly. The report under is a dummy report showing the structure. The names of each cell has been simplified. It is important to note that each cell name string is unique. The hierarchy of names will show what part of the circuit the cell is, and the cell name at the end will show what type of cell it is. What identifies a cell might be an abbreviation for the technology name or simply the string reg as shown in the example power report in listing 3.1.

Listing 3.1: Example power report from Primetime

```
****************************************
Report : Averaged Power
        -cell_power
        -leaf
        -sort_by cell_internal_power
        -power_greater_than    0
        -nosplit
Design : name
Version: G-2012.06-SP3
Date   : Sat Feb 8 06:51:52 2014
****************************************



  Attributes
  ----------
      a  -  Annotated internal & leakage power
      b  -  Black-box (unresolved) cell
      c  -  Clock pin internal power only
      d  -  Does not include clock pin internal power
      h  -  Hierarchical cell

                        Internal Switching Leakage   Total
Cell                    Power    Power     Power     Power  (    \%)  Attrs
--------------------------------------------------------------------------------
hierarchy/structure/cell1reg 3.905e-06 8.377e-06 1.437e-10 1.228e-05 ( 0.18%)
hierarchy/structure/cell2gate 3.835e-06 6.768e-06 1.435e-10 1.060e-05 ( 0.15%)
hierarchy/structure/cell3buffer 3.722e-06 2.144e-06 1.437e-10 5.866e-06 ( 0.08%)
....(80 000+ lines skipped)
hierarchy/structure/cell3reg 1.032e-09 6.677e-08 3.145e-11 6.577e-08 ( 0.00%)h
--------------------------------------------------------------------------------
Totals (85793 cells)  7.537e-04 1.114e-03 5.134e-03 7.002e-03 (100.0%)
```

```
1
```

The lines we are interested in are listed between the to dashed lines. Each line is simply a string followed by different numbers. Each number is separated by a space and represents internal power, switching power, leakage power, total power and percentage power. Exceptions of this rule happen in the percentage field, where a space can come within the parenthesis. Sometimes a letter will follow at the end of the numbers as in the last line of the example report. This letter represents a certain attribute shown in the attribute list. As a more sporadic exception there would sometimes be a line break behind the cell name instead of a space, effectively splitting the interesting data over two lines.

Location file

The location file is generated by extracting only the location data from the IC-compiler. The example file in listing 3.2 shows the standard for the location files that we have to parse.

Listing 3.2: Example input for cell location data

```
hierarchy/structure/cell1register;register_type4;3401.000 2631.000
hierarchy/structure/cell2gate;gate_type1;3380.000 2823.000
hierarchy/structure/cell3buffer;buffer_type3;3369.000 2762.000
....(80 000+ lines skipped)
hierarchy/structure/cell3register;register_type4;824.800 1749.52
```

The file is a comma-separated values(CSV)[15] file with semicolons as separators. This type of file is possible to open directly in excel. The fields contain unique cell name, reference name and a (x,y) tuple split by a space. The (x,y) tuple gives the location of the cell.

Parsing the files

Parsing files for information is a trivial problem with Python, especially when they are as simple as these. You can take in a file line by line and handle them with functions in the string library. I will quickly explain the way I did it for this project, but this will be different for any other report. The important thing is to know what data you need to come out of the parser.

For the power report simply ignore any lines outside of the two dashed lines. To avoid the problem of the extra space in the "% of total power" field, you can remove all instances of "(" and "(". To separate the last letter that sometimes appear at the end of the line replace "%)" with space. Then you can use tools as string.split() to separate words so you have the names and numbers in separate strings as a list. You can then pick the fields in the list you are interested and cast the number strings to float. Python handles casting directly from scientific notation strings to float. From the power file the parser return a list where each item contains the data for each cell. Each data element is again a list with the name as the first element, and the power from the different power categories following as floats. It is important to add a way to ignore the letter that may show up at the end of the line. This can be done by always only returning the first 6 elements of the list for each line.

From the location file we are not interested in the reference name, but the cell name and the x and y position of the cell. Python has good a built in library for parsing .csv files. However since the x and y positions are in the same field, it has to be split with space as a separator before being cast to floats.

3.3.2 Power report analysis

The power reports contain the power for each cell, with tens of thousands to hundreds of thousand lines the files are practically too big to use without any tools. The files do contain total power at the bottom, but this summary could be more detailed. A more useful summary would not only show total power, but also differentiate between what types of cells consumed the power. This is achieved by allowing the user to define a file with several categories. Each category can have several search words that will add the power of a cell name matching the search word to that category. I found out that the engineers were all quite used to using regular expressions to search files. A simple configuration file for this tool would look like this:

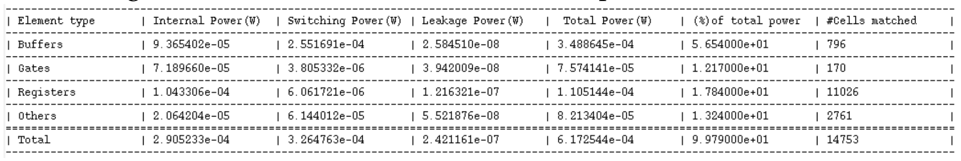
Listing 3.3: Example configuration file for quick analyze tool

```
----Buffers
^.*buffer.*$
----Gates
```

```
^.*gate.*$
^specificGateName1$
^specificGateName2$
----Registers
^.*reg.*$
```

The quadruple dash signifies a new category, while the the regular expressions in between gives the search phrases that should mach the cells that fit in that category. A user can also add a list of specific names for cells to the category since some cells might be hard to match within their right category with a more general regular expression. The program will output reports in the following format:

Figure 3.1: Screen shot from terminal output from summarizer



```
---------------------------------------------------------------------------------------------------------------------------------------------
| Element type        | Internal Power(W)  | Switching Power(W) | Leakage Power(W)   |  Total Power(W)    | (%)of total power  | #Cells matched     |
---------------------------------------------------------------------------------------------------------------------------------------------
| Buffers             | 9.365402e-05       | 2.551691e-04       | 2.584510e-08       | 3.488645e-04       | 5.654000e+01       | 796                |
---------------------------------------------------------------------------------------------------------------------------------------------
| Gates               | 7.189660e-05       | 3.805332e-06       | 3.942009e-08       | 7.574141e-05       | 1.217000e+01       | 170                |
---------------------------------------------------------------------------------------------------------------------------------------------
| Registers           | 1.043306e-04       | 6.061721e-06       | 1.216321e-07       | 1.105144e-04       | 1.784000e+01       | 11026              |
---------------------------------------------------------------------------------------------------------------------------------------------
| Others              | 2.064204e-05       | 6.144012e-05       | 5.521876e-08       | 8.213404e-05       | 1.324000e+01       | 2761               |
=============================================================================================================================================
| Total               | 2.905233e-04       | 3.264763e-04       | 2.421161e-07       | 6.172544e-04       | 9.979000e+01       | 14753              |
---------------------------------------------------------------------------------------------------------------------------------------------
```

Any cells that cannot be matched by the different categories are put in the "Others" category. In this particular example, only a simple category file is present, and all exceptions from the simple naming scheme end up in others. The program runs within a second, so refining the categories file with additional categories is possible. This tool tool turned out to be uninteresting once the graphing tool was finished. The graphing tool was suspected to be slower and not usable for fast summaries. However, as you will see in this chapter, the aspect of user defined categories were included in the graphing tool in a similar manner, and the run times were unproblematic.

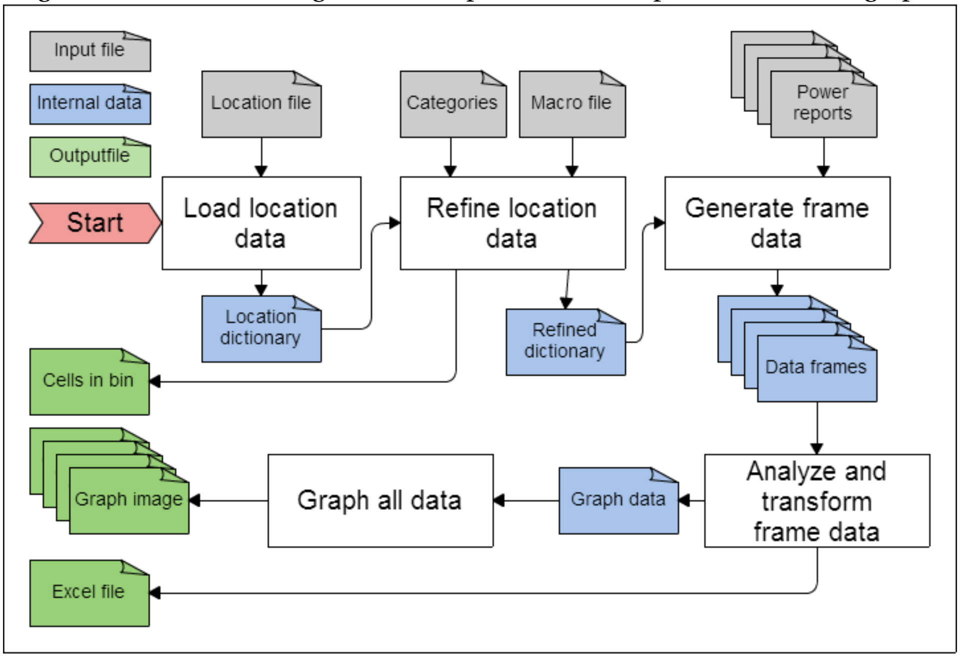
3.4 Graphing tool

The graphing tool is a larger project than the analysis tool. The amount of data involved is significantly higher and the result more refined. It is supposed to be able to take in several consecutive reports, and graph the power consumption over the surface of the chip in a animated graph. This chapter goes through the features of the program in the order they were designed. This represents the user centered design methodology followed in this project. Each new feature was added after feedback with the users after they saw the newest prototype at work.

3.4.1 Program work flow

The work flow of the program is not similar to the order tasks are presented in this section. To get a grip of where in the work flow each feature is added, I will first quickly present the final work flow of the program as seen in figure 3.2. The introduction will be brief, as every feature is explained in more detail in their own section.

Figure 3.2: Work flow diagram for the process from reports to finalized graphs



Load location data Reads location data from a file and stores it in a dictionary sorter by cell name.

Refine location data Finds the right bin for every cell, and figures out what categories each cell belong to. Prints out the file containing the cell names held in each bin.

Generate frame data Reads the power reports and creates a data set for every frame by adding the power number for each cell to the correct bin.

Analyze and transform frame data Finds maximum values in each frame and transforms the data to logarithmic plotting.

Draws all graphs Reads all the graph data and sequentially draws each frame.

3.4.2 Constructing data for surface plots

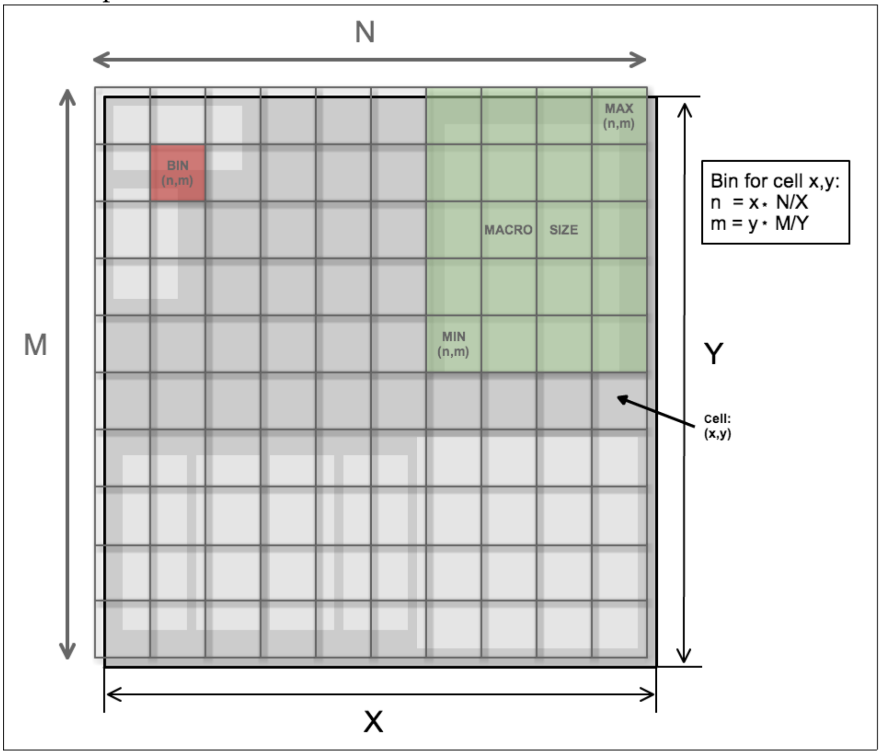
This is the core function of the program. The data needed to make a single image for the animation needs to be gathered and constructed from two different data sets. The plot data needs to be a N*M matrix containing the amount of power used in total within the area of the chip corresponding to each n,m bin. For each time slice there will be a unique power report, resulting in an unique data plot. N and M are numbers chosen by the user.

The way the tool creates this power density data changed a few times during the course of the project. The way I describe the process here is a result of optimization over several iterations. From the programs first version to its last, the time the program used to construct the data was reduced by 75%. However I judged that software optimization and effective algorithms were not the main focus of this thesis, so the progress in run times are not highlighted as results. There are probably still room for improvement, but the work flow is thought out to avoid unnecessary repetition.

Loading location data

The locations of each cell stays the same for every report on the same chip. A design decision was made to make the program only able to remember one chip at the time, meaning you would have to run it two times for two separate chip designs. Since the cell locations stay the same the entire program, the program first loads these. The run time added to the program from loading the cell locations is not dependant on number of reports. In the power reports the cell names are not sorted. To quickly find the correct location for each cell, I chose to load the cell locations directly to a dictionary before the program loads the power reports. This way the program can very quickly look up the correct location of any cell name. The dictionary data structure has a key and a value, the cell name is used as a key, and the x,y location of each cell is stored in the value. A requirement for dictionaries are unique keys. This is fulfilled as all cell names are unique.

Figure 3.3: How the grid represents the chip area. Capital N,M,X and Y refers to the maximum number for each dimension. n, m, x and y represents specific values within the limits. Macros are defined by the minimum and maximum (n,m) tuples.



Refining location data

The value in the dictionary is an instance of a location class. The location class allows the user to access the x,y coordinates, calculate n,m coordinates for each cell if given X,Y and M,N. In order to find X,Y the location dictionary has to be searched for the highest x and y value. N and M are specified by the user. When the class has N,M;X,Y and x,y it can calculate n, and m as seen in figure 3.3. This is done before the report data is loaded to minimize the time used to place each cell power in the correct bin, as seen in the work flow in figure 3.2.

Loading power data

The reports are parsed one by one, loading the name and power usage of the cell in a unsorted list. The list is then traversed, looking up every cell name in the dictionary and adding the power to the bin the dictionary provides for that cell. The NxM grid is stored in a class instance. One instance per frame, and all the instances are contained in a list. In order to have static scales in the animation, the limits for all the axes has to be the same for each frame. The data for each frame is therefore traversed to find the max power value for any bin in the entire animation.

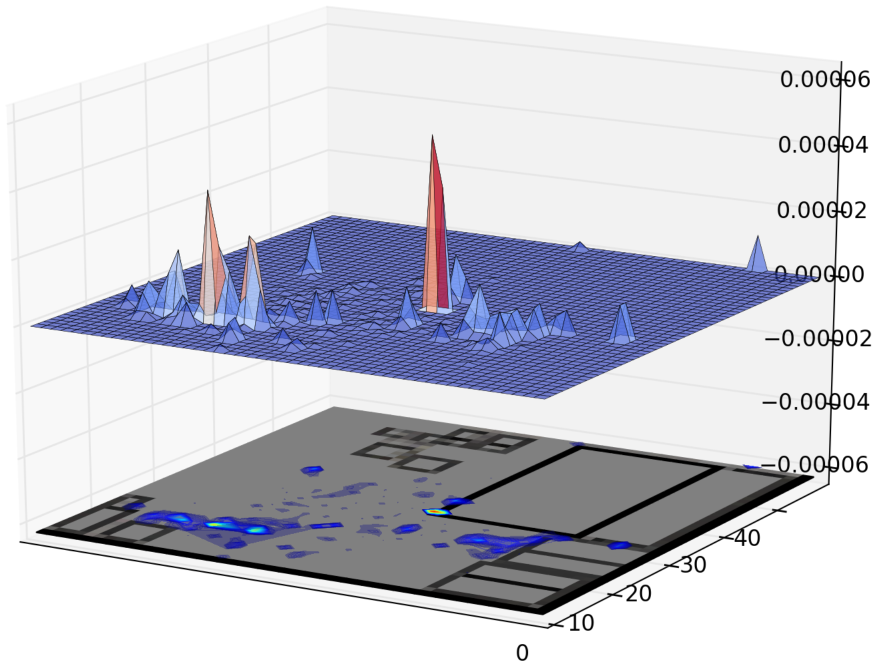
3.4.3 Circuit layout visualization

A design feature requested from Nordic was that the graphs was to have a drawing of the circuit visible under the plot, with a heat map showing exactly where on the circuit the power is used. A .pdf file with a visual representation of the circuit was given as a possible help. The two solutions described in this sections were early iterations. The third and final solution is described in section 3.4.4, as it is connected with extra input data.

Visualization by manually drawing

For the first iteration of this feature, a simple .png in black and white was made manually based of the contours of the circuit. The graphing tool then loaded the value of each pixel, and the picture was graphed as a heat map based of the png as seen in figure figure 3.4 . The .png file also had to have the same MxN

Figure 3.4: Manually drawn layout visualization

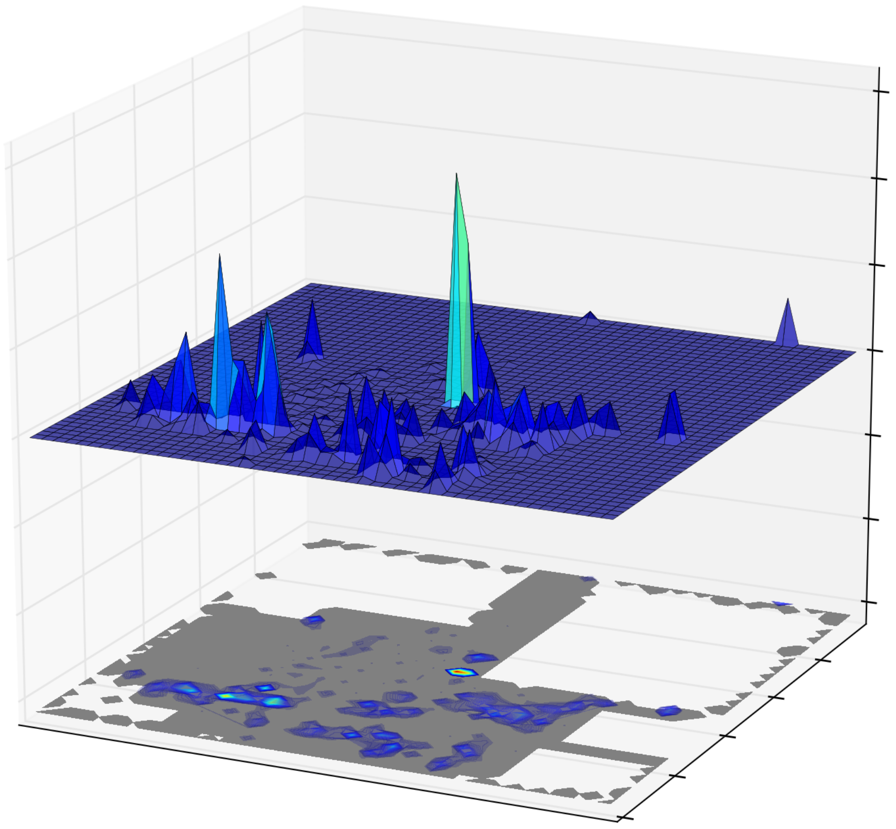


resolution as the graph resulting in either fixed graph resolution, or the need for many .png files. This method was slow, both because of the amount of work done manually, and the extra work done by the program. A second problem was being sure that the data displayed matched the coordinates and axes of the picture. This required knowledge of both the circuit, the plot and the graphing tool to get right.

Visualization by cell density

With automation in mind, a new scheme was created. One of the data files that is used to create the graph data, is a location file of all cells, giving the coordinates of their power connection. By making a new data set that kept the value of how many cells are connected to each bin, you would get a data set showing density. The thought was that this might give a recognizable pattern of the circuit as seen in the middle of figure 3.5. It turned out to be better to simply

Figure 3.5: Automaticly generated layout visualization based on cell density



differentiate between two values, either there are one or more cells in the bin, or there are none. This method shows the recognizable outlines of macros such as flash, radio and ram. This solved the main concerns with the manual layout since the data is based of the same data that generates the graph the orientation is always correct. It is also based of the data you already have, so there is no extra work for the user. A third more visually pleasing version is described later in this chapter, but requires extra input. If that input is not present, this method is used instead.

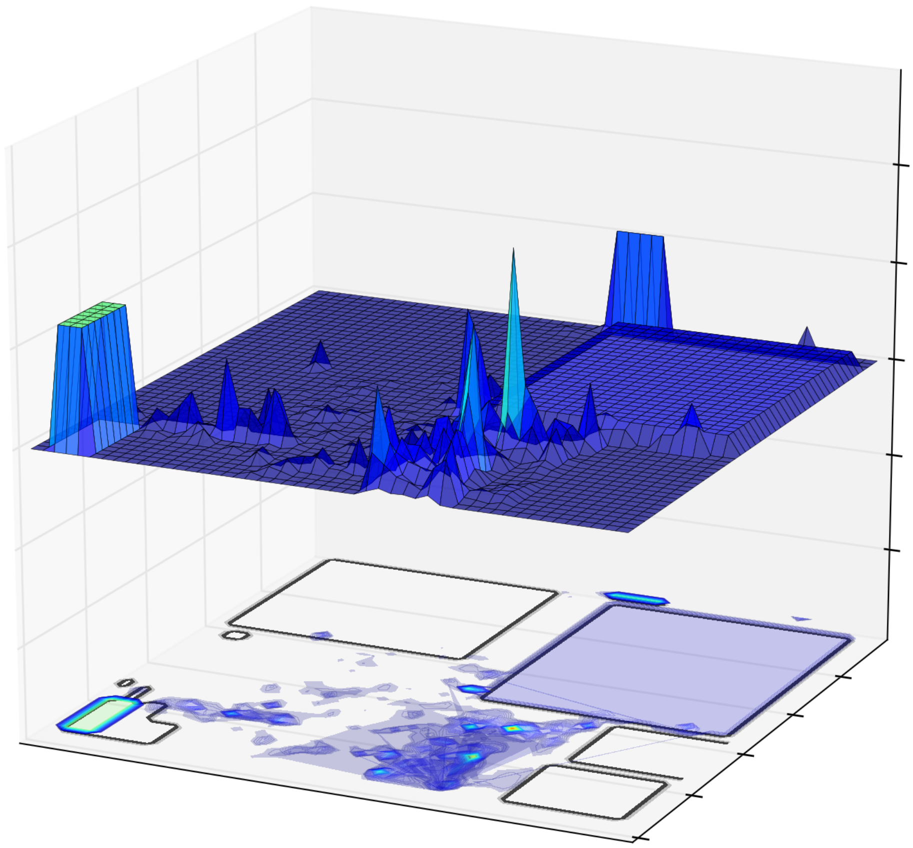
3.4.4 Problems when visualizing circuit macros

When the initial tests with the clock tree reports were successful I reported that the program was written as general as possible to enable plotting different data. Jan Egil Øye wished to test the visualization tool on a power report from a new simulation, and included reports including the entire cell library, not only the cells belonging to the clock tree. When attempting to run the tool on the full report containing all cells, there would be huge spikes. I searched the reports for high numbers and found macros like flash, radio transmitter and memory that are designed separately and inserted as black boxes in the design. In some simulation modes a macro can stand for up to 50% of total power. These cells would create peaks dwarfing all data, meaning the surface would be flat except for a couple of spikes showing macro activity. In the clock tree there is no macros, but the concept of visualizing power data from the full circuit was interesting. At first I created an ignore file, allowing to simply remove the spikes that would make the surface plot unreadable. Figure 3.5 is made by ignoring data from the large macros. However Jan Egil and I agreed I should spend the extra time needed to make sure the program could show the power used by macros in a informative manner.

Exceptions for macro data

In order to do this, more information about the macro cells is needed. I suggested to spread the power used in macros over several bins representing their actual size, as this would show an average power density in that area. The power is of course not dissipated uniformly in the entire macro, but since it is black boxed, it is a fair approximation. To do this correctly, additional information was needed for the macros. Printing a file similar to the location file but with cell name and bounding box instead of the location of the power connection is a small job. Deciding what macros are important to get on this list has to be done by the designer. However macros are decided early in a design, and does not change in a final phase where graphing is supposed to happen, so it is a one time job for each circuit. A new type of file was added with the format as seen in the example listing 3.4.

Figure 3.6: The surface plot area after the addition of bounding box for macros. Allows for showing power use in macro blocks and improves layout visualization



Listing 3.4: Example input file for macro bounding box data

```
memory1;memory_unit;{2100.000 1900.000} {3000.000 3500.000}
storage1;storage_unit;{100.000 120.000} {500.000 900.000}
radio1;radio_unit;{600.000 450.000} {1500.000 900.000}
```

The bounding box file contains the name of the macro matching the cell name in both the location and power file. Like the location file, it also contains a reference tag which we do not use. The last field contains min_x, min_y, max_x and max_y for the macros. This file is parsed similarly like the location file. The bounding box file is not mandatory, but if it is included the location data for the cells mentioned in the file is updated. This is done through a function in the location class which sets a flag for the cell to mark it as a bbox, and calculates min_n, min_m, max_n, max_m and area. Similar formulas used to find m and n for normal locations is used. The area is given in number of bins the macro cover. The other change that needs to be done is to add a check when the power data is added to each frame. If the cell power added has the bbox flag high, it adds power divided by area to every bin it covers. The resulting surface will have box-like representing the macros as seen in figure 3.6. This representation actually helps separate macros from other power use, as they have quite different characteristics. The feedback on this was positive from the engineers, as they could easily identify activity such as reading from flash and the memory being used.

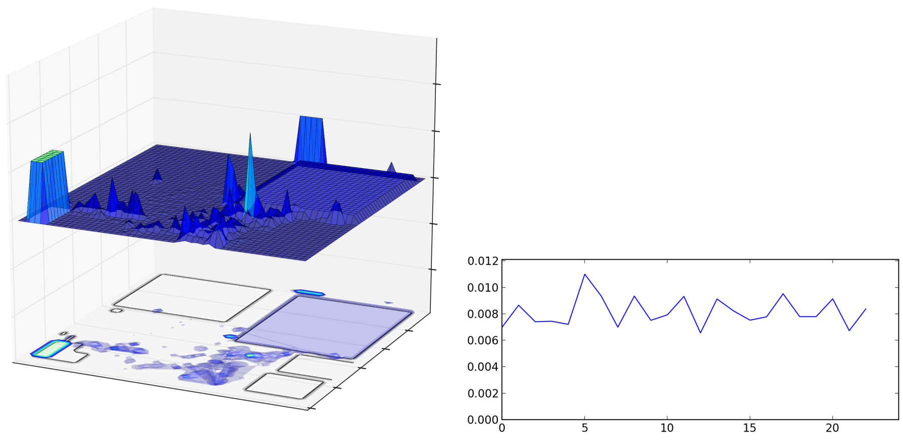
Using bbox to visualize circuit

In addition this macro data can be used to create a smoother visual representation of the circuit. By plotting a projection of a surface plot from 0.1 to 0.9 where 0 is the outside of macros, and 1 is the inside of macros, clear outlines can be made in the graphs as seen in figure 3.6. If a macro file is included, this technique is used instead of the density visualization described in section 3.4.3

3.4.5 Showing the total power used in each frame

The surface plot can be animated to show change over time, however, there is no direct way of comparing power at two different time slices against each other. In order to give the user a concept of the power used in the circuit over time, the total of each frame is gathered when the power reports are loaded. The data is plotted in a separate graph with frame number on the x axis, and power value on the y axis. The graph was added based on feedback that they wanted something similar to the graphs the generated by time based simulation in Primetime PX as seen in figure 2.3. By having this graph visible, they could directly start looking closer at areas they had found interesting in those other tools. It also helps the designer to spot interesting frames like the highest total power use, and inactive periods and actually compare the total power spent in each frame.

Figure 3.7: The view for the user after adding a total power per frame graph. The graph on the right graphs total power on the y axis to report number on the x axis



Total power spent by categories

Once the total power graph was up, it was apparent that the visualization was useful. It gave a clear signal what phases of the simulation used the most power, and you could clearly distinguish the simulation shifting running modes. This inspired the idea of being able to set up a list of categories that would be plotted together with the total power plot. The idea was that you for example could set up a category for all cells that are part of the processor, and observe in what modes those are most active. The need is similar to the categorization in the power report analysis tool mentioned in section 3.3.2.

Implementation

The category file is loaded after the cell location dictionary is made. For each category every cell id in the location dictionary is traversed. If a cell id matches a regular expression that belongs to a category that category name is added to the information on the cell held by the dictionary. This results in every cell also has a list containing all categories it is a part of when the power file is loaded, the cell data also tells the program what categories to add the power use to. This reduces the time the regular expressions has to be searched for to one, instead of once per report.

Plotting category data

After plotting many categories in the same graph, it was obvious that the scales of the different data could vary greatly. As a default the graph was scaled after the total graph, meaning some minor power consumers would not be visible. We decided not to logarithmically scale the y axis, as you would loose resolution on the temporal difference in a single plot. Instead the graph was split in two, the top one scaled after the total power use, as this was easy to recognize from other simulations, the other contained all user defined graphs, scaled by one of those graphs set by the user. This improved the readability of the graph, and allowed the user to follow certain lines of interest closely. The category file works similar to the one in section 3.3.2, but with more tags to create new categories, to enable the new features:

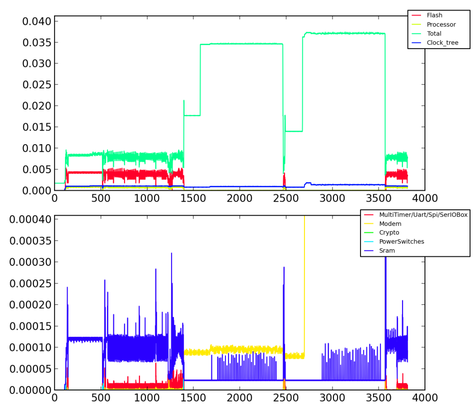
- –*[category name] marks the category that should be used to scale on the bottom plot
- –![category name] displays the category in the top graph together with the total power
- –[category name] displays the category in the bottom graph.
- –(mute)[category name] to hide the category from plotting, will still be printed in excel file as described in section 3.4.8.

As figure 3.8 shows, the different categories can give much more detailed understanding of the power used.

3.4.6 Identifying points of interest

As a design decision we agreed that the graph should give an intuitive view of the power use, not a detailed numerical view. For the details on specific cells and power, the engineers can search the report files. The graph should work as a way to find out where to look in those report files. When testing the program

Figure 3.8: Plots showing total power for user defined categories over time. Y axis: Power, X axis: time.



on some scenarios, different peaks appeared. Some static, and some sporadic. However it was hard to tell exactly what these spikes indicated, and what cells actually used the power. There was a need for a link between the graph and the report file, to quickly locate the right cells. To make this possible, first it had to be clear what exact n,m coordinates the peaks had. To display this, a list of peak positions n,m and what power they used was created for each frame. This was done by searching for the 10 highest values, and their indices in the N*M grid. This is done as part of the analyzing and transforming frame data as seen in figure 3.2

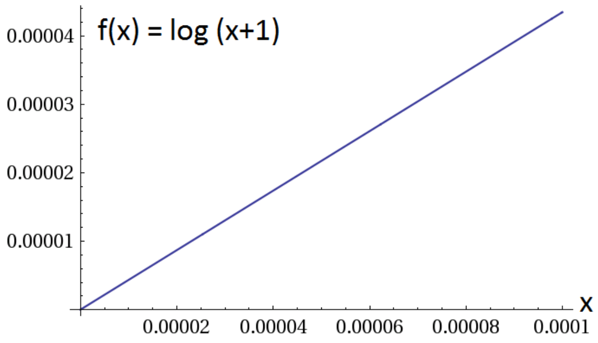
Look up bin

When the programmer has the bin number he or she could calculate the approximate x,y position they were looking for, but to do this, you needed to know the limits of x,y,n and m. This was solved by printing out a file containing a grid, each bin containing the cell names that are contained in that bin. Thew file is read by a small program that prints out the cell names of whatever coordinates you input. The print results can easily be copied and used as regular expressions in a category file. This will result in a bin being plotted on its own. This can be useful for identifying how often a spike appears. You can also see what types of cells are contributing to that spike, which is needed if you want to know if you can change the spike. By looking up the different cell names in the report, you can find the exact power drained in each cell for that time instance. This feature gives the designers a good bridge between the intuitive graphs and the detailed reports. The bridge allows them to use the impressions and ideas they get from the graphs to search the reports for key data.

3.4.7 Logarithmic plotting

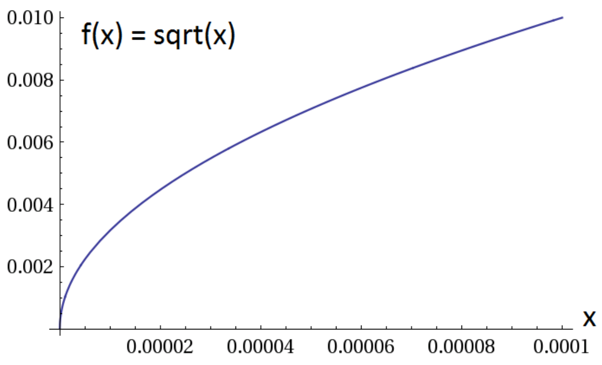
To identify details and small power usages even in graphs where there are also large spikes, logarithmic plotting is useful. It was a feature desired by the designers to get more visual details in certain plots. There is no automatic logarithmic plotting function for the surface graph, so this transformation had to be done on the data before it was graphed. Since the power numbers are in the magnitude from 1e-10 to 1e-4 using log(x) does not work as numbers between 0 and 1 become huge negative numbers. Reports can also sometimes contain negative numbers, which would result in complex numbers when put in a logarithmic function. The first attempt at fixing this included adding 1 to every number and then taking the logarithm giving $f(x) = log(x+1)$. However when x varies from 1e-4 go 1e-10, the transformation is nearly linear as seen in figure 3.9. Instead

Figure 3.9: Properties $f(x) = log(x)$ at low x values



the program uses square root of x. This works for all positive numbers, so negative numbers are transformed to 0 with an error message, as Jan Egil Øye meant they were not interesting in the plot. The square root graph in figure 3.10 has the same properties as a logarithmic plot would have for values over 1. Therefore I chose $f(x) = sqrt(x)$ to transform the surface plot data.

Figure 3.10: Properties of $f(x) = \sqrt{x}$ at low x values



3.4.8 Excel print

As a design decision, we wanted to keep the plot view clear of too many numbers. These should be available if you were looking for them. The category graphs in figure 3.8 contain a lot of information, but you can only have the plots scaled after total and one other graph. When looking at plots from a data set, the user got an urge to see in more detail what happened in these graphs. An extra function was made to simply print out the data points in a .csv format that can be opened directly in excel. This way the designer can open the excel file afterwords and plot all, or some of the lines in the categories graphs. It is also a good way to check up exact numbers from the categories, and with this addition, the quick analyses tool (mentioned in section 3.3.2) is completely included as a feature the graphing tool. Excel is a tool the designers were used to using when handling data. This feature allows the user to use the data for other purposes, and look more closely on certain data points.

3.5 View area

The visualization in this project is done with the Python library Matplotlib[6]. This task is possible with other libraries, so in this section we focus on how the data is presented rather than the graphing tool.

3.5.1 The surface plot

The central plot of the visualization is the 3d surface plot showing where the power is spent. The graph contains 3 plots, based of two data sets. The data for a surface or contour plot has to be organized in a two dimensional array where each bin n,m contains a value. In the power data, this value represents the power used within the bin, while for the layout the value is 1 if there is a macro defined within the bin, otherwise 0. The plot area contains three different plots, as seen in figure 3.11.

Surface plot of power used The surface plot gives clear visualization of spikes, which is important to look for. It is plotted with a slight opacity so that a spike does not block all data behind it.

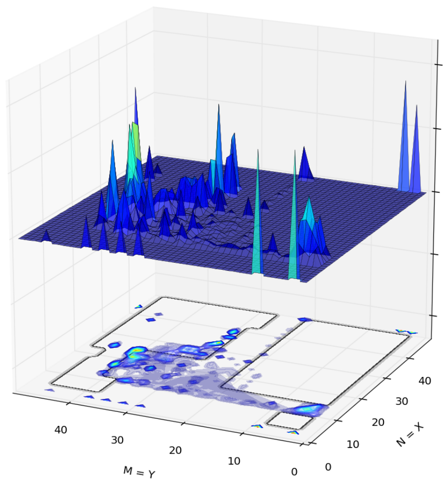
Contour of the power used The contour plot makes a heat map over the layout. This heat map represents where the power is spent and ties that information to the circuit layout. I chose not to plot any contour colors for bins that were exactly 0 as this improved the contrast between some power used and no power used and increased the visibility of the layout.

Contour of the circuit layout The contour data is graphed by excluding 0 and 1, and only graphing 0.01 to 0.99. This means you only get lines between the bins containing and bins not containing any macros.

3.5.2 Categories plot

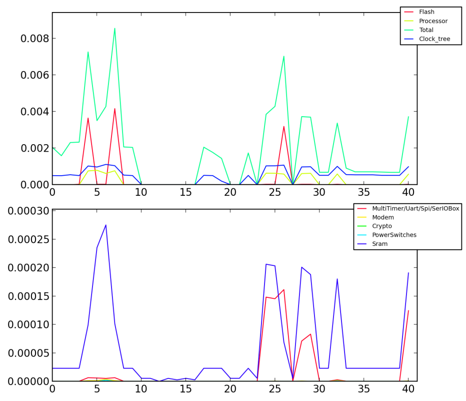
In figure 3.12 you can see the plot showing the category data. The graphs shows the total power per slide, giving an idea of the temporal change in power if the graphs are sequential time slices as in firuge 4.9. If the data is based of scenarios like in figure 3.12 it shows an comparison between different scenarios. The plot on the top is total power used against time, and can include extra graphs set by the user for total power spent by a category. Time or slide number is on the x axis, and power used in watts are on the y axis. This main graph is

Figure 3.11: Three dimensional surface plot with circuit layout and heat map underneath



also accompanied by a second graph that can be filled categories that need a different scale than the total power graph. This allows the user to mark certain data or points as interesting and make them clear on the slides. The x and y axis are set to the same limits for all images in a series so only the line changes.

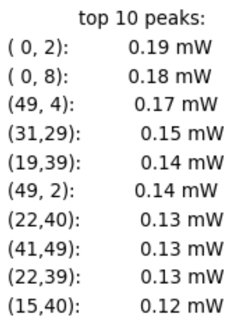
Figure 3.12: Plots showing total power for user defined categories in each frame. Y axis: Power, X axix: frame number. Each frame is for a differen scenario.



3.5.3 Top 10 peaks

In order to identify specific points of interest based on the data in the surface plot, a list of the 10 highest peaks is shown as seen in figure 3.13. This also displays the power used in that point, and gives a good way to read exact numbers out of the graphs and get an idea of the scale.

Figure 3.13: Example of visual style of the top 10 list.



top 10 peaks:

(0, 2):	0.19 mW
(0, 8):	0.18 mW
(49, 4):	0.17 mW
(31,29):	0.15 mW
(19,39):	0.14 mW
(49, 2):	0.14 mW
(22,40):	0.13 mW
(41,49):	0.13 mW
(22,39):	0.13 mW
(15,40):	0.12 mW

3.5.4 Animation

Matplotlib has a function to create animated graphs. However with the old version of linux running on the servers, it was not compatible. Instead each report were made into a single image file, sequentially numbered. This way they could be animated through ffmpeg[5] by a one-liner in the linux command window. Ffmpeg can generate both gif and movies from a series of numerated images. This proved to be a good solution, as the designers then could animate if wanted, but otherwise scroll through the images in a photo viewer in windows. Generating one image at a time is not as time effective as generating an animation, as the animation optimizes by for example only rendering the changes in each frame. It also meant extra care when choosing what to plot to make sure the limits on the graphs stayed the same during the whole animation.

3.6 Testing the visualizer

Prototypes of the program was tested throughout the design process as is usual in user centered design. These tests had two types of results:

Results for the design These tests of the program allow us to find faults that need to be improved and find out what user cases we have not yet enabled. These types of results are discussed earlier in this chapter, as part of the design process

Results interesting for the engineers These results are presented in chapter 4 and discussed further in the discussion. Here are the types of data we decided to try the system on:

Sequential power estimations The reports are from power estimations based of simulation. They are time intensive to generate(4000 reports would take around a day), but give a detailed picture of the power. We wanted to test this to see how informative these graphs would be compared to more static graphs. The largest test done was 4000 consecutive images of a chip that already had simulation files ready for estimation. The results of these are presented in section 4.3.

Categories Static power estimations cannot give time consecutive reports. A single estimation run can however still give several reports. The different reports each represent a scenario meant to show different sides of the power consumption. The only change made to the program to handle this was to have a different limit for each power surface graph, instead of one for the entire run. This is because animating the graphs from scenarios does not make any sense. The resulting graphs that show this are presented in section 4.4

3.6.1 Real life application test

The goal of this thesis is to use visualizing to improve the understanding of clock trees in a design phase. The tool turned out to be done in time for use during the clock design phase of a Nordic Semiconductor chip on its way to tape out. A line was added in the script generating and simulating different clock trees to print out reports for all trees, allowing the program to be tested on a real user case by a real user. The test was handled by Jan Egil Øye, who was working on the clock tree. He used the graphs to look at the clock tree power using the static estimation report technique Nordic normally use, but with all cells printed instead of a single estimate number. He was then able to use the graphs as additional information about the clock tree. This was the main test for the program, as it is the exact user case it was meant to do.

Chapter 4

Results

The graphing tool was constructed and fulfills all specifications set by Nordic Semiconductor. In this chapter some chosen plots will be presented. The graphs displayed in the results have had information removed on request of Nordic as they are generated from competition sensitive data. This is done so the thesis can be open for the public. It is also important to note that it was a design choice to not have units on all graphs. The graphs are meant to give intuitive understanding, not exact information, as that can be found in the reports. This thesis results are the graphs themselves and not the data in them.

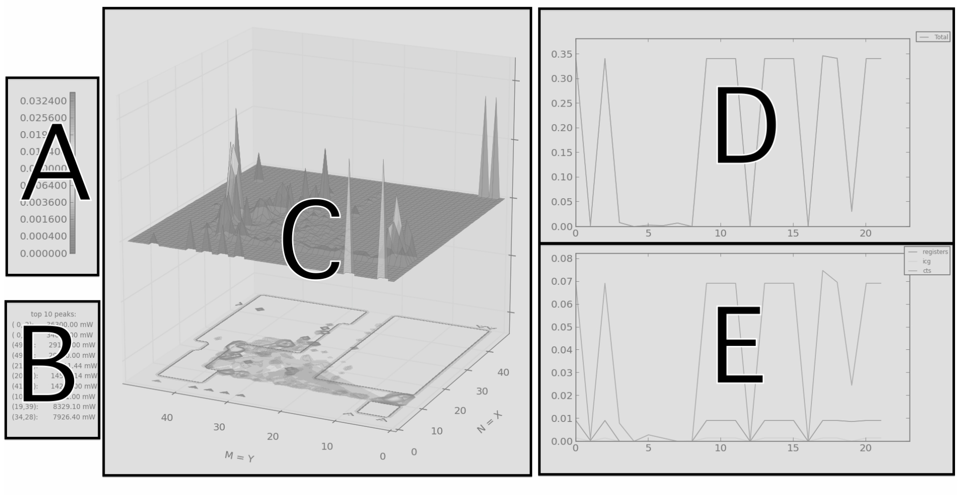
4.1 Key to the output graphs

All graphs in this chapter have the same view area. If you choose to graph a folder of reports, there will be one resulting image for each report. Each image contains several graphs. Figure 4.1 indexes the different areas. They represent the following:

A: Color reference bar shows the approximate value the colors used in color coding of area C.

B: Top 10 peaks Displays the top 10 peaks in the current plot in the following format: (n,m) 0.0 mW, where n and m is the position of the bin containing the peak, and the power number represents the amount of power spent in that bin.

Figure 4.1: The areas of the resulting image file, explanation in section 4.1



All graphs except figure 4.2 to 4.4 were generated by an old version of this program which printed the power numbers several magnitudes too large. This is fixed in the tool, but the power files made to create these graphs are deleted, so recreating them is not possible.

C: Surface plot shows power density over the chip area. Underneath the surface plot is a plot representing the layout of the chip with the contours of the surface plot to show power density relative to the layout.

D: Total power and power per category is graphed with power on the Y axis and image number on the X axis. If the reports are from consecutive time intervals, the graph will represent total power consumed by chip over time. If the reports are from different scenarios, this plot simply shows total power consumed in each frame. The user sets the categories, with the names and what cells to go in them. The names of the categories are listed in the legend in the top right corner.

E: Additional categories can be put in this graph to have the scale of a specific category instead of the total power consumed scale. Otherwise it works exactly the same as D

4.2 Test in design process

Figure 4.2, 4.3 and 4.4 are quite similar. They are all results of testing of the graphing tool in the actual design phase as explained in section 3.6.1. An extra line to execute the tool was added to the scripts that are already run to do power simulations. So there is no extra work in graphing these reports, they simply appear in a folder together with the original reports. This section contains images received back from Jan Egil Øye after the clock tree generation was finalized for tape out.

Total power reduction During the first to last iteration the clock tree power was reduced from 389 mW to 343 mW. A power cut of 11.8%.

Run time A run with 23 reports took 72 seconds to gather the data and draw 23 graphs. On average these reports contained ca 39 000 cells. The time used to gather the data was 25 seconds (1.1 second per report). The graphing took 46 seconds(2 seconds per graph).

Figure 4.2 This is one of the graphs produced from an early iteration of the clock tree. There were 23 graphs produced for each iteration. One graph per scenario. On the total power graph, you can see that several of the scenarios produce exactly the same total power. The surface plots with the same total power were all completely identical to each other. The different scenarios were meant to represent different parts of the clock tree.

Figure 4.3 This graph shows the same scenario as figure 4.2, but for the last iteration of the clock tree. The two graphs look very similar but some differences should be pointed out. The total power consumption is visibly lower than the previous graph. Comparing the two, you can see that the top peak (22,40) in figure 4.2 has been reduced from 0.20mW to 0.12mW. In the two "top 10 peaks" lists, there are several differences. Also note that the category graph for CTS Buffers is lower in figure 4.3 than figure 4.2.

Figure 4.4 Shows the power scenario from the same estimation run as 4.3. It is the scenario that includes the most cells. It has slightly higher power consumption than the other graphs in this section, comparing the two, you can only see one difference. A spike at 25,45 that is there in figure 4.4 and not in figure 4.3.

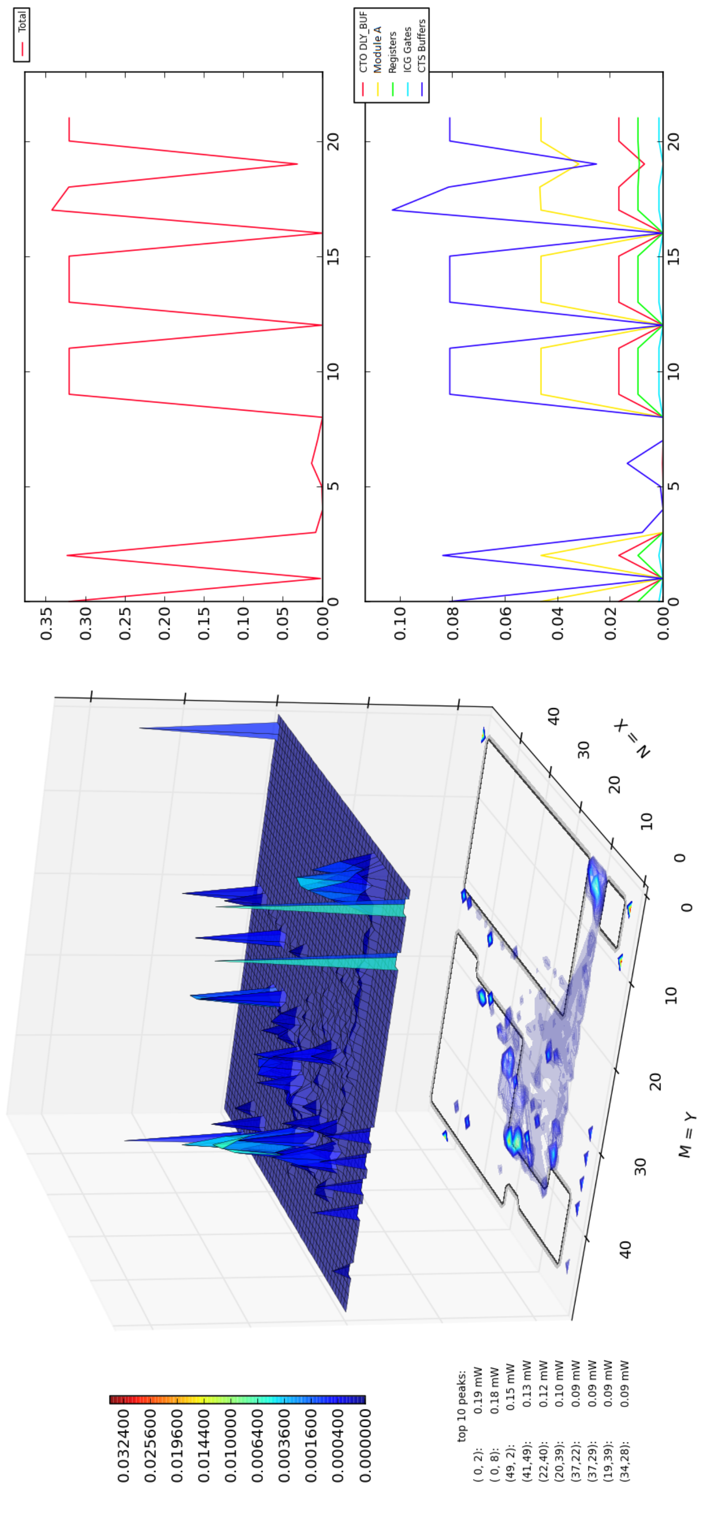


Figure 4.2: Power consumption of clock tree on chip to be sent to tape out. Early iteration. The frame shown is image 21 of 23.

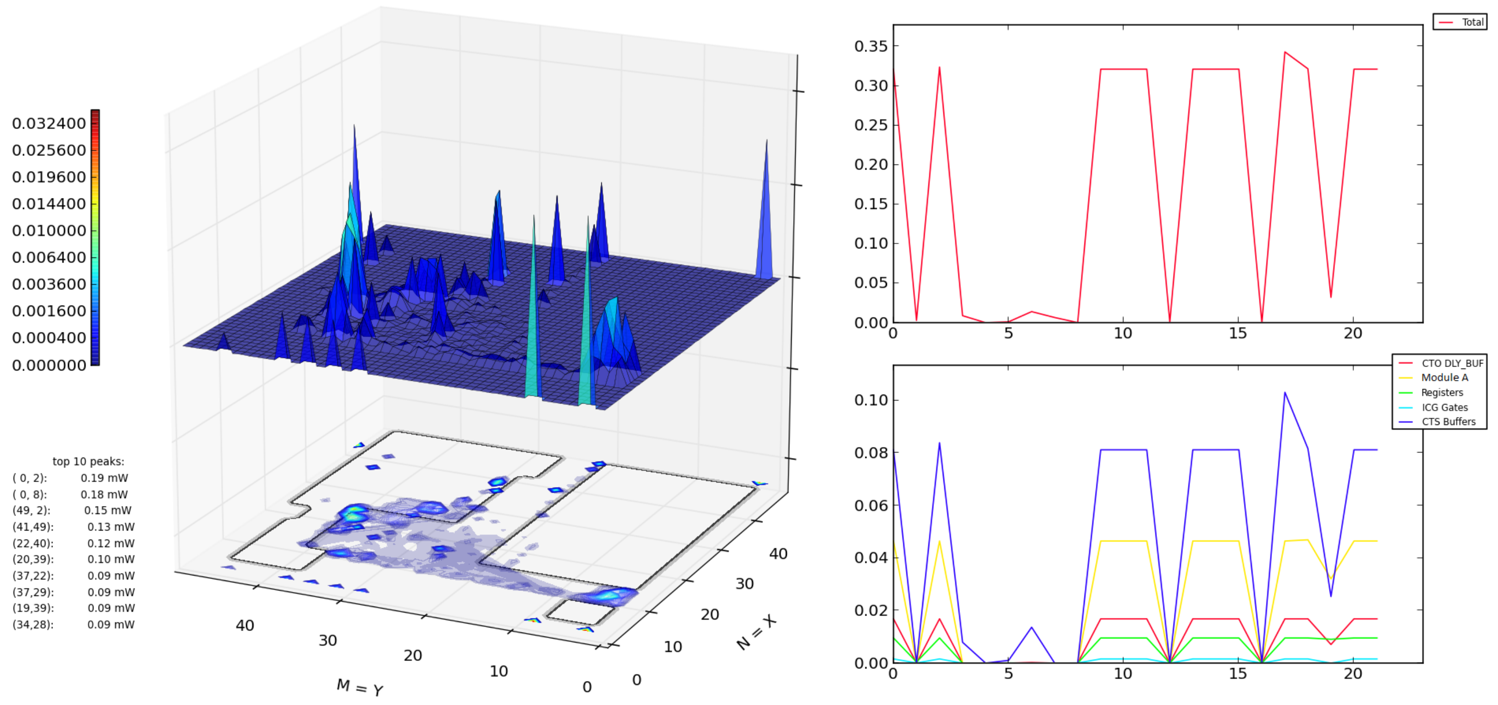


Figure 4.3: Power consumption of clock tree on chip to be sent to tape out. Final iteration. The frame shown is image 21 of 23.

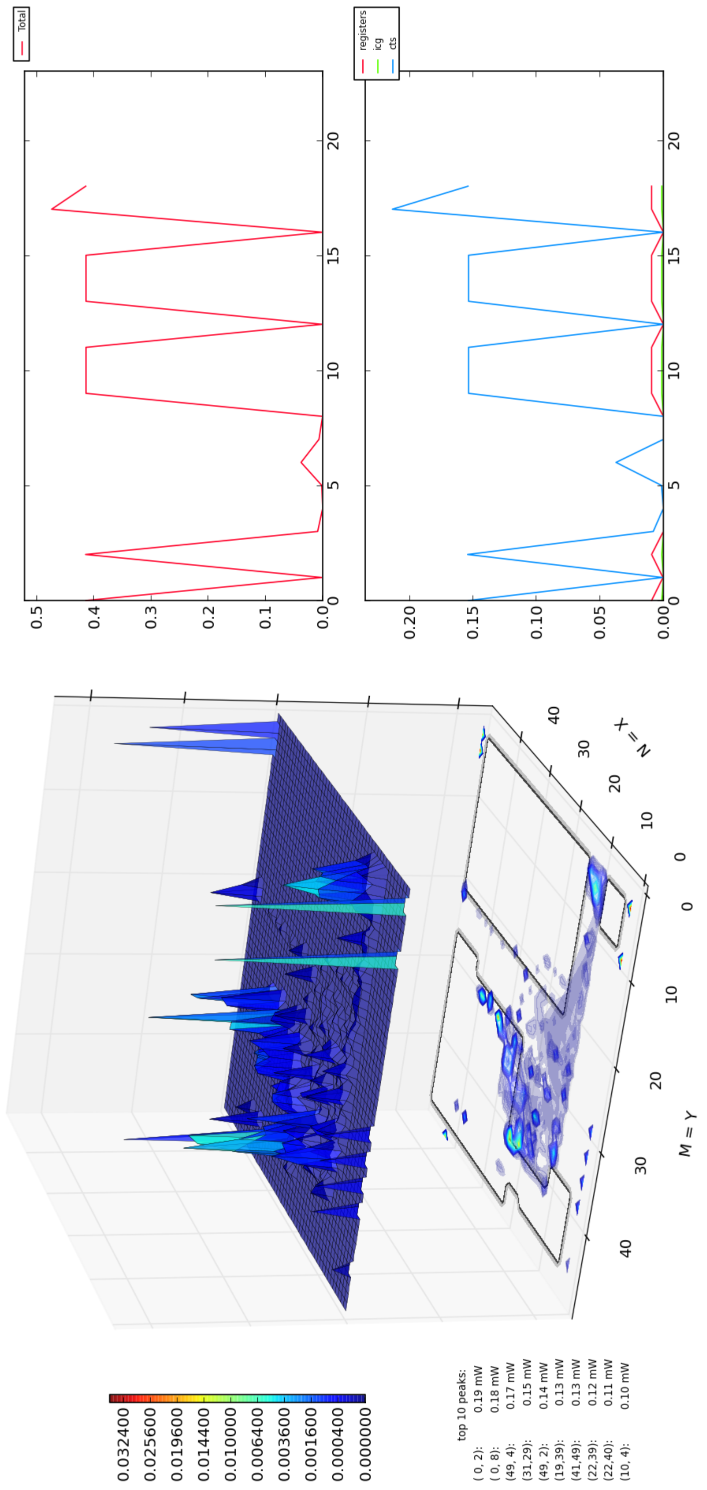


Figure 4.4: Power consumption of clock tree on chip to be sent to tape out. Final iteration. Full clock tree.

4.3 Animated graphs

Since displaying animations on paper is not possible, a few screen shots showing the progression of the animation is displayed. All figures 4.5 to 4.9 have incorrect data in the top 10 peaks list. The numbers are 1000 times larger than the actual values. The 5 screen shots are chosen out of the 4000 to show the surface plots of different modes the circuit run in. These graphs are based of reports of another circuit than the previous images. The power estimation these reports are from is time based, which means it is based on a simulation of a test bench. In this section the important parts of these figures will be noted. The notation (n,m) is used to describe points in the surface plot, and areas will be described by two points (nMin , mMin),(nMax , mMax) representing two opposite corners of a macro as seen in figure 3.3.

Run time A set of 4000 reports formed this animation. It took in 4 hours to run the program. Out of this 1 hour and 35 minutes was used to gather and organize the data, an average of 1.4 s per report. Drawing the graphs took 1 hour and 25 minutes, which averages out to 2.2 seconds per graph. The reports contained approximately 100 000 cells each.

Figure 4.5 Frame 136 is the frame with the highest power consumption in the start up phase. From the categories graph on the right you can see that the sram is using power. This can be seen in the surface plot too, it is the macro in the closest corner. The active sram unit stretches from approximately (1,1) to (8,12).

Figure 4.6 In this mode of operation, the flash and sram both have varying power consumption. This can be observed by the categories graph, where the total categories graphs for sram and flash. Which means they are periodically active for data operations.

Figure 4.7 Frame 2029 shows a macro dominating the power use in the surface plot. In the categories graph the flash is no longer active, the sram has less frequent sporadic accesses and the modem is active. If you compare figure 4.7

to figure 4.6, you can see that the activity in the area between (1,5) and (20,20) has greatly decreased, but a new area around (30,39) has risen.

Figure 4.8 In frame 3525 the categories graph shows that the modems power consumption has risen drastically. The only real change in the surface plot is a new active area on the chip surrounding the point (15,46).

Figure 4.9 The last frame shows the entire overview graphs on the right. It also shows a huge spike at (42,0) it only occurs a few times in single frames. Another thing that was discovered from this animation was that a cluster at (22,23) remains constant the entire run time.

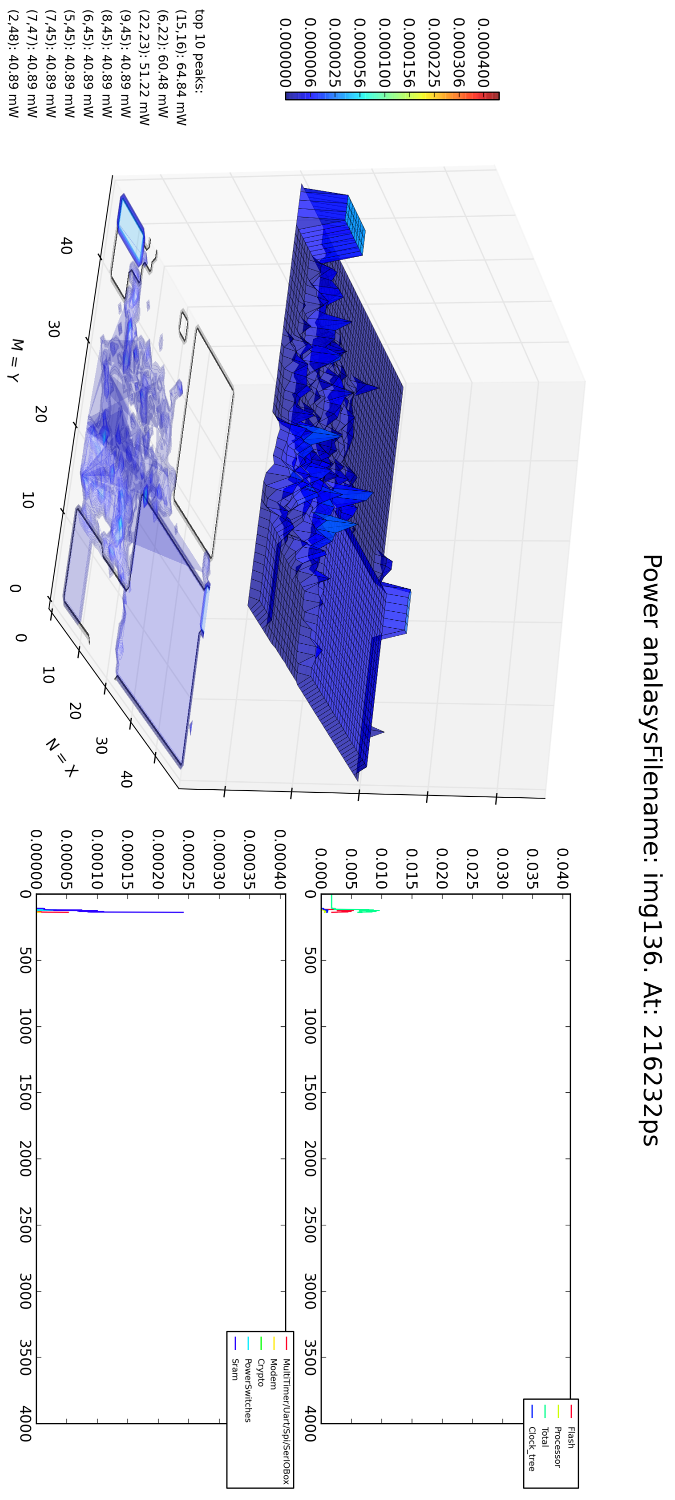


Figure 4.5: Stillframe from animated sequence. Frame 136 of 4000

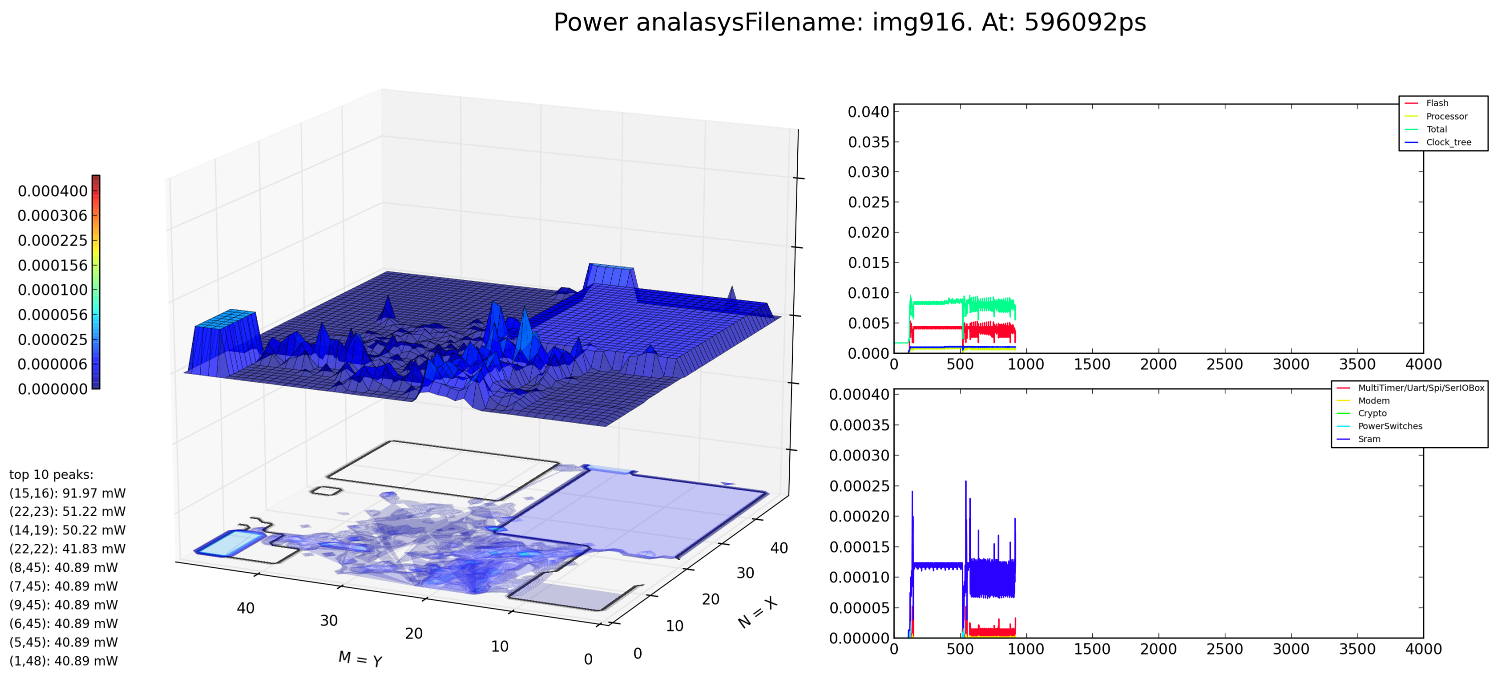


Figure 4.6: Stillframe from animated sequence. Frame 916 of 4000.

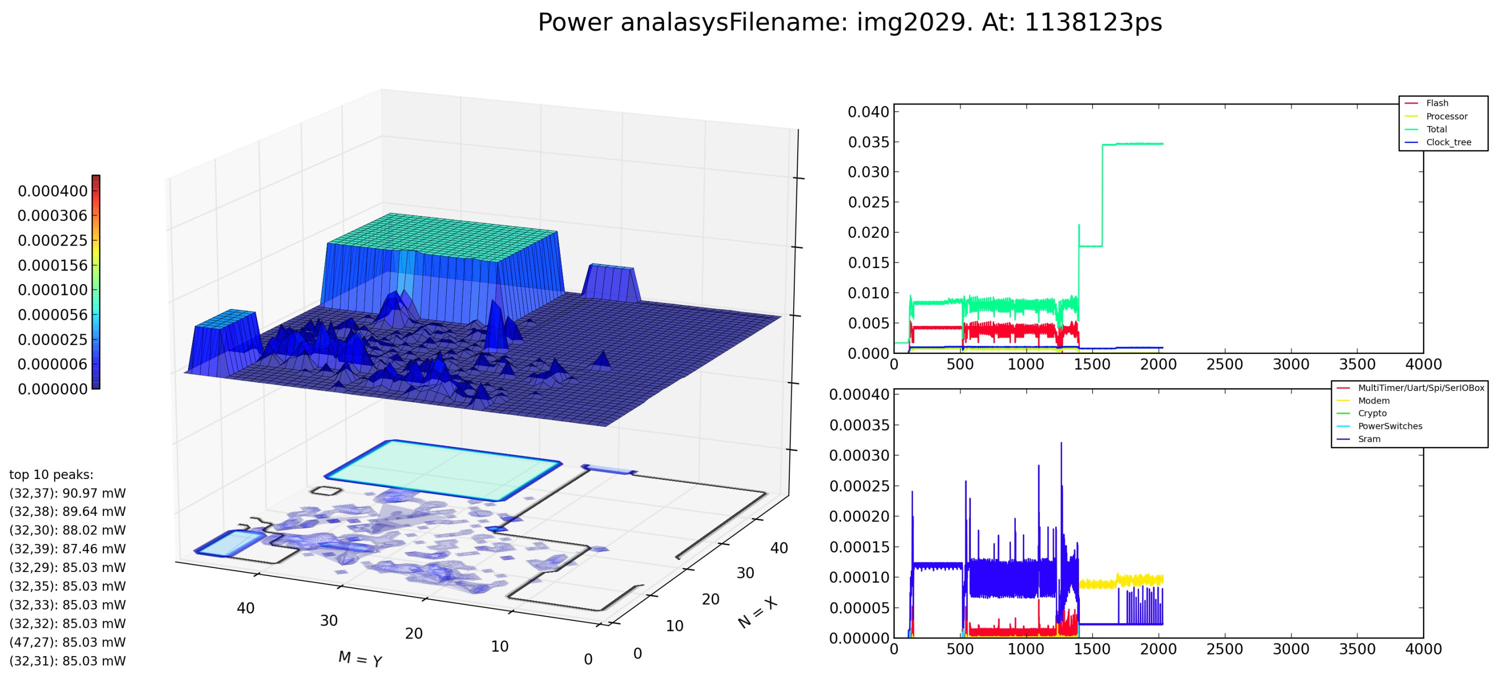


Figure 4.7: Stillframe from animated sequence. Frame 2029 of 4000

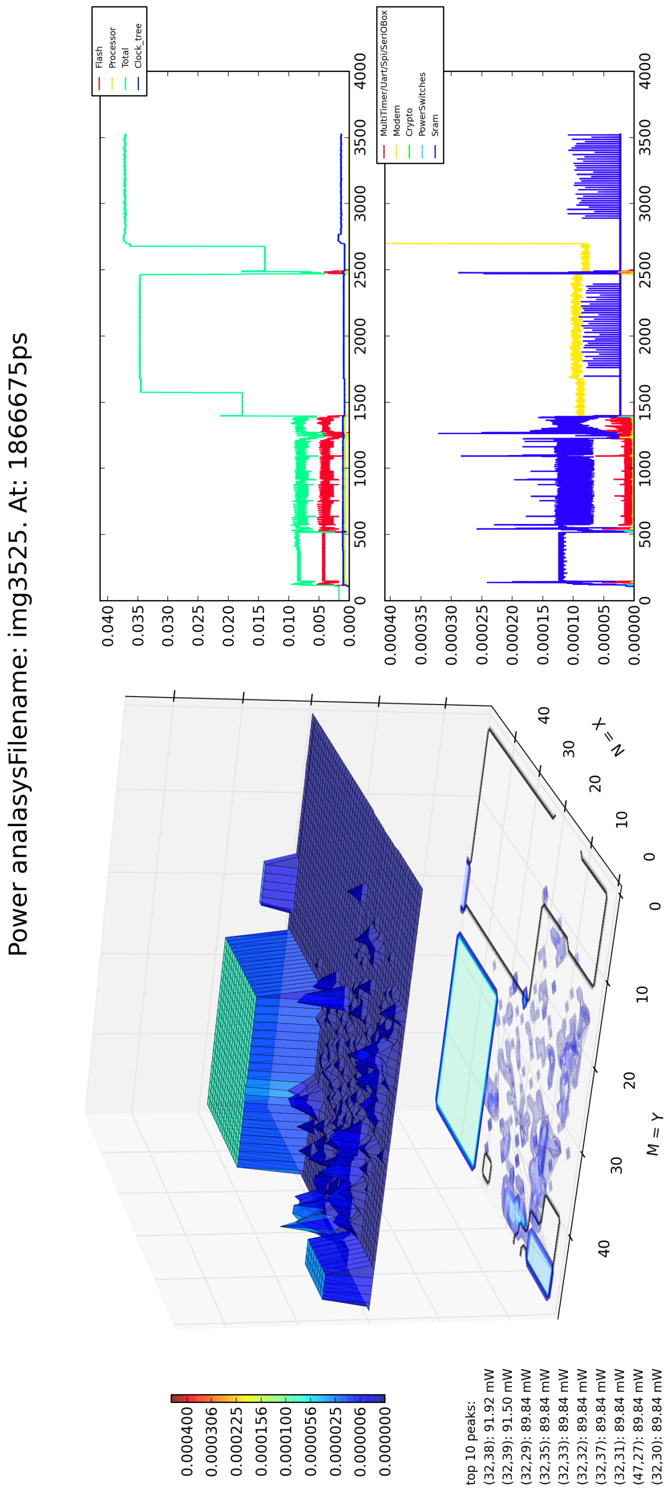


Figure 4.8: Stillframe from animated sequence. Frame 3525 of 4000

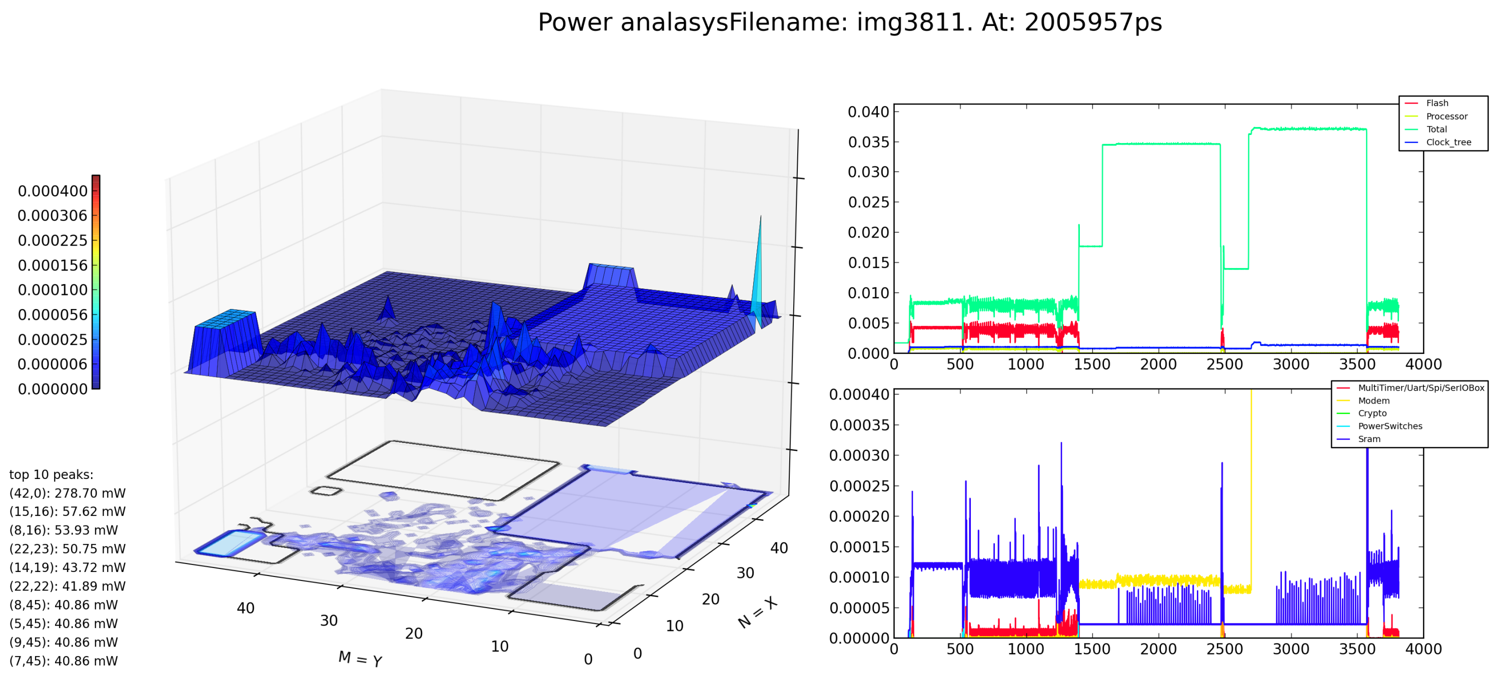


Figure 4.9: Stillframe from animated sequence. Frame 136 of 4000

4.4 Scenario plotting

This section shows two graphs in figure 4.10 and figure 4.10. These two images are part of a series of 41 scenarios. This test was done with no categories defined, so there is only one overview graph on the right side. The top 10 peaks list contains wrong, inflated values. They are included to show how the program handles plotting a series of scenarios. One of the requirements of plotting graphs for a video is for the limits of the graphs to stay the same. For scenarios, the limits on the surface plot is only based of the current frame. This allows for high detail even in frames with lower power consumption.

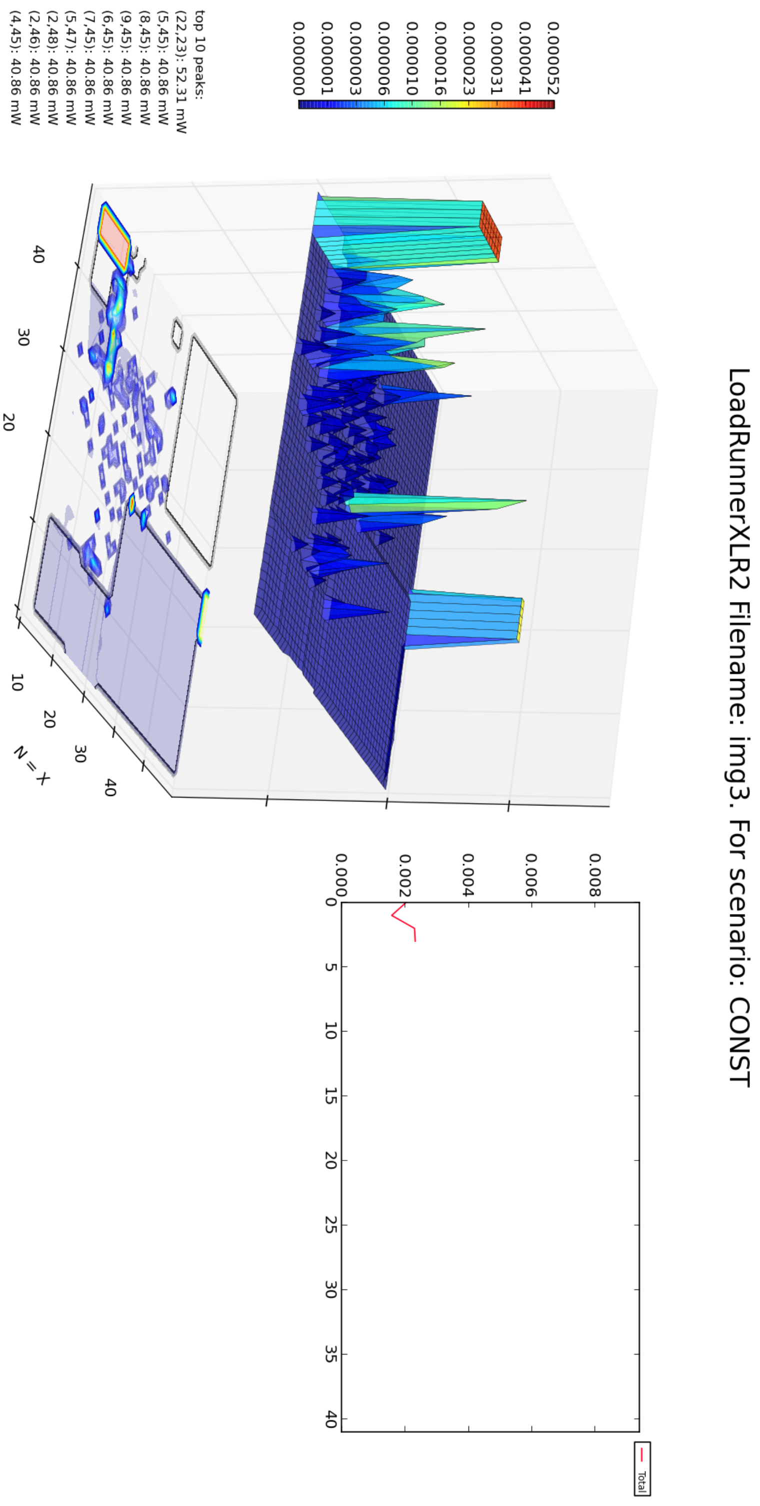


Figure 4.10: Power consumption of clock tree first time after being generated.

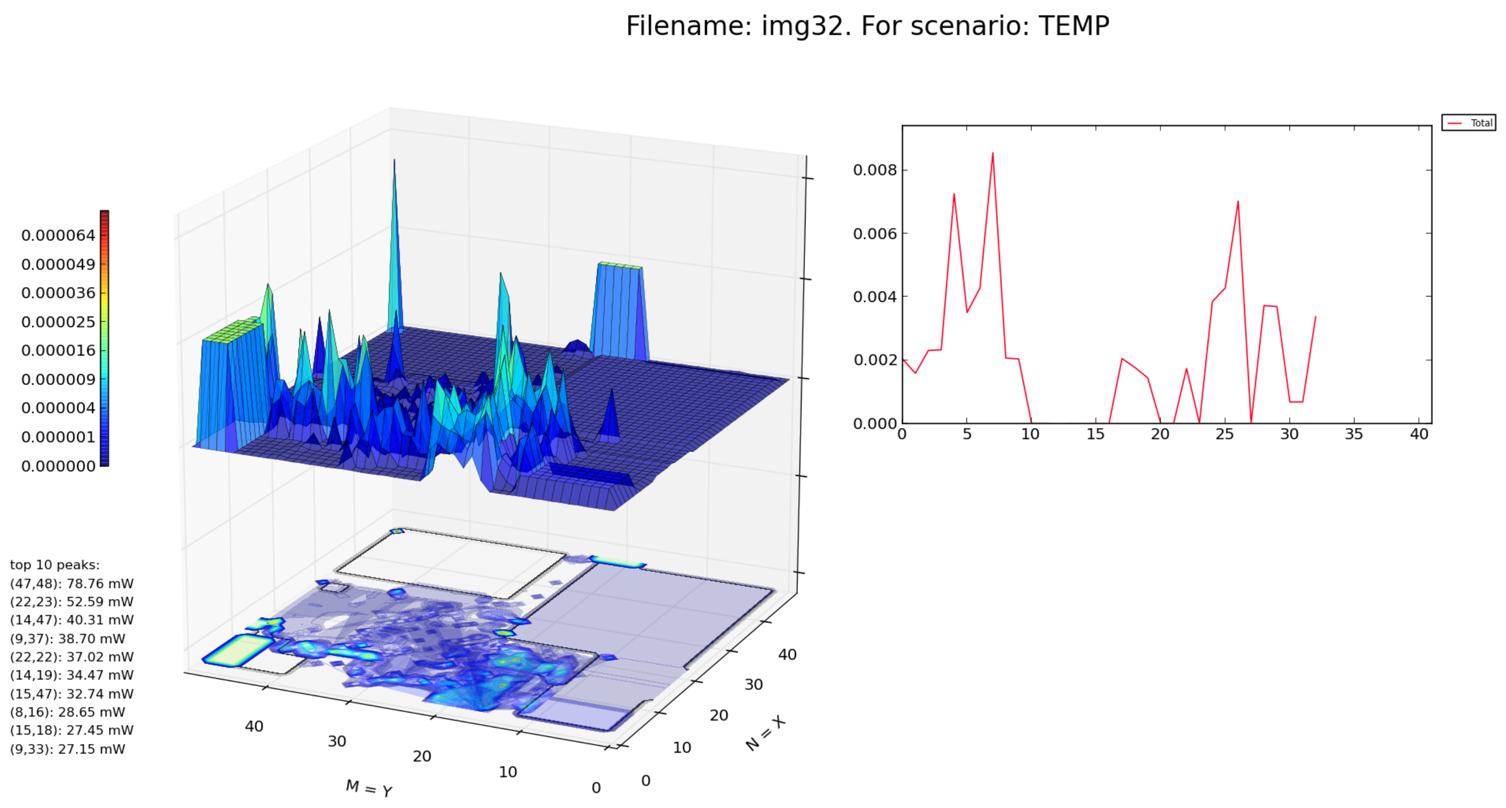


Figure 4.11: Power consumption of clock tree first time after being generated.

Chapter 5

Discussion

The tool in this thesis is designed to be as universal as possible. Its application depends on what data you choose to graph. While the goal of the thesis was to use the program to reduce clock power in the sign-off phase, new use cases were suggested by the people I worked with in Nordic. In this chapter you will find discussion on the results, the original goal as well as other valuable use cases for this tool.

5.1 Merits of visualizing

A basis of this entire project is that visualizing data is important. To reduce repetition, I will handle this first, so we can assume for the rest of the discussion that being able to visualize large data sets is a positive thing in itself. The human brain is very good at recognizing patterns. Even when we do not know what we are looking for, we will find patterns. We even have a tendency for finding patterns when we know they are not supposed to be there, its called "Apophenia"[14]. Algorithms can be designed to quickly find something in large data, but they need to be designed. To design an algorithm, you need to know what you are looking for. Circuit behaviour is very complex. We understand some of it, enough to make things work, but our models are always simplified, and you seldom personally understand more than one abstraction level down in much detail. For a designer that knows a circuit, access to data from a simulation in graph form, will enable her or him to discover weird behaviour or problems by looking for suspicious patterns. With a easy way to get extract numbers from the same data, the designer can check if the pattern she or he sees is real or just random. This is a powerful tool to increase understanding and find unexpected results. Increased understanding can help to improve design, and identifying mistakes can save time and money in a design process. It is hard to quantify the value visualization adds to data, but easy to argue that a good visualization enriches data. Especially in large data sets handled by a engineer. With the value of visualization in mind, we will proceed to the more specific points of this paper.

5.2 Analyzing test results

There were several test results presented in chapter 4. In this section they are discussed in order to reach a conclusion.

5.2.1 Utilizing clock visualization in the design phase

Run time The sign-off phase, where the clock is generated, is a time sensitive design phase. Making the visualization a natural part of the design phase, required the tool to be easy to use, and execute fast enough. In Nordics user case, 23 reports of the estimated power used in the clock tree were generated from Primetime, a analysis tool. This process takes in the vicinity of 10 to 15 minutes. The additional time used to make graphs of the results is 72 seconds, or around 1 minute and 10 seconds. This is not a problematic run time increase, even in a time sensitive design phase.

Notes on the graphs From figure 4.2 and 4.3 we can see that the iterative design refinement achieved a power reduction. The exact numbers were pulled from the excel output and they showed a power reduction of 11.3%. This is however not a new result based of a contribution from the thesis, as the design process was executed as it normally is at Nordic. The goal of this thesis was to produce rich graphs for the designers to have more information available when generating the clock tree. The graphs are by any standard far more informative than the average power number previously used to evaluate the clock constraints. Power consumption density charts with circuit layout is not available through the conventional tools Nordic use today.

The graphs from the iterations of the clock tree did not turn out very different from each other. This means the input of the tool is uninformative, not that the tool does not do its job. As you can see in figure 4.3 9 of the 23 scenarios used to check clock power consumption actually show exactly the same data. Running 9 of these instead of 1 is unnecessary overhead. It is also important knowledge that the reports show exactly the same data, as they are meant to show different aspects of the power. This means the tests have to have different constraints to give the results they are intended to.

W hen tested in a real world design phase the tool did its job, but the it not help tremendously in evaluating the clock tree. The reports did not show the information they were intended to, and the deadline could not be moved, so there was no time to fix it. The graph does still show some interesting points. Figure 4.4 shows a spiky surface plot that represents the clock tree. Big spikes can contain clusters of buffers made to split out the clock signals in several wires. The spike at (22,40) on the top 10 list in figure 4.4 turned out to contain 115 buffers, an unreasonable amount, that should be spread out. This was found by using the look up tool to check the cells in the bin, and searching for buffers. Splitting wires with buffers as late as possible is generally better. It decreased the area used for weiring, and the buffers become more spread out, resulting in more even power dissipation. This was not used to improve the clock tree this time, but the graphs and the tools around them can help a designer find points like this, with potential for optimization.

5.2.2 Animated graphs from time based estimation

The graphs for the time based analysis can be found in section 4.3.

Run time The full animation with 4000 graphs took 4 hours to generate and render. This is a significant run time, but the analysis reports took 2-3 days to generate. In comparisons to the report generation time, 4 hours is an acceptable run time. As a user you can choose to only graph 5 reports first, in order to configure the settings the way you want. This will not take more than half a minute, which is fast enough to get the configuration right before running on 4000 reports.

Graphs This plot gathers data from over 100 GB of reports, they hold an immense amount of information. In section 4.3 different points of interest is highlighted from the graphs. A designer that knows the circuit will be able to see these without help, and find even more. The engineers that had been working with this chip could instantly recognized behaviour patterns and modules even without proper category graphs. The tool unlocks information never visible to the designers from other graphs. This can be seen in Figure 4.6 which contains the same time slice as the graph from Primetime PX in figure 2.3. Time based power analasys is not typically available before the circuit is prototyped, as the test benches will not be ready yet. This means these detailed graphs will not be able to help the designer in the initial clock tree generation process.

Utilizing time based animation When the circuit prototype returns from tape out it is tested and power use is measured. Is is only possible to measure power consumed by the circuit over time. There is no way to split that number up into what parts of the circuit used what, except measuring different scenarios with known activity level in the chip. Having a detailed graph of where the power is used is very useful for understanding power measurements. The test benches that run on the prototype, can also be simulated, and the activity data can be analyzed in Primetime PX. Comparing measurements and analysis results from the same test benches can improve the understanding of the relation, and give insight in the actual power dissipation. This is done already with the power graphs from Primetime PX, but the tool can give extra insight.

Understanding the current design There are also other uses for properly understanding the current design. In the modern circuit industry, reuse of modules is very common. So finding parts of the circuit has potential for power optimization can enable the designers to improve the next generation circuit. The tool gives the designer a way to access the immense amount of information hidden in hudred GB of reports. An example is mentioned in figure 4.6 4.6, a spike that only occurred on a few frames. This was something the engineers reacted too, as it seemed odd, checking on a later iteration of the design, this was gone. This type of information would not be detectable in the power graphs from Primteime PX.

5.2.3 Graphing scenarios

The graphs with scenarios is part of the results to show of the scaling on the Z axis in the surface plot. If you compare figure 4.10 with figure 4.11 you can clearly see that they have different scale, even though they are in the same sequence. This is done as scenario data is not interesting to animate, like time based data is, but will work as separate graphs. With a scale that changes, you get a better visual comparison between high and low power peaks in the plot.

5.3 Future work

Nordic will continue to use the tool in their clock tree generation phase, as well as for plotting Primetime PX power estimations. For this thesis my supervisor at Nordic, Jan Egil Øye, has provided me with reports from Primetime PX. The process of generating these reports is part of the design work flow already, but the tool sets new standards to the reports. To gain more out of the visualization program, the power estimations done by Primetime PX and the data printed has to be reevaluated. With the tool ready to visualize any resulting reports, this can give the designers much more informative reports than the ones presented in section 4.2.

5.3.1 Keeping uniform chip temperature

Another way to take this work forward is to use the tool to create a visualizer for another specific domain. One domain where looking at total power consumption of a chip might help to optimize is to locate heat spikes in a circuit. Big temperature difference within a circuit can lead to timing problems as delay in wires are related to temperature. In ultra low power circuits, high power operations should never run too long, meaning the case should be extreme for heat to be a problem. In higher performance processors, heat is a limiting factor. For this type of work, a well represented visualization of heat or power density can show problem areas. Power dissipation is the source of the heat, so a heat graph could possibly be created from a power density graph. This transformation could be based on additional data and be added in the transformation phase of this tool, or a new tool could be built from scratch using the experiences made in this thesis.

Chapter 6

Conclusion

It proved to be possible to visualize power consumption density on-chip without significantly adding run time to the power analysis process. This gives the designer the possible to utilize data from power analysis reports that are far more detailed than what was previously taken into account when evaluating clock trees. The visualized power consumption density proved to be a useful way to handle data for a designer. In addition the tool creating the graphs enables the designer to quickly find potential optimization points and research the details through an index that connects the cells names to the plot surface bins. For power aware designs, the clock tree is a tactical place to optimize as it often responsible for a large part of circuit power consumption.

The tool combines reports from IC Compiler containing location data with reports from power analysis executed by Primetime PX. This was tested in the design phase where Nordic generates the clock for their circuit. The reports usually generated for this, were not good for graphing, but the tool still showed some potential optimization points. Graphing of 4000 reports, each containing power information from around 100 000 cells, was achieved within four hours. The resulting animation unlocks huge amounts of data that would otherwise not be easily accessible for the designers. The graphing tool developed in this thesis fills a gap for a more thorough investigation of power analysis reports in the commercial tools Nordic use. It will continue to be a part of their clock tree design process, and they will focus on refining reports to better utilize the visualization to find points of interest in the clock tree.

Appendix A

Acronyms

ULP	Ultra Low Power
UCD	User Centered Design
vcd	Value Change Dump
SPEF	User Centered Design
SDC	User Centered Design
CTS	Clock Tree Synthesis

Bibliography

[1] Monica Donno, Enrico Macii, and Luca Mazzoni. “Power-aware clock tree planning”. In: *Proceedings of the 2004 international symposium on Physical design*. ACM. 2004, pp. 138–147.

[2] Karthik Kannan Duane E. Galbi. “Measuring Active Power Using Primetime PX”. In: (2010).

[3] D. Duarte, V. Narayanan, and M.J. Irwin. “Impact of technology scaling in the clock system power”. In: *VLSI, 2002. Proceedings. IEEE Computer Society Annual Symposium on*. 2002, pp. 52–57. DOI: 10.1109/ISVLSI.2002.1016875.

[4] *Notes on User Centered Design Process (UCD)*. URL: http://www.w3.org/WAI/redesign/ucd.

[5] *Official homepage of ffmpeg*. URL: http://www.ffmpeg.org/.

[6] *Official website of Matplotlib*.

[7] *Official website of Pypy*. URL: http://pypy.org/.

[8] M. Olsson et al. “Extracting vectors from application traces for power integrity analysis”. In: *Signal Propagation on Interconnects (SPI), 2011 15th IEEE Workshop on*. 2011, pp. 39–42. DOI: 10.1109/SPI.2011.5898836.

[9] *Python regular expression module*. URL: https://docs.python.org/2/library/re.html.

[10] P. Ramanathan, A.J. Dupont, and K.G. Shin. “Clock distribution in general VLSI circuits”. In: *Circuits and Systems I: Fundamental Theory and Applications, IEEE Transactions on* 41.5 (1994), pp. 395–404. ISSN: 1057-7122. DOI: 10.1109/81.296331.

[11] *Regular expessions info page.* URL: http://www.regular-expressions.info/.

[12] *Synopsis Ic Compiler infor page.* URL: http://www.synopsys.com/Tools/Implementation/PhysicalImplementation/Pages/ICCompiler.aspx.

[13] *Synopsis Primetime info page.* URL: http://www.synopsys.com/Tools/Implementation/SignOff/Pages/PrimeTime.aspx.

[14] *Wikipedia article on "Apophenia".*

[15] *Wikipedia article on Comma-separated values.* URL: http://en.wikipedia.org/wiki/Comma-separated_values.

Appendix B

Quick user guide

This program can create images to animate graphs from power reports. The supported format is Primetime PX

To run the program on a folder with no configuration use:
./run_on_folder.py [folder name]
This folder needs to contain some things:
Power reports. If they are from consecutive time frames their names need to end with: .[numbers].rpt Otherwise they will be handled as scenario reports.
Location report with the name: outputCellLocation.csv

Macro reports with the name: outputCellBbox.csv
A categories file with the name: Category

To create a category file, or run the program with more guidance and configuration opportunities:
./animate.py
Follow the instructions. The program will create a settings file for you which you will have to modify to make work.

Appendix C

The code

The source code in this project is included below. The program can be run with a command line interface that helps the user set up the program and shows the format for the categories file. run_on_folder.py takes only a single input, that is the folder name. There are some restrictions to that folder. It needs to contain the location file, categories file and bbox file with specific names. The graph_reports.py is the one currently included in the script used to analyse the clock tree. This chapter includes all important code made for this thesis.

Listing C.1: Run_on_folder.py

```
#!/usr/local/etc/python/current/bin/python2.7
#Graph_reports.py



# This file is just to give an easy user interface. It is not strictly needed to
     graph, it is a file to set up the graphing for a folder allready containing
     the location file, macro file and categories file.
# It graphs the folders with the easiest settings, only allowing the user to change
     category data
import sys, getopt, re, datetime, pickle, os, operator
import parser, graph_classes, graph, math

def main(argv):
  folder = argv[1]
  if not folder.endswith("/"): folder += "/"

  location_path = folder + "outputCellLocation.csv"
  bbox_path = folder + "outputCellBbox.csv"
  graph_folder = folder + "graphs/"
  if not os.path.exists(graph_folder):
        os.makedirs(graph_folder)
  if os.path.isfile(folder+"category"):
        categories_file = folder+"category"
  else:
        categories_file = "clock_categories"
        print "No file named category found in target folder, using file
             \"Clock_categories\" instead"


  starttime = datetime.datetime.now()

  #animation = graph_classes.Animation(["TITLE"],["LOG_MODE"],["N"],["M"])
  animation = graph_classes.Animation("Surface plot of power report", True, 50, 50)

  #animation.load_frames(["REPORT_FOLDER"],["LOCATION_FILE"],["BBOX_FILE"],["START_FRAME"],["END_FRAM
  animation.load_frames(folder, location_path, bbox_path,0, 0, "clock_categories"
       ,power_index = 3)

  load_time = (datetime.datetime.now())

  animation.graph_all_frames(graph_folder)

  graph_time = (datetime.datetime.now())
  print "time used to load all data",(load_time-starttime).seconds
  print "time used to graph all graphs",(graph_time-load_time).seconds
  print "total time used", (graph_time-starttime).seconds

if __name__ == "__main__":
  #Expexted input: [ Folder containing reports, cell locatons file , bbox file ,
       category file]
  main(sys.argv)
```

Animate has a settings file. If it does not get it as an input, it will create one for you. It can also create a working example categories file.

Listing C.2: animate.py

```
#!/usr/local/etc/python/current/bin/python2.7


# This file is just to give an easy user interface. It is not strictly needed to
     graph, it just makes sure the correct input is given to the graph_class.
# To use the graphing tool directly, use the following functions as they are used
     in this file:
# animation_object = graph_classes.Animation(inputs) - Sets up the animation
     framework
```

```
# animation_object.load_frames(input) Gathers the data from reports and creates
     frame data ready to be graphed
# animation_object.graph_all_frames(input) Graphs all the data one by one and saves
     them as images in a folder
# Look in the Settings class in this document to see a description of all input
     variables.

import sys, getopt, re, datetime, pickle, os, operator
import parser, graph_classes, graph, math

def main(argv):
  s = Settings()
  if not len(argv) == 1:
    print "Loading settings from: " + argv[1]
    s.load_config_file(argv[1])
    print "Following settings were loaded: "
    for key, value in s.settings.iteritems():
      print key, " = ", value.value
  else:
    s.print_default_config_file()
    print "default config file created. Please modify it before running again"

  if not s.settings_ready:
    print "Program not running because of lacking configration"
    if not len(argv) == 1:
      print "Some settings that can't be are same as default settings."
      print "See warnings above and modify the config file: " + argv[1]
    return



  starttime = datetime.datetime.now()
  animation = graph_classes.Animation(s.settings["TITLE"].value,
       s.settings["LOG_MODE"].value, s.settings["N"].value, s.settings["M"].value)
  animation.load_frames(s.settings["REPORT_FOLDER"].value,s.settings["LOCATION_FILE"].value,s.setting
       power_index = s.settings["PARSER_COLUMN"].value)
  load_time = (datetime.datetime.now())
  animation.graph_all_frames(s.settings["PLOT_FOLDER"].value)
  graph_time = (datetime.datetime.now())
  print "time used to load all data",(load_time-starttime).seconds
  print "time used to graph all graphs",(graph_time-load_time).seconds
  print "total time used", (graph_time-starttime).seconds

class Settings:
  def __init__(self, filename = None):
    self.filename = filename
    self.settings = {}
    self.settings["TITLE"]= Single_setting("This is the title on top of the
         screen", "string", "Default_tile", True)
    self.settings["LOG_MODE"]= Single_setting("Plot the power surface logarithmicly
         Yes/No", "bool" , "Yes" , True)
    self.settings["N"] =Single_setting("Decides how many bins for there to be in
         the N direction, NxM = XxY ", "int" , "50", True)
    self.settings["M"] =Single_setting("Decides how many bins for there to be in
         the N direction, NxM = XxY", "int" , "50", True)
    self.settings["LOCATION_FILE"] =Single_setting("The path to the file giving the
         location of every cell", "Valid path" , "path", False)
    self.settings["BBOX_FILE"] =Single_setting("The path to the file giving the
         bboxes of macros that should have destribiuted power, optional, but gives
         better result", "Valid path" , "None" , True)
    self.settings["PLOT_FOLDER"] =Single_setting("The path to the desiered folder
         for the finished plots to be saved", "Valid path" , "finished_plots" , True)
    self.settings["REPORT_FOLDER"] = Single_setting("The path to the folder
         containing all the reports", "Valid Path" , "power_folder_path", False)
    self.settings["START_FRAME"] =Single_setting("What report number to start
         making animations on", "int" , "0", True)
    self.settings["END_FRAME"] =Single_setting("What report number to stop making
         animation images on, if 0, all reports will be used", "int" , "0", True)
    self.settings["CATEGORY_CONFIG"] =Single_setting("A config file for the
         categories in the right summary graphs to generate an example category
         config file with guiding comments set this field to **generate**", "Valid
         path" , "None" , True)
    self.settings["PARSER_COLUMN"] = Single_setting("What column to be used in the
         parser. 0 indexed, starting with the first number. If None, the parser
         default will be used", "int", "None", True)
    self.key_list=["TITLE", "REPORT_FOLDER", "LOCATION_FILE","BBOX_FILE",
         "CATEGORY_CONFIG", "PLOT_FOLDER","N","M","LOG_MODE", "START_FRAME",
         "END_FRAME", "PARSER_COLUMN"]#This is here to get a sensible ordering of
         the settings in the file
    self.settings_ready = False

  def print_default_config_file(self):
    command = raw_input("No configuration file specified. Do you want to create a
         default one? y/n > ")
    if command == "y":
      filename = raw_input("Please specify a filename > ")
    else: return
    overwrite = False
    while os.path.exists(filename) and not overwrite:
      want_to_overwrite = raw_input("That file allready exsists. Do you want to
           overwrite it? y/n > ")
      if want_to_overwrite == "y": overwrite = True
      else:
```

```
      command = raw_input("No configuration file specified. Do you want to create
           a default one? y/n > ")
      if command == "y":
        filename = raw_input("Please specify a filename > ")
      else: return
  self.filename =filename
  self.write_config_file(filename)

def write_config_file(self, filename, write_default= True):
  with open(filename, "w+") as f:
    for key in self.key_list:
      f.write("#" + self.settings[key].description + ". Needs type: " +
           self.settings[key].type+ "\n")
      if write_default: value = self.settings[key].default_value
      else:
        value = str(self.settings[key].value)
      f.write( key + "=" + value + "\n" )

def load_config_file(self, config_file):
  self.filename = config_file
  with open(config_file, "r") as f:
    for line in f:
      if line.startswith("#") or line.isspace():
        continue
      if line.strip().split("=")[0] in self.settings:
        self.settings[line.strip().split("=")[0]].set_value(line.strip().split("=")[1])
  all_settings_ready = True
  for setting, key in [(self.settings[key], key)for key in self.key_list]:
    if not setting.non_default_value:
      print "WARNING:Only default value is set for " + key + ". " + key + " needs
           to be defined to something else than: " + str(setting.value)
      all_settings_ready = False
    elif setting.type == "Valid path" and not setting.value == None and not
         os.path.exists(setting.value):
      print "WARNING: Path: " + setting.value + " does not exist please choose a
           valid path"
      all_settings_ready = False
    if key == "CATEGORY_CONFIG" and setting.value == "**generate**":
      self.generate_category_config_file()
      return

  self.settings_ready = all_settings_ready

def generate_category_config_file(self):
  print "Generate category config file option enabled in CATEGORY_CONFIG setting."
  filename = raw_input("Please specify a filename for the category_config file >
       ")
```

```
overwrite = False
while os.path.exists(filename) and not overwrite:
  want_to_overwrite = raw_input("That file allready exsists. Do you want to
       overwrite it? y/n > ")
  if want_to_overwrite == "y": overwrite = True
  else:
    command = raw_input("Do you still want to generate a category_config file?
         y/n > ")
    if command == "y":
      filename = raw_input("Please specify a filename for the
           category_config_file> ")
    else:
      self.settings["CATEGORY_CONFIG"].set_value("None")
      print "Changed settings file to "
      return
with open(filename, "w+") as f:
  f.write("#Any line that starts with # or is emtpy will be ignored\n")
  f.write("#Categories can be placed in the total plots on the right. To add a
       category write --[category name]\n")
  f.write("#The lines following a category contains the regular expressions
       that should be connected to that category\n")
  f.write("#There are some options when creating a category:\n")
  f.write("# --![categor name]   This marks a category that will be displayed
       in the top graph together with total. all other will be displayed at the
       bottom\n")
  f.write("# --*[caegory name]   marks the focus graph that the scale of the
       lower plot will be adjusted for\n")
  f.write("# --(mute)[category_name marks a category that should be gathered
       data on, but not showed in the graph. It will end up in the ecxell
       file]\n")
  f.write("#\n")
  f.write("#Following a category are several regular expressions. If you want
       to import regular expressions from a file\n")
  f.write("# You can instead of writing a regular expression write:\n")
  f.write("#(source)[file path]\n")
  f.write("# This is usefull if you have a list containing all cells that are
       for example in the clock tree, or in a certain bin.\n")
  f.write("#\n")
  f.write("#EXAMPLE SETUP. CHANGE FOR YOUR OWN NEEDS\n")
  f.write("--!registers\n")
  f.write("#The registers category will be plotted with Total power in the top
       graph \n")
  f.write("^.*reg.*$ \n")
  f.write("--*s_ram\n")
  f.write("#This will be the focus of the bottom graph, determining what the
       focus is\n")
  f.write("^SRAM REGEG$\n")
```

```
      f.write("--(mute)bin34,45\n")
      f.write("#bin34,45 will only show up in the excell summary\n")
      f.write("(sourse)filename_for_file_to_source\n")
      f.write("#And it will include all cells written in this file, can be
           generated by lookup_bin.py 34,45 > filename_for_file_to_source \n")
      f.write("#\n")
    self.settings["CATEGORY_CONFIG"].set_value(filename)
    self.write_config_file(self.filename, False)

  def make_config_file_from_prompts(self):
    command = raw_input("You did not specify a config file. Do you wish to create
         one? press y/n: ")
    need_command = True
    if command == "y" or command == "n": need_command = False
    while need_command:
      command = raw_input("Not valid answer, please answer y or n")
      print command
      if command == "y" or command == "n": need_command = False
    if command == "n":
      return
    need_command = True
    print "Setting up config file. For a guided creation type yes, to exit at any
         time type exit. \n To create default config file and not run the program,
         write a filename with a .cnfg extension"
    print "THIS FEATURE IS NOT YET IMPLEMENTED"

  def promt_for_category():
    pass

class Single_setting():
  def __init__(self, description, type, default_value, default_value_functional):
    self.description = description
    self.type = type
    self.default_value=default_value
    self.default_value_is_functional = default_value_functional
    self.set_value(default_value)
  def set_value(self, value):
    if value == "None":
      self.value = None
    elif self.type == "int":
        self.value = int(value)
    elif self.type == "bool":
      self.value = value in ["True", "true", "yes" , "ON", "on" ,"On" ," Yes " ," 1
           ", True]
    else:
      self.value = value
    self.non_default_value = (not (self.value == self.default_value)) or
         self.default_value_is_functional


if __name__ == "__main__":
  main(sys.argv)
```

Listing C.3: graph_classes.py

```
import os, parser, graph, re, math, pickle
from numpy import *

class Animation:
  def __init__(self, animation_title, log = False, range_n = 50, range_m = 50):
    self.title = animation_title
    self.log = log
    self.range_n = range_n
    self.range_m = range_m #Todo: Clean up
    self.frames=[]
    self.max_z= 0.0
    self.max_value_bottom_frame = 0
    self.max_value_top_frame = 0
    self.report_paths = []


  def write_category_data_to_excell_file(self, filename):
    with open(filename, "w+") as f:
      line = "Categories;"
      for key, value in self.frames[0].cat_dict.iteritems(): line += str(key) + ";"
      f.write(line[:-1] + "\n")
      for frame in self.frames:
        line = frame.tag +";"
        for index, (key, value) in enumerate(frame.cat_dict.iteritems()): line +=
             str(value) + ";"
        line = line.replace(".",",")
        f.write(line[:-1] + "\n")

  def graph_all_frames(self, target_folder):
    top_graphs={}
    bottom_graphs = {}
    for index, (key, value) in enumerate(self.frames[0].cat_dict.iteritems()):
      if key in self.cats_in_top_graph:
        top_graphs[key]=[]
      elif key in self.cats_in_bottom_graph :
        bottom_graphs[key]=[]
    if self.log:
```

```
    print "Logarithm mode on"
    self.max_z = 0
    for frame in self.frames:
      frame.change_data_to_log()
      self.max_z = max(frame.max_z, self.max_z)
  self.write_category_data_to_excell_file(target_folder + "/excell_summary.csv")
  for frame_nr, frame in enumerate(self.frames):
    for index, (key, value) in enumerate(frame.cat_dict.iteritems()):
      if key in top_graphs:
        top_graphs[key].append(value)
      elif key in bottom_graphs:
        bottom_graphs[key].append(value)
    if self.reports_for_animation:
      graph.single_z_frame(frame.data,frame_nr,self.max_z, self.circuit_layout,
          100, len(self.frames), frame.tag, frame.top_ten_peaks, self.log,
          top_graphs, bottom_graphs, self.max_value_top_frame,
          self.max_value_bottom_frame, self.title, target_folder)
    else:
      graph.single_z_frame(frame.data,frame_nr,frame.max_z, self.circuit_layout,
          100, len(self.frames), frame.tag, frame.top_ten_peaks, self.log,
          top_graphs, bottom_graphs, self.max_value_top_frame,
          self.max_value_bottom_frame, self.title, target_folder)

    print "graph number %d is done" %frame_nr

def load_frames(self,report_folder,location_file, macro_file = None, first_graph
     = 0 , last_graph= 0, categorize_power = None, power_name= None, power_index
     = None):
  self.find_all_reports_in_folder(report_folder)
  self.load_locations(location_file, macro_file)
  self.load_power_categories(categorize_power)
  if int(last_graph) > 0:
    self.report_paths = self.report_paths[int(first_graph):int(last_graph)]
  print "Loading report data from", len(self.report_paths)," reports"
  for index,report_path in enumerate(self.report_paths):
    #This makes it possible to choose the index of the power value in a parsed
        file either by name(a few standard names used) or by index number(this is
        more robust)
    if power_name: cells_power= parser.to_list_of_name_value_touple(report_path,
        value_name=power_name)
    elif power_index:
      cells_power= parser.to_list_of_name_value_touple(report_path,
          value_index=power_index)
    else: cells_power= parser.to_list_of_name_value_touple(report_path) # Default
        chooses total

    if not cells_power:
        print "WARNING: Parser could not find data in ", report_path, ". No graph
             will be generated for this report. Change parser in parser.py to fix
             this"
        continue
      frame_tag = 0
      if self.reports_for_animation: frame_tag = "At: " +
           report_path.split(".")[-2] + "ps"
      else: frame_tag = "For scenario: " + report_path.split("/")[-1][:-4] + "
           (Contains %d cells)" %len(cells_power)
      frame = Frame(frame_tag, self.range_n, self.range_m, self.all_category_names)

      for cell_power in cells_power:
        if cell_power[0] in self.cell_locations:
          frame.add_data(self.cell_locations[cell_power[0]], float(cell_power[1]))

      print "Data loaded for frame: ",index, frame_tag
      self.max_z = max(self.max_z, frame.analyze())
      self.max_value_bottom_frame = max(self.max_value_bottom_frame ,
           frame.cat_dict[self.focus_category])
      self.max_value_top_frame = max(self.max_value_top_frame ,
           frame.cat_dict["Total"])
      self.frames.append(frame)

  def load_locations(self, location_file, macro_file = None):
    self.cell_locations = {}
    with open(location_file, "r") as f:
      for line in f:
        cell_name =line.split(";")[0]
        cell_coordinates =line.split(";")[2]
        self.cell_locations[cell_name] = Location(cell_coordinates)
    self.max_x_value = max(self.cell_locations.iteritems(), key =
         lambda(k,v):v.x)[1].x+0.1
    self.max_y_value = max(self.cell_locations.iteritems(), key =
         lambda(k,v):v.y)[1].y+0.1
    for k, v in self.cell_locations.iteritems():
      self.cell_locations[k].calculate_nm_locations(self.max_x_value,
           self.max_y_value, self.range_n, self.range_m)

    if macro_file:
      with open(macro_file, "r") as file:
        for line in file:
             self.cell_locations[line.split(";")[0]].update_to_bbox(line.split(";")[2].strip())
      self.circuit_layout= [[5 for n in xrange(self.range_n)] for m in
           xrange(self.range_m)]
      for _ , cell in self.cell_locations.iteritems():
        if cell.is_bbox:
          for n_delta in range(cell.size_n):
```

```
        for m_delta in range(cell.size_m):
          self.circuit_layout[cell.n_min + n_delta][cell.m_min + m_delta] = 0
  else:
    self.circuit_layout = [[0 for n in xrange(self.range_n)] for m in
         xrange(self.range_m)]
    for _,v in self.cell_locations.iteritems():
      self.circuit_layout[v.n][v.m] = 5

  self.cells_in_bin={}
  for k, v in self.cell_locations.iteritems():
    for n , m in v.bin_list:
      self.cells_in_bin.setdefault(str(n)+","+str(m), ["x_min:"
           +str(n*self.max_x_value/self.range_n)+ " y_min:"
           +str(m*self.max_y_value/self.range_n)+ " x_max:"
           +str(n+1*self.max_x_value/self.range_n)+ " y_man:"
           +str(m+1*self.max_y_value/self.range_n)]).append(k)
  pickle.dump(self.cells_in_bin, open( "lookup_bin.p", "wb" ))


def find_all_reports_in_folder(self,report_folder):
  if not report_folder.endswith("/"): report_folder+= "/"
  for root, dirs, files in os.walk(report_folder):
    if files:
      for file in files:
        if file.endswith(".rpt"):
          if re.match("^\d*$",file.split(".")[-2]):
            self.reports_for_animation= True
          else: self.reports_for_animation = False
          if root.endswith("/"): self.report_paths.append(root+file)
          else: self.report_paths.append(root+"/"+file)
  print len(self.report_paths), " Reports found."
  if self.reports_for_animation: self.report_paths = sorted(self.report_paths,
       key = lambda x: int(x.split(".")[-2]))
  else: self.report_paths.sort()

def load_power_categories(self, filename):
  self.all_category_names = ["Total"]
  self.focus_category = "Total"
  current_category = False
  self.cats_in_top_graph = ["Total"]
  self.cats_in_bottom_graph = []
  if filename:
    with open(filename, "r") as file:
      for line in file:
        if line.startswith("--"):
          if line.startswith("--!"):
            current_category = line.strip()[3:]
           self.cats_in_top_graph.append(current_category)
          elif line.startswith("--*"):
           current_category = line.strip()[3:]
           self.focus_category = current_category
           self.cats_in_bottom_graph.append(current_category)
          elif line.startswith("--(mute)"):
           current_category = line.strip()[8:]
          else:
           current_category = line.strip()[2:]
           self.cats_in_bottom_graph.append(current_category)
          self.all_category_names.append(current_category)
        elif not line.startswith("#") and not line.startswith(" ") and not
           len(line.strip()) == 0 and current_category:
          if line.startswith("(source)"):
           sourcefile = line.strip()[8:]
           with open(sourcefile) as f: regex_list = f.readlines()
           if regex_list[0].strip()[0:2]== "Bin" and
              regex_list[0].strip()[-1]==":":
           regex_list = regex_list[1:] #To remove first line in files generated
              by lookup bin
           for regex in regex_list:
           for cell_name, box in self.cell_locations.iteritems():
            if re.match(regex.strip(), cell_name):
              self.cell_locations[cell_name].add_category(current_category)
          else:
           for cell_name, box in self.cell_locations.iteritems():
           if re.match(line.strip(), cell_name):
            self.cell_locations[cell_name].add_category(current_category)


class Frame:
def __init__(self, tag, range_n, range_m, categories):
  self.tag = tag
  self.data = zeros((range_n, range_m))
  self.categories = []
  self.cat_dict= {category_name:0 for category_name in categories}
def add_data(self, location, power):
  if location.is_bbox:
    for n_delta in range(location.size_n):
      for m_delta in range(location.size_m):
        self.data[location.n_min+n_delta][location.m_min+m_delta] +=
           power/location.area
  else:
    self.data[location.n][location.m] += power
  self.cat_dict["Total"]+= power
  for category_name in location.categories:
    if category_name in self.cat_dict:
      self.cat_dict[category_name]+= power
def analyze(self):
  self.top_ten_peaks = [ unravel_index(i, self.data.shape) for i in
       argsort(self.data, axis = None)[-10:]]
  self.max_z= self.data.max()
  return self.max_z
def change_data_to_log(self):
  for index, x in enumerate(nditer(self.data, op_flags=['readwrite'])):
    if  x > 0: x[...] = math.sqrt(x)
  self.max_z = math.sqrt(self.max_z)


class Location:
def __init__ (self, xy_string):
  self.x= float(xy_string.split()[0])
  self.y= float(xy_string.split()[1])
  self.is_bbox = False
  self.categories = []
  self.bin_list = []
def calculate_nm_locations(self, range_x, range_y, range_n, range_m):
  self.range_x = range_x
  self.range_y = range_y
  self.range_n = range_n
  self.range_m = range_m
  self.n = int(self.x * range_n / range_x)
  self.m = int(self.y * range_m / range_y)
  self.bin_list = [(self.n , self.m)]
def update_to_bbox(self, bbox_string):
  self.is_bbox = True
  x_min, y_min, x_max, y_max = [float(a) for a in re.sub("[{}]", " ",
       bbox_string).split()]
  self.n_min = int(x_min * self.range_n / self.range_x )
  self.m_min = int(y_min * self.range_m / self.range_y )
  self.n_max = int(x_max * self.range_n / self.range_x )
  self.m_max = int(y_max * self.range_n / self.range_y )
  self.size_n = self.n_max - self.n_min + 1
  self.size_m = self.m_max - self.m_min + 1
  self.area = self.size_n *self.size_m
  self.bin_list = [(self.n_min +delta_n,self.m_min +delta_m) for delta_n in
       range(self.size_n) for delta_m in range(self.size_m)]
def add_category(self, category):
  self.categories.append(category)
```

Listing C.4: parser.py

```
#parser.py
```

```
import re

def to_list_of_name_value_touple(filename,value_index = 4, value_name= None ):
  name_to_index= {"total" : 4 , "Total" : 4,"Total Power" : 4 , "Leak" :3 ,
       "leak":3, "leakage":3, "Leakage":3, "Leakage Power":3, "Switching Power":1,
       "Internal Power":1,"Internal":1, "Switching Power":2, "Switching" :2}
  print "parsing" , filename
  if value_name: value_index = name_to_index[value_name]
  with open(filename, "r") as file:
    valid_data_line=False
    output_list = []
    full_line = []
    for line in file:
      line = re.sub("[()%]", "", line)
      words = [x for x in line.strip().split() if not x == ""]
      if valid_data_line:
        if
            words[0].startswith('-------------------------------------------------------------------
          valid_data_line = False
        else:
          if len(words) == 1:
            full_line = []
            full_line.append(words[0])
          else:
            full_line.extend(words)
            output_list.append((words[0] ,words[value_index]))
      elif len(words) == 1 and words[0].startswith(
          '--------------------------------------------------------------------------------'):
        valid_data_line = True
  return output_list
```

Listing C.5: graph.py

```
#!/usr/local/etc/python/current/bin/python2.7

import sys, datetime
import graph_classes
from matplotlib import cm
from matplotlib.ticker import FuncFormatter
import matplotlib.ticker
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import animation
import numpy as np
from matplotlib._png import read_png
from pylab import imread
from matplotlib.patches import Rectangle as rectangle
```

```
import os

def single_z_frame(Z, nr,max_z,layout, max_layout, total_frames, frame_tag,
     top_ten_peaks, log_mode, top_graphs, bottom_graphs, top_max, bottom_max,
     title_text, target_folder = "finished_plots"):


  y = np.arange(0, len(Z), 1)
  x = np.arange(len(Z), 0, -1)
  X , Y = np.meshgrid(x,y)
  #fig = plt.figure(figsize=(17,8), dpi = 200 )
  fig = plt.figure(figsize=(16,9), dpi = 120 )
  fig.suptitle(title_text + ' Filename: img%d. '%nr + frame_tag , fontsize=20)
  min_z = -6*(max_z/5)

  #Axsis for 3d figure
  ax = fig.add_axes([0.05, 0.01, 0.55, 0.98], projection = "3d")
  #Plotting surface plot
  im = ax.plot_surface(X, Y, Z, rstride = 1, cstride = 1, cmap= cm.jet, alpha=0.7,
       linewidth=0.2,vmin=0, vmax = max_z )


  #Create legend with coordinates for peaks
  extra_list = [rectangle ((0, 0), 1, 0.2, fc="w", fill=False, edgecolor='none',
       linewidth=0) for i in xrange(len(top_ten_peaks)+1)]
  #if log_mode: top_ten_coordinates = ["%22s" % "top 10 peaks:"]+["(%2d,%2d):
       %13.2f mW"%(i[0],i[1],(Z[i[0]][i[1]])*(Z[i[0]][i[1]])*1000 ) for i in
       reversed(top_ten_peaks)]
  #else:
  top_ten_coordinates = ["%25s" %"top 10 peaks:"]+["(%2d,%2d): %13.2f
       mW"%(i[0],i[1],Z[i[0]][i[1]]) for i in reversed(top_ten_peaks)]
  lg = ax.legend( extra_list,top_ten_coordinates,bbox_to_anchor=(0.1,
       0.35),prop={'size':8} )
  lg.get_frame().set_linewidth(0)

  ax.set_xlabel('M = Y')
  ax.set_ylabel('N = X')


  #removing surface plot tick lables
  ax.w_xaxis.set_ticklabels([len(Z)-len(Z)*(n+1)/5 for n in range(5)])
  #ax.w_yaxis.set_ticklabels([])
  ax.w_zaxis.set_ticklabels([])

  # Plot the layout on the bottom
  #cset = ax.contourf(X, Y, layout, zdir= "z", offset =
       min_z,levels=np.linspace(1,1000,2),cmap=cm.gray, cstride=1,alpha=1)
```

```
cset = ax.contourf(X, Y, layout, zdir= "z", offset =
     min_z,levels=np.linspace(1,4,3),cmap=cm.gray, cstride=1,alpha=1)
cset = ax.contourf(X, Y, Z, zdir= "z", offset =
     min_z*0.999,levels=np.linspace(0+max_z/100,max_z,100),cmap=cm.jet,
     cstride=1,alpha=0.2)

#fix correct viewing og the 3d plot
ax.set_zlim(min_z,max_z*6/5, auto = False)
#ax.invert_xaxis()
ax.view_init(18,295)

#add colorbar
cb_ax = fig.add_axes([0.07, 0.45,0.005,0.3])
cb = fig.colorbar(im, cax = cb_ax)
def antilog(x, pos): return "%f" %((x*max_z)*(x*max_z)) # Data is never actually
     taken the logarithm of.
if log_mode:
  formatter = FuncFormatter(antilog)
  cb.ax.yaxis.set_major_formatter(formatter)
cb.ax.yaxis.set_ticks_position('left')



colormap = plt.get_cmap('gist_rainbow')
#bottom graph
if bottom_graphs:
  colormap = plt.get_cmap('gist_rainbow')
  ax2= fig.add_axes([0.6,0.1,0.35,0.35])
  ax2.set_color_cycle([colormap(1.*i/len(bottom_graphs)) for i in
       range(len(bottom_graphs))])
  for index, (name, plot_list) in enumerate(bottom_graphs.iteritems()):
    color = colormap(1.*index/len(bottom_graphs)) # color will now be an RGBA
         tuple
    ax2.plot(range(len(plot_list)),plot_list, label = name)
  ax2.legend(bbox_to_anchor=(1.13, 1.05),prop={'size':7})
  ax2.set_xlim(0,total_frames,auto=False)
  ax2.set_ylim(0,bottom_max*1.1, auto=False)

# Top graph
ax3= fig.add_axes([0.6,0.5,0.35,0.35])
ax3.set_color_cycle([colormap(1.*i/len(top_graphs)) for i in
     range(len(top_graphs))])
for index, (name, plot_list) in enumerate(top_graphs.iteritems()):
  color = colormap(1.*index/len(top_graphs)) # color will now be an RGBA tuple
  ax3.plot(range(len(plot_list)),plot_list, label = name)
ax3.legend(bbox_to_anchor=(1.13, 1.05),prop={'size':7})
ax3.set_xlim(0,total_frames,auto=False)
ax3.set_ylim(0,top_max*1.1, auto=False)
```

```
   #plt.savefig("finished_plots/img%d.png" %nr,dpi= 200, format="png")
  if not target_folder[-1] == "/" : target_folder += "/"
  if not os.path.exists(target_folder): os.makedirs(target_folder)
  plt.savefig(target_folder + "/img%d.png" %nr,dpi= 120, format="png")

  plt.close()
```

```
#!/usr/local/etc/python/current/bin/python2.7

import sys, getopt, re, datetime, pickle, os, operator


def main(argv):
  folder = argv[0]
  if not argv[0][-1] == "/": folder = argv[0]+"/"
  dictionary = pickle.load( open(folder+ "lookup_bin.p", "rb" ) )
  if argv[1] in dictionary:
    for cell in dictionary[argv[1]][:1]:
      print "Bin " + argv[1] + "has the restrictions: " + cell + " contains the
           following cells:"
    for cell in dictionary[argv[1]][1:]:
      print "  " + cell
  else:
    print "No cells in that bin, make sure your format is in the correct form:
         folder_name n,m"

if __name__ == "__main__":
  main(sys.argv[1:])
```