Master's thesis

NTNU
Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

Embla Trasti Bygland

Power Modeling of Complex Designs

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Snorre Aunet, Knut Austbø
July 2020

NTNU
Norwegian University of Science and Technology

Embla Trasti Bygland

Power Modeling of Complex Designs

Master's thesis in Electronics Systems Design and Innovation
Supervisor: Snorre Aunet, Knut Austbø
July 2020

Norwegian University of Science and Technology
Faculty of Information Technology and Electrical Engineering
Department of Electronic Systems

NTNU
Norwegian University of
Science and Technology

Abstract

In this project, a tool for making power models of designs at the Register Transfer Level (RTL) is implemented. The generated power model is intended to be used with a power estimation tool, to give an early, fast and accurate power estimate. Nordic Semiconductor ASA issues this masters project with the motivation of making RTL simulations power-aware. Discovering power bugs early in the implementation of a design may save iterations in the Application Specific Integrated Circuit (ASIC) design flow, and thus reduce time to market for a product.

The method for estimating power at the RTL called the top-down method was chosen for the implementation. Among other desired qualities, it does not require a gate-level representation of the design to produce a power estimate. This allows for power estimation to be done concurrently to simulations for functional verification of the RTL, before synthesis of the design.

The power modeling problem is divided into three tasks:

1. Extracting structural information from an elaborated SystemVerilog representation of the design.
2. Extracting information about available cells and their power consumption characteristics from the cell library.
3. Combining the structural representation with the cell- and power information retrieved, in order to create a power model.

In the implementation, the structure of the design is represented by a node tree, while a cell library object was created to represent available cells from the cell library and their power data. In order to produce a power model, the implementation takes sequences of generic cells from the structure tree and replace them with cells obtained from the cell library. The power model consists of several power-aware node trees. The power model representation is more similar to the gate-level netlist than the elaborated SystemVerilog representation. However, more work is needed to obtain a proper comparison between them.

The implementation shows promise for accurate and fast power estimation. Several ab-

stractions are done in the process so that fast estimations can be made, and their effect on the power consumption have been evaluated together with other alternatives. When creating the power-aware node tree, cells from the generic cell library are grouped to more complex cells from the cell library. This grouping ensures a reduction in the number of cells, which brings the model closer to the gate-level representation.

Some work remains to complete the power model; the most complex generic cells from the elaborated SystemVerilog file need to be constructed from several cells from the cell library. Complex cells with no equivalent yet are those representing arithmetic operations, shifters and comparators. When these cells have a representation, switching activity can be propagated through the structure trees in order to get a power consumption estimate for each of them. The final job of the power estimation tool is to solely use the activity data from the RTL simulation, together with the power values from each structure tree to yield the power estimate.

Sammendrag

I dette prosjektet implementeres et verktøy for å lage effektmodeller av RTL design. Den genererte effektmodellen er ment å brukes sammen med et effektestimeringsverktøy for å gi et tidlig, raskt og nøyaktig effektestimat. Nordic Semiconductor ASA utsteder dette masterprosjektet med motivasjonen å gjøre RTL simuleringer effektbevisste. Å oppdage power bugs tidlig i implementeringen av en design kan spare iterasjoner i ASIC designflyten, og dermed redusere tiden som kreves for å få et produkt på markedet.

Metoden for å estimere effekt på RTL kalt top-down metoden ble valgt for implementeringen. Blant andre ønskede kvaliteter krever det ikke en syntetisert nettliste-representasjon av designet for å produsere et effektestimat. Dette gjør at effektestimering kan gjøres samtidig med simuleringer for funksjonell verifisering av RTL, før syntesen av designet.

Effektmodelleringen er delt inn i tre deler:

1. Hente ut strukturell informasjon fra en prosessert SystemVerilog representasjon av designet.
2. Hente ut informasjon om tilgjengelige celler og deres effektforbruk fra cellebiblioteket.
3. Kombinere den strukturelle representasjonen med celle- og effektinformasjonen, og lage en effektmodell.

I implementasjonen er strukturen til et design representert av et nodetre, mens et cellebibliotekobjekt er laget for å representere tilgjengelige celler fra cellebiblioteket og effektforbruket deres. For å produsere en effektmodell tar implementasjonen sekvenser av generiske celler fra strukturtreet og erstatter dem med celler hentet fra cellebiblioteket. Effektmodellen består av flere effektbevisste nodetrær. Den effekt-bevisste representasjonen har mange likheter med den syntetiserte nettlisten. Dog, mer arbeid er nødvendig for å lage en god sammenligning mellom representasjonene.

Implementasjonen er lovende for nøyaktig og rask høy-nivås estimering av effektforbruk. Flere abstraksjoner blir gjort i prosessen slik at estimasjonen er rask. Hvordan abstraksjonene påvirker effektestimatet er evaluert sammen med andre alternativer. Når et effektbevisst nodetre lages, grupperes generiske celler til mer komplekse celler fra cellebiblioteket.

Denne grupperingen gjør at antall celler i representasjonen reduseres, noe som bringer modellen nærmere den syntetiserte nettlisten.

Noe arbeid gjenstår for å gjøre effektmodellen komplett; flere komplekse generiske cellene fra den prosesserte SystemVerilog-filen må settes sammen av tilgjengelige celler fra cellebiblioteket. Komplekse generiske celler som ennå ikke har noen ekvivalent effektbevisst representasjon er de som representerer aritmetiske operasjoner, skiftere og komparatorer. Når disse generiske cellene har en representasjon i effektmodellen, kan signaler propageres gjennom strukturtrærne, og et effektestimat lages for hvert nodetre. Jobben til effektestimeringsverktøyet som skal bruke effektmodellen er å kombinere aktivitetsdata fra en RTL-simulering med effektverdiene fra hvert nodetre i et vilkårlig design, og gi et effektestimat for designet.

Preface

This Master's Thesis concludes a five-year M.Sc. degree at the Norwegian University of Science and Technology (NTNU) at the programme Electronics Systems Design and Innovation, with a specialisation in Design of Digital Systems.

Preliminary research was done during the fall semester of 2019, which resulted in an unpublished literary review on Register Transfer Level power estimation. Methods discussed in this review are reconsidered, and one method is selected for the implementation of a power model.

The thesis is written in cooperation with Nordic Semiconductor ASA. They have contributed with the required Synopsys licenses, a workplace with a computer, and a wonderful supervisor Knut Austbø, who does not seem to mind late-night readthroughs of text with too few commas in it. I also had great supervision from Snorre Aunet from the Institute of Electronic Systems at NTNU. They both have my gratitude.

I would also like to thank my friends from my study programme for providing companionship and focus through video chat during long days of working from home. I am grateful for their help, and the opposite of grateful to the corona virus and the backache working from my kitchen table has given me.

Lastly, I wish to thank my mom, and if there is a Best Mom Award given by any reader of this thesis, I hereby nominate her.

Contents

Glossary

Dennard Scaling A MOSFET scaling law claiming the power density stays constant as transistors scale, thus making it possible to reduce power consumption by reducing the design size. This has held until recently, as leakage power is not negligible anymore with the smaller gate lengths in newer technology

fan-in is the reduction of signals caused by several signals being connected to a cell with fewer outputs than inputs. E.g. a 3-inputs AND gate has a fan-in of 3.

fan-out is the number of input gates that is driven by an output of a logic gate

JSON stands for JavaScript Object Notation and is a format for representing structured data.

Liberty is a widely adopted library format. The format is managed by the Liberty Technical Advisory Board, which is sponsored by Synopsys [1]

one-hot is form of signal encoding where only one bit of the signal can be high at a time

power bug is a fault with the design causing the power consumption to behave unexpectedly. It may cause the design to violate its power constraints.

SystemVerilog is a hardware description- and hardware verification language

VHDL is a hardware description- and hardware verification language

Acronyms

ASIC Application Specific Integrated Circuit

BDD Binary Decision Diagram

BN Boolean Network

CDFG Control flow Data Flow Graph

CMOS Complementary Metal-Oxide-Semiconductor

HDL Hardware Descriptive Language

I/O Input/Output

IC Integrated Circuit

IEEE Institute of Electrical and Electronics Engineers

LUT Lookup Table

RT-level Register Transfer Level

RTL Register Transfer Level

SV SystemVerilog

List of Tables

List of Figures

1 Introduction

1.1 Motivation

Power consumption is becoming increasingly important in Integrated Circuit (IC) design with the emergence of more and more battery-driven devices [5]. Transistor dimensions have continuously been shrinking to lower the power consumption of ICs. However, with the breakdown of Dennard Scaling [6] in the 2000s, leading to an increase in power density with smaller dimensions, downscaling has less effect on power consumption than it used to. Designers are now pushed to focus more on power consumption in their designs, and designing circuits for low power usage is becoming just as important as designing for high performance. The latter may be easier for designers, while many may lack the intuition to create circuits with low power in mind. To aid designers in this endeavour, tools for estimating the power consumption are essential.

Power estimation can be done at all design stages, until, in the end, it can be measured on the physical IC. The closer one is to the final implementation; the more accurate the power estimation typically can get. The less abstract the design representation is, the more one knows about parameters critical to power consumption. This is illustrated in Figure 1.1.

The system-level representation of a design is very abstract, and few aspects of the physical endproduct are known. The RTL representation is less abstract than the system-level representation, but still much remains unknown about the physical IC. The gate-level representation is closer to the endproduct than the two others, and many parameters relating to power consumption are determined at this level. The accuracy of power estimation will typically follow the trend of the graph; being more accurate the less abstract the design representation is.

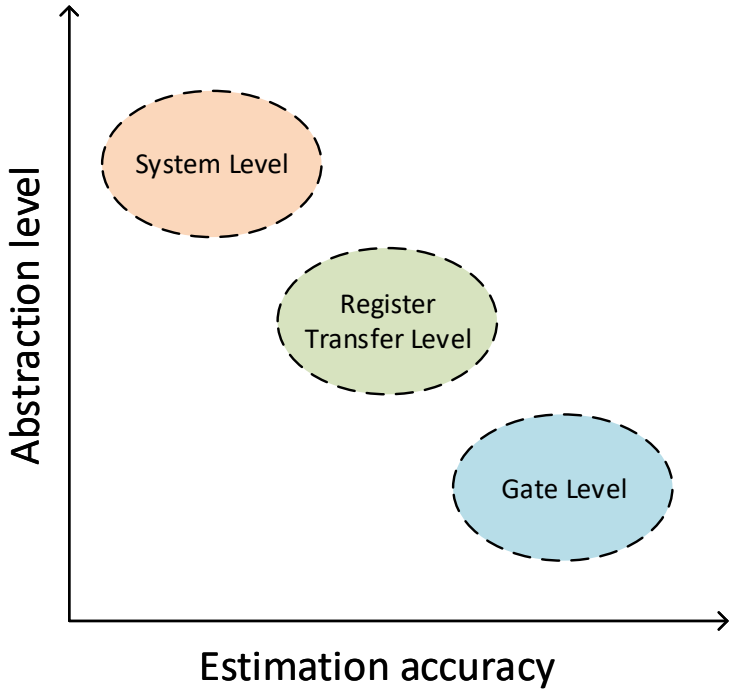


Figure 1.1: Graph relating design abstraction level and power estimation accuracy.

The ASIC design process is an iterative process, described in Section 2.2. Discovery of issues at a particular stage might bring one back to earlier stages in the design process, where more significant changes can be done. Each iteration is costly in development time and effort, and may increase the time to market for a product. Ideally, the design should be made with as few iterations as possible. Discovering and fixing power bugs already at the RTL is thus beneficial, possibly reducing the number of design iterations necessary.

There is a lack of suitable tools for estimating power at the RTL. They tend to be either too time consuming to run or too inaccurate to give assured results. Many also output an average power estimate with no granularity in time and space, which is needed if one is to use this estimate to deal with power bugs.

1.2 Problem description

This project aims to investigate and develop power models for use in power estimation at the RTL. A general method for making these models is found and it holds for all types of RTL designs. To make this model, information about the cell library used is necessary together with an RTL description of the design.

Nordic Semiconductor ASA requested this project, and their motivations are to be able to discover power bugs early and enhance their design flow by developing a power estimation tool able to yield power estimates corresponding to the RTL simulations. It is necessary to have a low spatial and temporal granularity in the model made, in order to discover power bugs.

In Chapter 4, different approaches to RTL power estimation are investigated, and an implementation using the top-down method is decided upon. An advantage of the top-down method is its ability to yield a power estimate before having a gate-level representation of the design. This way, the estimation method does not introduce extra iterations to the design flow described in Section 2.2.

The top-down method tends to be less accurate than other options for RTL power estimation. The approach presented in this thesis tries to atone for this by using the Liberty file to get accurate power information about the cells to be used in the design, and combine this cell information with elaborated Hardware Descriptive Language (HDL) structural information, which potentially yields a better structural representation than an unprocessed HDL representation. The HDL elaboration will be done using Synopsys HDL Compiler.

This project implementation is divided into three main tasks:

1. **Analysing the structure of a design**

   By using an elaborated HDL implementation, information about the structure of a design, necessary for estimating power, will be retrieved. This information could be the number of gates, number of registers, amount of combinatorial logic, how the signals are connected, and so on.

2. **Obtaining power characteristics from the technology library**

   Finding a means to retrieve information about the available cells and their power characteristics from the cell library. Power characteristics being the leakage power

and the dynamic power of the cells. It is also necessary to retrieve information about the cells in question to be able to relate their functionality to the power data.

3. **Creating a power model**

   Combining information about the structure retrieved in task **1** and information about the available cells retrieved in task **2** to make a power model of a design, representing all the signals and logic in the HDL representation.

The novelty of this power model is its generality and its use of the elaborated HDL and a cell library. The generality allows power models being made for any RTL as long as it can be elaborated by Synopsys HDL Compiler. The power models can be made with any cell library, their dimensions being irrelevant. The use of a cell library in the power model generation and a structural representation derived from elaborated SystemVerilog aims to achieve a high accuracy to future power estimations at the RTL.

1.3 Report structure

After this introduction this report consists of the following chapters:

2 Theory	In this chapter some relevant and useful theory for the project is presented.
3 Background	This chapter presents related work and gives an introduction to RTL power estimation.
5 Design tools and file formats	Here relevant design tools and file formats are presented.
6 Extracting design structure	Describes how the structural information is retrieved from the RTL representation.
7 Extracting library information	Describes how the power relevant information in the cell library is found, stored and used.
8 Generating a power model	This chapter combines the information retrieved in the two preceeding chapters to create a power model.

9 Conclusion Concludes the work done.

10 Future work Suggestions towards future work of improving the power model and applying it in top-down RTL power estimation.

2 Theory

2.1 Terminology

Some terminology that will be used in this report is shown below.

GATE"N"	Logic gates will be referred to in capital letters annotated with the number of inputs the gate has. For instance, a 2-input and gate will be written AND2. For more complex gates the numbers annotated refer to clusters of inputs, if this number is one it often skips the firs operator. For example ANDOR21 is a AND2 gate followed by an OR2 gate, where one of the OR2 inputs is the output of the AND2 gate. These more complex gates will be explained with logic functions or figures to make this clearer.
$*$	Logical AND operation
$+$	Logical OR operation
!	Logical NOT operation
cell	A building block in ASIC design. A transistor circuit encapsulated into a logic function, such as an AND gate. Could also describe building blocks with other purposes, but in this project this will not be visited. All available cells are gathered in a cell library.
generic gate/cell	A cell of the generic cell library used by Synopsys HDL Compiler when doing the design elaboration. In cases where the cell represents a logic gate, it may be referred to as a generic gate instead of a generic cell.

2.2 The ASIC design flow

The ASIC design flow is a mature design flow used in the making of Integrated Circuits. This flow allows one to, step-by-step, go from an abstract design description, towards the layout sent to a foundry for manufacturing the physical IC. The flow is iterative and may, at any point before the physical IC is produced, return to an earlier stage, where larger changes can be made [2].

In Figure 2.1 different ways to represent the circuit with decreasing abstraction is shown. The steps in the design flow is described below.

System Level — At this level the design is described as a set of functionalities, characteristics and constraints.

Algorithmic level — Here the design is described and verified on an algorithmic level, often using high level programming languages.

Register Transfer Level — This representation makes use of a hardware descriptive language, to describe the design as digital signals, logic operations and registers, and verified as such, for instance using SystemVerilog or VHDL.

Logic Level — The Register Transfer Level description of the design can be *synthesised* into a Logic level description. Here the description is mapped to the available logic cells in the cell library. The synthesis process also checks timing and area constraints so one knows whether these hold or not at this level.

Physical Layout — For the physical layout representation the cells from the logic level representation are placed and connected on a theoretical chip. Analog phenomenon, eg. wire capacitances, are taken into concern in an attempt to model the physical IC.

IC — Here the endproduct of the process is made in a foundry. based on a GDSII file from the physical layout, which contains all information necessary to produce the IC.

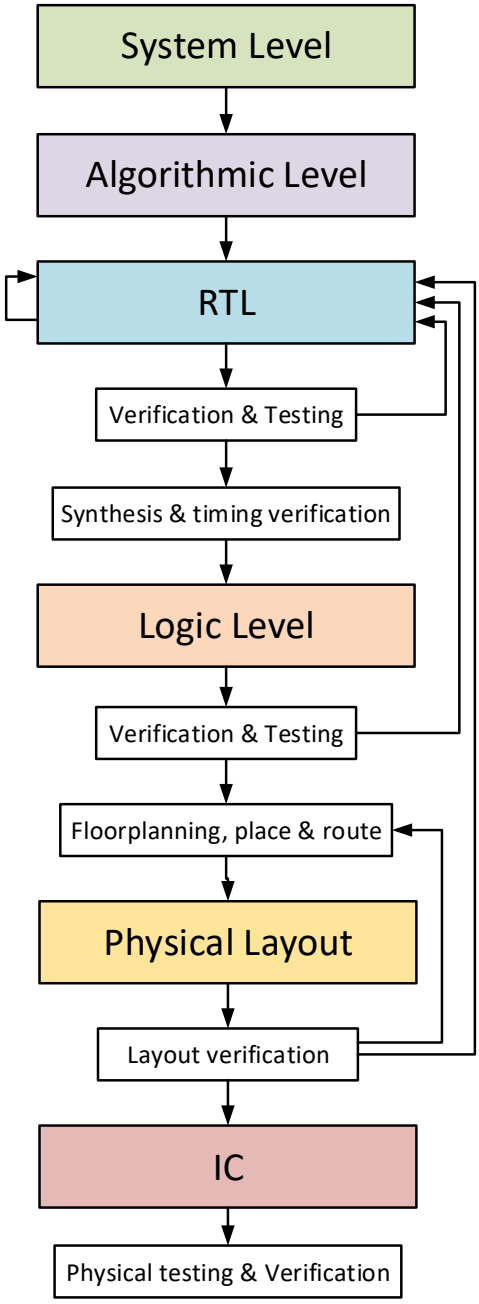


Figure 2.1: Illustration of the iterative ASIC design flow, [2]

2.3 CMOS power consumption

The power consumption in digital CMOS-based circuits can be divided into dynamic and static power consumption. The dynamic power consumption is caused by the switching activity in the system, while the static power consumption is caused by leakage in the CMOS transistors [4]. The total power consumption of the system is the sum of these two as is given by Equation (2.1).

$$P_{total} = P_{dynamic} + P_{static} \tag{2.1}$$

2.3.1 Dynamic power consumption

The dynamic power consumption can be divided into switching power and short-circuit power [4].

The main contributor to the dynamic power consumption is the switching power, which is the power it takes to charge and discharge the output capacitance of a logic gate. It can be calculated as shown in Equation (2.2), where α is an activity factor describing how often the output switches (changes value). C_L is the load capacitance on the gate output, V_{dd} is the supply voltage and f_{clock} is the clock frequency.

$$P_{SW} = \frac{\alpha}{2} C_L V_{dd}^2 f_{clock} \tag{2.2}$$

Another contributor to dynamic power consumption is the short-circuit current. When CMOS logic is in the middle of switching both the NMOS and the PMOS transistor will be partially open, allowing some current to flow from V_{dd} to ground. This is illustrated in Figure 2.2.

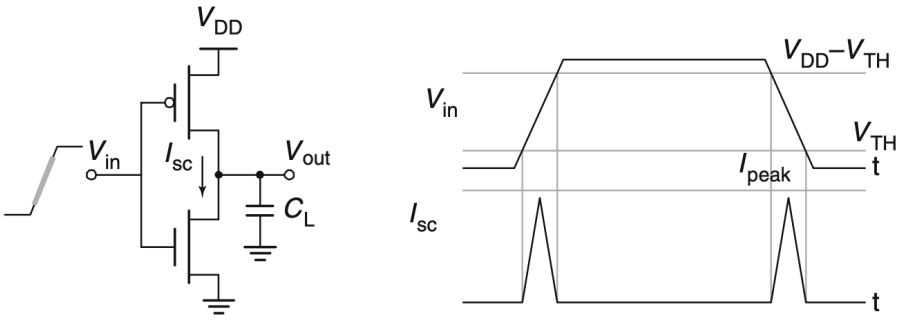


Figure 2.2: Illustration of the short-circuit power in CMOS logic [3]. When V_{IN} rises and falls I_{SC} will flow from V_{DD} to ground for a short period of time.

The short-circuits contribution to power consumption can be calculated using the expression shown in Equation (2.3), where t_{sc} is the duration of the short circuit current, V_{dd} is the supply voltage of the system, I_{sc} is the average short-circuit current and f_{clock} is the clock frequency.

$$P_{SC} = t_{sc} V_{dd} I_{sc} f_{clock} \tag{2.3}$$

The total dynamic power consumed in the circuit will be the sum of the switching power and the short-circuit power consumed by all the transistors in a design, shown in Equation (2.4). N is the number of transistors, P_{SW_t} is the switching power- and P_{SC_t} is the short-circuit power of transistor t.

$$P_{dynamic} = \sum_{t=0}^{N} \left(P_{SW\_t} + P_{SC\_t} \right) \tag{2.4}$$

Dynamic power consumption from a logic circuit perspective

A different way of viewing switching power consumption, more suited for logic designs, is gotten from dividing the power consumption in two contributions: The contribution from switching of nets in a design, the **switching power**, and the contribution from the switching of internal signals in a logic cell, which also includes the short-circuit contribution, **internal power**. The total dynamic power of a design can thus be seen as a sum of the **switching power**, P_{SW}, of all nets, and the **internal power**, P_{IN}, of all cells. This is shown in Equation (2.5).

$$P_{dynamic} = P_{SW} + P_{IN} \tag{2.5}$$

2.3.2 Static power consumption

The static power consumption in the CMOS transistors is caused by leakage current. This leakage has traditionally been negligible compared to the switching power, but the down-scaling of the technologies and the lower supply voltages, which in turn has lead to lower threshold Voltage, V_t. Nowadays the static power consumption of transistors, is just as significant as the switching power. Contributions to the leakage current come from the sub-threshold leakage, the gate leakage and the junction leakage [4].

- Sub-threshold leakage is current leaking from source to drain while the transistor is operating in the weak inversion region ($V_G < V_t$). It increases exponentially when lowering V_t [4] and is the largest contributor to the static power consumption.
- Gate leakage is current caused by electrons tunnelling through the oxide layer of the gate.
- Junction leakage is caused by potential differences between the drain diffusion region and the substrate. It is often negligible compared to the other two contributors.

2.4 Process, Voltage and Temperature corners

Variations in the manufacturing and the environment will lead to significant changes in the characteristics of a transistor. These changes may cause the IC behaviour to vary. To make a circuit operate as expected these variations should be taken into account. The sources of these variation are **process variation**, **supply voltage** and **temperature** [4].

The **process variation** is caused by slight variations in the manufacturing process, like the concentration of dopants or the oxide thickness.

These variations lead to manufactured transistors having varied characteristics. These are described as; F (fast) and S (slow) for the corner-cases, where the transistor will operate faster and slower than expected, and T (typical) describing an average transistor. For a CMOS transistor, consisting of one PMOS and one NMOS transistor this yields four operating corner-cases describing a constricted area, in which the pair og transistors will always operate within. FF, SS, SF and FS. The center of this area is (TT), the average transistor. This is illustrated in Figure 2.3.

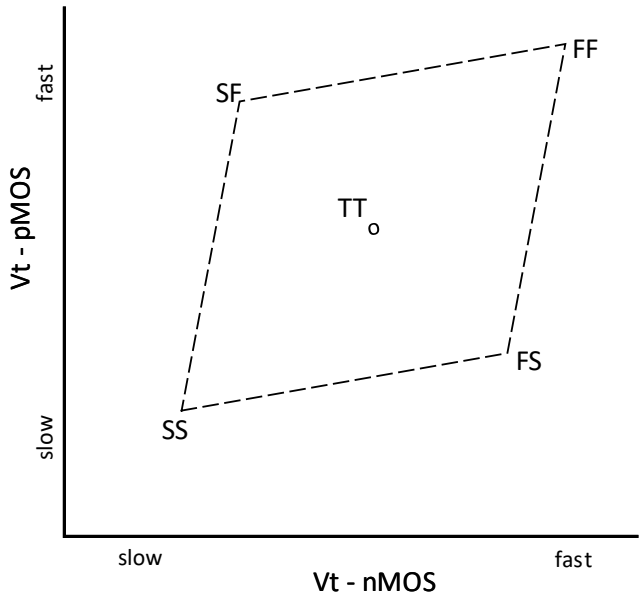


Figure 2.3: CMOS design corners [4]

The variation in **temperature** also affects the transistor's operation significantly as it lowers the threshold voltage. If the operating temperature is high the transistor will have a higher leakage, which increase its power consumption.

Lastly, the **supply voltage** can deviate from the intended value for many reasons, such as the tolerance of the voltage regulators and noise.

Thus, it is not enough to only take an average transistor in the TT corner, operating in room temperature with the intended supply voltage into account. One also needs to consider the transistor in its slow corner, operating on a high temperature with a low voltage, and all other corner-cases.

3 Background

As demands to power consumption rise, the size of battery-driven devices sink and smaller transistor dimensions lead to higher power density, the need for power estimation tools rise. The designers wish to optimise the ASIC design flow and minimise the time to market, while still keeping up with state of the art power demands. Power can be estimated at all stages in the design process described in Section 2.2.

At the system level, accurate power estimation tools are few, but maybe not for much longer. In 2019 Institute of Electrical and Electronics Engineers (IEEE) released a new standard for power modeling at the system level [7]. There has not been a standard way of representing power data at the system level before. The organisation suggests the lack of such a standard could be why the industry is still inadequate in this field.

At the functional level, there power estimation tools exist, but they are mainly meant to speed up the simulations. One can argue that there are two main reasons one can wish to estimate power at a high level;

1. One wants to get an approximate indication of the power consumption at this level before lower-level representations are made.

2. Simulations at this level is faster than low-level simulation

In 2. one returns to a more abstract design representation to run faster simulations. Increasing the simulation speed is the primary motivation for the functional level power estimation tools. They are based on already existing gate-level representations.
Zhong et al. [8] try to estimate power at the functional level using some RTL power models derived from a gate-level representation of the design. Here a cycle-accurate functional description is merely an abstraction of the known RTL, in order to to speed up the RTL power estimation by going up an abstraction level for the simulation. In another paper, Zhong et al. [9] further improve their solution. Lee and Gerstlauer [10] annotates a functional model of a design with constructs allowing the capturing of activity. Using machine learning, power models can be synthesised from this functional model. An advantage of this method is that it allows for high-speed simulations. However, the functional model requires an existing gate-level representation of the design to train the power model.

The methods for estimating power at the RTL can be divided into two main methods of implementation. They will be referred to as bottom-up and top-down methods. The bottom-up method starts with a less abstract representation of the design, such as the gate-level representation, and tries to relate power estimates done at this level to factors that are also known at the RTL [11], [12], [13]. The estimation method then returns to the RTL representation of the design and does power estimation on different scenarios there. The top-down method, on the other hand starts at the abstract RTL and tries to estimate lower-level information about the design in order to estimate power directly [14], [15], [16].

3.1 Bottom-up power estimation

In Figure 3.1 a typical estimation flow of bottom-up power estimation can be seen. The available input data is a gate-level netlist with corresponding simulation data. However, this requires that synthesis and layout with the desired technology library have been performed. A power estimation tool is then run on the gate level representation with a broad set of activity data. This results in a set of power estimates and simulation data that can be used to characterise the design, often relating the Input/Output (I/O)-switching to the power consumption. To get a power estimate, characterisation variables or a Lookup Table (LUT) are then fed to a general power model at the RTL, together with the simulation data of the scenario from which one wishes to estimate power.

Ravi et al. [11] makes an extensive set of macromodels from RTL components. These models are then translated into simulatable power model libraries. The creation of new designs then solely make use of these components for which power estimates are available.

Gupta et al. [12] made a macromodel relating gate-level power estimates to the hamming distance between consecutive input vectors. A complicated characterisation stage is necessary to exploit this relation.

Mehta et al. [13] also takes basis in making a macromodel for every possible RTL component. A clustering algorithm is used to group input vectors leading to similar power consumption in the circuit. These groups are then placed in a LUT. This clustering makes their model faster, as there are fewer values to look up.

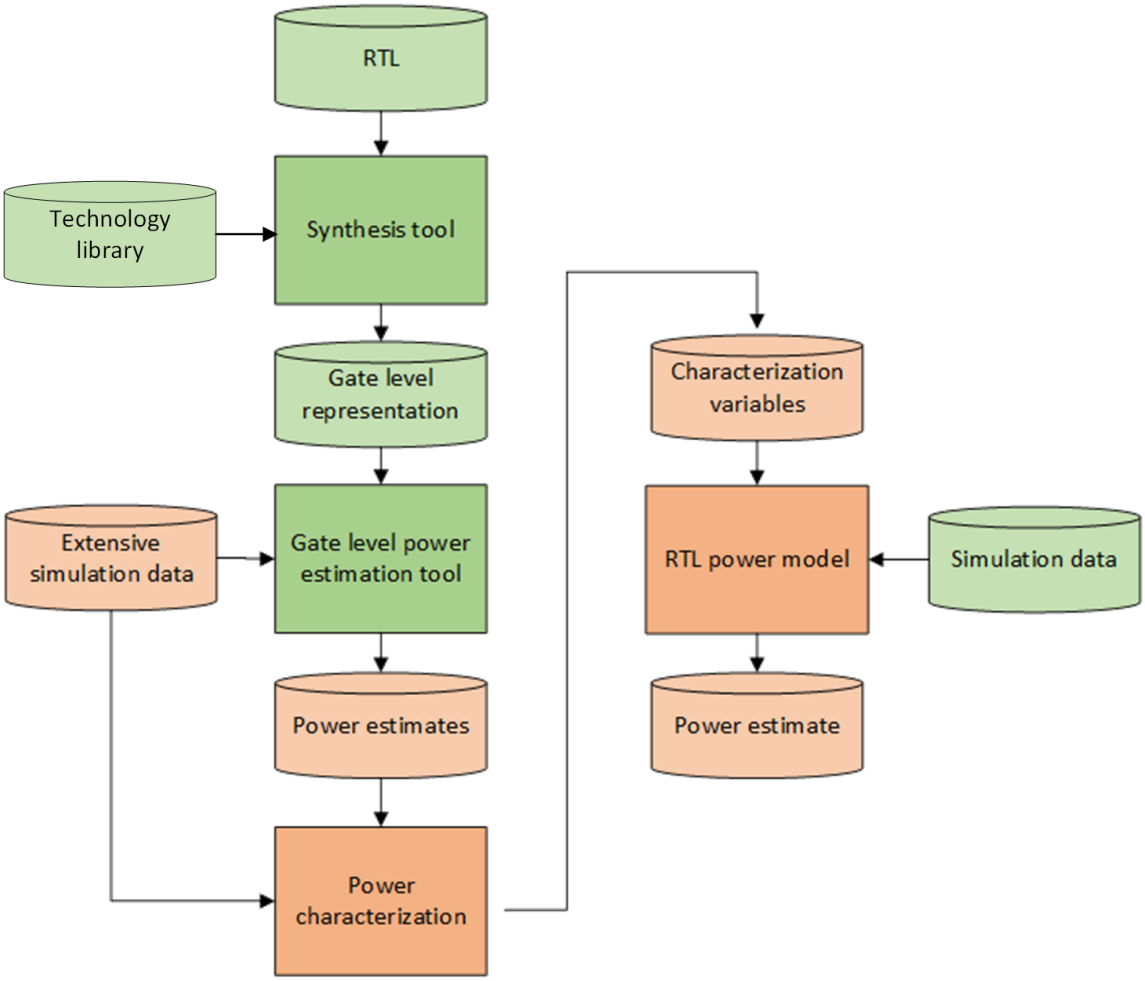


Figure 3.1: The estimation flow in the case of bottom-up power estimation

3.2 Top-down power estimation

The estimation flow of a top-down power estimation approach can be seen in Figure 3.2. The method needs to take in information about the structure, readily available at the RTL, for instance, a HDL description. It also needs to take the cell library into account. The cell library can be considered by, for example, knowing the power characteristics of a standard gate from the cell library, or by processing the entire cell library as an input. It could also be possible to do some characterisation. If one, for instance, has a design that will be synthesised with strict timing constraints, this will increase its power consumption compared to a design with less strict timing constraints.

Zafalon et al. [14] have developed both a top-down and a bottom-up technique for power

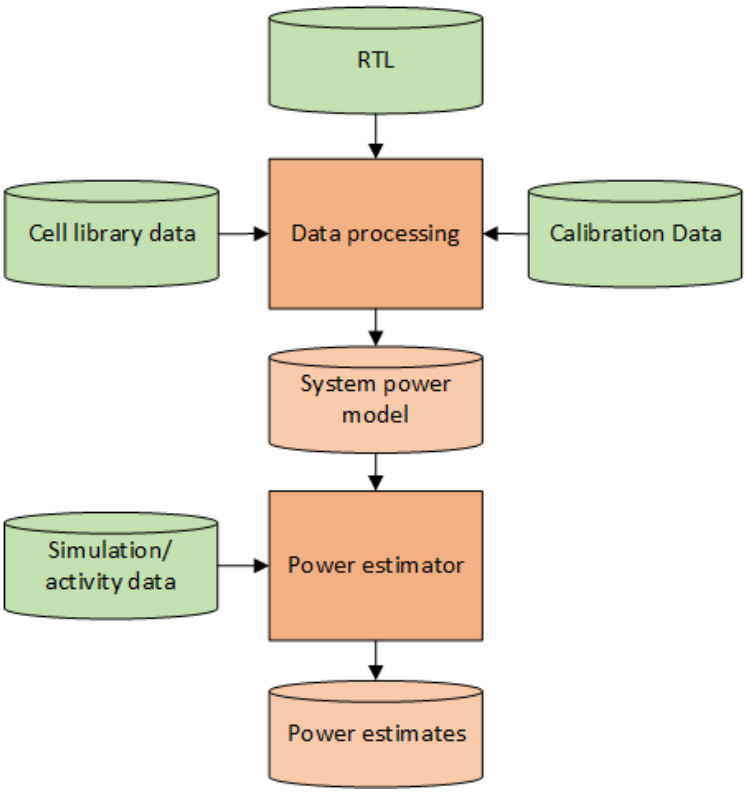


Figure 3.2: A top-down estimation flow.

estimation. Their top-down approach is based on using a Binary Decision Diagram (BDD) to represent the circuit. Representing a design as a BDD is the same as making the design using only 2-to-1 multiplexers. This design is then optimised to some degree decided by the user, and the power estimate is tuned to the target technology. The user decides whether the actual synthesis will focus on power, timing or area and the model is also tuned based on that input.

Buyuksahin and Najm [15] make use of a Boolean Network (BN), a directed acrylic graph where each node is a boolean function, and its edges represent the connection between nodes. They use this network to estimate the gate count of the design, which yields an estimate of the circuit's total capacitance.

Sambamurthy et al. [16] use a Control flow Data Flow Graph (CDFG) to represent the circuit. This graph allows for modeling both the data operations done and conditionals. The number of stages necessary to implement a function is then estimated from the maximum input number of gates in the target technology and the function's size to be computed. The probability of switching at each node is then estimated from input switching from simulation

or the input switching probabilities and the likelihood of that switching propagating all the way to the logic depth of the function. The method of Logic Effort is used to make a capacitance estimate. All of the above is then combined into a power estimate.

3.2.1 Fast synthesis power estimation

Several vendors provide tools for estimating power at the RTL. To mention a few; Ansys has PowerArtist [17], Synopsys has Spyglass Power [18], Mentor Graphics has PowerPro [19] and Cadence has Joules RTL Power Solution [20]. These are typically based on some variant of fast synthesis power estimation, mapping the RTL description to cells in a cell library and estimating the power consumption based on these cells. This method is applied by vendors already providing synthesis tools to provide a power estimation tool faster than gate-level estimation.

The power estimation flow of such tools is shown in Figure 3.3. The figure is simplified as the internal synthesis, and power estimation flow is undisclosed information private to the tool vendors. It is based on a synthesis tool that omits information not crucial to power estimation in order to speed up the synthesis. After the fast synthesis, an estimation tool will be used to estimate power. It gets its parameters from the "synthesised" design, activity data and possibly calibration data. As these methods bring the design closer to a gate-level representation, they allow for accurate power estimates but introduces a synthesis process which, though it is faster than a regular synthesis, may still be slow.

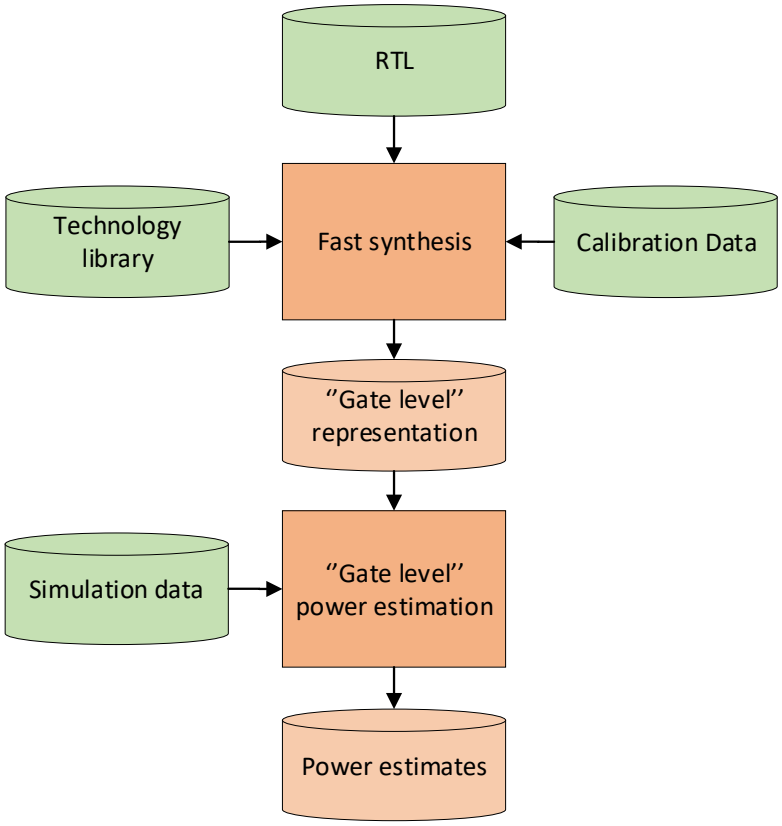


Figure 3.3: The estimation flow in the case of fast synthesis estimation

3.3 Prestudy

This thesis is written in collaboration with Nordic Semiconductor ASA. An unpublished literary review has been conducted on RTL power estimation to find a method suiting their motivations, which can be summarised as:

- Wanting to make RTL simulation power-aware.
- Being able to use this power awareness to detect power bugs.

The prestudy can be found on GitHub [21]. The following is a quick outline of the main differences between the top-down and bottom-up estimation flows and a summary of the prestudy conclusion.

The bottom-up methods have their foundation at the gate level and thus tend to have a

more accurate power estimation due to more information about the design being available as the power model is made. The challenge of bottom-up power estimation is to get the power estimates to correlate well with the input and output switching statistics of the design so that the model can be used at the RTL. The top-down methods tend to be less accurate, but lack the time-consuming characterisation stage of the bottom-up methods, making them faster for new designs and possibly more suited for design exploration if they take the internals of the RTL description into account.

To make the RTL simulation power-aware a power estimation tool for the RTL is needed. It is a significant advantage if this model is available before the design has been synthesised. Otherwise, it will introduce an extra iteration into the design flow, which may be avoided using the top-down method.

To detect power bugs with this power estimation, it needs time/cycle awareness. It could either work for smaller time-frames or do estimation cycle-by-cycle in the simulation. The latter is preferable. In addition to this temporal granularity, the tool should also have some spatial granularity. When running simulations on larger modules and observing unexpected power behaviour, it is an advantage to see where this behaviour occurs.

If the desire had been to increase simulation speeds when running power scenarios, then going from a gate-level representation to a RTL representation makes sense. Otherwise, this introduces an extra iteration to the design flow, which may be avoided using the top-down method.

If a top-down estimation approach does not provide enough accuracy, it could be supported by bottom-up models for existing design blocks to increase the estimation accuracy.

4 Suggesting a solution

The top-down method has been chosen for implementation due to its desirable estimation flow. The top-down flow is simple and starts at the RTL and makes a power estimate directly. For a bottom-up flow, on the other hand, a gate level representation of the design is needed **before** a RTL power estimate can be made. Using the top-down method a design can be changed or discarded because of power concerns early in the design flow, without ever needing to be synthesised, if the power estimates are accurate enough. With the top-down method it is possible to verify the power behaviour concurrently to the functional verification of the RTL.

The suggested estimation flow can be seen in Figure 4.1. Here the RTL representation of a design and data from the cell library is retrieved and processed separately, to later be combined into a power model. The power model is used together with activity data by a power estimation tool to yield a power estimate. Already existing data and tools are highlighted in green, while the parts highlighted in orange would have to be implemented.

It is necessary to implement a system processing the structural information found in the RTL representation, and another system processing power information related to the cell library. Then, the retrieved information from both systems can be combined into a power model, which will serve as an input to a power estimator together with simulation- or activity data.

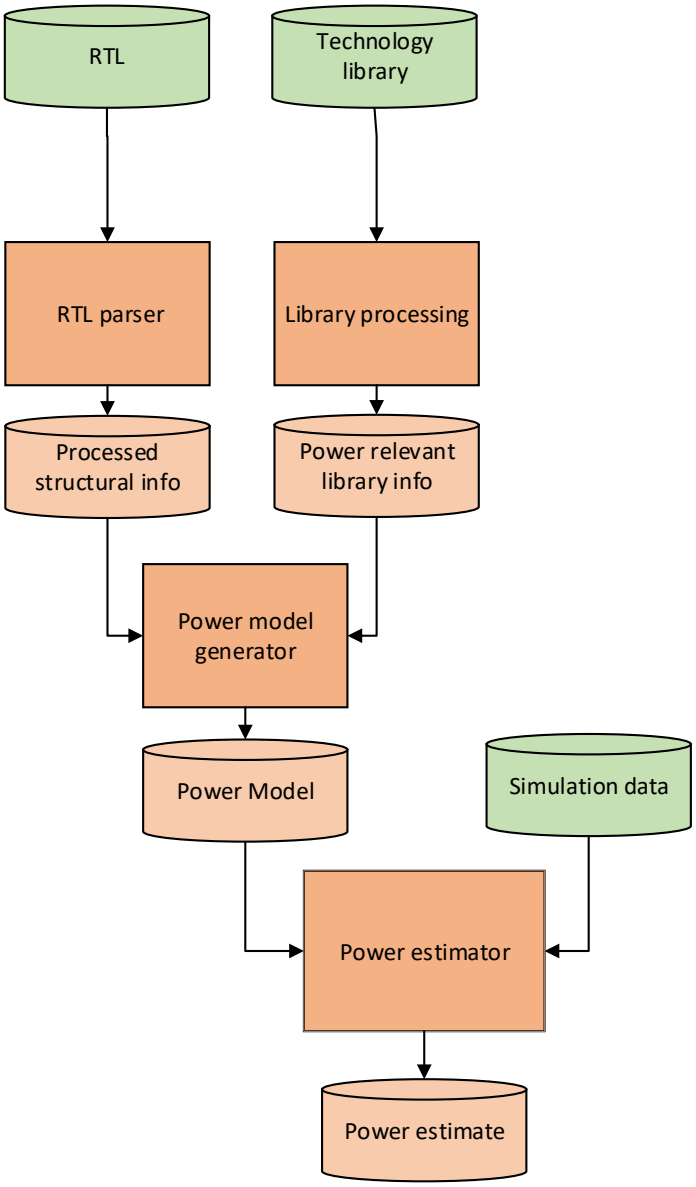


Figure 4.1: The intended estimation flow of the top-down power estimation. The blocks highlighted in green are already existing, while the orange ones have to be implemented to make the top down power estimator.

4.1 Structural information

The *processed structural info*-block in Figure 4.1 should contain information about which *operations* are done on which *signals* and how they are all connected. Later, in the *Power model generator*-block this will be related to power information. The structural information should also allow for some estimation of activity in the structure, depending on observable, (input, output and/or register), switching activity. It is also important that the structure remains relatable to the RTL it represents.

Most synthesis tools have an *elaboration* stage where they retrieve structural information from the RTL as a pre-processing stage for the *synthesis*. This is done by breaking down coding constructs and compiler directives and mapping the code to cells from a *generic* library. This library does not correspond to any physical library and the *generic cells* represent logic- and arithmetic functions on the signals only. With this representation as a foundation the *operations* are the generic cells in the elaborated netlist and the *signals* are their connections.

Using the elaborated structural information, rather than unprocessed RTL, brings one a bit closer to the gate level representation of the design and possibly towards more accurate power estimates. It is not desirable to go all the way to a gate level representation as the synthesis process is time consuming, especially for larger designs. It is interesting to see what kind of power model can be developed with this elaborated design as a starting point rather than the RTL it is elaborated from or the netlist it is synthesised into. Detailed information about the elaborated SystemVerilog format can be seen in Section 5.1

4.2 Cell library information

A common approach in high level power estimation is to abstract away the cell library by using a general gate representing all the gates in the design instead of differentiating between gates. Such a cell is commonly a NAND2 cell with the correct gate length and power characteristics corresponding to the cell library. This project attempt to lay the foundation of accurate RTL power estimation and thus want differentiate between the cells in the design to some extent. Knowing what cells are where and what they are affected by will possibly improve the accuracy of temporal and spatial power estimates even if the average power estimate remains the same. Finding out *what* cells are available and what power consumption these cells have will be the job of the *Library Processing*-block in Figure 4.1. The library power information is commonly stored in a Liberty file. Liberty is

a standard format for representing timing and power characteristics of a cell library. More information on the format is found in Section 5.2.

Nordic Semiconductor ASA has a Liberty parser that can retrieve information from the Liberty file, but further processing is necessary to structure and select the information necessary to do power estimates, which is information relating to the static and dynamic power consumption of the cells. Synopsys has a HDL compiling tool doing design elaboration, but it will be necessary to retrieve structural information from the elaborated SystemVerilog file. Lastly this project will combine the structural information from the elaborated SystemVerilog and the power-focused information from the cell library into a power-aware representation of the design, a *power model*, which can in turn be used for power estimation.

4.3 The power modeling flow

Figure 4.2 is a refined version of Figure 4.1. It goes more into detail on the estimation flow adding the Liberty parser and the elaborated SystemVerilog. The highlighted blocks are those involved in developing a power model, and thus the scope of this project. The power model will combine the structural information retrieved from the elaborated SystemVerilog and the power- and cell information retrieved from the Liberty file.

The elaborated file can be made using Synopsys HDL Compiler. A tool part of the Synopsys synthesis flow shown in Figure 4.3. In their flow the HDL is first compiled into an elaborated SystemVerilog netlist. The elaborated design is then fed to Synopsys DesignCompiler together with the Liberty file to yield the gate level netlist.

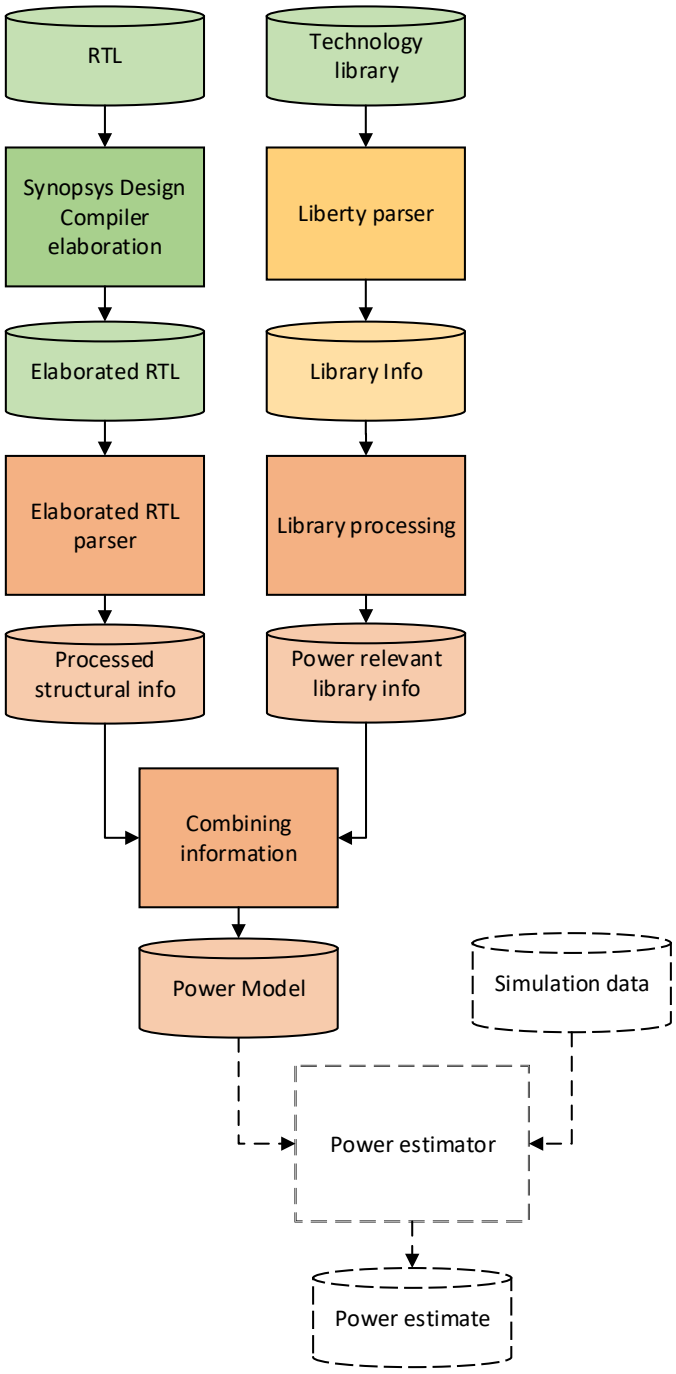


Figure 4.2: A refined flow for the top-down power estimation. The already existing Liberty parser is highlighted in yellow. The part of the flow that is out of scope is drawn in dotted lines.

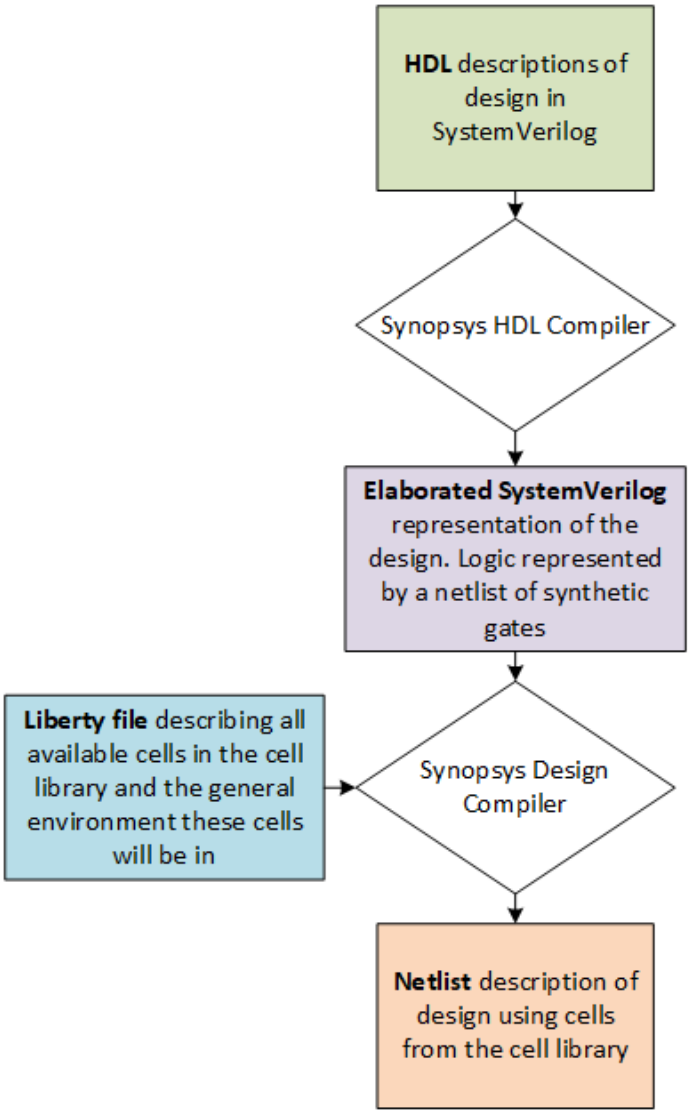


Figure 4.3: An overview of the synthesis process from RTL to netlist

5 Design tools and file formats

5.1 Design elaboration

When synthesising a design, the constructs in the RTL are mapped to cells in the cell library, creating a hardware design with equivalent functionality as the one described in the RTL. This representation is called a netlist. Most synthesis tools do this by going through an elaboration stage. Here the constructs in the RTL are first optimised and mapped to cells from a *generic* cell library. A *generic* cell library is a library with functional cells not corresponding to physical ones. They do not have any power- or timing data. The elaboration also goes through the compiler directives, which are direct instructions on how to process the HDL, such as **'ifdefs**. The results of the elaboration stage is an intermediate file, similar to the netlist, using cells from a generic library, rather than cells from the library used in synthesis.

To complete the synthesis process, the elaborated file is optimised further and mapped to the cells in the cell library. In this project, Synopsys HDL Compiler has been used to get an elaborated representation of the design. In Table 5.1, a simplified list of these elaboration constructs made by this synthesis tool can be seen. The *module*, *input*, *output* and *assign* constructs are the same as in the RTL file. The *wire* represents all connections between objects. The rest of the objects have replaced the more complex RTL with simple, generic gates like an AND2 gate. The elaborated netlist is not technology-specific and, thus, does not contain any power information.

Table 5.1: Elaborated cells

Construct	**generic cells**	**Description**
Module	module	A SystemVerilog module declaration or instantiation
Input	input	An input port of variable bitwidth
Output	output	An output port of variable bitwidth
Wire	wire	A wire of variable bitwidth

Assign	assign	Assigning one wire to another wire or a constant
Multiplexer	MUX_OP	A multiplexer with variable datawith and select signal width
Register	SEQGEN	A one bit register
AND2	GTECH_AND2	A two-input AND gate
OR2	GTECH_OR2	A two-input OR gate
XOR2	GTECH_XOR2	A two-input XOR gate
Select	SELECT_OP	This sends one of several data signals out, depending on a control signal
Adder	ADD_UNS_OP, ADD_UNS_CI_OP, ADD_TC_OP, ADD_TC_CI_OP	Adder with inputs and outputs of variable width
Subtractor	SUB_UNS_OP, SUB_UNS_CI_OP, SUB_TC_OP, SUB_TC_CI_OP	Subtractor with inputs and outputs of variable width
Shift	ASH_UNS_UNS_OP, ASH_UNS_TC_OP, ASH_TC_UNS_OP, ASH_TC_TC_OP, ASHR_UNS_UNS_OP, ASHR_UNS_TC_OP, ASHR_TC_UNS_OP, ASHR_TC_TC_OP, SRA_UNS_OP, SRA_TC_OP	Shifting a signal in a certain direction, possible to take the sign into account
Barrel shift	BSH_UNS_OP, BSH_TC_OP, BSHL_TC_OP, BSHR_UNS_OP, BSHR_TC_OP	Shifting, rolling the bit shifted out to the opposite side of the signal instead of shifting in zeros or ones

Shift-and-add	SLA_UNS_OP, SLA_TC_OP	Shift signal before adding
Multipliers	MULT_UNS_OP, MULT_TC_OP	Multiply two signals and output the result
Division	DIV_UNS_OP, MOD_UNS_OP, REM_UNS_OP, DIVREM_UNS_OP, DIVMOD_UNS_OP, DIV_TC_OP, MOD_TC_OP, REM_TC_OP, DIVREM_TC_OP, DIVMOD_TC_OP	Divide a signal by another and output the result
Comparators	LT_UNS_OP, LT_TC_OP, GT_UNS_OP, GT_TC_OP, LEQ_UNS_OP, LEQ_TC_OP, GEQ_UNS_OP, GEQ_TC_OP, EQ_UNS_OP, NE_UNS_OP, EQ_TC_OP, NE_TC_OP	Compare two signals of variable width
Not	GTECH_NOT	An single bit inverter
Buffer	GTECH_BUF	An single bit buffer

Many of the more complex generic cells are grouped into the same constructs. These cells differ depending on the representation of their input signal representations but are otherwise similar in functionality. The Synopsys elaboration differs between *unsigned*, UNS, and *twos' complement*, TC, representations. In the table, cells with different signal representations but otherwise the same functionality is put in the same group.

Many comparators have been grouped together into one. It can be argued that their power consumption is quite similar, as the logic needed to implement them are the same. However, the larger-than and smaller-than comparisons introduce more complexity than the equal and not equal, so it can also be an option to divide the comparators into two (or more) groups.

Table 5.2 shows the elaborated cell groups sorted by functionality.

Table 5.2: Groups of elaborated SystemVerilog constructs

Group	**Construct**
Connects	inputs outputs wire
Buffers	Buffer
Multiplexer	Multiplexer
Register	Register
Logic operators	AND2 OR2 XOR2 Comparator Not Shifter Barrel shift
Arithmetic operators	Adder Subtractor Shift-and-add Multiplier Divisor

The generic *select* cell is unique as it does not have a cell equivalent in any cell library.

An *if* or *case* statement is elaborated into a generic *select* cell by Synopsys HDL compiler unless it is specified in the HDL representation that one wants it to be inferred as a multiplexer. The *select* statement is then synthesised by Synopsys DesignCompiler into either logic or a multiplexer depending on the available cells and unknown DesignCompiler conditions.

5.2 Liberty file format

The cell library's timing and power characteristics are found in a Liberty file. The Liberty file format is an industry standard used to describe cells of a particular technology. Information regarding timing, power, area, functionality and operating conditions of cells in the cell library can be found in this file.

The Liberty file consists of three types of statements:

- **Group statements**
  A collection of statements grouped together. In a library, the uppermost group is a *library* group, and no other such groups can be made in a Liberty file. A group internal to the library can, for example, be a *cell* group, and a group internal to the cell can be a *pin* group.
- **Attribute statements**
  A statement used to describe the characteristics of objects (groups) in the library. Such attributes can, for instance, be the size of a cell or the unit of leakage current in the library.
- **Define statements**
  Used to define new attributes. Which kind of group they are meant to describe is also specified.

Values are often specified without units, and the units of different values are described at a higher level, as library attributes.

5.2.1 Power characteristics

The same cell library is characterised in the different design corners described in Section 2.4, resulting in different Liberty files for different process conditions.

In Chapter 2, it was described how power consumption could be divided into dynamic and

static power consumption. Here, the Liberty groups and attributes relating to the two types of power consumption will be investigated.

Static power

The liberty **cell** group has a sub-group called **leakage_power**. In this group, a leakage power value is given. The group also has an optional **when** attribute and a **related_pg_pin** attribute, set to the supply pin of the cell. The **when** attribute describes the different states of the input pins. For instance, if the cell is a two-input **AND** gate, the attribute would be one of the 4 possible input cases; $A1$ & $A2$, $A1$ & $!A2$, $!A1$ & $A2$ or $!A1$ & $!A2$. If the **when** attribute is not given, the average leakage power is the one given. Depending on how accurate data one wants, one can choose to retrieve the average leakage power or all the state-specific leakage power values from the liberty file. In Listing 5.1, an example of the **leakage_power** group can be seen.

Listing 5.1: leakage_power group example

```
1  leakage_power () {
2     value              : 93.1982;
3     when               : "A1 & A2";
4     related_pg_pin     : "VDD";
5   }
```

The unit of values of different groups are described at a higher level in the library.

Dynamic power

For a logic design, one can say that there are two contributions to the switching power:

- **Switching power**
  The charging and discharging of the output load capacitance, which is determined by the input pins the output is connected to.
- **Internal power**
  The internal switching of transistors within the cell, both as a result of an input transition leading to an output transition and an input transition only causing some transistors *within* the cell to switch.

Both of these contributions are found in the **internal_power** group in the Liberty file. This group is a sub-group of the **pin** group, which in turn is part of a **cell** group. The

internal_power group of an input pin will describe the internal power consumption of the cell, while the **internal_power** group of an output pin will describe the switching power.

Listing 5.2: pin group examples

```
    pin (Z) {
        direction           : "output";
        related_power_pin   : "VDD";
        related_ground_pin  : "VSS";
        power_down_function : "(!VDD) + (VSS)";
        function            : "A1*A2";
        max_capacitance     : 0.078;
        timing () {...}
        timing () {...}
        internal_power() {
            related_pin     : "A1";
            when            : "A2";
            related_pg_pin  : "VDD";
            rise_power (lookup_table_template) {
                // lookup table data
            }
            fall_power (lookup_table_template) {
                // lookup table data
            }
    }
        internal_power() {...}
    pin (A1) {
        direction           : "input";
        related_power_pin   : "VDD";
        related_ground_pin  : "VSS";
        max_transition      : 10;
        capacitance         : 0.0268;
        rise_capacitance    : 0.0045;
        rise_capacitance_range (0.0071, 0.0089);
        fall_capacitance    : 0.0067;
        fall_capacitance_range (0.0032, 0.0045);
        receiver_capacitance () {...}
        internal_power() {...}
    }
```

5.2.2 Power related library attributes and groups

In Table 5.3 some groups and attributes related to power consumption at a library level can be seen.

Table 5.3: Library group and attribute overview

Group/attribute	**Description**
voltage_unit	the voltage unit used for the cell library voltage values
capacitive_load_unit	The unit for capacitive loads in the cell library
library_features (group)	
default_cell_leakage_power	default value for cell leakage power if cell lacks this group, if not specified it is zero
lu_table_template (group)	Describes buildup of a lookup table that can be filled with characterisation values
cell (group)	See Section 5.2.3

5.2.3 Cell attributes and groups

In Table 5.4 some groups and attributes related to the power consumption on a cell level can be seen. The cell in itself is a group in the library.

Table 5.4: Power related cell groups and attributes overview

Group/attribute	**Description**
footprint	used to relate cells with same functionality
area	the area of the cell
leakage_power (group)	
value	the leakage power value
when	pin logic values for value to be valid
related_pg_pin	related supply voltage pin
pin (group)	See Section 5.2.4

5.2.4 Pin attributes and groups

Important groups and attributes of the pin group are shown in Table 5.5. The pin group itself is a group in a cell.

Table 5.5: Pin power related groups and attributes overview

Group/attribute	**Description**
direction	Whether pin is input or output pin
related_power_pin	What is the power pin relative to this pin
related_ground_pin	What is the ground pin relative to this pin
capacitance (input pin)	Capacitance of pin
function (output pin)	Boolean function describing pin function
max_capacitance (output pin)	Maximum capacitance the pin can drive
internal_power (group)	
related_input	input relating to this group instantiation
when	conditions of other related pins
related_pg_pin	the related power-ground pin
rise_power (group)	the power consumption if related_input rises (a LUT)
fall_power (group)	the power consumption if related_input falls (a LUT)

5.3 Test files and modules

5.3.1 Test modules

The system will be tested on several of Nordic Semiconductors designs. Here they are listed together with a short description of their functionality:

Module1	An activity monitor
Module2	A memory management module
Module3	A queue module
Module4	A data management module
Module5	A filter module

5.3.2 Calibration netlist

The system will in Chapter 7 make use of a *calibration netlist*. This file is the netlist of a full chip made by Nordic Semiconductor ASA.

5.3.3 Liberty file

The Liberty parser developed in Chapter 7 has been tested on one Liberty file. This file is representing a library in the sub-micro dimensions, with typical process values and operating conditions.

5.3.4 Project files

The code implemented as a part of this project can be found on GitHub [21] and also in the Appendix of this report. References to the Appenix are given in Table 5.7

Table 5.7: Code listings and code documentation

Description	**Code listing**	**Documentation**
Implementation for retrieving the structural information from the HDL description	Appendix D	Appendix A
Implementation of retrieving and organising library information from the Liberty file	Appendix E	Appendix B
Implementation of power model generator combining structural information from the HDL description of a design with the cell library information from the Liberty file	Appendix F	Appendix C

6 Extracting design structure

This chapter presents how the structural information of a design is retrieved, what comprises this information, and how it is shaped into a useful representation. The scope of this chapter, relative to the rest of the project, is highlighted in Figure 6.1.

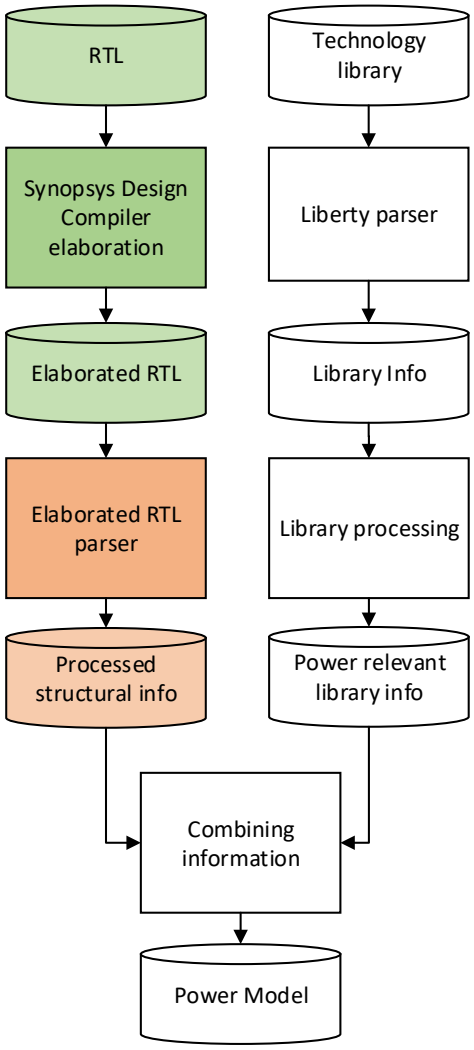


Figure 6.1: The modeling flow with the part of the flow relevant to this chapter highlighted

The structural representation consists of information about operations done on signals and how the signals are connected as described in Section 4.1. It is needed as an input to the power model generator. In this power model generator, the structural representation will be combined with information about the cell library, such as which cells are available and information on the power consumption of these cells.

In Chapter 4 an estimation flow was settled upon. The flow makes use of the elaborated SystemVerilog made by Synopsys HDL compiler, rather than unprocessed RTL. The elaborated SystemVerilog format is presented in Section 5.1. Using a gate-level representation would introduce a slow synthesis process, and is thus undesirable. Using the elaborated RTL the code constructs and compiler directives are dealt with and a structural library-independent netlist is available. Assuming the elaboration tool does this well, using this representation as an input to get a structural representation of the design is ideal, and will save some implementation time.

6.1 Elaborated SystemVerilog

The available structural information from the elaborated HDL is low level and fine-grained, being a netlist of the design using a generic cell library. This representation does not introduce any abstraction, except the abstraction already present as a gap between the Register Transfer Level (RT-level) and the gate-level. On the contrary, it reduces the gap between these representations by removing code constructs, like *if*, *case* and *generate* statements in the HDL, transforming it to a netlist of generic gates. However, if any compiler directives or parameters change, the design will have to be re-elaborated, and the structural representation regenerated.

The information needed for the structural representation is:

- What building blocks make up the design
- How they are connected to compose the design

In the elaborated SystemVerilog, the *building blocks* are the cells from the generic cell library and their *connections* are represented as wires, input and outputs in the elaborated design representation, listed in Table 5.1.

It is necessary to retrieve enough information about the arrangement of generic cells, so that a good representation of the design can be made. A goal for this project as a whole is

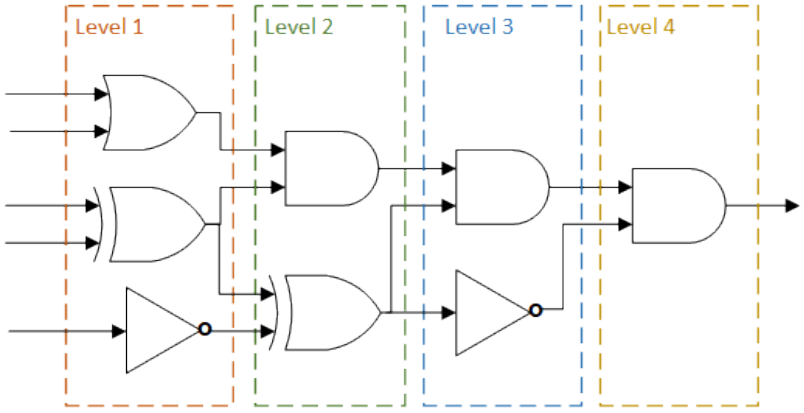
to be able to differentiate between the different types of cells and their power consumption. This requires distinguishing between different types of cells in the structural representation.

In Table 5.2 groups are made of the generic cells in elaborated SystemVerilog. These groups will be used, rather than representations for every generic cell type being differentiated between.

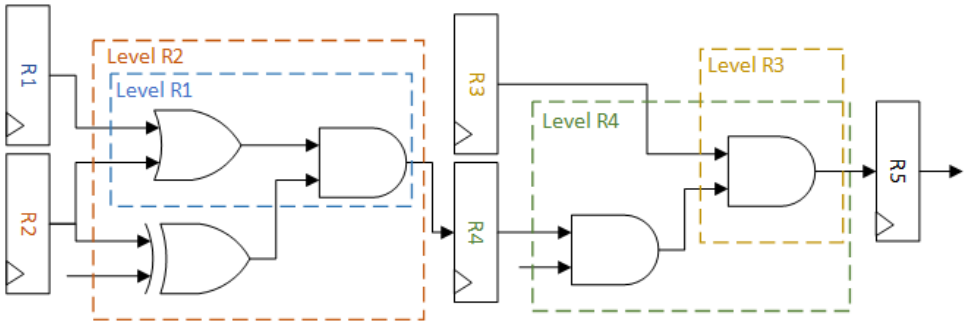
6.2 Structural representation of a design

There are many ways to represent the structure of the design, a few methods have been evaluated:

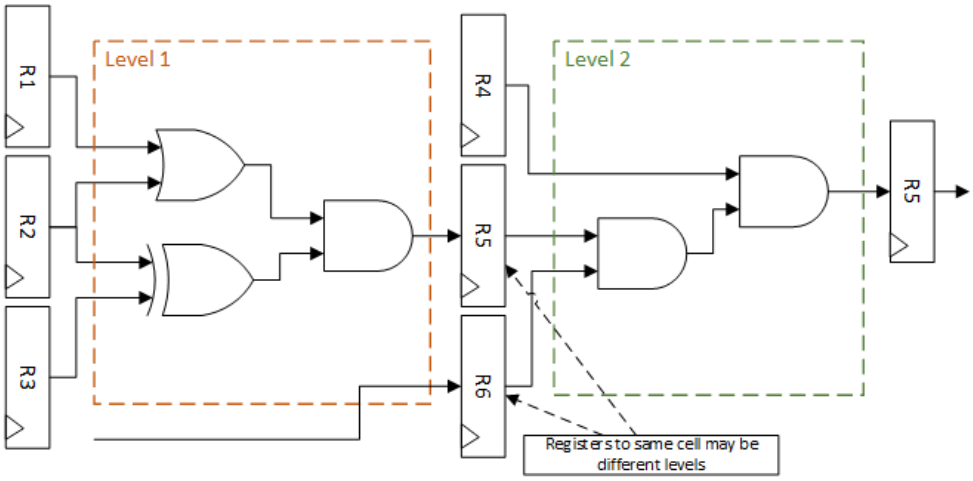
- The **levelised** circuit representation annotates gates with a level value. This annotation makes it possible to deal with each level of switching separately and propagate switching probabilities through the design. In [12], Gupta and Najm use this representation to estimate power by calculating the average capacitance at each level. However, this works for combinatorial circuits, not sequential. An illustration of this representation can be seen in Figure 6.2a
- If one can monitor the switching of registers during simulation, another way to structurally represent the circuit is to **levelise it by register**. One can monitor each register and the logic on each register output affected by its switching. This representation is illustrated in Figure 6.2b. One thing to take notice of is how each cell may be part of more than one register level.
- Another option is **large-scale register levelisation**, putting all the logic between a set of registers in one group and making a power estimate for this group depending on switching activity. This is illustrated in Figure 6.2c. A problem this introduces can also be seen in the figure. As "Level 2" in the figure technically also is "Level 1" due to the input of R6 not going through "Level 1". Clear definitions on how to group registers must be in place.
- It is also possible to represent the design **module-by-module**. This representation is made by having a model/representation for each module in the design. This representation can be useful for calculating average power, but power variations over time will be hard to consider. An advantage of this method compared to the register levelisation is that the module borders are clear and non-ambiguous.



(a) A levelised circuit



(b) A register-levelised circuit



(c) A circuit with groups of registers making up one level.

Figure 6.2: Different methods to levelise a logic circuit

The problem with logic being in more than one level is present in all the design representations except the **module-by-module** representation as the boundaries between modules are strict, and the **register-levelised** method, as the boundaries here are very loose. Ne-

mani and Najm [22] eliminates this problem by differentiating between the lowest level that a gate is used at and other possible levels by saying the first level a gate is encountered is where it is *generated*, while, on other levels the gate is *used*.

When settling on a structural representation, it must have the right level of complexity. A too complicated representation would need more time to process the simulation data presented to it by the power estimation tool. At the same time, a too simple model might not be able to give accurate enough power estimates. It is necessary to make abstractions to simplify the design representation, as this will reduce the time required to run a power estimation. On the other hand, it is essential to retain enough information to give useful power estimates.

For the **levelised** representation each gate would have to be considered which would result in a lot of data processing during simulation, if we want the simulation to be fast this is not suitable, even if considering each gate would give the most accurate results. The **large-scale register levelisation** and the **module-by-module** representation both abstract away too much information for the power estimation to be accurate as too much logic will be in the same groups, and the different impacts of different signals will not be seen. Their simulations would be fast, but the results not satisfactory. The **register-levelised** representation considers the impact of one register at a time, which is a manageable granularity, still being fine-grained enough the possibility for accurate estimation remains.

6.3 Abstractions made

As mentioned making abstractions are important to achieve the desired simulation speeds. As information can later be abstracted away by the *power model generator*, doing too many abstractions while making the structural representation is, however, deemed unnecessary. It is better to abstract away information after the structural representation is related to power information. Some abstractions done at this level are given below:

- **Grouping of generic cells**
  In Section 5.1 generic cells are grouped together based on their basic functionalities. This is done in Table 5.2. The biggest generic cell groups were the ones representing complex arithmetic operations such as multiplication, but also shifters and comparators have big groups.
- **Register levelisation**

Depending on how it is implemented, the register levelisation may introduce abstractions. If all information outside the register level loses its relation to information within that level, the correlation between inputs and the impact these correlations have on power consumption is lost. It will also make it harder to predict switching activity as inputs from different register levels may not be able to relate to each other, leading to a more inaccurate output switching probability.

Before relating the representation to the cell library used, it is good to retain as much information as possible to make the relation simpler.

6.4 Elaborated SystemVerilog parser implementation

The elaborated SystemVerilog parser is implemented in Python. It is class-based and has one class for each of the constructs listed in Table 5.1. It takes in the elaborated HDL representation of the design, created by Synopsys HDL Compiler and yields a set of node trees as the structural representation. The flow of the parser is illustrated in Figure 6.3.

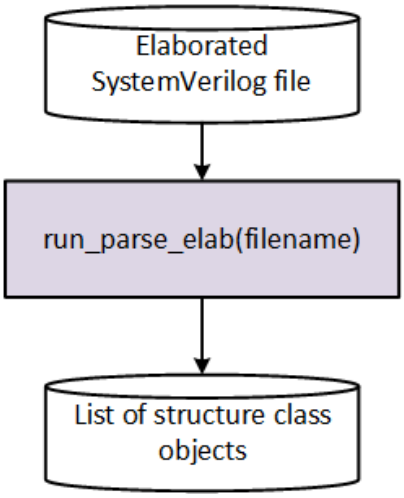


Figure 6.3: The elaborated SystemVerilog file is parsed and a set of *structure* class objects are made

The functions of the system, their hierarchy and the different classes are described in depth in Appendix A. The problem can be divided into two parts;

1. Parsing the elaborated SystemVerilog and storing the information retrieved in objects.

2. Processing these objects to make a register-levelised structural representation.

The implementation is briefly described in the sections below. The code of the parser is given in Appendix D and on GitHub [21].

6.4.1 Parsing

The parser begins by going through the whole input file line-by-line. It makes objects for each instantiation of a generic cell or connection it encounters and stores them in lists.

The lists are part of a module object, representing a module instantiation. When a new module declaration is reached, a new module object is made. A module instantiation inside another module is another object type referring to the module objects they represent. After a new object is created from a generic cell instantiation, its connection ports are registered and found in the existing input-, output- or wire- object lists. The cell connection is then registered in the related connection object.

When all the objects are made, and the parsing is done, one has all the structural information in the lists of inputs, outputs and wires as all generic cell objects are now connected to these.

6.4.2 Post-processing

The information obtained from the parsing needs to be made into the register-levelised structural representation.

The object lists in the module object make it possible to start from an input or register and see how the signals propagate, fan-in and fan-out until they do not propagate any further. A signal stops propagating when it reaches a top-module output, another register or a *select* or *control* signal for a SELECT or MUX generic cell. This propagation is done, and a structure tree is made from the elements.

The structure tree is a node tree with one parent and multiple children per node. As the goal of the register-levelised representation is to see all the cells affected by the switching of a cell fan-out is necessary to take into consideration. Fan-in is not considered as different inputs to a cell, may originate with different registers.

This structural representation leads to every cell with more than one input having itself and everything connected to its output duplicated as many times as its number of inputs.

To deal with this expansion, and reduce the processing time caused by it, the output node tree from a cell is stored in the cell the first time it is encountered. Later, if the cell is reencountered as part of another structure, the node tree stored in the cell object will be reused.

6.4.3 Register levelised structure trees

The circuit from Figure 6.2b represented as the structure trees developed from parsing elaborated SystemVerilog and processing the retrieved data is shown in Figure 6.4. The circuit becomes four structure trees, one for each register. Gates with inputs coming from elsewhere will also be part of other structure trees.

It is only possible to go through the tree in the direction of the arrows, and each tree head, starting with each of the four registers, are oblivious to the sharing of structures with the other trees.

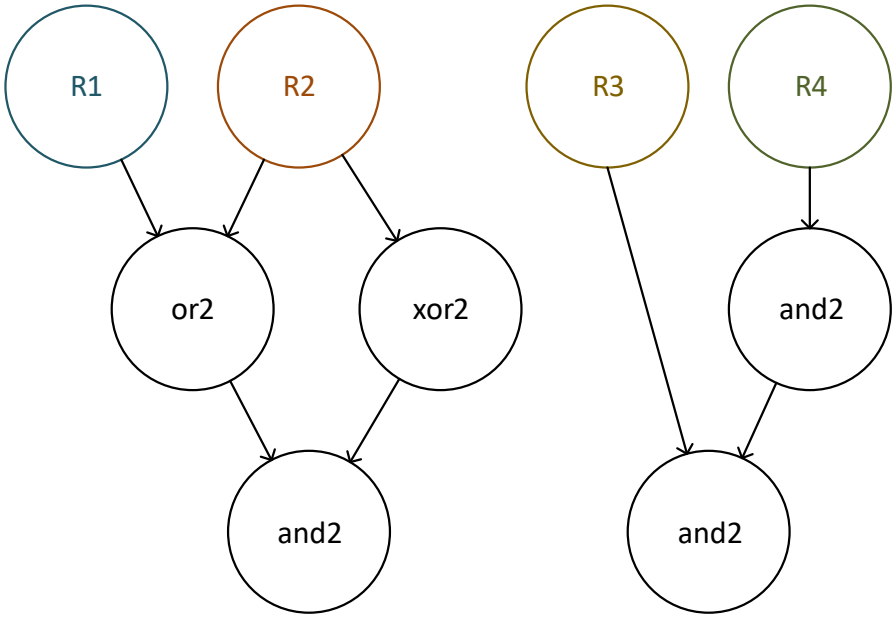


Figure 6.4: A structural representation of the circuit in Figure 6.2b as a tree of *structure* objects

A text representation of this structural representation is shown in Listing 6.1. Here the relation between the structure trees are more obscured.

Listing 6.1: Structural representation example

```
R1
reg
  |gtech_or2
  |   |gtech_and2
R2
reg
  |gtech_xor2
  |   |gtech_and2
R3
reg
  |gtech_and2
R4
reg
  |gtech_and2
  |   |gtech_and2
```

The first time a generic cell instantiation is encountered, the structure object created from it is stored in the cell object. If this cell has more than one input, it will be reencountered. When this happens, the structure object stored in the cell object will be reused.

6.5 Comparing cell counts

In Table 6.1 cell counts from the elaborated SystemVerilog parser and the synthesised file are given. The counts are sorted into different groups based on cell functionality. The *select* statement has no equivalent in the synthesised file, and *others* is a group representing cells in the cell library with no generic equivalent functions, such as *decap* cells, preventing an IR drop by providing current when much of the logic switches at once.

Table 6.1: Gate counts from elaborated structure and synthesised file

	constructs	**elaborated SV parser count**	**synthesised file count**
Module1	registers	164	175
	muxes	6	56
	inverters	406	88
	buffers	332	6
	arithmetic	2	0
	logic	1128	1196

	selects	2450	-
	others	-	2
	total	4488	1521
Module2	registers	15	13
	muxes	4	0
	inverters	92	13
	buffers	57	8
	arithmetic	0	0
	logic	411	183
	selects	81	-
	others	-	0
	total	660	217
Module3	registers	12	12
	muxes	0	11
	inverters	22	9
	buffers	20	1
	arithmetic	5	0
	logic	23	55
	selects	26	-
	others	-	0
	total	108	88
Module4	registers	16	16
	muxes	1	16
	inverters	99	22
	buffers	70	0
	arithmetic	0	0
	logic	89	156
	selects	51	-
	others	-	0
	total	326	210
Module5	registers	180	180
	muxes	0	0
	inverters	6	99
	buffers	5	11
	arithmetic	19	226

	logic	5	708
	selects	4	-
	others	-	1
	total	219	1225

6.6 Structural representation discussion

6.6.1 Cell counts

The most noticeable trend in Table 6.1 is the difference in total cell count. In most cases, the number of cells after synthesis is drastically reduced. The exception to this trend is the modules containing arithmetic cells. The generic cells for arithmetic operations may be large, as their input signals' width can vary. For example, a 32-bits adder could be represented as one generic cell instantiation. This is seldom the case for arithmetic cells in the non-generic cell library. These cells are often small and brilliantly combined to perform complex arithmetic operations. One generic cell adding together two 32-bits non-constant values will, after synthesis, very likely be represented by more than one arithmetic cell.

The elaborated SystemVerilog of Module5 contains 19 *arithmetic* cells. It is a filtering module performing several multiplications. The synthesis of the arithmetic cells leads to an increase to 226 *arithmetic* cells. Together with four select statements, they contribute to increasing the logic gate count from 5 to 708. The buffer counts also sink considerably in all test modules except Module5 where the buffer count increases from 5 to 11.

The select cells do not exist in the synthesised design, and the synthesis process transforms these to either multiplexers or logic cells. In most cases, it seems they become logic cells. The large Module1 begins with 2450 selects in its elaborated representation, and less than 56 of these are synthesised into multiplexers. All the other modules have similar trends, except for Module3. Here 26 selects become 11 multiplexers and the logic count increase from 23 to 55, but this may also be due to the modules' 5 arithmetic cells.

For modules where arithmetic operations are not done, the total gate count is reduced with more than 2/3 from the elaborated netlist to the synthesised one.

These general trends will hold, but the difference between the elaborated netlist and the synthesised netlist is heavily dependent on the cell library. If the cell library is similar to the generic cell library used during elaboration the reduction in gate count will mostly be

due to clever optimisation. If, on the other hand, the cell library contains a wide selection of more complex cells, the mapping from the generic library which contains simple cells to the more complicated cells will in itself reduce the gate count significantly. The further from the synthetic cells the actual cell library is, the more inaccurate the structural representation made by the parser will be.

An example of this can be seen from the inverter count. It is lowered due to many gates inverting the output being present in the cell library compared to the generic library, like NAND-, NOR- and XNOR gates. Cells with several inputs also lowers the gate count, as one OR5 gate can be used instead of four OR2 gates and so on.

6.6.2 The register-levelised node tree

The structural representation as a node tree introduces some limitations to the possibility of optimising the structural representation. It allows one to follow one bit through the circuit, but is unaware of the fan-in introduced by cells. It only sees the fan-out introduced by wires and is thus ever-expanding but never shrinking.

When re-encountering objects, their structure will be reused. This reuse of structures prevents this expansion from impairing the size of the structural representation, reducing the processing time and preventing duplication of structural representations for one cell.

Some data is lost in this representation, as only one bit is considered at a time in multiple-input cells. This loss of data could introduce an issue when this representation is later used for power estimation due to the status of all inputs impacting the probability of an output switching.

This data loss could be worked around, either by assuming a probability for the other signals being 0 or 1 or by storing some switching information in the structure object before propagating a switching probability. The latter of these will yield a more accurate result than assuming probabilities and considering the correlation between inputs. However, propagating input probabilities this circumspectly may make the power estimation slow, which is not desired.

Another obstacle the structure tree faces is later being combined with power data from the cell library. If cells with more operations and higher numbers of inputs exist in the cell library, switching to these cells in the structural representation will be hard as we are only able to move forward in the structure, not backward (from parent to children, not from child to parent).

6.6.3 Abstractions introduced by generic cell groups

Already in Section 5.2 some choices were made abstracting away information. Several generic cells were grouped together based on their functionality. For instance, multipliers using signed- or unsigned input signals, (or a combination of the two), has been put into a *multiplier* group. All comparator cells were also grouped together.

These generic cells often have no equivalent in the cell library. It is the job of the synthesis tool to combine available cells to create the functionalities needed. These types of generic cells will be subject to massive optimisations in the synthesis. For instance, if one of the inputs to an addition or multiplication is constant, this can significantly reduce the logic needed.

Predicting some of these optimisations may be more important than whether the inputs to the cell are signed or unsigned.

The problem of how to represent these cells aside, grouping cells that will be represented differently, will introduce inaccuracy. For the comparators the *equality* and *inequality* comparators are grouped together with the *"greater than"*, *"lesser than"*, *"greater than or equal"* and *"lesser than or equal"* comparators. The logic needed for different types of comparators will vary and dividing this group in three should be considered in the future.

6.6.4 Registers being optimised away

In Table 6.1, it can be seen that the register counts are not always the same between the synthesised design and the elaborated SystemVerilog file.

In some cases, if input signal buses are more narrow than the module is made to handle, some registers within the module will be optimised away as they are not needed. Predicting this with the structural representation is hard. As each register is the head of a structure tree, it is hard to know whether registers will be inferred. A *has_parent* variable introduced in the register object and only registers having a data input get this variable set to *True*. If a register has a parent, and its corresponding structure tree has children, the register is assumed inferred. In most cases, this assumption is valid, but in Module1 it results in fewer registers in the elaborated count than in the synthesised one, and in Module2 it results in more.

6.6.5 Possible optimisations

A disadvantage with the chosen structural representation is that different branches in the node tree do not have any relation to each other even if they are parents of the same structure. More ideally, they would be aware of each other or even be part of the same representation of a cell in the power model.

Such awareness would make propagating the activity data more accurate. One would have a probability for each of the inputs rising, and thus be able to calculate a more accurate probability of the output switching. It could also make it easier to relate the information on available cells in the cell library to the structure trees of generic cells.

On the other hand taking several inputs into account before propagating switching activity is complicated and such an approach may be too time consuming.

7 Extracting library information

This chapter presents and discusses how the information related to power consumption is retrieved from the cell library. An illustration of the flow, highlighting the steps relevant to this chapter, can be seen in Figure 7.1.

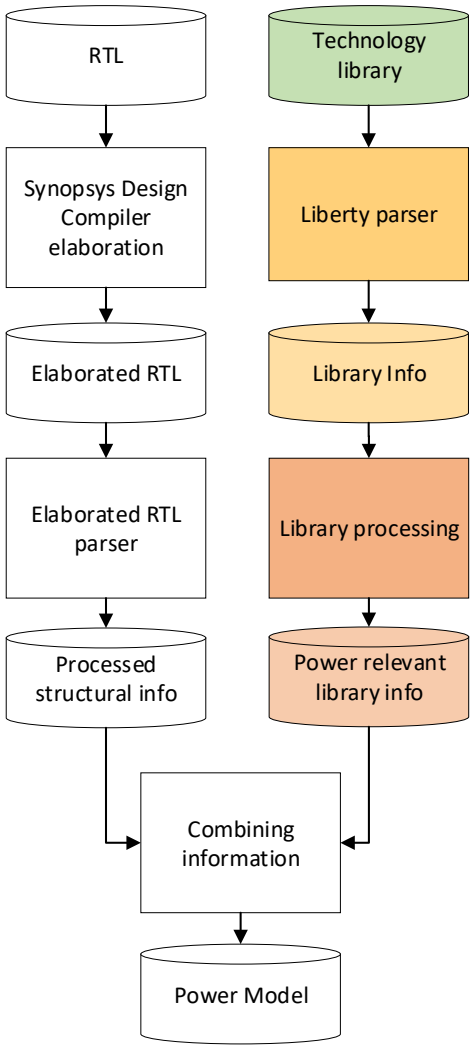


Figure 7.1: The modeling flow with the part related to retrieving power information from the cell library highlighted.

The resulting representation of cells in the cell library and their power consumption will later be used in the power model generator. The power model generator will relate the cell information to the structural representation of the design.

The motivation for using the Liberty file for this is given in Section 4.2. As information about the different cells available and their power characteristics is not normally introduced to a design before synthesis. Introducing it earlier and trying to use it in power estimation will lead to better accuracy of the power estimations if this information is related to the design one tries to estimate the power consumption of in a good way.

7.1 Relevant power data

The information to be retrieved from the cell library consists of certain groups and attributes. These are described in Section 5.2. Different cells have different power consumptions. An AND2 gate and a Multiplexer, for instance, will not have the same leakage power or switching power. Differentiating between types of cells will lead to more accurate power estimates, than, for instance, using the power data of an average cell for all cells. Using the average cell may yield the same average power consumption for the design, with a loss of spatial and temporal accuracy. Being able to improve accuracy with this differentiation is depending on relating the cell information well to the RTL representation of the design. Using the structural representation developed in Chapter 6, relating different cells and their power data to different generic cells in the structure trees is possible.

This information will allow for estimating the switching power by adding together the power values from the input pins' internal_power group, which makes up the *internal power* of the cell, and the power value in the output pins internal_power group, which depends on the capacitance of the pins they drive (if any) and constitutes the *switching power* of the cell. It also allows for estimating the leakage power by looking up the value in the leakage power group.

7.2 Abstractions

7.2.1 The difference between fall- and rise power

Often it is either the fall- or the rise power consuming power in a transition. The rise power is described in the rise_power group. It describes the power consumed as the output pin rises when a related input pin rise. The fall power is part of the fall_power group and

describes the power consumed as an output pin falls after the related input pin rises. In this implementation, these two contributions to power, the rise- and fall power, are added together, as a signal rising implies that it has previously fallen, and conversely. This way, one needs only observe rising transitions, but can still take the fall power into account. This combination of the rise- and fall power effectively halves the amount of data needing to be represented. The temporal accuracy of the power estimation will, however, be affected by this combination. If a cell consumes the most switching power when an input falls, instead of when the input rises, the power consumption will happen later than estimated. After a signal rises, it is impossible to say when it will fall, and the power related to the fall of the input will be consumed.

This inaccuracy may be worth having half the amount of data to process in a simulation. An illustration of the abstraction is shown in Figure 7.2.

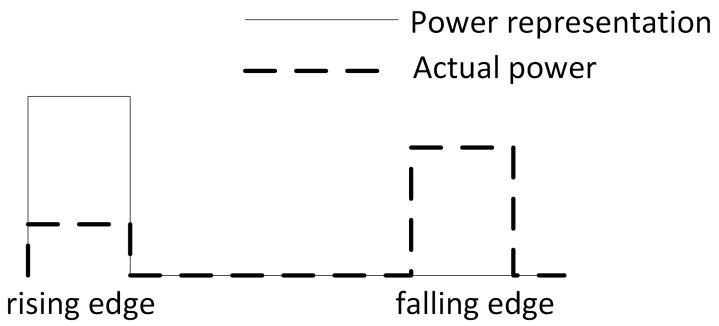


Figure 7.2: The impact on power estimation when summarising rise- and fall power

7.2.2 The difference between data input pins

The difference between data inputs in a cell will be abstracted away. The first input pin is the only input pin for which power data will be stored. This will lead to power estimated for different input pins in a cell to all be based on the power characteristics of the first input pin.

A good example of this abstraction can be given with the ANDOR21 cell, shown in Figure 7.3, two of the input signals go through the AND2 gate, before the OR2 gate, while the third input signal only goes through the OR gate.

The ANDOR21 cell does not exist in the generic cell library. Its equivalent is a generic AND2 gate, combined with an OR2 gate. The two first inputs of these cells will go through

both the AND2 gate and the OR2 gate and have the structural representation given in Listing 7.1, while the third input will only see the OR2 gate and be oblivious to the AND2 gate, as shown in Listing 7.2. The two first input transitions will naturally consume similar power, but the last input is not going through the same operations and will have different power characteristics.

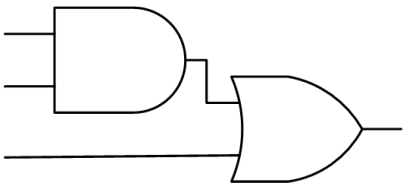


Figure 7.3: ANDOR21

Listing 7.1: ANDOR21 gate seen from the two first inputs

```
gtech_and2
    |gtech_or2
    |     |...
```

Listing 7.2: ANDOR21 gate seen from the last input

```
gtech_or2
    |...
```

A comparison between two cells from the cell library; an ANDOR21 cell, as shown in Figure 7.3, and a regular OR2 cell is done. The comparable scenarios are the third input of the ANDOR21 rises while the result of the AND operation is 0, and a rising input on the OR2 cell while the other input is 0. It turns out that the ANDOR21 consumes approximately 41% more power in this switching.

Being able to take the found difference in power consumption into account would be advantageous as a more accurate power model could be made. With the structural representation from Chapter 6, where only one bit is considered at a time, it is impossible to recognise a cell with more operations than those the bit in question propagates through. Thus, abstracting away all inputs but the most complex one, will not make the representation more abstract, as the same limitation is already present from the structural representation. If a way to work around this abstraction is found in the power model generator, or the structural representation is re-implemented, however, only saving data from the first input pin of a cell is introducing some inaccuracy.

7.2.3 The state-dependency of leakage power

The leakage power in a cell is state-dependent. The library used contains leakage power groups with the *when* condition for all possible states of the cell to model this. There is also one leakage group containing the average leakage power between all the other leakage groups. In this project, the average leakage power group will be used, and the state of the cell will not be taken into account. Making use of the average leakage power removes the need to calculate the states of cells in the power model. Thus, it is still an opportunity for the power model to abstract away cells entirely, and only have a sum of power values in their place when calculating the power consumption in simulation.

Knowing the leakage power as a cycle variant contribution would increase the temporal accuracy of the power estimations, but it may introduce more computation than it is worth. Cells that spend equal time in all states will contribute a total power equal to the leakage power value used in this project. It is unlikely this is the case for any cell, but the total average power contribution from leakage power will likely be close to the total leakage power consumption nevertheless, as some cells will consume more than expected and others less.

The difference between the states of cells is significant and shown in Table 7.1. For a regular AND2 gate the leakage power can vary with up to 161%. If the AND2 gate is always open, this will result in a larger actual leakage power consumption than the estimated one, and if the gate is always closed it will result in a smaller actual leakage contribution than the estimated value.

Table 7.1: Increase in leakage power from least consuming to most consuming state

Cell	**Increase in power consumption**
AND2	161%
MUX2	51.9%
NOT	147%

Knowing how the leakage power varies in time is not relevant in most cases as the leakage power contribution to power is several orders of magnitudes less than the switching power. It is the steady, relentless power contribution every cycle that makes leakage power a big part of a design's power consumption. As long as the average leakage power is accurate enough, knowing the leakage variation in a cell over time is deemed redundant for this

project. When qualitative results are obtained, the accuracy of the estimated leakage power contribution should be investigated.

7.3 Cells with same functionality

In a cell library, there are several cells with the same functionality, that have different properties. They can differ in timing-, area- and power characteristics, and the load capacitance the cell can handle and so on. Depending on requirements to a specific location in the design, the synthesis tool may choose any of these cells.

For large fan-outs, cells with high enough driving capacity are needed. For critical paths, faster cells may be necessary to avoid breaking the timing constraints. If it is a priority to reduce the power consumption of a design, cells with low power consumption will be used wherever possible.

As these cells have varying power characteristics, it is necessary to know which of them are most likely to be used in synthesis to improve the estimation accuracy.

From the pin attributes, the maximum capacitance an output pin can drive is given. If the total capacitance of the pin(s) this output is connected to exceeds this capacitance for any cell, it can be discarded as a possible candidate.

Trying to choose a cell only based on the driving capacitance, however, ignores design constraints. These will be reflected in the cells used at the gate level representation.

A representative design can be used as a *calibration netlist* for the Liberty parsing to take design constraints into account. It must have been synthesised with the same frequency and voltage as intended for the design one wants to model as which cells chosen during synthesis depend heavily on this information. The *calibration netlist* needs to be large enough to give a clear indication of which cells are likely to be used and which are less likely.

In the *calibration netlist*, the number of occurrences for all the available cells is counted. Later, when comparing cells of the same functionality, this count can be compared to the counts of other cells, and used as an indication of whether or not a cell is more likely to be used than another.

7.4 Implementation

Running the Liberty parser is time-consuming. The liberty file used in this project exceeds 11 million lines, containing around 300 cells. For each cell approximately 20 lines of information is needed. This amounts to 6000 lines constituting the information wanted. Temporarily storing the power- and cell information outside the Liberty file is deemed necessary. The flow of the implementation is shown in Figure 7.4.

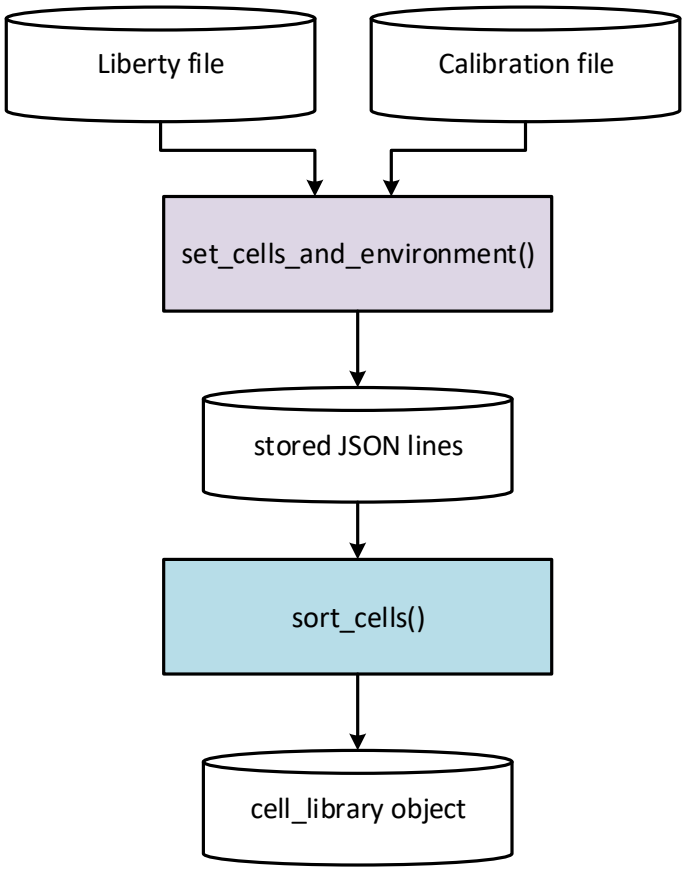


Figure 7.4: The dataflow of retrieving the relevant Liberty data.

The implementation has been divided in two:

1. Reading out data from the Liberty file and calibration data from the *calibration netlist* and storing it in JSON object lines.

2. Reading out the JSON lines from the intermediate file and construct an object representing the cell library.

The implemented code can be found in Appendix E and on GitHub [21]. A detailed overview of the functions and classes in the implementation can be found in Appendix B.

7.4.1 Parsing Liberty and storing data

Using the Liberty parser provided by Nordic Semiconductor ASA, the Liberty file was parsed, and the information interesting for power estimation of a cell was put into a list then transformed to a JSON object. In addition to the cell information, each cell's occurrences are counted from a *calibration netlist*, a large synthesised design used by Nordic Semiconductor ASA. This count can later be used to see which variations of each cell type are more probable to be used in synthesis.

Each cell is thus made into one object on one line and all unwanted information is removed from the cell representation. A function for reading out the representations from the JSON file was also made. A JSON line looks like this:

```
([cellName, footprint,leakagePower, occurences_in_calibration_file, [
    input_pins, output_pins])
```

The output_pins is a list of output pin objects:

```
[pinName, pin_direction, pin_function, pwrPin, gndPin, related_pin,
    when_condition, [rise_cap, powerSumList]]
```

rise_cap is an array of load capacitance values, and PowerSumList is an array of power values relating to each capacitance value in rise_cap. The values in PowerSumList is the sum of the rise- and fall power values for the capacitance value in question. The input pin object is identical to the output_pin object, except for having a capacitance value instead of a function.

Running the liberty parser of Nordic Semiconductor ASA on the more than 11 million line Liberty file takes 11-12 minutes. If in addition to this calibration is done on an almost 3 million line calibration netlist, the time required to get the JSON library representation gets close to 20 minutes.

7.4.2 Putting together a cell library object

When the power library representation is needed, the file with the JSON objects is parsed, and each cell found is put in a cell object. This object contains all the information on the JSON line and a sequence of generic gates from the elaboration library corresponding to the cell behaviour. The cells are then grouped into a cell_group object based on their functionality. All the cell groups are then put in a cell_library class object representing the cell library.

7.4.3 Summary

The relevant power information from the cell library can now be retrieved and stored in a library object. The library object contains lists of groups sorted after the number of inputs and separate lists for registers, multiplexers and empty groups for the more complex generic cells. This library object can be used to find the power information one wants, which was previously found in the liberty file.

To avoid the time-consuming parsing of the liberty file every time the power estimation is done, the Liberty information is intermediately stored in JSON lines. This intermediate format reduces the time it takes to get the information from the liberty file and the calibration data from almost 20 minutes to instantaneous.

7.5 Discussion

7.5.1 Choosing a cell from a group

When deciding which cell in a cell group to use in the power model, there are different ways to do so. One can use the characterisation data to get the cell in a group with the highest weight, or one can use the weights to calculate some average cell in a group based on a weighted average.

Another option is to not care about the characterisation data and choose a cell with suitable driver strength depending on the total input capacitance it has to drive. Although, this selection may depend just as much on the speed needed for switching.

As this power representation contains no timing information, choosing cells based on timing is not an option. This may introduce inaccuracy if the circuit is synthesised with strict performance constraints. The calibration data tries to make up for this but depends on

the design subject to power estimation being synthesised under similar constraints as the calibration netlist.

Choosing a cell in a cell group can be up to the user of the library by adding procedures to it, giving out a cell depending on the user's choice. Such a procedure could ask for a weighted average cell or a cell corresponding to some load capacitance.

7.5.2 Other representations

A representation that does not abstract away the fall power and leakage power states is also a possibility. Differentiating between rise- and fall power should allow for more cycle accurate power estimation. The fall power can lead to more power consumption than the rise power and signals may remain high for a long or short duration before falling again. A more state-dependent estimation of the leakage power could also be possible.

The abstracting away of power data from all inputs, but the first one could be skipped to make the library representation more general. When it is going to be combined with the structural information from Chapter 6, however, having power data from only the first cell input is sufficient.

7.5.3 On the calibration

The calibration script is simplistic and goes through the entire calibration netlist for each cell to count how many times it appears. If runtime is critical, the algorithm should be improved. A good alternative is to go through the calibration netlist looking for all cells in a group at once, or even going through it only once counting occurrences of all library cells simultaneously.

It could also be an option to move the calibration to a different pointing the flow, especially if its runtime is improved. This way, the Liberty parsing remains indifferent to synthesis settings, just extracting the information from the liberty file, and the parsing of the Liberty file will not have to be done again if one wants to test the estimation with another calibration netlist. The calibration data could, for example, be an input to the power model generator instead. Using the calibration data as an input here requires the power model generator user to be aware of the cell library, so the calibration netlist is undoubtedly from the same library as the one used for power estimation.

8 Generating a power model

This chapter will present how the structural information retrieved from the elaborated SystemVerilog file can be combined with the cell library information retrieved from the Liberty file in order to create a power model of the design. Figure 8.1 illustrates the scope of this chapter.

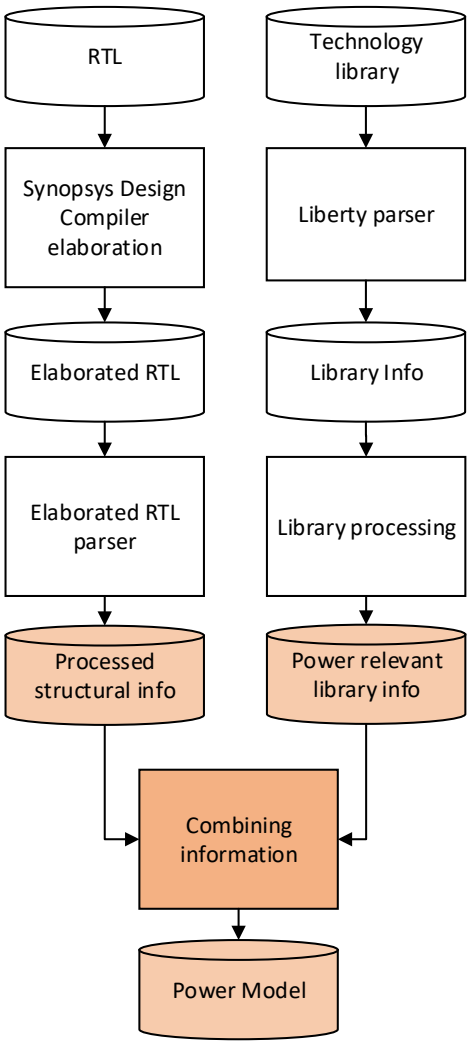


Figure 8.1: The modeling flow with the flow relevant to this chapter highlighted.

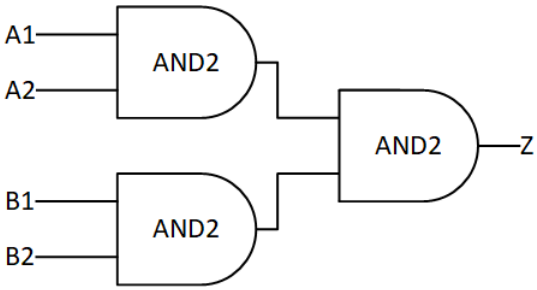
The *processed structural info* will, in this project, be the *structure* tree from the elaborated SystemVerilog parser. The *power relevant library info* is the *cell_ library* object acquired from the Liberty file. Combining the data retrieved is necessary to relate the structural representation to realistic power data and is the last step towards getting a power model.

8.1 Limitations introduced by the structural representation

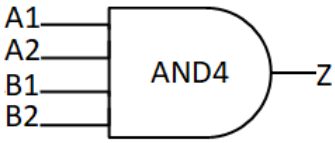
The most major limitation introduced by the implementation in Chapter 6 is the nature of the structure trees. The nodes in the structure tree allow for finding the children of a structure, but not its parents. It fans out whenever a signal fans out, but does not fan in when a cell does so. Two inputs to the same cell remain oblivious to each other in such a structural representation.

On one hand this allows one to easily parse through all the structures affected by a register or input changing value and estimate the amount of logic. On the other hand, not knowing the other parents of a node can make it harder to estimate the switching activity.

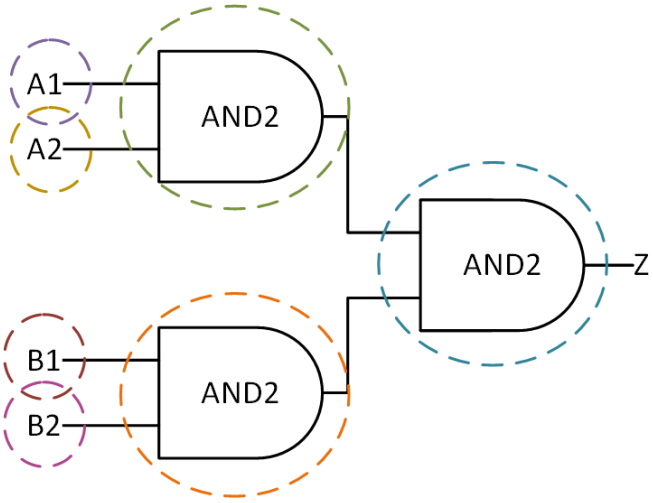
Not knowing the parents of a node in the structure tree makes doing many optimisations on the structure tree impossible. Only sequences of cells will be recognised and possibly replaced with more complex cells. For instance, the three AND2 gates in Figure 8.2a in an elaborated SystemVerilog representation is very likely to be optimised to one AND4 cell during synthesis, if one is available in the cell library. As the last AND2 gate in the structure is unaware of its parents the parents remain oblivious to each other. The structure objects created by the implementation in Chapter 6 is shown in Figure 8.2c. And the tree made up of the structure objects is illustrated in Figure 8.2d. All the four input pins, will, even though they share structure objects be unaware of their relation to each other.



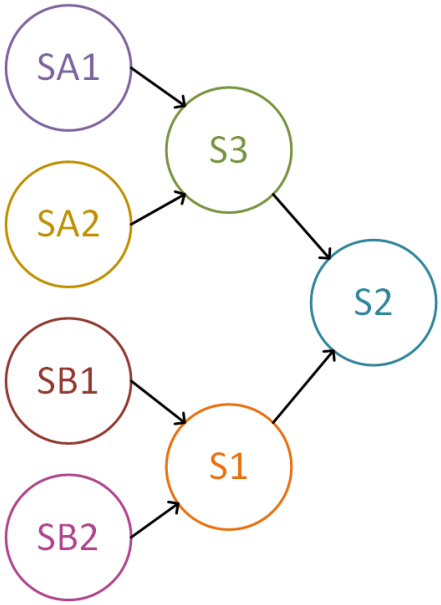
(a) AND4 represented in elaborated SV



(b) AND4 after synthesis



(c) AND4 in structural representation. Structure objects are represented as dotted circles around the object they represent.



(d) AND4 structure tree

Figure 8.2: Different representations of an AND4 gate

8.2 Limitations introduced by the cell library representation

Three abstractions were introduced in the implementation of the library processing in Chapter 7. They are discussed further in Section 7.2.

- **Combining rise- and fall power**

  Reducing the temporal accuracy of the power estimates. Combining the power contributions will not introduce any inaccuracy on the switching power values, but the time where power contributions happen will be inaccurate.

- **The difference between input pins**
  Power data is only saved for the first input pin of a cell. Ignoring the difference may complicate grouping of generic cells to one cell from the cell library, as one is unable to differentiate between inputs and their impact on power. It was seen in Section 7.2.2 that the OR2 operation done in a regular OR2 cell and the OR2 operation that is part of an ANDOR21 cell consume different power.

- **State dependency of leakage power**
  The different leakage power values for different states of a cell were not retrieved from the cell library. Only the average value between all the states is available after processing the library data. As the leakage power can increase with as much as 160% from a low-output state to a high-output state as shown in Table 7.1, the inaccuracy using only the average leakage power of cells may introduce significant inaccuracy.

8.3 Combining the structural information and the liberty data

The cells in the actual cell library are different from the cells in the generic cell library used in the structural representation. To combine the power information with the structural representation, a mapping of the generic cells to cells from the cell library is necessary.

As a cell is not aware of its parents, this can only be done by going through the structural representation and replacing generic cells or sequences of generic cells with cells from the cell library.

After being able to determine which *cell group* fits one or more of the generic cells, a *cell* has to be chosen from that group to get power data. As the calibration data states, the probabilities of each cell in a group being used, some weighted average power consumption of the *cell group* can be made from this. The driver strength of cells could also be used when choosing a cell, excluding those that can drive loads differing by some margin from the actual load.

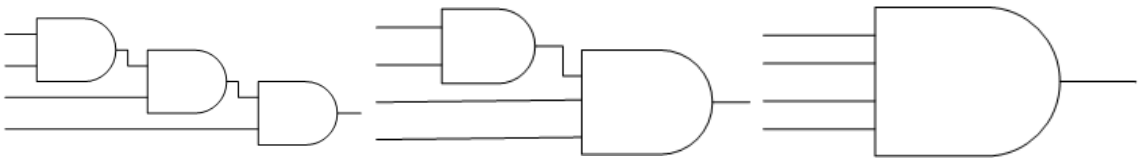
For the generic arithmetic cells, there is seldom a suitable cell in the cell library. This is discussed further in Section 8.4.

As so few of the buffers were still present after synthesis, as seen in Table 6.1, it is chosen to remove buffer cells entirely from the power representation of the system. Another option could be to let buffer cells remain only if their fan-out is high enough for it to be deemed necessary.

8.3.1 Need for optimisation

When Synopsys DesignCompiler elaborates the HDL, the representation is optimised and mapped to the generic cell library. When mapping it to corresponding cells, one option is to find an equivalent cell for each of the generic cells, and simply replace all generic cells with their cell library equivalent from the Liberty file. Systematic errors this may introduce can be investigated and adjusted for to the best ability. Another option is to do some "optimisation" and for instance map an AND2 cell followed by a NOT cell to a NAND2 cell if one is available in the cell library.

To investigate whether the generic cells can be mapped directly to library cells by using only the cell library equivalents of generic cells, or if several generic cells should be mapped to one more complex cell from the cell library when possible, three options of implementing an AND4 gate have been examined: One representation consists of three AND2 gates, one consists of one AND2 gate and one AND3 gate, and lastly, one consists of one AND4 gate. Power information for each of these have been retrieved from the cell library that is used in this case. The three AND4 implementation options are shown in Figure 8.3.



(a) 3 AND2 **(b)** 1 AND2, 1 AND3 **(c)** 1 AND4

Figure 8.3: Different implementations of a 4-input AND gate

The different cells in the cell library have different power characteristics as shown in Table 8.1. Here the cells are all driving the same output capacitance having one of their inputs rise as the other rises for the dynamic power and the average leakage power value is used. The leakage power values are a bit lower for the bigger cells and the dynamic power consumption is a bit higher.

Table 8.1: Power consumption in AND cells of different sizes using AND2 as the reference

	AND3	AND4
Leakage power	−8.4%	−20.1%
Dynamic power	+14.4%	+18.9%

The average leakage power for Figure 8.3b is 36% lower than for the three AND2 gates in Figure 8.3a. For the AND4 gate in Figure 8.3c, it is 73% lower. This is due to the leakage power being the average of all the states in the cell and the gates with higher inputs having more states with low power consumption. Combining the three generic cells into one will reduce the calculated leakage power, simply by making use of the cell with more inputs.

In reality, however, the three AND2 gates in Figure 8.3a will not all consume their average leakage power. As one input from the second AND2 gate comes from another, the probability of that input being high is lower than if it was coming from an input propagating through less cells. For the third AND2 gate the probability of the gate output being high is lowered once again. As the model does not monitor states or take state-dependency into account when calculating leakage power this is not taken into account, and the estimated leakage power of the three AND2 cells will be higher than in reality. However, the reduction in leakage power from combining generic cells when mapping them to the cell library is so significant that it improves the power consumption either way. The leakage power consumption of an open AND4 cell with all inputs high, for instance, is 49% lower than that of three open AND2 cells.

For the switching power, the scenarios that leads to the output of the AND4 gate switching has been considered. This means that three inputs are already high, and one rises. For the AND gate combination in Figure 8.3a, this can cause one-, two- or all of the AND2 cells to switch. The mean case will be considered. For the second AND4 gate implementation, in Figure 8.3b, this means either both cells will switch, or only the AND3 cell will. Lastly, for the AND4 cell in Figure 8.3c, one cell will switch.

Using the unoptimised combination of three AND2 gates in Figure 8.3a as a basis, the switching power will on average be 19% lower for the option in Figure 8.3b, and 41.3% lower for the AND4 gate in Figure 8.3c if they drive the same output. The comparison in shown in Table 8.2.

Table 8.2: Power consumption in AND4 optimisations, in comparison to the three AND2 gate implementation in Figure 8.3a

	1 AND2 1 AND3, Figure 8.3b	1 AND4, Figure 8.3c
Leakage power	−36%	−73%
Dynamic power	−19%	−41%

Thus, combining the generic cells to suit more complex cells from the cell library is desirable to get a more accurate power estimate. This will reduce the error in leakage power by introducing cells with more states and reduce the error in dynamic power by using cells more likely to be chosen by the synthesis tool.

The generic library does not contain logic cells with inverted outputs, such as *NAND*, *NOR* and *XNOR*. Typically these cells need less transistors than their non-inverting equivalents. A NAND2- and an AND2 gate will be used to understand the significance of this.
In figure 8.4 CMOS logic for a NAND2 gate and an AND2 gate is shown. An AND2 gate consists of a NAND2 gate and an inverter. The generic cell equivalent of the NAND2 gate is a AND2 gate followed by an inverter. If the NAND2 is implemented directly as such, four extra transistors are needed, compared to using a NAND2 gate directly, if one is available in the cell library.

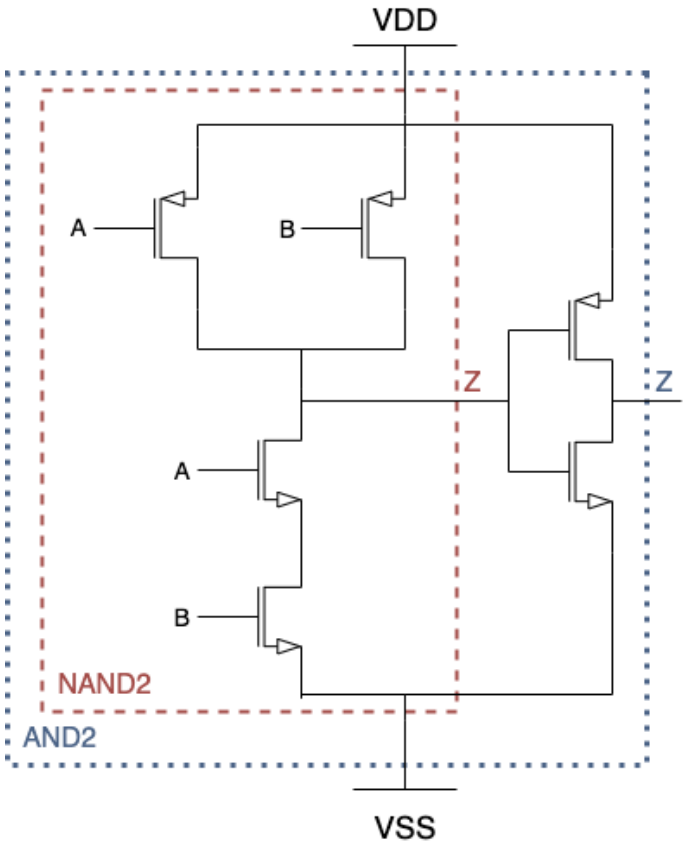


Figure 8.4: A common CMOS schematic for a NAND2 and an AND2 gate. The AND2 schematic is the same as the NAND2 but with an added inverter.

The amount of logic needed, and thus, the power consumed, is reduced by optimisations in the synthesis process as shown in Table 6.1. It is necessary not only to map the structure from Chapter 6 to cells from the cell library, but to also do this intelligently as many cells in the cell library are complex and does not have generic cell equivalents. The same goes for the other way around, as many generic cells, specially the arithmetic ones will never have an equivalent in the cell library.

8.4 Generic cells with no library equivalent

In the power model generator it is assumed that all the logic cells in the generic library have an equivalent. This means cell libraries used **must** have the following cells: *NOT*, *AND2*, *OR2*, *XOR2*. It is also assumed that the cell library contains a *register* cell and a *multiplexer* cell.

It is possible to make a power model generator without these assumptions, but then alternatives to the generic cells would have to be found and proposed. An alternative to an AND2 cell, for instance, will be a NAND2 cell followed by an inverter.

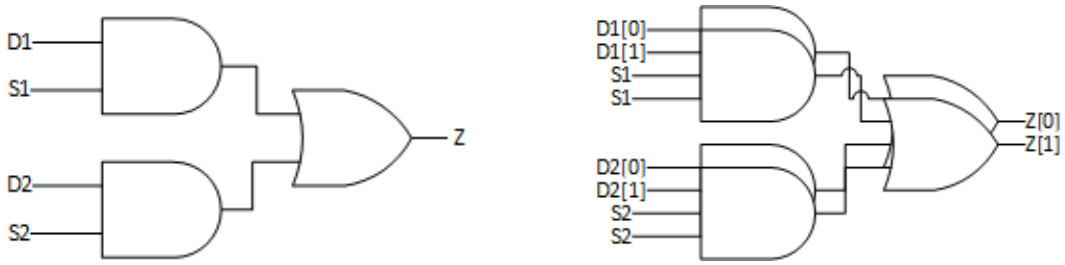
Several of the generic cells are complex and have no equivalent in the cell libraries, as their implementation will depend heavily on the inputs to the cells and their size. Most of the generic arithmetic cells; *multipliers*, *divisors*, *adders* and *subtractors* fall under this category and need an alternative implementation with cells from the cell library. For now, all of these are left as empty shells, containing no power information and no cells from the cell library. Alternative representations for *selects*, *shifters* and *comparators* will have to be made as well.

In Section 8.4.1 a representation is investigated for the generic *select* cell. A similar approach can be used upon making representations for the other cells, however, the synthesis of the arithmetic generic cells will depend highly on their inputs. An addition or multiplication of an arbitrary number and a constant, for instance, require less logic than an addition or multiplication of two arbitrary numbers.

8.4.1 The *select* cell

The elaborated SystemVerilog netlist contains the 'select' cell, which does not have an equivalent in any cell library. It has the functionality of a one-hot multiplexer and is at times synthesised using a multiplexer, and at other times synthesised using logic cells.

It is necessary with a consistent way to represent the 'select' cell to incorporate it into the power model. From the results in 6.5, it can be seen that in most cases the select statement is not made into a multiplexer, but rather implemented in logic. A 2-input one-hot multiplexer can be represented as two AND2 gates and one OR2 gate, which is also equivalent to an AO22 gate. This representation can be seen in Figure 8.5. This implementation can also be extended to an N-bits one-hot multiplexer by using N AND2 gates and one N-inputs OR gate or its equivalent. For datawidths larger than one, the whole MUX-structure will be duplicated for each data bit



(a) One-hot MUX2 logic representation with datawidth 1

(b) One-hot MUX2 logic representation with datawidth 2. Twice the amount of logic is needed compared to the MUX2 with half the datawidth

Figure 8.5: One-hot multiplexers with different datawidth

The power consumption of possible replacements has been calculated, using a MUX2 as a baseline. The AO22 gate is a suitable replacement with a 4.9% increase in the leakage power and a 7.9% decrease in switching power. Two AND2 gates followed by an OR2 gate performs worse with 37% higher leakage and a 40.8% increase in the switching power. If an AO22 gate is available in the cell library, replacing the SELECT statement with that one, rather than a MUX, is a good option. As a second option, if no such gate is available, substituting the *select* operator with a MUX would be better than the three-cell alternative in Figure 8.5.

The AO22 gate is equivalent to a one-hot MUX, but has two separate select signals instead of a select input and the same select input inverted, compared to a regular MUX. Using the AO22 gate solely to replace select statements should not introduce more than this error in the cases where a MUX is used instead of other logic cells. The optimisation done by the synthesis tool, concerning *select* operators, is still unaccounted for, however.

8.5 Estimating the switching power

When estimating the switching power of a design, how switching activity propagates through the cells is important. With the register-levelised structural representation each input- and register bit is the head of a structure tree containing all the logic affected by the input- or register bit switching. Two methods for calculating the switching probabilities will be investigated. Calculating the complete signal propagation, and calculating the signal propagation for each structure tree separately.

- **Determine the complete propagation of the signals**
  It is possible to know the exact propagation of signals by waiting with calculating the output switching probability of a cell until all input switching activity is known. Then one would know exactly how many gates switched each cycle and can estimate power from it. However this is very complex, especially for larger modules and would take a considerable amount of time. This is what a simulator does.
- **Calculate propagation in each structure tree separately**
  Another option, that do not require as much calculation each cycle would be to calculate the propagation probabilities beforehand. Then *one* power value can be present for each structure tree. During simulation this value can be added to the total power consumption each time the head of the structure tree (a register or an input) rises.

As determining the complete propagation of signals is too complex for the method to be used efficiently in parallel with RTL simulations the second option is better suited for propagating the switching activity through a structure tree. Having one switching power value per structure tree is a huge advantage when it comes to calculating the power consumption with activity data from a simulation.

8.6 Implementation

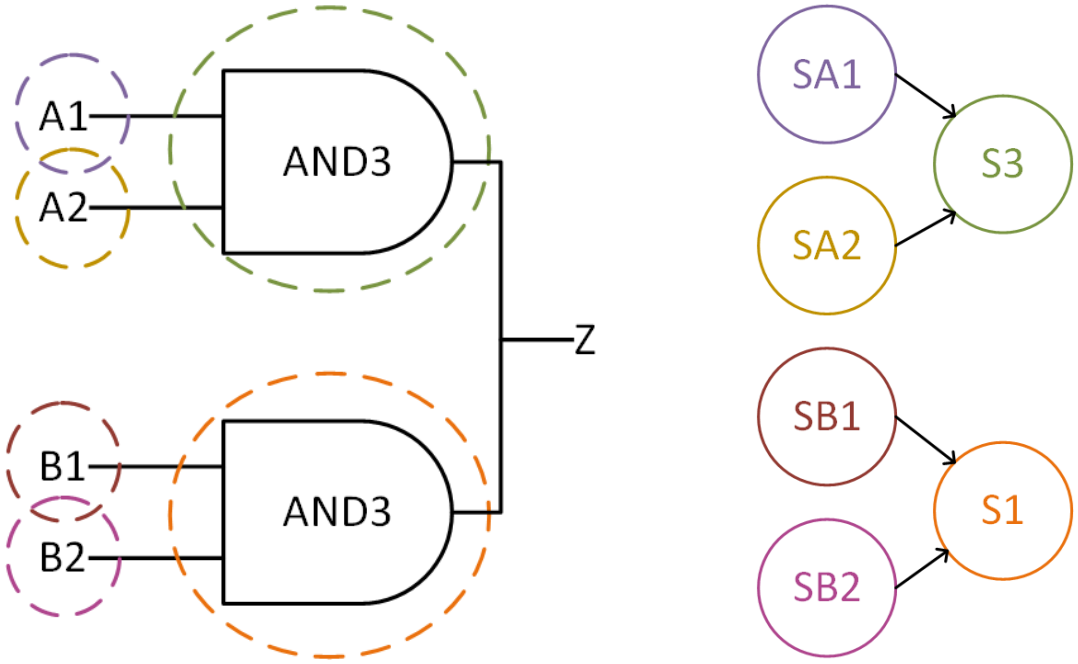
A description of the system functions, classes, and behaviour can be seen in Appendix C. The full implementation can be found in Appendix F and on GitHub [21].

The power model generator is implemented in Python. It scans the structure trees from the structural representation implemented in Chapter 6 and replaces generic cells, or sequences of generic cells, with cells from the cell library and their power information. These cells and corresponding power information is stored in a *cell_group* object by the library processing

implemented in Chapter 7. The *cell_group* is a group of cells with equivalent functionality. All cell groups existing in the cell library are stored in a *cell_library* object.

The structure from the elaborated SystemVerilog parser, implemented in Chapter 6, is scanned and a corresponding structure is made containing power information, in the form of a *cell_group* object. This new structure tree is also doing optimisations where more cells follow each other without fanning out. If any sequence of cells correspond to the functionality of a more complex cell, the sequence will be replaced with that cell from the cell library. For instance if an AND2 gate is followed by an OR2 gate, and that OR2 gate only the sequence will be optimised to an ANDOR21 gate if one is present in the cell library. If the AND2 gate output fans out and drives more inputs, however the optimisation will not be done.

In Figure 8.6a the power structure representation of the AND4 gate in Figure 8.2 is shown. The sequences of AND2 gates are combined together and optimised to two AND3 gates.



(a) AND4 in optimised structural representation. **(b)** Optimised AND4 structure tree

Figure 8.6: An AND4 gate as made by the power model generator

The implementation of the switching propagation through the structure tree to yield a power

estimate for each structure tree was deemed too time consuming. A bit more work done on the propagation on switching probabilities can be seen in Chapter 10. The implemented power model generator is a structure tree of cell groups and contain all the necessary information to implement the switching power estimate suggested in Section 8.5.

8.7 Results

The structural representation of the elaborated SystemVerilog and the library information retrieved from the Liberty file have been successfully combined into a tree structure of cells from the cell library.

One structure tree from Module2 is given in Listing 8.1 and the same structure from Module2, after it is processed by the power model generator is given in Listing 8.2. It can be seen that all *select* operators have been replaced with an AO22 cell and that all buffers are removed. In addition the module is able to unite sequences of generic cells corresponding to a sequence of operations that can be done by a more complex cell from the cell library, granted there is no fan-out between the generic cells that are being merged. The first OR gate after the register on line 2 in Listing 8.1 is followed by a NOT gate, in the power structure these are put together into a NOR2 operation.

Another structure tree from the elaborated SystemVerilog parser is shown in Listing 8.3, and can be compared to the power structure tree in Listing 8.4. Here the generators ability to combine cells is once again demonstrated as for instance the sequence on line 11-13 of Listing 8.3, becomes the NOR3 cell on line 6 in Listing 8.4.

Listing 8.1: Structure from the elaborated SV model

```
'reg'
  |'gtech_or2'
  |   |'gtech_not'
  |   |   |'gtech_buf'
  |   |   |   |'select_op'
  |   |   |'gtech_or2'
  |   |   |   |'gtech_or2'
  |   |   |   |   |'reg'
  |   |   |   |   |'reg'
  |'gtech_not'
  |   |'gtech_and2'
  |   |   |'gtech_buf'
  |   |   |   |'select_op'
  |   |   |'gtech_or2'
  |   |   |   |'gtech_or2'
  |   |   |   |   |'reg'
  |   |   |   |   |'reg'
  |   |'gtech_or2'
  |   |   |'gtech_not'
  |   |   |   |'gtech_buf'
  |   |   |   |   |'select_op'
  |   |   |   |'gtech_or2'
  |   |   |   |   |'reg'
  |   |   |   |   |'reg'
  |   |'gtech_or2'
  |   |   |'gtech_not'
  |   |   |   |'gtech_buf'
  |   |   |   |   |'select_op'
  |   |   |   |   |'select_op'
  |   |   |   |   |'select_op'
  |   |   |'gtech_buf'
  |   |   |   |'select_op'
  |   |   |   |'select_op'
  |   |   |   |'select_op'
```

Listing 8.2: Power structure from the power model generator

```
'reg '
  |'nor2 '
  |   |'andor22 '
  |   |'or3 '
  |   |   |'reg '
  |   |   |'reg '
  |'not '
  |   |'and2 '
  |   |   |'andor22 '
  |   |   |'or3 '
  |   |   |   |'reg '
  |   |   |   |'reg '
  |   |'nor2 '
  |   |   |'andor22 '
  |   |   |'or2 '
  |   |   |   |'reg '
  |   |   |   |'reg '
  |   |'or2 '
  |   |   |'not '
  |   |   |   |'andor22 '
  |   |   |   |'andor22 '
  |   |   |   |'andor22 '
  |   |   |''
  |   |   |   |'andor22 '
  |   |   |   |'andor22 '
  |   |   |   |'andor22 '
```

Listing 8.3: Structure from the elaborated SV model

```
'reg'
  |'gtech_or2'
  |  |'gtech_not'
  |  |  |'gtech_and2'
  |  |  |  |'select_op'
  |  |  |  |  |'reg'
  |  |  |  |'gtech_and2'
  |  |  |  |  |'gtech_or2'
  |  |  |  |  |  |'gtech_buf'
  |  |  |  |  |  |  |'select_op'
  |  |  |  |  |  |'gtech_or2'
  |  |  |  |  |  |  |'gtech_or2'
  |  |  |  |  |  |  |  |'gtech_not'
  |  |  |  |  |  |  |  |  |'select_op'
  |  |  |  |  |  |'gtech_not'
  |  |  |  |  |  |  |'gtech_and2'
  |  |  |  |  |  |  |  |'select_op'
  |  |  |  |  |  |  |'gtech_and2'
  |  |  |  |  |  |  |  |'gtech_and2'
  |  |  |  |  |  |  |  |  |'select_op'
  |  |  |  |'gtech_and2'
  |  |  |  |  |'gtech_or2'
  |  |  |  |  |  |'gtech_or2'
  |  |  |  |  |  |  |'gtech_not'
  |  |  |  |  |  |  |  |'select_op'
  |  |  |  |  |'gtech_and2'
  |  |  |  |  |  |'select_op'
  |  |  |  |'gtech_not'
  |  |  |  |  |'gtech_and2'
  |  |  |  |  |  |'gtech_and2'
  |  |  |  |  |  |  |'select_op'
  |  |  |'gtech_or2'
  |  |  |  |'reg'
  |  |  |'gtech_and2'
  |  |  |  |'select_op'
  |  |  |  |  |'reg'
  |'sub_op'
  |  |'select_op'
  |  |  |'reg'
  |'select_op'
  |  |'reg'
```

Listing 8.4: Power structure from the power model generator

```
'reg '
  |'nor2 and2 '
  |  |'andor22 reg '
  |  |'and2 or2 '
  |  |  |'andor22 '
  |  |  |'nor3 andor22 '
  |  |  |'not '
  |  |  |  |'and2 andor22 '
  |  |  |  |'and3 andor22 '
  |  |'and2 '
  |  |  |'nor3 andor22 '
  |  |  |'and2 andor22 '
  |  |  |'not and3 andor22 '
  |  |'or2 reg '
  |  |'and2 andor22 reg '
  |'andor22 reg '
  |'andor22 reg '
```

8.8 Discussion

Some work remains before the power model is ready to be integrated into a power estimation tool, and the remaining work is outlined in Chapter 10, together with suggestions on how the power estimation tool making use of the power model can be implemented. Discussions regarding the implementation done so far is given in the subsections below.

8.8.1 The quality of the cell mapping

Mapping the structure shown in Listing 8.3 to cells from the cell library, one is left with the structure shown in Listing 8.4. As the generic cells are not directly mapped but grouped when possible the cell count is reduced. It was concluded in Section 8.3.1 that reducing the number of logic cells needed would increase the overall accuracy of the representation.

Some mappings done in the power model generator are evidently increasing the accuracy of the estimations, like combining a sequence of AND2, NOT into a NAND2 gate, as this reduces the number of transistors needed by four. In other cases, evaluating whether the cell mapping is bringing the representation closer to the synthesised one, is harder. The AND4 gate in Figure 8.2 is represented as two AND3 gates in the power structure, as shown in Figure 8.6a. Here the original elaborated representation had three AND2 gates. Changing the AND2 gate to an AND3 gate will reduce both the leakage power and the switching power, as the number of cells is reduced from three to two. The difference between the power consumption of AND cells with differing numbers of inputs are not big, as seen in Table 8.1. Further reducing these two AND3 cells to one AND4 cell would be beneficial. This last combination is harder to do as the power structure trees implemented are oblivious of their parents and can only be parsed in one direction. It is necessary to change the structure trees and the cell mapping algorithm if one wishes to increase the accuracy of the estimation further. Then, before deciding which cell from the cell library to map a structure to, one would be able to look at the parents of a generic cell.

When mapping the structural representation to a power-aware structure, the complex cells with many inputs and logic operations will only be chosen where a match to their most complex paths is found. This is discussed in Section 7.2.2, using an ANDOR21 cell as an example. Calculations done there show the difference between the OR2 gate and the input only going through the OR operation in an ANDOR21 gate consumes differentiating switching power.

Another option is to range the library cells by complexity and tag the generic cells with the

library cell of which they are included. This tag can then be changed if a more complex cell is used. Then, the generic OR2 cell that is part of the ANDOR21 operation would know it was part of an ANDOR21 and not just an OR2. If this method is used, it is necessary to keep track of which input pin of a library cell is connected to a structure and to differentiate between the power consumption of different cell inputs.

Such a method would also require a more sophisticated algorithm for mapping the generic cells to the library cells and that the different data inputs are kept track of and their power information stored when parsing the Liberty file.

The choice of removing all generic buffer cells in the power model has also been made. As buffers, when present, consume a significant amount of power, how to determine when buffers will be inferred could be advantageous. In most cases, as observed in Table 6.1, generic buffers are not inferred. A generic buffer cell will be inferred if the cell output has a load capacitance it is not able to drive by itself. Such a load is typically present where large fanouts occur. Buffer cells consume considerable power, and trying to predict their inference is a possibility, rather than assuming they will never be inferred.

8.8.2 Consequences of abstractions

No state retention

The leakage power in cells from the cell library is state-dependent and the implemented system retains no state information. Only the average leakage power of a cell is used. If it is discovered that this has a very negative impact on the accuracy of the estimated leakage power, attempting to predict cell states can be considered. As the leakage power contribution is several magnitudes lower than the switching power it was argued that estimating leakage power with a spatial and temporal accuracy may not be necessary to get good power estimates.

Adding state information can be done by introducing a probability of how likely a gate is to be in a specific state. Calculating that probability is hard unless fall power is also considered, as the state of a cell depends on the cell input values. When only considering the signal rise times, and adding the contribution from rise- and fall power together, the time between rising and falling transitions is lost. This means one loses the information on how long the cell has been in an open state versus a closed state.

When storing state probabilities in objects, simplifying the power model will become harder as the power consumption will be dependent on variables in each structure object rather

than a constant power value. An option could be to try to estimate some state probabilities in cells as the switching probabilities are calculated, as this could remove the need to include the fall power contribution by itself while still increasing the accuracy of power estimates.

Fall power and rise power combined

Abstracting away the fall power by combining it with the rise power in the implementation in Chapter 7 allows for much faster estimation later on, by only needing half the amount of data, but temporal information is lost. The estimates will have a lower cycle accuracy as fall power can contribute more than the rise power to power consumption in many cases, depending on the cell. Knowing when a cell output is probable to rise, but not when it is likely to fall also makes state prediction harder if it is later decided that this is necessary.

8.8.3 Evaluating the power model

The power model generated has been visually evaluated in Chapter 8.7, but more evaluation is needed to determine the quality of the model. An option to do such an evaluation would be to create a power estimation tool using the power model, and compare the power estimates yielded to those of state-of-the-art tools for power estimation, both at the RTL and the gate-level. To get the power model to work with a power estimation tool the library cell implementation of the arithmetic operations comparisons and shifts must first be made, and the signal propagation through structure trees must be implemented. After that is done, a power estimation tool making use of the power model must be made. When implementing the power estimation tool, the register- and input switching must be monitored and related to its respective structure tree. Each cycle the contribution to power can be added together, and give a cycle-by-cycle estimate, or transitions within a time frame can be counted and added up to a power estimate for that frame.

It would also be useful to use smaller and simpler designs than those used in this project to be able to investigate the relationship between the elaborated- and the synthesised netlist further, and see how the model made compares to this. These investigations could also be useful when developing the internals of the arithmetic cells, comparators and shifters as one could see how such generic cells are treated by the synthesis tool in different cases.

After evaluating the power model, it would also be possible to determine if there are systematic errors. These can be atoned for by adjusting the resulting power values, and the

model can be improved.

8.8.4 Improvements to consider

Improved optimisation

An improvement that could have a positive effect on power estimates is to add more optimisation to the power model generator. It is currently only looking for sequences of cells corresponding to larger, more complex cells. If it were able to reduce the amount of logic needed in other ways, we would get closer to the synthesised representation.

If parents of nodes in the structure tree were known, the structure would be able to do optimisations also in that direction. For an ANDOR21 gate, with logic function $(A1 * A2) + B$ this would mean the B input knows it is an input to the ANDOR21 gate and not to an OR2 gate. Such optimisations would lower the power consumption and bring the representation closer to the synthesised one. For an AND4 gate, such as the one in Figure 8.2a this would mean that the structure tree could transform it to an AND4 gate if one is available in the cell library, rather than the two AND3 gates it currently becomes.

Other kinds of optimisations could also be considered. If two inverters are placed after each other, there is a possibility they could be removed, and operation reordering may also reduce the number of logic. Such optimisations could be done by viewing the gates as logic operations and using boolean algebra to simplify the expressions. How effective such optimisations are in bringing the design closer to its synthesised representation, depends on the optimisations done during the elaboration of the HDL.

Constant signals

Another factor the implementation in this project is not considering is constant values. If a signal value is set to 1 or 0, it is not necessary to consider it a signal that may have either value. For instance, if a signal set to 1 enters an AND2 gate, the other input will always be propagated through, and the AND2 gate can be removed. Similar conclusions can be reached with a signal being set to 0.

Alternatives if generic cells do not exist

In the implementation it was assumed that *multiplexers*, *AND2*, *OR2*, *XOR2* and *NOT* cells were present in the cell library as equivalents to the generic logic cells. Such an assumption may not always be the correct, and a possibility of replacing some of these cells

with logic equivalents should be implemented, to make the power modeling available for cell libraries diverging from this assumption.

Not giving all registers a power structure

As the model is now **all** registers become the top structure of a structure tree. For modules with many registers, this may result in too many structure trees and a more fine-grained power estimation than needed. If the system is modified such that which registers have their own structures can be chosen by the user instead of taking all registers into account, the module would have modifiable granularity. The propagation of switching probabilities through registers will have to be implemented for this method, as registers can now be part of structure trees. Otherwise only minor changes are required in the elaborated SystemVerilog parser.

Using switching probability to adjust leakage power

The difference in leakage power in different states of a cell is significant, as seen in Table 7.1. An attempt to adjust for this could be added after the switching probabilities are calculated. If a cell output is likely to be high as a result of a register or input rising, the leakage power increase after a transition changing its state happen. This could be taken into account.

Still, when not considering the fall power contribution by itself when the state changes back to a state with low leakage is unknown. It could be that by assuming a state for a given amount of cycles after a rising output, or by calculating some state duration from the activity data higher accuracy can be reached, than by simply using the average leakage values.

8.8.5 The accuracy/speed trade-off

Making the power estimates more accurate and bringing the representation closer to the IC it will become is advantageous, but solely increasing the accuracy of RTL power estimation is not the goal of this project. Some accuracy can be sacrificed to yield fast power estimates, as the estimation speed is also critical. Finding a balance between execution speed and estimation accuracy is paramount.

Separating the structure trees and modeling the power without state-dependency are important steps toward fast power estimation. Both abstractions seem promising, but it remains to be seen whether they are accurate enough. Steps towards improving the model

accuracy can be taken if it proves to be inaccurate. For instance using the predicted activity of structure trees, (depending on the register- or input bit switching), to partly give the leakage power state dependency, or improving the mapping from the generic cells to the cells from the cell library, to better imitate the power characteristics of the synthesised design.

If the power estimation yields systematic errors, such as estimating a consequently too high switching probability, they can easily be adjusted for.

9 Conclusion

A power estimation flow for top-down power estimation at the RTL have been chosen. Parts of this flow have been implemented to make a power model generator. The power model generator makes use of an elaborated SystemVerilog representation of the design, and process the format to yield a node tree for each input or register bit. This node tree representing the design structure is then combined with information from the cell library to be used when manufacturing the IC. The Liberty file is parsed and information regarding all the cells available in the cell library is organised and stored in a *cell_library* object. When combining the structural information and the library information some optimisation is done on the structure tree depending on the available cells in the cell library. The resulting structure trees contain the information needed to estimate the power consumption of the design, and the representation has similarities to the netlist.

The implementation shows promise in finding a power-aware representation of a design without first synthesising it. By parsing the Liberty file of a cell library, realistic power values are obtained and integrated into the structure tree representing the design. The grouping and mapping from generic cells to library cells ensures a reduction in the number of cells inferred, which is shown to have a positive effect on the accuracy of power estimates and brings the design representation closer to that of the synthesised design.

Thorough evaluation of the power model is needed to determine whether the speed/accuracy trade-off is satisfactory. Such an evaluation would be more easily conducted after integrating the model into the suggested power estimation flow. The speed and accuracy of power estimations done using the implemented power model can then be compared to power estimates from state-of-the-art tools for power estimation at the RTL and the gate-level.

Some work remains in the implementation. Power-aware replacements for the arithmetic generic cells in the structural representation will have to be made in the power model generator. Replacements for shifters and comparators are also needed. Lastly, it is necessary to combine the power information in each node together into one power value representing the structure tree for a rising input or register bit. To do so, the propagation of the signal have to be estimated. In addition to this the leakage power should be added together for

all the cell objects present to represent the leakage per cycle.

10 Future work

10.1 Finishing the power model

Some work remains for the power model to be finished. That work is listed below:

- **Compose arithmetic cells from cell library**
  The generic, arithmetic cells need an equivalent. The amount of logic one bit from a signal with a certain width must go through needs to be estimated together with the switching- and leakage power of such propagation. This must be done for
  - Multipliers
  - Dividers
  - Adders
  - Subtractors
- **Make equivalent for other complex generic cells**
  The shifter and comparator generic cells also need an equivalent consisting of cells from the cell library with corresponding power data.
- **Make estimate for each structure tree**
  The switching activity needs to be propagated through the structure tree for the corresponding input- or register and a power estimate for each tree must be made in addition to a leakage power per cycle estimate.

10.2 Implementing a power estimation tool

The power model, when finished, must be integrated into a power estimation tool to yield a power estimate. Such a tool must be made. The key points in developing such a tool are given below:

- **Connect registers and inputs to structure trees**
  The registers and inputs corresponding to the structure trees in the power model must be found and "connected".
- **Use activity data to calculate switching data**
  The activity data from the simulation must be used to add up switching power consumptions from the structure trees.
- **Calculate leakage power**
  The structures in the structure trees must have their leakage power added together to get a leakage-per-cycle estimate.

Bibliography

[1] I. S. IEEE and T. Organiztion. https://ieee-isto.org/member_programs/liberty-technical-advisory-board/, 2020. [Online; accessed 27-May-2020].

[2] S. Palnitkar, *Verilog HDL: A Guide to Digital Design and Synthesis, Second Edition*. USA: Prentice Hall PTR, 2nd ed., 2003.

[3] J. Rabaey, *Low Power Design Essentials*. Integrated Circuits and Systems, Springer US, 2009.

[4] N. Weste and D. Harris, *CMOS VLSI Design: A Circuits and Systems Perspective*. USA: Addison-Wesley Publishing Company, 4th ed., 2010.

[5] J. Rabaey, "Introduction," in *Low Power Design Essentials*, pp. 1–23, Boston, MA: Springer US, 2009. Series Title: Integrated Circuits and Systems.

[6] R. Dennard, F. Gaensslen, H.-N. Yu, V. Rideout, E. Bassous, and A. LeBlanc, "Design of ion-implanted MOSFET's with very small physical dimensions," *IEEE Journal of Solid-State Circuits*, vol. 9, pp. 256–268, Oct. 1974. Conference Name: IEEE Journal of Solid-State Circuits.

[7] Design Automation Committee, "IEEE Standard for Power Modeling to Enable System-Level Analysis," *IEEE Std 2416-2019*, pp. 1–63, July 2019.

[8] L. Zhong, S. Ravi, A. Raghunathan, and N. Jha, "Power estimation for cycle-accurate functional descriptions of hardware," in *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pp. 668–675, Nov. 2004. ISSN: 1092-3152.

[9] L. Zhong, S. Ravi, A. Raghunathan, and N. Jha, "RTL-Aware Cycle-Accurate Functional Power Estimation," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 25, pp. 2103–2117, Oct. 2006.

[10] D. Lee and A. Gerstlauer, "Learning-Based, Fine-Grain Power Modeling of System-Level Hardware IPs," *ACM Transactions on Design Automation of Electronic Systems (TODAES)*, vol. 23, pp. 30:1–30:25, Feb. 2018.

[11] S. Ravi, A. Raghunathan, and S. Chakradhar, "Efficient RTL power estimation for large designs," in *16th International Conference on VLSI Design, 2003. Proceedings.*, pp. 431–439, Jan. 2003. ISSN: 1063-9667.

[12] S. Gupta and F. Najm, "Energy and peak-current per-cycle estimation at RTL," *IEEE Transactions on Very Large Scale Integration (VLSI) Systems*, vol. 11, pp. 525–537, Aug. 2003.

[13] H. Mehta, R. M. Owens, and M. J. Irwin, "Energy characterization based on clustering," in *33rd Design Automation Conference Proceedings, 1996*, pp. 702–707, June 1996.

[14] R. Zafalon, M. Rossello, E. Macii, and M. Poncino, "Power macromodeling for a high quality RT-level power estimation," in *Proceedings IEEE 2000 First International Symposium on Quality Electronic Design (Cat. No. PR00525)*, pp. 59–63, Mar. 2000. ISSN: null.

[15] K. Buyuksahin and F. Najm, "Early power estimation for VLSI circuits," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 24, pp. 1076–1088, July 2005.

[16] S. Sambamurthy, J. A. Abraham, and R. S. Tupuri, "A Robust Top-Down Dynamic Power Estimation Methodology for Delay Constrained Register Transfer Level Sequential Circuits," in *21st International Conference on VLSI Design (VLSID 2008)*, pp. 521–526, Jan. 2008. ISSN: 2380-6923.

[17] Ansys. https://www.ansys.com/products/semiconductors/ansys-powerartist, 2019. [Online; accessed 22-June-2020].

[18] Synopsys. https://www.synopsys.com/verification/static-and-formal-verification/spyglass/spyglass-power.html, 2019. [Online; accessed 22-June-2020].

[19] Mentor. https://www.mentor.com/hls-lp/powerpro-rtl-low-power/power-estimation, 2019. [Online; accessed 22-June-2020].

[20] Cadence. https://www.cadence.com/en_US/home/tools/digital-design-and-signoff/power-analysis/joules-rtl-power-solution.html, 2019. [Online; accessed 22-June-2020].

[21] E. T. Bygland. https://github.com/emblatbyg/thesis, 2020. [].

[22] M. Nemani and F. N. Najm, "Towards a high-level power estimation capability [digital ICs]," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 15, pp. 588–598, June 1996.

A Technical implementation of the elaborated SystemVerilog parser

Figure A.1 shows the function hierarchy of the elaborated SV parser. In Table A.1 a function overview can be seen, while A.2 shows the helper functions of the parser.

The Table A.3 gives a complete overview of the elaborated SV parser classes, class variables and procedures can be seen.

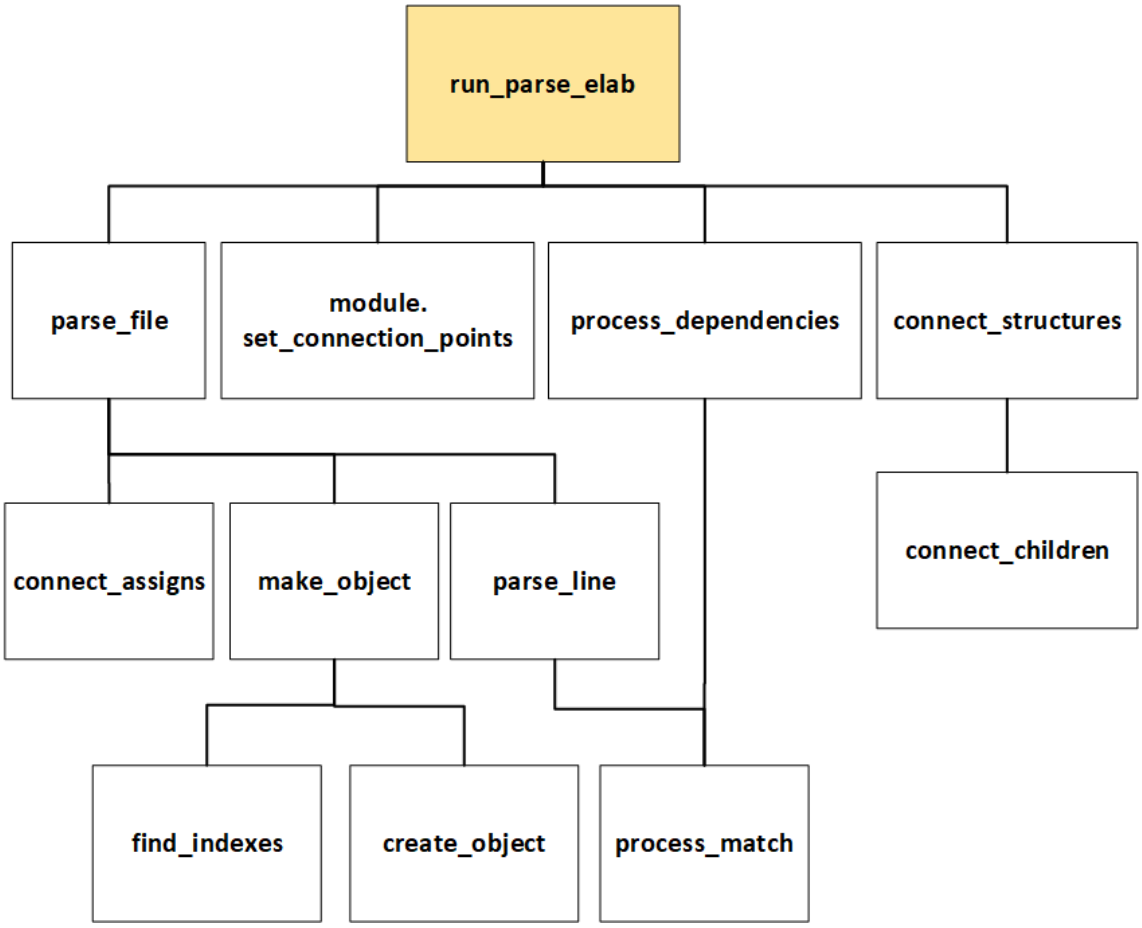


Figure A.1: The function hierarchy of the elaborated SV parser. Functions at the same level are called from left to right.

Table A.1: Overview of the functions in the elaborated SV parser.

Function	**Description**
parse_file(filename)	Parses the elaborated SystemVerilog file object by object
module. set_connection_points()	reads the description on how to connect the inputs and outputs of a module from the module declaration stored in the module object
process_dependencies()	goes through the dependencies of a module and connects them to the rest of the module
connect_structures (top_module)	Goes through all inputs of the top module and all registers, creating their structure trees
connect_assigns ()	Sets the two signals assigned to each other equal each other
make_object (line, object_type)	processes the object line sent in from parse_file
create_object (object_type)	calls the init function of the type specified by make_object
parse_line (line, object_handle)	goes through the object internals and set their connection points
process_match (match, object_handle, connection_type, port_name)	connects one connection point to an input, output or wire
connect_children (parent, i1, i2)	make a node tree including everything connected to a specified parent (input signal or register output)

Table A.2: Helper functions for the elaborated SV parser

Function	**Description**
search_list(list, object_name)	looks for object with object_name in the list of objects
find_module(line)	looks for regex matching start of a module in line
find_endmodule(line)	looks for regex matching end of module in line
empty_global_lists()	empties global lists removing a module environment
set_global_lists(module)	setting global lists to match environment of module
connect_nodes(n1, n2)	set two nodes to equal each other
hline find_indexes(string)	looks for an index declaration in the string

Table A.3: Overview of classes in the elaborated SV parser and their variables and procedures.

Class	Variables	Procedures
register	id name Q QN clear preset next_state clocked_on data_in enable synch_clear sync_preset synch_toggle synch_enable output_nodes_q output_nodes_qn	__init__(self) initialize object, returns object handle
gtech_or2 gtech_xor2 gtech_and2	id name A B Z output_nodes	*__init__(self)* initialize object, returns object handle
gtech_not gtech_buf	id name A Z output_nodes	*__init__(self)* initialize object, returns object handle
shift_op b_shift_op	id name SH A	__init__(self) initialize object, returns object handle

	Z output_nodes	
shift_add_op	id name SH output_nodes	*__init__(self)* initialize object, returns object handle
comp_op	id name A B output_nodes	__init__(self) initialize object, returns object handle
add_op sub_op mult_op div_op	id name A B Z output_nodes	*__init__(self)* initialize object, returns object handle
mux_op	id name D S Z datawidth output_nodes	*__init__(self)* initialize object, returns object handle
select_op	id name D S Z selectwidth datawidth output_nodes	*__init__(self)* initialize object, returns object handle
input_obj	id name connection_nodes	*__init__(self)* initialize object, returns object handle

	width depth widthoffset	*add_node_input_connection(self,i1,i2,l,index_type)* add a connection to the node specified by indexes and index_type
output_obj	id name connection_nodes width depth widthoffset	*__init__(self)* initialize object, returns object handle *add_node_input_connection(self,i1,i2,l,index_type)* add a connection to the node specified by indexes and index_type *add_node_output_connection(self,i1,i2,l,index_type,k)* add a connection to the node specified by indexes and index_type
connection	id name connection_nodes width depth widthoffset	*__init__(self)* initialize object, returns object handle *init_connection_nodes(self)* initialize an array of node objects corresponding to the size of the connection *add_node_input_connection(self,i1,i2,l,index_type)* add a connection to the node specified by indexes and index_type *add_node_output_connection(self,i1,i2,l,index_type,k)* add a connection to the node specified by indexes and index_type
node	id connected_inputs connected_outputs i1 i2	*__init__(self)* initialize object, returns object handle *add_input_connection(self,l)* add connection to connected_inputs *add_output_connection(self,l,J)*

		add connection to connected_outputs and adds node to output_nodes of connected object
dependency	id name modulename module_handle connections	*__init__(self)* initialize object, returns object handle *add_connections(self,list)* add a connection to connections
assign	id lhs rhs lhs_i1 lhs_i2 rhs_i1 rhs_i2	*__init__(self)* initialize object, returns object handle
module	name connection_string connection_points regs nots bufs and2s or2s muxes selects connects inputs outputs dependencies shifters comparators xor2s multipliers subtractors b_shifters adders	*__init__(self)* initialize object, returns object handle *set_lists(self)* set module environment lists *set_connection_points(self)* set the connection_points variable from the connection_string

	shift_adders divisors assigns	
structure	id children represented- _object_handle	*__init__(self)* initialize object, returns object handle *add_child(self, child)* add child structure to children *print(self)* prints node structure

B Technical implementation of the liberty parser

In Figure B.1 the function hierarchy of parsing the Liberty file and storing the power data in JSON objects can be seen. In figure B.2 the function hierarchy of reading out the cells from the JSON lines and putting them in a cell library can be seen.

Table B.1 shows the functions of the system processing the data from the liberty file, extracting the relevant power information. Table B.2 shows a class overview of the liberty data retrieving system.

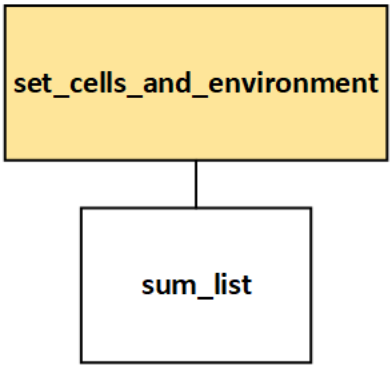


Figure B.1: The function hierarchy of the liberty parser

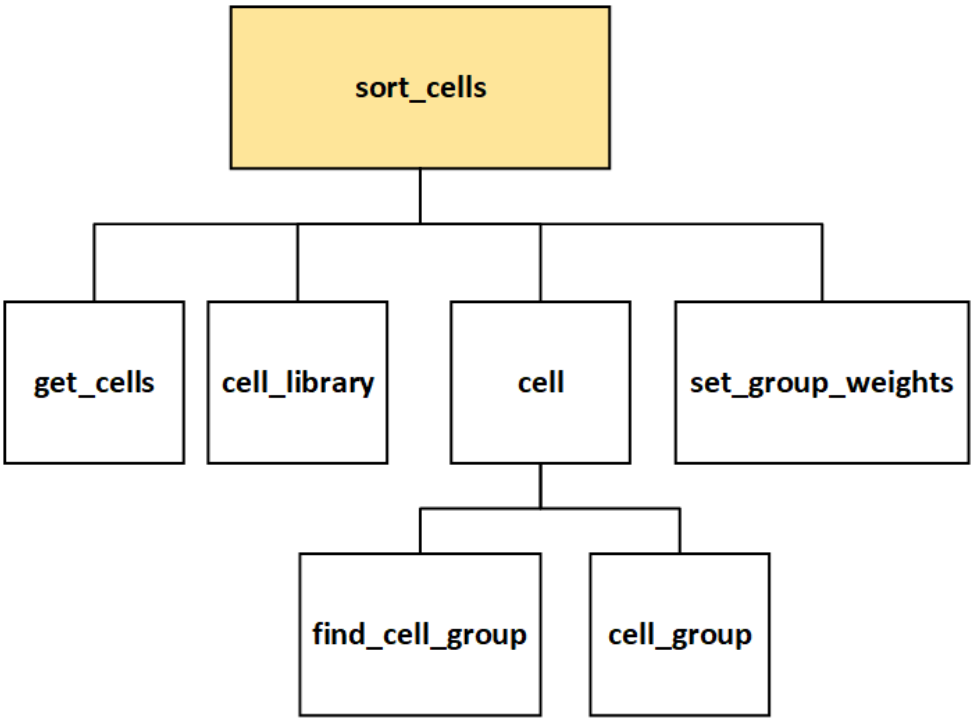


Figure B.2: The function hierarchy of the liberty power data retrieving

Table B.1: functions for processing the liberty file information

Function	**Description**
set_cells_and_environment()	parses the liberty file and retrieves the cell library information. Stores it in a JSON line. Also counts the occurence of each cell in a calibration netlist.
get_cells(filename)	Reads cell JSON lines from a file and returns a list of these lines
sort_cells(path)	makes cell object from list of cells, makes a library object containing the cells. Calibrates the library.
get_dict_N(N)	get dictionary with regexes and synthetic gate sequences corresponding to cells with N inputs
count_ocurrence(word)	count occurence of word in calibration netlist

Table B.2: Class overview for processing the liberty file information

Class	**Variables**	**Procedures**
cell	footprint name leakage_power synthetic_gate_list def_list input_pins output_pins	__init__(self, def_list, cell_lib) initialise object adds it to cell group. Adds group to cell library if new group is made
cell_group	matching_key synthetic_gate_list cells cellcounts weights	__init__(self, sequence, matching_key) initialise object append_cell(self, cell) add cell to cells, and calibration count to cellcounts get_weights() set weights based on cellcounts
cell_library	cells_6 cells_5 cells_4 cells_3 cells_2 cells_1 muxes regs group_lists	__init__(self) initialise object get_list(self, N) get list corresponding to cells with N inputs find_cell_group(self, list, group_key) look for cell group with matching_key equal to group_key set_group_weights(self) call set_weights for all cell_groups in library print_available_cells(self) print available cells

C Technical implementation of the power model

Figure C.1 describes the function hierarchy of the power model generator and Table C.1 describes the functions in greater detail. Table C.2 describes the two classes in the power modeling system. The value class is used to make a mutable number, while the power_structure class make up the nodes of a node tree.

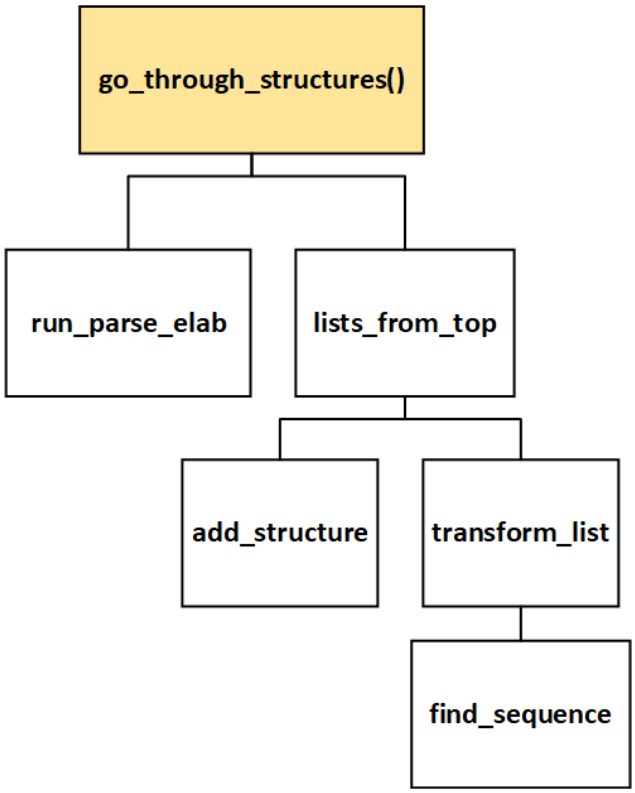


Figure C.1: Function hierarchy for the power model implementation

Table C.1: functions for making the power model

Function	**Description**
go_through_structures()	Iterate through structures from the parsed elaborated SystemVerilog and create power structure equivalents
run_parse_elab(filename)	Called after importing the elaborated SystemVerilog parser. Runs the parser and returns the structure trees.
lists_from_top(s, power_s)	recursively goes through all nodes in structure tree from parse_elab(). Makes a parallell power structure tree.
transform_list(cellLib, l)	Transforms list of synthetic cells to list of cells from cell library
find_sequence(to_find, cell_group)	see if cell group's synthetic gate sequence has any match(es) in list. Return positions of occurence(s) if that is the case.

Table C.2: Class overview for the power model

Class	**Variables**	**Procedures**
power_structure	name cell_lib_list structural_rep_list children parent	__init__(self, def_list, cell_lib) initialise object
value	i	

D Code implemented in Chapter 6

```
# File for parsing .elab files made by synopsys compiler

import numpy as np
import re
import pandas as pd
import sys

#list of module objects
modules = []
#list of structure trees
top_level_parents = []

#lists setting module environment
regs            = np.array([])
nots            = np.array([])
bufs            = np.array([])
and2s           = np.array([])
or2s            = np.array([])
muxes           = np.array([])
selects         = np.array([])
connects        = np.array([])
inputs          = np.array([])
outputs         = np.array([])
dependencies    = np.array([])
shifters        = np.array([])
comparators     = np.array([])
xor2s           = np.array([])
multipliers     = np.array([])
subtractors     = np.array([])
b_shifters      = np.array([])
adders          = np.array([])
shift_adders    = np.array([])
divisors        = np.array([])
assigns         = np.array([])

#gate counts
reg_n       = 0
not_n       = 0
buf_n       = 0
and2_n      = 0
or2_n       = 0
mux_n       = 0
select_n    = 0
shift_n     = 0
comp_n      = 0
xor2_n      = 0
mult_n      = 0
sub_n       = 0
b_shift_n   = 0
add_n       = 0
shift_add_n = 0
div_n       = 0

#empty global lists relating to a module environment
def empty_global_lists():
    global regs
    global nots
    global bufs
    global and2s
    global or2s
    global muxes
    global selects
    global connects
    global inputs
    global outputs
    global dependencies
    global shifters
    global comparators
    global xor2s
    global multipliers
    global subtractors
    global b_shifters
    global adders
    global shift_adders
```

```
    global divisors
    global assigns

    regs         = np.array([])
    nots         = np.array([])
    bufs         = np.array([])
    and2s        = np.array([])
    or2s         = np.array([])
    muxes        = np.array([])
    selects      = np.array([])
    connects     = np.array([])
    inputs       = np.array([])
    outputs      = np.array([])
    dependencies = np.array([])
    shifters     = np.array([])
    comparators  = np.array([])
    xor2s        = np.array([])
    multipliers  = np.array([])
    subtractors  = np.array([])
    b_shifters   = np.array([])
    adders       = np.array([])
    shift_adders = np.array([])
    divisors     = np.array([])
    assigns      = np.array([])

#set global lists relating to a module environment
def set_global_lists(module):
    global regs
    global nots
    global bufs
    global and2s
    global or2s
    global muxes
    global selects
    global connects
    global inputs
    global outputs
    global dependencies
    global shifters
    global comparators
    global xor2s
    global multipliers
    global subtractors
    global b_shifters
    global adders
    global shift_adders
    global divisors
    global assigns

    regs         = module.regs
    nots         = module.nots
    bufs         = module.bufs
    and2s        = module.and2s
    or2s         = module.or2s
    muxes        = module.muxes
    selects      = module.selects
    connects     = module.connects
    inputs       = module.inputs
    outputs      = module.outputs
    dependencies = module.dependencies
    shifters     = module.shifters
    comparators  = module.comparators
    xor2s        = module.xor2s
    multipliers  = module.multipliers
    subtractors  = module.subtractors
    b_shifters   = module.b_shifters
    adders       = module.adders
    shift_adders = module.shift_adders
    divisors     = module.divisors
    assigns      = module.assigns

#go through elaborated systemverilog file line by line
def parse_file(path):
    #make lists and treat lists to make objs later
    print ("Parsing file:" + path )

    #SEARCH FOR START OF OBJECT
    object_handle   = 'false'
    objectstring    = ""
    bitwidth = 0
    modulename = ''
    in_module = False
    moduleline = ''
    with open(path, 'r') as svfile:
        line = svfile.readline()
        linenum = 1
        while line:
            #search for start of module
            if (in_module):
                #handle end of module
```

```
            if (find_endmodule(line)):
                global modules
                in_module = False
                module_handle = module(modulename)
                module_handle.set_lists()
                module_handle.connection_point_string = moduleline
                connect_assigns()
                empty_global_lists()
                modules.append(module_handle)
            #search for start of object
            else:
                key, match = parse_line(line, 'false')
        else:
            #search for start of module
            modulename, in_module = find_module(line)
            #print("Started module: "+modulename)
            match = False
            module_declaration_ongoing = True
            #get lines until end of module declaration and catch things in regex
            while module_declaration_ongoing:
                for k, rx in rx_dict_end.items():
                        objectstring = objectstring+line
                        #print("objectstring to search for end: \n\t"+objectstring)
                        match = rx.search(line)
                        if match:
                            # found end, leave while loop
                            module_declaration_ongoing = False
                            moduleline = objectstring
                            #print(moduleline)
                        else:
                            # read new line until end is found
                            line = svfile.readline()
                            linenum = linenum +1
            objectstring = ''
            match = False

        if match and in_module:
            #if connection split items by comma and make objects of all
            offset = 0
            if match.group(1) == 'input' or match.group(1) == 'output' or match.group(1) == 'wire':

                if match.group(2) != None:
                    bitwidth = int(match.group(2)) - int(match.group(3)) + 1
                    line = " ".join(line.split()[2:])
                    offset = int(match.group(3))
                else:
                    bitwidth = None
                    line = " ".join(line.split()[1:])
            #else prepare for finding attributes of single object
            else:
                #object_handle = make_object(match.group(1), key)
                if(key == 'dep'):
                    object_handle = make_object(match.group(2), key)
                    object_handle.modulename = match.group(1)
                    #print("made dep: "+match.group(2)+" of module "+match.group(1))
                else:
                    object_handle = make_object(match.group(1), key)
            # while taking in list of equally attributed connections or single item
            # get new lines until all of object(s) is in one string
            in_object = True
            while in_object:
                #merge lines until end of regster is found
                #print ("looking for end of object")
                for k, rx in rx_dict_end.items():
                    objectstring = objectstring+line
                    #print("objectstring to search for end: \n\t"+objectstring)
                    match = rx.search(line)
                    if match:
                        # found end, leave while loop
                        in_object = False
                    else:
                        # read new line until end is found
                        line = svfile.readline()
```

```
                            linenum = linenum +1

                #make single object or make several connect objects
                objectstring = "".join(objectstring.split())
                if (key == 'input' or key == 'output' or key == 'wire'):
                    objectstring = objectstring.strip(";")

                    objectstring = objectstring.translate({ord(i): None for i in '}{ '})

                    objectlist = objectstring.split(',')
                    for name in objectlist:
                        object_handle = make_object(name, key)
                        #if (object_handle != None):
                        if (bitwidth != None):
                            object_handle.width = bitwidth
                            object_handle.widthoffset = offset
                            object_handle.init_connection_nodes()
                        #print ("made new connection object with name "+name+"\nand width "+ str(bitwidth))
                else:
                    # add connection node info to object
                    parse_line(objectstring, object_handle)
                # when done with an object, empty object string
                objectstring = ""

            line = svfile.readline()
            linenum = linenum + 1

# create object
def make_object(line, name):
    #print("Making object: "+name)

    line = line.translate({ord(i): None for i in '}{\ '})

    i1, i2, object_name, index_type = find_indexes(line)
    if ((i1 != -1) or (i2 != -1)) and name != 'register' and name != 'assign':
        object_handle = None
        foundbool = False
        if name == 'input':
            object_handle, foundbool = search_list(inputs, object_name)
        elif name == 'output':
            object_handle, foundbool = search_list(outputs, object_name)
        elif name == 'wire':
            object_handle, foundbool = search_list(connects, object_name)
        if (foundbool):
            #print("Found object in already existing connetion object")
            if (object_handle.width <= i1): object_handle.width = i1+1
            if (object_handle.depth <= i2): object_handle.depth = i2+1

        else:
            object_handle = create_object(name)
            object_handle.name = object_name
            if i1 != -1: object_handle.width = i1+1
            if i2 != -1: object_handle.depth = i2+1
            object_handle.init_connection_nodes()
    else:
        object_handle = create_object(name)
        object_handle.name = line
        if(name == 'input' or name == 'output' or name == 'wire'):
            object_handle.init_connection_nodes()
        if(name == 'assign'):
            #print("making fancy assign")
            object_handle.i1 = i1
            if i2 != -1: object_handle.i2 = i2
            foundbool = False
            connected_handle, foundbool = search_list(outputs, object_name)
            if foundbool != True:
                connected_handle, foundbool = search_list(inputs, object_name)
                if foundbool != True:
                    connected_handle, foundbool = search_list(connects, object_name)
            if foundbool:
                object_handle.lhs = connected_handle

    return object_handle
```

```
311
312 #looks for indexes at end of string, returns i1, i2, str(w/o)
    indexes
313 def find_indexes(string):
314     i1 = -1
315     i2 = -1
316     index_type = ''
317     indexes = re.compile(r"(?:\[(\d{1,4})\])(?:\[(\d{1,4})\])?$"
    )
318     slices = re.compile(r"(?:\[(\d{1,4})\:(\d{1,4})\])$")
319     indexfind = indexes.search(string)
320     newline = indexes.sub("",string)
321     if indexfind != None:
322         index_type  = 'index'
323         if indexfind.group(1) != None:
324             i1 = int(indexfind.group(1))
325             if indexfind.group(2) != None:
326                 i2 = int(indexfind.group(2))
327     else:
328         slicefind = slices.search(string)
329         if slicefind != None:
330             index_type = 'slice'
331             i1 = int(slicefind.group(1))
332             i2 = int(slicefind.group(2))
333             newline = slices.sub("", string)
334             #print("found a slice")
335     return i1, i2, newline, index_type
336
337 #create an object of a class specified by objectname
338 def create_object(objectname):
339     objectselect = {
340         'wire'       : connection,
341         'input'      : input_obj,
342         'output'     : output_obj,
343         'SELECT_OP'  : select_op,
344         'MUX_OP'     : mux_op,
345         'GTECH_NOT'  : gtech_not,
346         'GTECH_BUF'  : gtech_buf,
347         'GTECH_AND2' : gtech_and2,
348         'GTECH_OR2'  : gtech_or2,
349         'GTECH_XOR2' : gtech_xor2,
350         'register'   : register,
351         'dep'        : dependency,
352         'COMP_OP'    : comp_op,
353         'SHIFT_OP'   : shift_op,
354         'SUB_OP'     : sub_op,
355         'ADD_OP'     : add_op,
356         'MULT_OP'    : mult_op,
357         'DIV_OP'     : div_op,
358         'B_SHIFT_OP'    : b_shift_op,
359         'SHIFT_ADD_OP' : shift_add_op,
360         'DIV_OP'     : div_op,
361         'assign'     : assign
362
363     }
364     #get function
365     #print("making object: "+objectname)
366     func = objectselect.get(objectname, lambda: None)
367     if func == None:
368         print("found no object with objectname: "+str(objectname
    ))
369         return None
370     else:
371         #print("lookup successful, func = "+ str(func))
372         retval = func()
373         return retval
374
375 #return true if module declaration is on line
376 def find_module(line):
377     module_start = re.compile(r"module\s+(\S+)\s?\(")
378     match = module_start.search(line)
379     if (match):
380         return match.group(1), True
381     else:
382         return None, False
383
384 #return true if line contains endmodule
385 def find_endmodule(line):
386     module_end = re.compile(r"endmodule")
387     match = module_end.search(line)
388     if (match):
389         return True
390     else:
391         return False
392
393 #look for object with name objectname in a list of objects.
    return handle if match, None otherwise
394 def find_object(objectlist, objectname):
395     for i in range(0, len(objectlist)-1):
396         if objectlist[i].name == objectname:
```

```
            return objectlist[i]
    return None

# parse one line, looking for start of object or internal parameters of object
def parse_line(line, object_handle):
    # print ("parsing line: \n"+ line + "\n in object:\n" + str(object_handle))
    key = ""
    match = ""
    if (object_handle == 'false'):
        for key, rx in rx_dict_start.items():
            #look for start of object to determine object type
            match = rx.search(line)
            if match:
                return key, match
    #look for connect objects in non-connect objects?
    elif (object_handle.id == 'reg'):
        #look for objects inside register dict and end
        for key, rx in rx_dict_reg.items():
            match = rx.search(line)
            #print("line: " + line)
            #print ("found match for: " + key +" group captured: " +  match.group(1) )
            if (key == 'clear'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.clear = match.group(1)
            elif (key == 'preset'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.preset = match.group(1)
            elif (key == 'next_state'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.next_state = match.group(1)
            elif (key == 'clocked_on'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.clocked_on = match.group(1)
            elif (key == 'data_in'):
                process_match([match.group(1)], object_handle, 'input', key)
                object_handle.data_in = match.group(1)
            elif (key == 'enable'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.enable = match.group(1)
            elif (key == 'Q'):
                process_match([match.group(1)], object_handle, 'output', key)
                object_handle.Q = match.group(1)
            elif (key == 'QN'):
                process_match([match.group(1)], object_handle, 'output', key)
                object_handle.QN = match.group(1)
            elif (key == 'synch_clear'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.synch_clear = match.group(1)
            elif (key == 'synch_preset'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.synch_preset = match.group(1)
            elif (key == 'synch_toggle'):
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.synch_toggle = match.group(1)
            elif key == 'synch_enable':
                process_match([match.group(1)], object_handle, 'control', key)
                object_handle.synch_enable = match.group(1)
            else:
                print ("no matching key: " + key + " in register " + object_handle)
        #look for end
        for key, rx in rx_dict_end.items():
            match = rx.search(line)
    elif object_handle.id == 'gtech_or2' or object_handle.id == 'gtech_and2' or object_handle.id == 'gtech_xor2':
        for key, rx in rx_dict_AND2.items():
            match = rx.search(line)
            if (key == 'A' and match):
                process_match([match.group(1)], object_handle, 'input', key)
                object_handle.A = match.group(1)
            elif (key == 'B' and match):
                process_match([match.group(1)], object_handle, 'input', key)
                object_handle.B = match.group(1)
```

```
            elif (key == 'Z' and match):
                process_match([match.group(1)], object_handle, 'output', key)
                object_handle.Z = match.group(1)
            else:
                print ("\nNo attribute of object " + str(object_handle) + " matches key " + key)
                print("from line: "+line+"\n")
        for key, rx in rx_dict_end.items():
            match = rx.search(line)
            if (match == False):
                print ("looked for end in "+ object_handle + " could not find it..." )
    elif object_handle.id == 'gtech_not' or object_handle.id == 'gtech_buf':
        for key, rx in rx_dict_BUF.items():
            match = rx.search(line)
            if (key == 'A' and match):
                process_match([match.group(1)], object_handle, 'input', key)
                object_handle.A = match.group(1)
            elif (key == 'Z' and match):
                process_match([match.group(1)], object_handle, 'output', key)
                object_handle.Z = match.group(1)
            else:
                print ("No attribute of object " + str(object_handle) + " matches key " + key)
            #look for end
        for key, rx in rx_dict_end.items():
            match = rx.search(line)
            if (match == False):
                print ("looked for end in "+ str(object_handle) + " could not find it..." )
                return key, False
        return key, match
    elif object_handle.id == 'mux_op':
        for key, rx in rx_dict_MUX.items():
            match = rx.findall(line)
            if (key == 'D' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'input', key)

                object_handle.datawidth = datawidth
                object_handle.D = matchlist
                for i in range(0,len(match)):
                    object_handle.D[i] = matchlist
            elif (key == 'S' and match):
                process_match(match, object_handle, 'control', key)
                #number of selects is length of match, only one bit widths.
                object_handle.S = np.append(object_handle.S, match)
                #if match is list of matches to key
            elif (key == 'Z' and match):
                #print( "Added Z to mux")
                process_match(match, object_handle, 'output', key)
                #will have same datawidth as D.
                object_handle.Z = match

            else:
                print("did not find attributes of object: "+ object_handle.name)
    elif object_handle.id == 'select_op':
        for key, rx in rx_dict_SELECT.items():
            #returning everything matching given key in a list
            match = rx.findall(line)
            if (key == 'DATA' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'input', key)
                object_handle.datawidth = datawidth
                object_handle.D = matchlist
            elif (key == 'CONTROL' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'control',key)
                object_handle.datawidth = datawidth
            elif (key == 'Z' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'output', key)
                object_handle.Z = matchlist
            else:
                print("did not find attributes of object: "+ object_handle.name)
    elif object_handle.id == 'comp_op' or object_handle.id == 'add_op' or object_handle.id == 'sub_op' or object_handle.id == "mult_op" or object_handle.id == "div_op":
        for key, rx in rx_dict_SUB_ADD_MULT.items():
            match = rx.findall(line)
```

```
            if (key == 'A' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'input', key)
                object_handle.a_width = datawidth
                object_handle.A = matchlist
            elif (key == 'B' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'input', key)
                object_handle.b_width = datawidth
                object_handle.B = matchlist
            elif (key == 'Z' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'output', key)
                object_handle.z_width = datawidth
                object_handle.Z = matchlist
            else:
                print("did not find attributes of object: "+object_handle.name)
    elif object_handle.id == 'shift_op' or object_handle.id == 'b_shift_op':
        for key, rx in rx_dict_shift.items():
            match = rx.findall(line)
            if (key == 'A' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'input', key)
                object_handle.a_width = datawidth
                object_handle.A = matchlist
            elif (key == 'SH' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'control', key)
                object_handle.sh_width = datawidth
                object_handle.SH = matchlist
            elif (key == 'Z' and match):
                dataN, datawidth, matchlist = process_match(match, object_handle, 'output', key)
                object_handle.z_width = datawidth
                object_handle.Z = matchlist
            else:
                print("did not find attributes of object: "+object_handle.name)
    elif(object_handle.id == 'dep'):
        #look for dep objects and add them to list
        for key, rx in rx_dict_dep_internals.items():
            match = rx.findall(line)
            if match:
                object_handle.add_connections(match)
    elif(object_handle.id == 'assign'):
        #print("Assign statement")
        for key, rx in rx_dict_assign.items():
            match = rx.search(line)
            if key == 'rhs':
                rhsline = match.group(1)
                rhsline = rhsline.translate({ord(i): None for i in '}{\ '})
                if rhsline == "1'b0" or rhsline == "1'b1":
                    object_handle.rhs = "constant"
                else:
                    i1, i2, new_rhsline, indextype = find_indexes(rhsline)
                    object_handle.rhs_i1 = i1
                    #look for new_rhsline in connections.
                    if (i2 != -1): object_handle.rhs_i2 = i2

                    element, foundbool = search_list(outputs, new_rhsline)

                    if foundbool == False:
                        element, foundbool = search_list(inputs, new_rhsline)
                        if foundbool == False:
                            element, foundbool = search_list(connects, new_rhsline)
                    if foundbool:
                        object_handle.rhs = element
                    else:
                        print("Did not find match of rhs in assign")
                        print(new_rhsline)
    else:
        print (" No match found for object handle id: "+object_handle.id)
    return key, match

#if signal is not constant, find out what it is connected to and the width and register connection
def process_match(match, object_handle, connection_type, port_name):
    #print("Running process match for object " + str(object_handle.name))
    dataN = len(match)
```

```
    processed_matchlist = []
    if match[0] == "":
        return 0, 0, []
    for i in range(0, len(match)):
        match[i] = match[i].translate({ord(i): None for i in '
    }{\ '})
        datawidth = match[i].count(',')+1
        matchlist = match[i].split(',')
        processed_matchlist.append(matchlist)
        datawidth_set = {'sub_op','select_op', 'mux_op', '
    shift_op', 'add_op', 'mult_op', 'comp_op', 'div_op', '
    b_shift_op', 'shift_add_op'}
        if(connection_type == 'output'):
            if (object_handle.id in datawidth_set):
                if(object_handle.output_nodes ==[]):
                    #declare output_nodes
                    if object_handle.id == 'mux_op' or
    object_handle.id == 'select_op':
                        object_handle.output_nodes = [None]*
    datawidth
                    else:
                        object_handle.output_nodes = [None]*
    datawidth
        j_increment = 0
        for j in range(0, len(matchlist)): #j is i1
            if matchlist[j] == '1\'b1' or matchlist[j] == '1\'b0
    ':
                datawidth = 1
            else:
                found = False
                if(port_name == 'dep'):
                    to_append = [object_handle, port_name,
    object_handle.id, connection_type, j+j_increment, i] #0]
                else:
                    to_append = [object_handle, port_name,
    object_handle.id, connection_type, j+j_increment]
                #print("j = "+str(j))
                if (connection_type == 'input' or
    connection_type == 'control'): #or connection_type == '
    control'):
                    i1, i2, matchobj, index_type = find_indexes(
    matchlist[j])
                    element, found = search_list(inputs,
    matchobj)

                    if found:
                        element.add_node_input_connection(i1, i2
    , to_append, index_type)
                    else:
                        element, found = search_list(connects,
    matchobj)
                        if found:
                            element.add_node_input_connection(i1
    , i2, to_append, index_type)
                        else:
                            element, found = search_list(outputs
    , matchobj)
                            if found:
                                element.
    add_node_input_connection(i1, i2, to_append, index_type)

                            else:
                                print("Did not find: "+matchobj)
                                print("["+str(object_handle.name
    )+" , "+str(port_name)+" , "+str(object_handle.id)+"]")
                    if found:
                        if index_type == '':
                            #whole signal width-1 added to j
                            j_increment = j_increment + element.
    width -1
                        elif index_type == 'slice':
                            #add width of slice to j'
                            j_increment = j_increment+ i1-i2
                        #print("j_increment: "+str(j_increment))

                elif (connection_type == 'output'):
                    i1, i2, matchobj, index_type = find_indexes(
    matchlist[j])
                    element, found = search_list(outputs,
    matchobj)
                    if found:
                        if(i1 == -1 and i2 == -1):
                            #print("Element width: "+str(element
    .width))
                            if element.width > 1 and (element.
    width != len(object_handle.output_nodes)):
                                #redefine output
                                if (object_handle.id in
    datawidth_set):
```

```
                                    for i in range(element.width
-1):
                                        object_handle.
output_nodes.append(None)
                                    #print(len(object_handle.
output_nodes))
                        if element.width == 1 and index_type ==
'':
                            index_type = 'bit'
                        element.add_node_output_connection(i1,
i2, to_append, index_type,j+j_increment)
                    else:
                        element, found = search_list(connects,
matchobj)
                        if found:
                            if(i1 == -1 and i2 == -1):
                                #print("Element width: "+str(
element.width))
                                if element.width > 1 and (
element.width != len(object_handle.output_nodes)):
                                #redefine output
                                    if (object_handle.id in
datawidth_set):
                                        for i in range(element.
width-1):
                                            object_handle.
output_nodes.append(None)

                            if element.width == 1 and index_type
 == '':
                                index_type = 'bit'
                            #print("calling
add_node_output_connection i1="+str(i1)+" i2: "+str(i2)+" j:
 "+str(j))
                            element.add_node_output_connection(
i1, i2, to_append, index_type,j+j_increment)

                        else:
                            print("Did not find: "+matchobj)
                            print(str(object_handle)+" "+str(
port_name)+" "+str(object_handle.id))
                    if found:
                        #print("j = "+str(j))
                        if index_type == '':
                            #whole signal width-1 added to j
                                #print("Modified j_increment")
                                j_increment = j_increment + element.
width -1
                            elif index_type == 'slice':
                                #add width of slice to j'
                                #print("Modified j_increment")
                                j_increment = j_increment+ i1-i2
                            #print("j_increment: "+str(j_increment))
                    else:
                        print("Did not find\t"+str(matchlist[j])+"\
tANYWHERE")

    #print("\nFinished matchlist for: "+object_handle.name+"\
    nwidth: "+str(datawidth)+"\nN:" +str(dataN)+"\nMatchlist:" +
     str(processed_matchlist)+"\n")
    return dataN, datawidth,  processed_matchlist

#return object in list with name matching searchstring or None,
    False
def search_list(list, searchstring):
    element = None
    for element in list:
        if (element.name == searchstring):
            #if searchstring[0] == 'e': print("FOUND: "+
searchstring)
            return element, True
    return element, False

#set nodes involved in assign statements equal to each other
def connect_assigns():
    #print("RUNNING CONNECT ASSIGNS")
    #print(assigns)
    for a in assigns:
        #print(a.name)
        #print(a.lhs.name+" "+str(a.lhs))
        #print(a.rhs)
        #print(a.i1)
        #print(a.i2)
        if(a.i1 != -1):
            #print(a.lhs.connection_nodes)
            lhs_node = a.lhs.connection_nodes[a.i1-a.lhs.
widthoffset][a.i2]
        else:
            #print("should connect whole signal")
            pass
```

```
        if a.rhs != None and a.rhs !="constant":
            #print(a.rhs.connection_nodes)
            rhs_node = a.rhs.connection_nodes[a.rhs_i1-a.rhs.
    widthoffset][a.rhs_i2]
        #print("Defined two connection nodes")
        if (a.i1 != -1 and a.rhs != None):
            #Connect bits of single node
            if a.rhs == "constant":
                a.lhs.connection_nodes[a.i1][a.i2].constant =
    True
            else:
                n1 = a.lhs.connection_nodes[a.i1-a.lhs.
    widthoffset][a.i2]
                n2 = a.rhs.connection_nodes[a.rhs_i1-a.rhs.
    widthoffset][a.rhs_i2]
                connect_nodes(n1, n2)
        else:
            #connect whole node if a.rhs exists
            if a.rhs != None and a.rhs != "constant":
                #print("Connect whole signal")
                for i in range(len(a.lhs.connection_nodes)):
                    for j in range(len(a.lhs.connection_nodes[i
    ])):
                        n1 = a.lhs.connection_nodes[i][j]
                        n2 = a.rhs.connection_nodes[i][j]
                        connect_nodes(n1,n2)

#set nodes equal to each other
def connect_nodes(n1, n2):
    if n1 == n2:
        return
    else:
        for i in range(len(n1.connected_inputs)):
            n1_con = n1.connected_inputs[i]
            found = False
            for j in range(len(n2.connected_inputs)):
                n2_con = n2.connected_inputs[j]
                n1.connected_inputs.append(n2_con)
            n2.connected_inputs.append(n1_con)
        if n1.connected_outputs == []:
            n1.connected_outputs = n2.connected_outputs
        elif n2.connected_outputs == []:
            n2.connected_outputs = n1.connected_outputs
        if n1.constant == True: n2.constant = True
        if n2.constant == True: n1.constant = True


#classes
class register:
    id = 'reg'
    name = ""
    clear = 0
    preset = 0
    next_state = 0
    clocked_on = 0
    data_in = 0
    enable = 0
    Q = 0
    QN = 0
    synch_clear = 0
    synch_preset = 0
    synch_toggle = 0
    synch_enable = 0
    output_structure_taken = False
    structurecount = 0
    def __str__(self):
        #return "Register: \n clear = " + str(self.clear) + "\n
    preset = " + str(self.preset) + "\n next_state = " + str(
    self.next_state) +"\n clocked_on = " + str(self.clocked_on)
    +"\n data_in = " + str(self.data_in) +"\n enable = " + str(
    self.enable) +"\n Q = " + str(self.Q)
        return "Register: " + self.name
    def __init__(self):
        #print("made reg!")
        global regs
        regs = np.append(regs, self)
        self.output_nodes_q = [None]*1
        self.output_nodes_qn = [None]*1
        self.has_parent = False
class gtech_or2:
    id = 'gtech_or2'
    name = ""
    A = 0
    B = 0
    Z = 0
    structurecount = 0
    def __str__(self):
        return "OR2: " + self.name
    def __init__(self):
        #print("made or2!")
```

```
        global or2s
        or2s = np.append(or2s, self)
        self.output_nodes = [None]*1
class gtech_xor2:
    id = 'gtech_xor2'
    name = ""
    A = 0
    B = 0
    Z = 0
    structurecount = 0
    def __str__(self):
        return "XOR2: " + self.name
    def __init__(self):
        #print("made or2!")
        global xor2s
        xor2s = np.append(xor2s, self)
        self.output_nodes = [None]*1
class gtech_and2:
    id = 'gtech_and2'
    name = ""
    A = 0
    B = 0
    Z = 0
    structurecount = 0
    def __str__(self):
        return "AND2: " + self.name + " A: "+str(self.A)+" B: "+
    str(self.B)+" Z: "+str(self.Z)
    def __init__(self):
        global and2s
        and2s = np.append(and2s, self)
        self.output_nodes = [None]*1
class gtech_not:
    id = 'gtech_not'
    name = ""
    A = 0
    Z = 0
    structurecount = 0
    def __str__(self):
        return "NOT: " + self.name
    def __init__(self):
        global nots
        self.output_nodes = [None]*1
        nots = np.append(nots, self)
class gtech_buf:
    id = 'gtech_buf'
    name = ""
    A = 0
    Z = 0
    structurecount = 0
    def __str__(self):
        return "BUF: " + self.name
    def __init__(self):
        self.output_nodes = [None]*1
        global bufs
        bufs = np.append(bufs, self)
class mux_op:
    id = 'mux_op'
    name = ""
    D = np.array([])
    S = np.array([])
    Z = np.array([])
    datawidth = 0
    structurecount = 0
    def __init__(self):
        global muxes
        self.output_nodes = []
        muxes = np.append(muxes, self)
    def __str__(self):
        return "mux: " + self.name + "# inputs: " + str(self.
    d_size()) + " datawidth: " + str(self.datawidth)
    def d_size(self):
        D = self.D
        print (str(D))
        #print("number of Ds " + str(len(D)))
        return len(D) #D.size()
    def s_size(self):
        S = self.S
        return len(S)
    #def datawidth(self):
    #    D = self.D
    #    return len(D.item(0))
    def print(self):
        print("mux: " + self.name + "# inputs: " + str(self.
    d_size()) + " datawidth: " + str(self.datawidth))
class select_op:
    #NB select also has width of select to take into account
    id = 'select_op'
    name = ""
    D       = np.array([])
    CONTROL     = np.array([])
```

```
    Z           = np.array([])
    datawidth   = 0
    selectwidth = 0
    structurecount = 0
    def __init__(self):
        global selects
        self.output_nodes = []
        selects = np.append(selects, self)
    def __str__(self):
        return "select: " + self.name + " inputs: \n" + str((self.DATA)) + " \nselect:\n" + str((self.CONTROL)) +"\n datawidth = " +str(self.datawidth)+ " selectwidth = "+ str(self.selectwidth)
class connection:
    id = 'connection'
    name        = ''
    width       = 1
    depth       = 1
    widthoffset = 0
    def __init__(self):
        global connects
        connects = np.append(connects, self)
        self.connection_nodes = []
        self.init_connection_nodes()
    def init_connection_nodes(self):
        self.connection_nodes = [[node(j,i) for i in range(self.depth)] for j in range(self.width)]
    def add_node_input_connection(self,i1, i2, l, index_type):
        #to increment l[4]
        add = 0
        if index_type == '':
            for i in range(self.width):
                li = l[:]
                li[4] = l[4] + add
                add = add+1
                for j in range(self.depth):
                    self.connection_nodes[i][j].add_input_connection(li)

        elif index_type == 'index':
            self.connection_nodes[i1-self.widthoffset][i2].add_input_connection(l)
        elif index_type == 'slice':
            for i in range(i2, i1):
                li = l[:]
                li[4] = l[4] + add
                self.connection_nodes[i-self.widthoffset][0].add_input_connection(li)
                add = add+1

    def add_node_output_connection(self,i1, i2, l, index_type,k):
        add = 0
        if index_type == '':
            for i in range(self.width):
                li = l[:]
                li[4] = l[4] + add
                k = li[4]
                for j in range(self.depth):
                    self.connection_nodes[i][j].add_output_connection(li,k)
                add = add+1
        elif index_type == 'bit':
            self.connection_nodes[i1-self.widthoffset][i2].add_output_connection(l,k)

        elif index_type == 'index':
            self.connection_nodes[i1-self.widthoffset][i2].add_output_connection(l,k)
        elif index_type == 'slice':
            for i in range(i2, i1):
                li = l[:]
                li[4] = l[4] + add
                k = li[4]
                self.connection_nodes[i][0].add_output_connection(li,k)
                add = add+1
class assign:
    id = 'assign'
    lhs = ''
    rhs = ''
    i1 = 0
    i2 = 0
    lhs = None
    rhs = None
    rhs_i1 = 0
    rhs_i2 = 0
    def __init__(self):
        global assigns
        assigns = np.append(assigns,self)
```

```
    #     self.lhs = lhs
    #     self.rhs = rhs
class node:
    id = 'node'
    def __init__(self,i1, i2):
        self.connected_inputs      = []
        self.connected_outputs     = []
        self.i1 = i1
        self.i2 = i2
        self.constant = False
    def add_input_connection(self,l):
        self.connected_inputs.append(l)
    def add_output_connection(self,l,j):
        self.connected_outputs.append(l)
        connected_object_handle = l[0]
        if l[2] != 'reg':
            #add output node to connected object
            connected_object_handle.output_nodes[j] = self
        else:
            #add output node to register
            connected_object_handle.output_nodes_q[0] = self

            connected_object_handle.output_nodes_qn[0] = self
    def print(self):
        print("Node:")
        print("\tinputs")
        print(self.connected_inputs)
        print("\toutputs:")
        print(self.connected_outputs)
        print("Endnode")
class input_obj():
    id      = 'input'
    name    = ""
    width   = 1
    depth = 1
    widthoffset = 0
    def __init__(self):
        global inputs
        inputs = np.append(inputs, self)
        self.connection_nodes  = []
    def init_connection_nodes(self):
        #print("\ninitializing connection nodes")
        self.connection_nodes = [[node(j,i) for i in range(self.
    depth)] for j in range(self.width)]
        #print(self.connection_nodes)
    def add_node_input_connection(self,i1, i2, l, index_type):
        add = 0
        if index_type == '':
            if (i1 == -1 and i2 == -1):
                for i in range(self.width):
                    li = l[:]
                    li[4] = l[4] + add
                    for j in range(self.depth):
                        #print("added input connection with l
    [4]: "+str(li[4]))
                        self.connection_nodes[i][j].
    add_input_connection(li)
                    add = add+1

        elif index_type == 'index':
            self.connection_nodes[i1-self.widthoffset][i2].
    add_input_connection(l)
        elif index_type == 'slice':
            for i in range(i2, i1):
                li = l[:]
                li[4] = l[4] + add
                self.connection_nodes[i][0].add_input_connection
    (li)
                add = add+1
class output_obj():
    id      = 'output'
    name    = ""
    width   = 1
    depth = 1
    widthoffset = 0
    def __init__(self):
        global outputs
        outputs = np.append(outputs, self)
        self.connection_nodes = []
    def init_connection_nodes(self):
        #print("\ninitializing connection nodes")
        self.connection_nodes = [[node(j,i) for i in range(self.
    depth)] for j in range(self.width)]
        #print(self.connection_nodes)
    def add_node_output_connection(self,i1, i2, l, index_type,k)
    :
        add = 0
        if index_type == '':
            for i in range(self.width):
                li = l[:]
```

```
                li[4] = l[4] + add
                k = li[4]
                for j in range(self.depth):
                    self.connection_nodes[i][j].
    add_output_connection(li,k)
                add = add+1
                #print(add)
        elif index_type == 'bit':
            self.connection_nodes[i1-self.widthoffset][i2].
    add_output_connection(l,k)

        elif index_type == 'index':
            self.connection_nodes[i1-self.widthoffset][i2].
    add_output_connection(l,k)
        elif index_type == 'slice':
            for i in range(i2, i1):
                li = l[:]
                li[4] = l[4] + add
                k = li[4]
                self.connection_nodes[i][0].
    add_output_connection(li,k)
                add = add+1
    def add_node_input_connection(self,i1, i2, l, index_type):
        add = 0
        if index_type == '':
            if (i1 == -1 and i2 == -1):
                for i in range(self.width):
                    li = l[:]
                    li[4] = l[4] + add
                    for j in range(self.depth):
                        self.connection_nodes[i][j].
    add_input_connection(li)
                    add = add+1

        elif index_type == 'index':
            self.connection_nodes[i1-self.widthoffset][i2].
    add_input_connection(l)
        elif index_type == 'slice':
            for i in range(i2, i1):
                li = l[:]
                li[4] = l[4] + add
                self.connection_nodes[i][0].add_input_connection
    (li)
                add = add+1
class module:
    name = ""
    connection_point_string = ""
    def __init__(self, name):
        self.regs           = []
        self.nots           = []
        self.bufs           = []
        self.and2s          = []
        self.or2s           = []
        self.muxes          = []
        self.selects        = []
        self.connects       = []
        self.inputs         = []
        self.outputs        = []
        self.dependencies   = []
        self.shifters       = []
        self.comparators    = []
        self.xor2s          = []
        self.multipliers    = []
        self.subtractors    = []
        self.b_shifters     = []
        self.adders         = []
        self.shift_adders   = []
        self.divisors       = []
        self.assigns        = []
        self.name = name
        self.connection_points = []
    def set_lists(self):
        global regs
        global nots
        global bufs
        global and2s
        global or2s
        global muxes
        global selects
        global connects
        global inputs
        global outputs
        global dependencies
        global shifters
        global comparators
        global xor2s
        global multipliers
        global subtractors
        global b_shifters
        global adders
```

```
        global shift_adders
        global divisors
        global assigns

        self.regs           = np.copy(regs         )
        self.nots           = np.copy(nots         )
        self.bufs           = np.copy(bufs         )
        self.and2s          = np.copy(and2s        )
        self.or2s           = np.copy(or2s         )
        self.muxes          = np.copy(muxes        )
        self.selects        = np.copy(selects      )
        self.connects       = np.copy(connects     )
        self.inputs         = np.copy(inputs       )
        self.outputs        = np.copy(outputs      )
        self.dependencies   = np.copy(dependencies)
        self.shifters       = np.copy(shifters     )
        self.comparators    = np.copy(comparators  )
        self.xor2s          = np.copy(xor2s        )
        self.multipliers    = np.copy(multipliers  )
        self.subtractors    = np.copy(subtractors  )
        self.b_shifters     = np.copy(b_shifters   )
        self.adders         = np.copy(adders       )
        self.shift_adders   = np.copy(shift_adders)
        self.divisors       = np.copy(divisors     )
        self.assigns        = np.copy(assigns      )
    def set_connection_points(self):
        self.connection_point_string = self.
    connection_point_string.translate({ord(i): None for i in '\
    \n'})
        #print(self.connection_point_string)
        connectionpoints = []
        for key, rx in rx_dict_module_connections.items():
            match = rx.findall(self.connection_point_string) #rx
    .findall(line)
            #print("found "+str(len(match))+" matches in
    modulestring")
            if match:
                #print(str(match))
                for m in match:
                    if (m[0] == ''): connectionpoints.append(
    tuple([m[2]]))
                    else:
                        l =  m[1].translate({ord(i): None for i
    in '}{'})
                        l = l.split(',')
                        connectionpoints.append(tuple([m[0], l])
    )
                #connectionpoints.append(match)
        #print(connectionpoints)
        self.connection_points = connectionpoints
class dependency:
    #name of instantiation
    id = 'dep'
    name = ""
    #modulename
    modulename = ""
    module_handle = None
    possible_HINST = False
    def __init__(self):
        global dependencies
        dependencies = np.append(dependencies, self)
        self.connections = []
    def add_connections(self, list):
        self.connections.append(list)
class shift_op:
    id = 'shift_op'
    name = ''
    A = 0
    SH = 0
    Z = 0
    a_width = 0
    sh_width = 0
    z_width = 0
    structurecount = 0
    def __init__(self):
        global shifters
        shifters = np.append(shifters,self)
        self.output_nodes = []
class comp_op:
    id = "comp_op"
    name = ""
    A = 0
    B = 0
    Z = 0
    a_width = 0
    b_width = 0
    z_width = 0
    structurecount = 0
    def __init__(self):
        global comparators
```

```
        comparators = np.append(comparators, self)
        self.output_nodes = []
        #print("Made comparator")
class sub_op:
    id = "sub_op"
    name = ""
    A = 0
    B = 0
    Z = 0
    structurecount = 0
    def __init__(self):
        global subtractors
        subtractors = np.append(subtractors, self)
        self.output_nodes = []
        #print("made subtractor")
class add_op:
    id = "add_op"
    name = ""
    A = 0
    B = 0
    Z = 0
    a_width = 0
    b_width = 0
    z_width = 0
    structurecount = 0
    def __init__(self):
        global adders
        adders = np.append(adders, self)
        self.output_nodes = []
        #print("made adder")
class mult_op:
    id = "mult_op"
    name = ""
    A = 0
    B = 0
    Z = 0
    a_width = 0
    b_width = 0
    z_width = 0
    structurecount = 0
    def __init__(self):
        global multipliers
        multipliers = np.append(multipliers, self)
        self.output_nodes = []
        #print("made multiplicator")
class div_op:
    id = "div_op"
    name = ""
    A = 0
    B = 0
    Z = 0
    a_width = 0
    b_width = 0
    z_width = 0
    structurecount = 0
    def __init__(self):
        global divisors
        divisors = np.append(divisors, self)
        self.output_nodes = []
        #print("Made divisor")
class b_shift_op:
    id = "b_shift_op"
    name = ""
    A = 0
    SH = 0
    Z = 0
    a_width = 0
    sh_width = 0
    z_width = 0
    structurecount = 0
    def __init__(self):
        global b_shifters
        b_shifters = np.append(b_shifters, self)
        self.output_nodes = []
        #print("made barrelshift")
class shift_add_op:
    #dont know what content should be here, may need to
    implement as I go if I encounter it during testing
    id = "shift_add_op"
    name = ""
    structurecount = 0
    def __init__(self):
        global shift_adders
        shift_adders = np.append(shift_adders, self)
        self.output_nodes = []
        #print("Made shift adder")


#dictionaries containing regular expressions to handle different
    constructs from the elaborated systemverilog
```

```
rx_dict_module_start = {
    'begin'     : re.compile(r"module\s+(\S+)")
}
rx_dict_objectconnection = {
    'bit'       : re.compile(r"(1'b\d)"),
    'connect'   : re.compile(r"(\S+)(?:\[(\d{1,4})\])?(?:\[(\d{1,4})\])?")
}
rx_dict_start = {
    'register'      : re.compile(r"\\*\*SEQGEN\*\*\s+(\S+)\s" ),
    'GTECH_OR2'     : re.compile(r"GTECH_OR2\s+(\S+)\s"),
    'GTECH_NOT'     : re.compile(r"GTECH_NOT\s+(\S+)\s"),
    'GTECH_BUF'     : re.compile(r"GTECH_BUF\s+(\S+)\s"),
    'GTECH_AND2'    : re.compile(r"GTECH_AND2\s+(\S+)\s"),
    'GTECH_XOR2'    : re.compile(r"GTECH_XOR2\s+(\S+)\s"),
    'MUX_OP'        : re.compile(r"MUX_OP\s+(\S+)\s"),
    'SELECT_OP'     : re.compile(r"SELECT_OP\s+(\S+)\s"),
    #TODO: mux add, sub,mult, shifts and compares remain at least...
    #all of these can be single bit or multibit. if square brackets before name
    #multi, else single (group capture?)
    'input'         : re.compile(r"(input)\s+(?:\[(\d{1,3}):(\d{1,3})\])?"),
    'output'        : re.compile(r"(output)\s+(?:\[(\d{1,3}):(\d{1,3})\])?"),
    'wire'          : re.compile(r"(wire)\s+(?:\[(\d{1,3}):(\d{1,3})\])?"),
    'dep'           : re.compile(r"(\S+)\s+(\S*u_\S+)\s*\("),
    'COMP_OP'       : re.compile(r"^\s*(?:EQ_UNS_OP|NE_UNS_OP|EQ_TC_OP|NE_TC_OP|GEQ_UNS_OP|GEQ_TC_OP|LEQ_UNS_OP|LEQ_TC_OP|GT_UNS_OP|GT_TC_OP|LT_UNS_OP|LT_TC_OP)\s+(\S+)\s"),
    'SUB_OP'        : re.compile(r"SUB_(?:UNS_OP|UNS_CI_OP|TC_OP|TC_CI_OP)\s+(\S+)\s"),
    'ADD_OP'        : re.compile(r"ADD_(?:UNS_OP|UNS_CI_OP|TC_OP|TC_CI_OP)\s+(\S+)\s"),
    'MULT_OP'       : re.compile(r"MULT_(?:UNS_OP|TC_OP)\s+(\S+)\s"),
    'DIV_OP'        : re.compile(r"(?:DIV|MOD|REM|DIVREM|DIVMOD)_(?:UNS|TC)_OP\s+(\S+)\s"), #only div in Yoda
    'SHIFT_OP'      : re.compile(r"(?:ASH|ASHR|SRA)_(?:UNS|TC)_(?:UNS|TC|OP)(?:_OP)?\s+(\S+)\s"),
    'B_SHIFT_OP'    : re.compile(r"BSH(?:_UNS_OP|_TC_OP|L_TC_OP|R_UNS_OP|R_TC_OP)\s+(\S+)\s"), #not in Yoda
    'SHIFT_ADD_OP'  : re.compile(r"(?:SLA_UNS_OP|SLA_TC_OP)\s+(\S+)\s"), #not in Yoda
    'assign'        : re.compile(r"assign\s([^=]+)")
    #'SRA'           :
}
rx_dict_dep_internals = {
    #'dep'   : re.compile(r"\(\s?(?:\.([^\(\s,]+)\(([^\(\);\s]*)\),?)+\s?\);")
    'dep'   : re.compile(r"\s?(?:\.(?P<connection_point>[^\(\s,]+)\((?P<connected_to>[^\(\);\s]*)\),?)\s?")
}
rx_dict_assign = {
    'rhs' : re.compile(r"=([^=]+);")
}
rx_dict_shift = {
    'A'     : re.compile(r"\.A\(([^\)]*)\)"),
    'SH'    : re.compile(r"\.SH\(([^\)]*)\)"),
    'Z'     : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_module_connections = {
    #'reconnect'   : re.compile(r"\s?((?:\.(?P<connection_point>[^\(\s,]+)\((?P<connected_to>[^\(\);\s]*)\)))[,\)]\s?")
    #|([^,\.\(\)\{\}\[\]]+)[,\)]\s?"),
    #'plain'       : re.compile(r"\s*([^\(\s,]+)\s?[,\)]")
    'connection' : re.compile(r"(?:[,\(]\.([^\(\),]+)\(([^\(\)]+)\))|(?:[,\(]([^\.][^\)\(,;]*))")
}
rx_dict_end = {
    #'end'   : re.compile(r"\);"),
    'semi'  : re.compile(r";") #hopefully this does not ruin anything and all semicolons are ends
}
rx_dict_in_out_wire = {
    'varname' : re.compile(r"[^\s,]+") #one or more char not whitespace comma
}
rx_dict_SELECT = {
    'DATA'      : re.compile(r"\.DATA\d{1,2}\(([^\)]*)\)"),
    'CONTROL'   : re.compile(r"\.CONTROL\d{1,2}\(([^\)]*)\)"),
    'Z'         : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_comp = {
    'A'             : re.compile(r"\.A\(([^\)]*)\)"),
    'B'             : re.compile(r"\.B\(([^\)]*)\)"),
    'QUOTIENT'      : re.compile(r"\.QUOTIENT\(([^\)]*)\)")
```

```
}
rx_dict_SUB_ADD_MULT = {
    'A'      : re.compile(r"\.A\(([^\)]*)\)"),
    'B'      : re.compile(r"\.B\(([^\)]*)\)"),
    'Z'      : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_MUX = {
    'D'      : re.compile(r"\.D\d{1,2}\(([^\)]*)\)"), # SEEMS TO BE SOME THAT HAS UP TO D31 AS D INPUTS. HOW DO i HANDLE THese VARYING THINgS
                                                      # ALSO SOME ONLY GOING TO D3 BUT ATTACHING 5 BIT TO EACH D data width of d varies, number of D inputs varies
    'S'      : re.compile(r"\.S\d{1,2}\(([^\)]*)\)"), # make arrays for the mux
    'Z'      : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_BUF = {
    'A'      : re.compile(r"\.A\(([^\)]*)\)"),
    'Z'      : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_NOT = {
    'A'      : re.compile(r"\.A\(([^\)]*)\)"),
    'Z'      : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_AND2  = {
    'A'      : re.compile(r"\.A\(([^\)]*)\)"),
    'B'      : re.compile(r"\.B\(([^\)]*)\)"),
    'Z'      : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_OR2 = {
    'A'      : re.compile(r"\.A\(([^\)]*)\)"),
    'B'      : re.compile(r"\.B\(([^\)]*)\)"),
    'Z'      : re.compile(r"\.Z\(([^\)]*)\)")
}
rx_dict_reg = {
    'clear'           : re.compile(r"\.clear\(([^\)]*)\)"),
    'preset'          : re.compile(r"\.preset\(([^\)]*)\)"),
    'next_state'      : re.compile(r"\.next_state\(([^\)]*)\)"),
    'clocked_on'      : re.compile(r"\.clocked_on\(([^\)]*)\)"),
    'data_in'         : re.compile(r"\.data_in\(([^\)]*)\)"),
    'enable'          : re.compile(r"\.enable\(([^\)]*)\)"),
    'Q'               : re.compile(r"\.Q\(([^\)]*)\)"),
    'QN'              : re.compile(r"\.QN\(([^\)]*)\)"),
    'synch_clear'     : re.compile(r"\.synch_clear\(([^\)]*)\)"),
    'synch_preset'    : re.compile(r"\.synch_preset\(([^\)]*)\)"),
    'synch_toggle'    : re.compile(r"\.synch_toggle\(([^\)]*)\)"),
    'synch_enable'    : re.compile(r"\.synch_enable\(([^\)]*)\)")
}

#process module instantiations
def process_dependencies():
    for top_module in modules:
        set_global_lists(top_module)

        for dep in top_module.dependencies:
            found_dep = False
            #print("Looking for "+dep.modulename+" in dependencies")
            modulename = dep.modulename
            #find modulename in module list

            for dep_module in modules:

                if dep_module.name == modulename:
                    found_dep = True
                    dep.module_handle = dep_module
                    module_connection_list = dep_module.connection_points

                    for dependency_connection_tuple in dep.connections[0]:
                        cleaned_dependency_connection_point = dependency_connection_tuple[0].translate({ord(i): None for i in '\ '})
                        found = False

                        for module_connection_tuple in module_connection_list:

                            if cleaned_dependency_connection_point == module_connection_tuple[0]:
                                found = True
```

```
                            cleaned_dep_connectionlist = dependency_connection_tuple[1].translate({ord(i): None for i in '}{\ '})
                            cleaned_dep_connectionlist = cleaned_dep_connectionlist.split(',')

                            for i in range(len(cleaned_dep_connectionlist)):
                                if len(module_connection_tuple) > 1:
                                    i1, i2, module_connection, typeindex = find_indexes(module_connection_tuple[1][i])
                                else:

                                    i1, i2, module_connection, typeindex = find_indexes(module_connection_tuple[0])

                                dep_handle, found_dep = search_list(dep_module.inputs, module_connection)
                                if found_dep:

                                    print("sending "+str(cleaned_dep_connectionlist[i])+" into process match")
                                    dataN, datawidth, processed_matchlist = process_match([cleaned_dep_connectionlist[i]], dep_handle, 'input', 'dep' )



                                else:
                                    dep_handle, found_dep = search_list(dep_module.outputs, module_connection)
                                    #print(dep_connection)
                                    if found_dep:

                                        dataN, datawidth, processed_matchlist = process_match([cleaned_dep_connectionlist[i]], dep_handle, 'input', 'dep')
                                    else:

                                        print("Did not find dep "+dep_module.name)


                        #DEP NOT FOUND IN MODULES, MAYBE HINST
                        if found == False:
                            dep.possible_HINST = True

            if found_dep == False:
                print("Did not find dep: "+dep.modulename)
        top_module.set_lists()
        empty_global_lists()
                #find dep ports in inputs or outputs

#print name of all objects in a list
def print_list_names(l):
    for e in l:
        print("\t"+e.name)


#go structure heads and make structure trees
def connect_structure(module):
    global top_level_parents

    for inp in module.inputs:

        for nl in range(len(inp.connection_nodes)):
            for n in range(len(inp.connection_nodes[nl])):
                #print(str(nl)+" "+str(n))
                structure_handle = structure(None, inp, nl)
                top_level_parents.append(structure_handle)
                connect_children(structure_handle,nl,n)

    for m in modules:
        for reg in m.regs:
            #print(reg.name)
            structure_handle = structure(None, reg, 0)
            top_level_parents.append(structure_handle)
            connect_children(structure_handle, 0, 0)


#recursively connects all children to a parent and expand structure tree
def connect_children(parent, i1,i2):
    object_handle = parent.represented_object_handle

    i1 = parent.i1

```

```
    if object_handle.id != 'input' and object_handle.id != '
    output':

        datawidth_set = {'sub_op','select_op', 'mux_op', '
    shift_op', 'add_op', 'mult_op', 'comp_op', 'div_op', '
    b_shift_op', 'shift_add_op'}

        object_handle.structurecount = object_handle.
    structurecount+1

    global top_level_parents
    #print("Current object: "+object_handle.name)
    output_nodes = []
    output_nodes_q = []
    output_nodes_qn = []
    #print("parent: "+object_handle.name)
    if object_handle.id == 'output':
        output_nodes = []
    elif object_handle.id == 'reg':

        if object_handle.output_structure_taken == False:
            output_nodes_q = object_handle.output_nodes_q
            output_nodes_qn = object_handle.output_nodes_qn
            object_handle.output_structure_taken = True

        if parent != None:# and parent.connected_inputs[0] !=
    None:
            if parent.parent != None:
                object_handle.has_parent = True

    elif object_handle.id == 'input':

        output_nodes.append(object_handle.connection_nodes[i1][
    i2])

    elif object_handle.id == 'comp_op':
        output_nodes = object_handle.output_nodes
    else:

        output_nodes = [object_handle.output_nodes[i1]]

        #ADDED to shorten recursion
        object_handle.output_nodes[i1] = parent
    #print(output_nodes)
    for node in output_nodes:


        if node != None:
            if node.id == 'structure':

                parent = node

                return
            elif(node != None):

                for con in node.connected_inputs:
                    child_handle = con[0]

                    structure_handle = structure(parent,
    child_handle, con[4])
                    added = parent.add_child(structure_handle)
                    if added:
                        structure_handle.structure_type =
    child_handle.id
                        structure_handle.
    structure_connection_characteristic = con[3]

                        if con[2] == 'reg' and con[2] != '
    control':
                            con[0].has_parent = True
                        if structure_handle.
    represented_object_handle.id != 'reg' and structure_handle.
    represented_object_handle.id != 'input':
                            structure_handle.
    represented_object_handle.output_nodes[0] = structure_handle
                        if(con[3] != 'control' and con[2] != '
    reg' and con[2] != 'input'):
                            connect_children(structure_handle,
    con[4],node.i2)
                        elif(con[2] == 'input'):
                            #if input i2 needs to be set
    correctly
                            connect_children(structure_handle,
    con[4],con[5])#node.i2)
        else:
            pass
    for node in output_nodes_q:

        if(node != None):

            for con in node.connected_inputs:
```

```
                child_handle = con[0]

                structure_handle = structure(parent, child_handle, con[4])
                added = parent.add_child(structure_handle)
                if added:
                    structure_handle.structure_type = child_handle.id
                    structure_handle.structure_connection_characteristic = con[3]
                    if con[2] == 'reg' and con[2] != 'control':
                        con[0].has_parent = True
                    if structure_handle.represented_object_handle.id != 'reg' and structure_handle.represented_object_handle.id != 'input':
                        structure_handle.represented_object_handle.output_nodes[0] = structure_handle
                    if(con[3] != 'control' and con[2] != 'reg' and con[2] != 'input'):
                        connect_children(structure_handle, con[4],node.i2)
                    elif(con[2] == 'input'):
                        connect_children(structure_handle,con[4],con[5])

    for node in output_nodes_qn:
        if(node != None):

            if node.id == 'structure':
                parent = node

                return

            for con in node.connected_inputs:
                child_handle = con[0]
                #print(con)
                structure_handle = structure(parent, child_handle,con[4])
                added = parent.add_child(structure_handle)
                if added:
                    structure_handle.structure_type = child_handle.id
                    structure_handle.structure_connection_characteristic = con[3]
                    if con[2] == 'reg' and con[2] != 'control':
                        con[0].has_parent = True
                    if structure_handle.represented_object_handle.id != 'reg' and structure_handle.represented_object_handle.id != 'input':
                        structure_handle.represented_object_handle.output_nodes[0] = structure_handle
                    if(con[3] != 'control' and con[2] != 'reg' and con[2] != 'input'):
                        connect_children(structure_handle, con[4],node.i2)
                    elif(con[2] == 'input'):
                        connect_children(structure_handle,con[4],con[5])



#structure tree class
class structure:
    structure_type = ''
    structure_connection_characteristic = ''
    id = 'structure'
    def __init__(self, parent, represented_object_handle, i1):
        self.parent = parent
        self.children = []
        self.represented_object_handle = represented_object_handle
        self.i1 = i1
        self.powerStructure = None
    def add_child(self, child):
        if child.represented_object_handle == self.represented_object_handle:
            return False
        for c in self.children:
            if c.represented_object_handle == child.represented_object_handle:
                return False
        self.children.append(child)
        return True
    def print(self):
        if self.children != []: print("{", end = '')
        for child in self.children:
            print(child.represented_object_handle.id+" ,",end = '')
            child.print()

```

```
        if self.children != []: print("}", end = '')

    def __repr__(self, level=0):
        ret = "\t"*level+repr(self.represented_object_handle.id)
    +"\n"
        if level < 11:
            for child in self.children:
                ret += child.__repr__(level+1)
        return ret

#calls all the functions in the right order to create the
    structural representation
def run_parse_elab(filename):
    parse_file(filename)
    for m in modules:
        m.set_connection_points()
    process_dependencies()

    connect_structure(modules[0])

    #need to also count module instantiations with no content-
    assume hinst
    count_gates(modules[0])
    print_gates()
    return modules, top_level_parents

#count gates in representation
def count_gates(module):
    global reg_n
    global not_n
    global buf_n
    global and2_n
    global or2_n
    global mux_n
    global select_n
    global shift_n
    global comp_n
    global xor2_n
    global mult_n
    global sub_n
    global b_shift_n
    global add_n
    global shift_add_n
    global div_n
    m = module
    if m != None:
        for r in m.regs:
            if r.output_nodes_q[0] != None or r.output_nodes_qn
    [0] != None:
                if r.has_parent:
                #print(r.name)
                    reg_n       =  reg_n +1
                #else:
                   # reg_n = reg_n+1
        not_n       =  not_n           + len(m.nots         )
        buf_n       =  buf_n           + len(m.bufs         )
        and2_n      =  and2_n          + len(m.and2s        )
        or2_n       =  or2_n           + len(m.or2s         )
        mux_n       =  mux_n           + len(m.muxes        )
        select_n    =  select_n        + len(m.selects      )
        shift_n     =  shift_n         + len(m.shifters     )
        comp_n      =  comp_n          + len(m.comparators  )
        xor2_n      =  xor2_n          + len(m.xor2s        )
        mult_n      =  mult_n          + len(m.multipliers  )
        sub_n       =  sub_n           + len(m.subtractors  )
        b_shift_n   =  b_shift_n       + len(m.b_shifters   )
        add_n       =  add_n           + len(m.adders       )
        shift_add_n =  shift_add_n     + len(m.shift_adders)
        div_n       =  div_n           + len(m.divisors     )
        for d in module.dependencies:
            m = d.module_handle
            count_gates(m)

#print gate counts
def print_gates():
    print("regs\t\t"+ str(reg_n      ))
    print("muxes\t\t"+ str(mux_n      ))
    print("nots\t\t"+ str(not_n      ))
    print("bufs\t\t"+ str(buf_n      ))
    print("arithmetic:\t"+str(mult_n+sub_n+add_n+shift_add_n+
    div_n))
    print("logic:\t\t"+str(and2_n+or2_n+shift_n+comp_n+xor2_n+
    b_shift_n))

    print("selects\t\t"+ str(select_n   ))
    print()
    print("Total:\t\t "+str(reg_n+not_n+buf_n+and2_n+or2_n+mux_n
    +select_n+shift_n+comp_n+xor2_n+mult_n+sub_n+b_shift_n+add_n
    +shift_add_n+div_n))
```

E Code implemented in Chapter 7

```
from liberty.parser import parse_liberty
#spec = liberty.parser("liberty.parser", )
from pathlib import Path
import argparse
import numpy as np
import os,sys
import random
from datetime import datetime
import json
import re

path_to_libfile = sys.argv[1]
starttime = datetime.now()
path_to_calibration_file = sys.argv[2]
cells = ''
processed_library_path = "powerlib.txt"

input_set = {"A1", "I", "S", "I0", "TE", "CI", "CO", "A", "CDN",
     "D", "SDN"}
output_set = {"Z", "ZN", "Q", "QN", "ZN"}

def set_cells_and_environment(libfile):
    library = parse_liberty(open(libfile).read())
    #print("done parsing liberty after "+str((datetime.now()-
    starttime)/60)+" minutes")

    voltage_unit            = str(library.get("voltage_unit")
         ).replace("\"", "")
    current_unit            = str(library.get("current_unit")
         ).replace("\"", "")
    leakage_power_unit      = str(library.get("
    leakage_power_unit")  ).replace("\"", "")
    capacitive_load_unit    = str(library.get("
    capacitive_load_unit")).replace("\"", "")
    #what is unit of power in power templates?
    #also unit of leakage power

    cells = library.get_groups("cell")
    jsonstring = json.dumps([voltage_unit, current_unit,
    leakage_power_unit, capacitive_load_unit])
    fp = open(processed_library_path, "w")
    fp.write(jsonstring+"\n")
    for cell in cells:

        dynamic_current = cell.get_groups("dynamic_current")
        leakage_power   = cell.get_groups("leakage_power")
        pins            = cell.get_groups("pin")
        cellName = str(cell.args[0])
        occurences_in_calibration_file = count_occurence(
    cellName)
        leakagePower = str(leakage_power[-1].get("value"))

        footprint = cell.get("cell_footprint")

        #pinLists = []
        input_pins =[]
        output_pins = []
        for pin in pins:
            #pinList = []
            pinName = pin.args[0]
            direction = pin.get("direction")
            pinfuction = None
            intPwr_lists = []
            pin_cap = 0
            pwrPin = pin.get("related_power_pin")
            gndPin = pin.get("related_ground_pin")
            if (direction == "output"):
                pinfunction = pin.get("function")
            else:
                pin_cap = pin.get("capacitance")
                #get internal power groups:
            internal_power_groups = pin.get_groups("
    internal_power")
            for intPwr in internal_power_groups:
```

```
                related_pin = intPwr.get("related_pin")
                when        = intPwr.get("when")
                risePwrGroup = intPwr.get_group("rise_power")
                fallPwrGroup = intPwr.get_group("fall_power")

                rise_arg = risePwrGroup.args[0]
                fall_arg = fallPwrGroup.args[0]

                if rise_arg != "scalar":
                    rise_i1 = risePwrGroup.get_array("index_1")
                    rise_values = risePwrGroup.get_array("values")
                if fall_arg != "scalar":
                    fall_i1     = fallPwrGroup.get_array("index_1")
                    fall_values = fallPwrGroup.get_array("values")
                fall_values_i = []
                rise_values_i = []
                fall_cap = []
                rise_cap = []
                powersum_list = []
                if direction == "output":
                    if fall_arg != "scalar":
                        fall_cap_np = fallPwrGroup.get_array("index_2")
                        fall_cap = fall_cap_np.tolist()[0]
                        for timeIndex in range(0, len(fall_i1[0])):
                            #print(fall_i1[0][timeIndex])
                            if str(fall_i1[0][timeIndex]) == "0.46":
                                fall_values_i = fall_values[timeIndex]
                                break

                        fall_values_i = fall_values_i.tolist()
                    if rise_arg != "scalar":
                        rise_cap_np = risePwrGroup.get_array("index_2")
                        rise_cap = rise_cap_np.tolist()[0]
                        for timeIndex in range(0, len(rise_i1[0])):
                            #print(rise_i1[0][timeIndex])
                            if str(rise_i1[0][timeIndex]) == "0.46":
                                rise_values_i = rise_values[timeIndex]
                                break

                    rise_values_i = rise_values_i.tolist()
                    if rise_arg != 'scalar' and fall_arg != 'scalar':
                        powersum_list = sum_list(rise_values_i, fall_values_i)
                    elif rise_arg == 'scalar':
                        powersum_list = fall_values_i
                    elif fall_arg == 'scalar':
                        powersum_list = rise_values_i
                    else:
                        #both scalar
                        powersum_list = [float(0)]
                elif direction == "input":
                    if rise_arg != "scalar":
                        for timeIndex in range(0, len(rise_i1[0])):
                            #print(rise_i1[0][timeIndex])
                            if str(rise_i1[0][timeIndex]) == "0.46":
                                rise_values_i = rise_values[0][timeIndex]
                                break
                        rise_values_i = [float(rise_values_i)]
                    if fall_arg != "scalar":
                        for timeIndex in range(0, len(fall_i1[0])):
                            #print(fall_i1[0][timeIndex])
                            if str(fall_i1[0][timeIndex]) == "0.46":
                                fall_values_i = fall_values[0][timeIndex]
                                break
                        fall_values_i = [float(fall_values_i)]
                    if rise_arg != 'scalar' and fall_arg != 'scalar':
                        powersum_list = sum_list(rise_values_i, fall_values_i)
                    elif rise_arg == 'scalar':
                        powersum_list = fall_values_i
```

```
                    elif fall_arg == 'scalar':
                        powersum_list = rise_values_i
                    else:
                        #both scalar
                        powersum_list = [float(0)]


                related_pin = str(related_pin).replace("\"","")
                when = str(when).replace("\"","")
                #make one for scalar as well so not that many empty lists?
                if (direction == "output" and (related_pin in input_set)):
                    intPwr_list = [related_pin, str(when).replace("\"",""), [rise_cap, powersum_list]] #[rise_cap, rise_values_i], [fall_cap, fall_values_i]]
                    intPwr_lists.append(intPwr_list)
                #inputs do not have related pins, remove them from list?
                elif (direction == "input"):
                    intPwr_list = powersum_list #[related_pin, str(when).replace("\"",""), powersum_list]#rise_values_i, fall_values_i]
                    intPwr_lists.append(intPwr_list)

            if direction == "output": #and (str(pinName) in output_set):
                output_pins.append([str(pinName), str(direction).replace("\"",""), str(pinfunction).replace("\"",""), str(pwrPin).replace("\"",""), str(gndPin).replace("\"",""), intPwr_lists ])
            else:# (str(pinName) in input_set):
                input_pins.append([str(pinName), str(direction).replace("\"",""), str(pin_cap), str(pwrPin).replace("\"",""), str(gndPin).replace("\"",""), intPwr_lists ])
            #print(pinLists)
        jsonstring = json.dumps([str(cellName),str(footprint).replace("\"",""),leakagePower,occurences_in_calibration_file, [input_pins, output_pins]], separators=(',', ':'))
        fp.write(jsonstring+"\n")

        #make json line with dumps
    fp.close()
    print("done extracting data after "+str((datetime.now()-starttime)/60)+" minutes")
```

```

def get_cells(filename):
    #cellLib = cell_library()
    cell_list = []
    #open file
    #read file line for line
    with open(filename, 'r') as svfile:
        line = svfile.readline()
        linenum = 1
        while line:
            #json loads on line to get all variables
            decoded_cell_line = json.loads(line)
            cell_list.append(decoded_cell_line)

            line = svfile.readline()
    return cell_list

#set_cells()
def sum_list(l1, l2):
    returnlist = []
    if len(l1) == len(l2):
        for i in range(0, len(l1)):
            returnlist.append(float(l2[i])+float(l1[i]))
    else:
        print("trying to sum lists of different length")
    return returnlist

#sort cells in regs, mux and logic?
def sort_cells(processed_library_path):
    cells = get_cells(processed_library_path)
    cell_environment = cells.pop(0)
    cell_list = []
    cellLib = cell_library()
    for c in cells:
        one_cell = cell(c, cellLib)
        cell_list.append(one_cell)
    cellLib.set_group_weights()

    #make groups handling select_op, add_op and comp_op
    adder = cell_group(['add_op'], 'adder')
    cellLib.combination_cells.append(adder)
    comp = cell_group(['comp_op'], 'comp')
    cellLib.combination_cells.append(comp)


```

```
    mult = cell_group( ['mult_op'], 'mult')
    cellLib.combination_cells.append(mult)
    #cellLib.print_available_cells()
    return cellLib

class cell:
    footprint       = ''
    name            = ''
    leakage_power   = 0
    #synthetic_gate_list = [] # litst containing equivalent synthetic gate list
    def __init__(self,def_list, cell_lib):
        self.synthetic_gate_list    = []
        self.footprint              = def_list[1]
        self.def_list               = def_list
        self.name                   = def_list[0]
        self.leakage_power          = def_list[2]
        self.calibration_count      = def_list[3]
        self.pin_list               = def_list[4]
        self.input_pins             = def_list[4][0]
        self.output_pins            = def_list[4][1]
        self.N_inputs               = len(self.input_pins)
        self.N_outputs              = len(self.output_pins)
        #look through dict and set syn gate sequence and name of cell.
        #dict corresponds to # inputs

        #need to append to correct list in cell_lib
        #check if mux, check if reg, else, check N_inputs
        match = False
        for key, l in rx_dict_reg_cells.items():
            #look for name match among registers
            match = l[0].search(self.name)
            if match:
                self.synthetic_gate_list = l[1]
                #look through list in cell lib for cell_group with matching key,
                group = cell_lib.find_cell_group(cell_lib.regs, key)
                if group == None:
                    # make new cell group
                    group = cell_group(self.synthetic_gate_list, key)
                    #append cell to list in group
                    group.append_cell(self)
                    #append cell group to list in library
                    cell_lib.regs.append(group)
                else:
                    #else append cell to list in cell_group
                    group.append_cell(self)
                break
        if not match:
            for key, l in rx_dict_mux_cells.items():
                #look for name match among multiplexers
                match = l[0].search(self.name)
                if match:
                    self.synthetic_gate_list = l[1]
                    group = cell_lib.find_cell_group(cell_lib.muxes, key)
                    if group == None:
                        # make new cell group
                        group = cell_group(self.synthetic_gate_list, key)
                        #append cell to list in group
                        group.append_cell(self)
                        #append cell group to list in library
                        cell_lib.muxes.append(group)
                    else:
                        #else append cell to list in cell_group
                        group.append_cell(self)
                    break
        if not match:
            for key, l in rx_dict_sel_cells.items():
                #look for name match among multiplexers
                match = l[0].search(self.name)
                if match:
                    self.synthetic_gate_list = l[1]
                    group = cell_lib.find_cell_group(cell_lib.selects, key)
                    if group == None:
                        # make new cell group
                        group = cell_group(self.synthetic_gate_list, key)
                        #append cell to list in group
                        group.append_cell(self)
                        #append cell group to list in library
                        cell_lib.selects.append(group)
                    else:
                        #else append cell to list in cell_group
                        group.append_cell(self)
```

```
                    break
        #first look through mux dict and reg dict for matches, if none:
        if not match:
            dictionary = get_dict_N(self.N_inputs)
            for key, l in dictionary.items():
                match = l[0].search(self.name)
                if match:
                    #found regex, set list and break
                    self.synthetic_gate_list = l[1]
                    l = cell_lib.get_list(self.N_inputs)
                    group = cell_lib.find_cell_group(l, key)
                    if group == None:
                        # make new cell group
                        group = cell_group(self.synthetic_gate_list, key)
                        #append cell to list in group
                        group.append_cell(self)
                        #append cell group to list in library
                        l.append(group)
                    else:
                        #else append cell to list in cell_group
                        group.append_cell(self)
                    break

#group together all cells with equivalent functionality
#if no cell_group matching "matching_key" is found, make new match group
#put cell groups in library instead of cells
class cell_group:
    matching_key = ''
    def __init__(self, sequence, matching_key):
        self.matching_key           = matching_key
        self.synthetic_gate_list    = sequence
        self.cells                  = []
        self.cellcounts             = []
        self.weights                = []
    def append_cell(self, cell):
        #N = count_occurence(cell.name)
        self.cellcounts.append(cell.calibration_count)
        self.cells.append(cell)
        #print("Appended cell "+cell.name)
        #print("Occurences: "+str(N))
    def get_weights(self):
        total_count = 0
        for i in self.cellcounts:
            total_count = total_count +i
        for i in range(0,len(self.cellcounts)):
            count = self.cellcounts[i]
            if total_count > 0:
                self.weights.append(round(count/total_count,2))
            else:
                self.weights.append(round(1/len(self.cellcounts), 2))
        for c in self.cells:
            print(c.name+", ", end='')
        print()
        print(self.weights)


def get_dict_N(N):
    if N == 1:
        return rx_dict_1_cells
    elif N == 2:
        return rx_dict_2_cells
    elif N == 3:
        return rx_dict_3_cells
    elif N == 4:
        return rx_dict_4_cells
    elif N == 5:
        return rx_dict_5_cells
    else:
        return rx_dict_6_cells


rx_dict_1_cells = {
    'not' : [re.compile(r"INV"), ['gtech_not']],
}
rx_dict_2_cells = {
    'and2'  : [re.compile(r"AN2X?D"),   ['gtech_and2']],
    'ind2'  : [re.compile(r"IND2D"),    ['gtech_not', 'gtech_and2', 'gtech_not']],
    'nand2' : [re.compile(r"ND2D"),     ['gtech_and2', 'gtech_not']],
    'nor2'  : [re.compile(r"^NR2X?D"),  ['gtech_or2', 'gtech_not']],
    'xnor2' : [re.compile(r"XNR2"),     ['gtech_xor2', 'gtech_not']],
    'or2'   : [re.compile(r"^OR2X?D"),  ['gtech_or2']],
    'xor2'  : [re.compile(r"^XOR2"),    ['gtech_xor2']],
```

```
    'inor2' : [re.compile(r"INR2X?D"),  ['gtech_not','gtech_or2'
    , 'gtech_not']],
    'andor22'  : [re.compile(r"AO22D"), ['select_op']], #
    duplicated here to be found when list are short
}
rx_dict_3_cells = {
    'and3'      : [re.compile(r"AN3X?D"),   ['gtech_and2','
    gtech_and2']],
    'nand3'     : [re.compile(r"^G?ND3D"),  ['gtech_and2','
    gtech_and2', 'gtech_not']],
    'inand3'    : [re.compile(r"^IND3D"),   ['gtech_not', '
    gtech_and2','gtech_and2', 'gtech_not']],
    'nor3'      : [re.compile(r"^G?NR3"),   ['gtech_or2', '
    gtech_or2','gtech_not']],
    'inor3'     : [re.compile(r"^INR3"),    ['gtech_not', '
    gtech_or2', 'gtech_or2','gtech_not']],
    'iao21'     : [re.compile(r"^IAO21"),   ['gtech_or2', '
    gtech_not','gtech_or2','gtech_not']],
    'or3'       : [re.compile(r"^OR3X?D"),  ['gtech_or2', '
    gtech_or2']],
    'xor3'      : [re.compile(r"^XOR3"),    ['gtech_xor2', '
    gtech_xor2']],
    'andori21'  : [re.compile(r"^G?AOI21D"),['gtech_and2', '
    gtech_or2', 'gtech_not']],
    'orand21'   : [re.compile(r"OA21"),     ['gtech_or2', '
    gtech_and2']],
    'xnor3'     : [re.compile(r"XNR3"),     ['gtech_xor2', '
    gtech_xor2', 'gtech_not']],
    'iorand21'  : [re.compile(r"IOA21D"),   ['gtech_and', '
    gtech_not','gtech_and', 'gtech_not']],
    'andori222' : [re.compile(r"MAOI222"),  ['gtech_and2', '
    gtech_or2', 'gtech_or2', 'gtech_not']]
}
rx_dict_4_cells = {
    'and4'      : [re.compile(r"AN4X?D"),   ['gtech_and2','
    gtech_and2','gtech_and2']],
    'nor4'      : [re.compile(r"^NR4"),     ['gtech_or2','
    gtech_or2','gtech_or2','gtech_not']],
    'nand4'     : [re.compile(r"^ND4D"),    ['gtech_and2','
    gtech_and2','gtech_and2', 'gtech_not']],
    'xnor4'     : [re.compile(r"XNR4"),     ['gtech_xor2', '
    gtech_xor2','gtech_xor2', 'gtech_not']],
    'xor4'      : [re.compile(r"^XOR4"),    ['gtech_xor2', '
    gtech_xor2','gtech_xor2']],
    'orand211'  : [re.compile(r"OA211"),    ['gtech_or2', '
    gtech_and2', 'gtech_and2']],
    'or4'       : [re.compile(r"^OR4X?D"),  ['gtech_or2', '
    gtech_or2', 'gtech_or2']],
    'iinor4'    : [re.compile(r"IINR4"),    ['gtech_not', '
    gtech_or2', 'gtech_or', 'gtech_or', 'gtech_not']],
    'andor211'  : [re.compile(r"AO211D"),   ['gtech_and2', '
    gtech_or2', 'gtech_or2']],
    # identical to 21 'iandor22'  : [re.compile(r"IAO22D"),   ['
    gtech_or2','gtech_not', 'gtech_or2', 'gtech_not']],
    # identical to 21 'orandi22'  : [re.compile(r"OAI22D"),   ['
    gtech_and2','gtech_not', 'gtech_and2', 'gtech_not']],

    #'andor22'  : [re.compile(r"AO22D"), ['select_op']],
    'andori31'  : [re.compile(r"AOI31D"),   ['gtech_and2','
    gtech_and2', 'gtech_or2', 'gtech_not']],
    'andor31'  : [re.compile(r"AO31D"),     ['gtech_and2', '
    gtech_and2','gtech_or2']],
    'ind4'      : [re.compile(r"IND4D"),    ['gtech_not', '
    gtech_and2', 'gtech_and2', 'gtech_not']]
}
rx_dict_5_cells = {
    'and5'      : [re.compile(r"^G?AN5D"),  ['gtech_and2','
    gtech_and2','gtech_and2', 'gtech_and2']],
    'nor5'      : [re.compile(r"^G?NR5"),   ['gtech_or2','
    gtech_or2','gtech_or2', 'gtech_or2','gtech_not']],
    'or5'       : [re.compile(r"^OR5"),     ['gtech_or2','
    gtech_or2','gtech_or2', 'gtech_or2']],
    'xor5'      : [re.compile(r"XOR5D"),    ['gtech_xor2','
    gtech_xor2','gtech_xor2', 'gtech_xor2']],
    'xnor5'     : [re.compile(r"XNR5D"),    ['gtech_xor2','
    gtech_xor2','gtech_xor2', 'gtech_xor2', 'gtech_not']],
    'nand5'     : [re.compile(r"ND5D"),     ['gtech_and2','
    gtech_and2','gtech_and2', 'gtech_and2','gtech_not']],
    'oa221'     : [re.compile(r"OA221"),    ['gtech_or2', '
    gtech_and2', 'gtech_and2']],
    'ao221'     : [re.compile(r"AO221"),    ['gtech_and2', '
    gtech_or2', 'gtech_or']],
}
rx_dict_6_cells = {
    'and6'      : [re.compile(r"AN6D"),   ['gtech_and2','
    gtech_and2','gtech_and2', 'gtech_and2', 'gtech_and2']],
    'nand6'     : [re.compile(r"ND6D"),   ['gtech_and2','
    gtech_and2','gtech_and2', 'gtech_and2', 'gtech_and2', '
    gtech_not']],
```

```
    'xnor6'      : [re.compile(r"XNR6"),    ['gtech_xor2','gtech_xor2','gtech_xor2', 'gtech_xor2', 'gtech_xor2', 'gtech_not']],
    'nor6'       : [re.compile(r"^NR6"),    ['gtech_or2','gtech_or2','gtech_or2', 'gtech_or2','gtech_or2', 'gtech_not']],
    'or6'        : [re.compile(r"^OR6"),    ['gtech_or2','gtech_or2','gtech_or2', 'gtech_or2','gtech_or2']],
    'xor6'       : [re.compile(r"XOR6"),    ['gtech_xor2','gtech_xor2','gtech_xor2', 'gtech_xor2', 'gtech_xor2']],
}
rx_dict_reg_cells = {
    'reg'    : [re.compile(r"DF[C|S|Q|N][N|D|C]"), ['reg']],
    'reg' : [re.compile(r"(L[H|N]Q)"), ['reg']]
}
rx_dict_mux_cells = {
    'mux2n' : [re.compile(r"MUX2N"), ['mux_op', 'gtech_not']],
    'mux2'  : [re.compile(r"MUX2"), ['mux_op']]
}
rx_dict_sel_cells = {
    'andor22'  : [re.compile(r"AO22D"), ['select_op']], # duplicated here to be found when list are short
}


class cell_library:
    def __init__(self):
        self.cells_6 = []
        self.cells_5 = []
        self.cells_4 = []
        self.cells_3 = []
        self.cells_2 = []
        self.cells_1 = []
        self.muxes   = []
        self.regs    = []
        self.selects = []
        self.combination_cells = []
        self.group_lists = [self.cells_6, self.cells_5, self.cells_4, self.cells_3, self.cells_2, self.cells_1, self.muxes, self.regs]
    def get_list(self, N):
        list_dict = {
            1 : self.cells_1,
            2 : self.cells_2,
            3 : self.cells_3,
            4 : self.cells_4,
            5 : self.cells_5,
            6 : self.cells_6
        }
        l = list_dict.get(N, lambda: None)
        if l == None:
            print("could not get list, no corresponding N")
            return l
        else:
            return l
    def find_cell_group(self, l, group_key):
        for g in l:
            if g.matching_key == group_key:
                #found, return group
                return g
        #not found, return none
        #if none make new group outside function
        return None
    def set_group_weights(self):
        for l in self.group_lists:
            for g in l:
                g.get_weights()
    def print_available_cells(self):
        print("6 input")
        for c in self.cells_6:
            print("\t"+c.matching_key)
        print("5 input")
        for c in self.cells_5:
            print("\t"+c.matching_key)
        print("4 input")
        for c in self.cells_4:
            print("\t"+c.matching_key)
        print("3 input")
        for c in self.cells_3:
            print("\t"+c.matching_key)
        print("2 input")
        for c in self.cells_2:
            print("\t"+c.matching_key)
        print("1 input")
        for c in self.cells_1:
            print("\t"+c.matching_key)
        print("registers")
        for c in self.regs:
            print("\t"+c.matching_key)
        print("muxes")
```

```
        for c in self.muxes:
            print("\t"+c.matching_key)

# find cell group in list of cell groups
def find_cell(key, l):
    for cell_group in l:
        if key == cell_group.matching_key:
            return cell_group
    return None

# count occurence of a word in a file
def count_occurence(word):
    path = path_to_calibration_file
    fp = open(path, "r")
    f = fp.read()
    return f.count(word)


#make list of synthetic cells into list of cells from cell library
def transform_list(cell_lib, to_transform):
    if len(to_transform) < 6:
        i = len(to_transform)+1#6
    else:
        i = 6
    #make list of index structs, sort list, then go through list and append
    ind = []
    templist = list(to_transform)
    #look for regs
    regindexes = find_sequence(templist, cell_lib.regs[0])
    if regindexes != []:
        for r in regindexes:
            ind.append(r)
            templist[r[0]] = 0
    selindexes = find_sequence(templist, cell_lib.selects[0])
    if selindexes != []:
        for r in selindexes:
            ind.append(r)
            templist[r[0]] = 0


    muxindexes = find_sequence(templist, cell_lib.muxes[0])
    #for mux in muxindexes:
    if muxindexes != []:
        for m in muxindexes:
            ind.append(m)
            templist[m[0]] = 0
    l = cell_lib.combination_cells
    for element in l:
        indexes = find_sequence(templist, element)
        if indexes != []:
                for k in indexes:
                    ind.append(k)

                #print("Found "+str(element.synthetic_gate_list)+" in "+str(to_transform))
                for ii in indexes:
                    for r in range(ii[0], ii[1]):
                        #print(r)
                        #templist.pop(r)
                        templist[r] = 0
    #look through lists looking for matches to replace sequences
    while i != 0:
        l = cell_lib.get_list(i)
        #print("called cell_lib.get_list "+str(i))
        #go through dict with that many inputs:
        #need to relate this list to cell group somehow as well to have power info available
        for element in l:
            #print("looking for "+str(element.synthetic_gate_list)+" in "+str(to_transform))
            indexes = find_sequence(templist, element) #need element as well in list, not only indexes

            if indexes != []:
                for k in indexes:
                    ind.append(k)
                #print("Found "+str(element.synthetic_gate_list)+" in "+str(to_transform))
                for ii in indexes:
                    for r in range(ii[0], ii[1]):

                        templist[r] = 0


        i = i-1
    #replace found indexes when they are found so they cannot be found again
    ind.sort()
    elementlist = []
```

```

    for indexelement in ind:

        elementlist.append(indexelement[2])

    return elementlist

# returns none if indexes not in list or list of (startindex, stopindex) for each occurence
# also returns the index(es) it found
def find_sequence(to_find, element):
    l = element.synthetic_gate_list
    indexes = []
    temp = list(to_find)

    for i in range(len(temp)):

        if temp[i:i+len(l)] == l:
            temp[i:i+len(l)] = [0]*len(l)

            indexes.append((i, i+len(l), element))
    return indexes
```

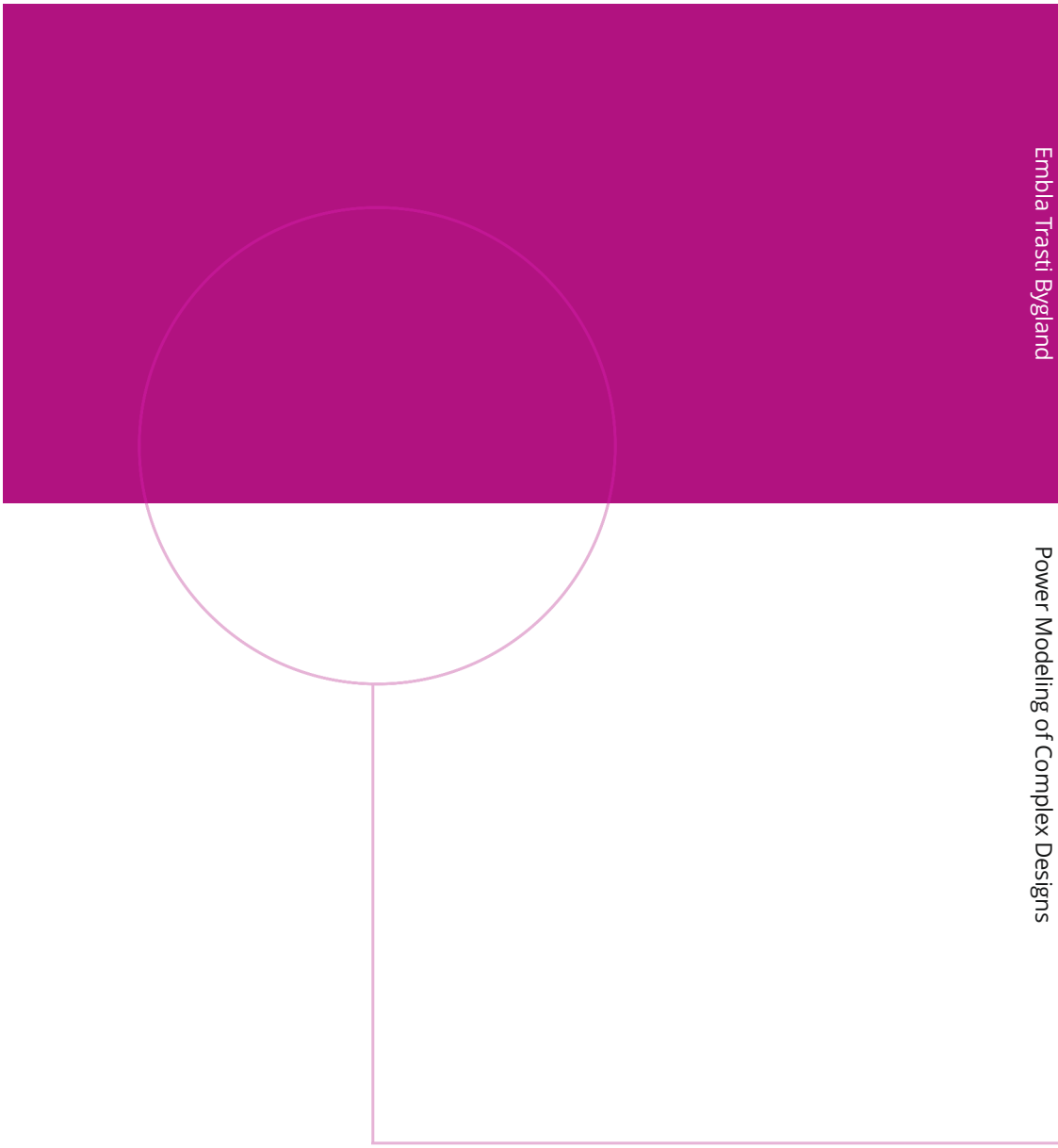
F Code implemented in Chapter 8

```
import parse_elab
import liberty_data
import sys

filename = sys.argv[1]




#list of all cells from the cell library
cellLib = liberty_data.sort_cells(liberty_data.
    processed_library_path)


powerStructures = []

#iterate through structures
def go_through_structures():
    modules, top_structures = parse_elab.run_parse_elab(filename
    )
    for s in top_structures:

        p = power_structure(s)
        p.parent = None
        p.name = s.represented_object_handle.name
        powerStructures.append(p)

        elementlist = liberty_data.transform_list(cellLib ,[s.
    represented_object_handle.id])
        p.cell_lib_list = elementlist
        p.structural_rep_list = [s]

        srepr = repr(s)
        print(s.represented_object_handle.name)
        print(srepr)
        count_powerStructure(p)
        lists_from_top(s,p)
    for p in powerStructures:

        print(p.name)
        #v = value()
        #p.print(v, 0)
        #print()

        r = repr(p)
        print(r)
    print_powercount()

def print_stuff(top_structures):
    for s in top_structures:
        print("Top structure: ")
        print(s.represented_object_handle.name)
        print("Children:")
        for c in s.children:
            print("\t"+c.represented_object_handle.name)
            #print("\tlvl3: ")
            for gc in c.children:
                print("\t\t"+gc.represented_object_handle.name)
                for ggc in gc.children:
                    print("\t\t\t"+ggc.represented_object_handle
    .name)


def lists_from_top(s, power_s):
    #print(s.represented_object_handle.name)

    l = []
    #while s != None:
    st = None
    for c in s.children:
        #s = c
        #l = []
        #structure_list = []
        #add_s_list = []
```

```
        ##goes through structure elements until fanout
        #st = add_structure(l, structure_list, c, add_s_list)
        ##print("\t",end='')
        ##print(l)
        ##go through cell lists compare to l
        #elementlist = liberty_data.transform_list(cellLib,l)
        #make power structure object with s as parent
        #p = power_structure(s, structure_list, elementlist)
        #power_s.children.append(p)
        if c.powerStructure == None:
            l = []
            structure_list = []
            add_s_list = []
            #goes through structure elements until fanout
            st = add_structure(l, structure_list, c, add_s_list)
            #print("\t",end='')
            #print(l)
            #go through cell lists compare to l
            elementlist = liberty_data.transform_list(cellLib,l)

            p = power_structure(power_s)
            for s in structure_list:
                s.powerStructure = p
            power_s.children.append(p)
            p.structural_rep_list = structure_list
            p.cell_lib_list = elementlist
            #count_powerStructure(p)
            #if output_nodes[0] == None or is reg
            if c.structure_connection_characteristic != 'control
':
                #if c.represented_object_handle.output_nodes[0]
!= None:
                count_powerStructure(p)
            for add_l in add_s_list:
                #after fanout, go through children until fanout
...
                #for c in st.children:
                lists_from_top(add_l, p)
        else:
            p = c.powerStructure
            power_s.children.append(p)

        #for e in elementlist:
        #    print(e.matching_key)
        #if st != None:
        #for add_l in add_s_list:
        #    #after fanout, go through children until fanout...
        #    #for c in st.children:
        #    lists_from_top(add_l, p)

def add_structure(l, structure_list, s, s_list):

    #look at what rep obj handle is and change it if select, add
     etc
    l.append(s.represented_object_handle.id)
    structure_list.append(s)
    #print("\t"+s.represented_object_handle.name)
    if len(s.children) == 1 and s.children[0] != None:
        #print(s.represented_object_handle.name)
        add_structure(l, structure_list, s.children[0], s_list)
    elif len(s.children) == 0:
        #no child
        return None
    else:

        s_list.append(s)
        return s

#save first elements parent, and last elements children,
#envelop with structure having old objects in list and new
    objects in list
class power_structure:
    name = ''
    countedBool = False
    def __init__(self, parent):
        self.structural_rep_list    = []
        self.cell_lib_list          = []
        self.children               = []
        self.parent                 = parent
    #def printstart(self):
    #    for c in self.cell
    def print(self, depth, startlvl):

        #print(self.represented_object_handle.id+", ",end = '')
        #if self.children != []: print("{", end = '')
        #print("{ ", end = '')
        if self.cell_lib_list != []:
            print("{", end = '')
            depth.i = depth.i+1
        #if self.children != []: print("{", end='')
```

```
        for c in range(len(self.cell_lib_list)):
            if c+1 == len(self.cell_lib_list):
                print(self.cell_lib_list[c].matching_key+" ",end = '')
                if self.children != []:
                    depth.i = depth.i+1
                    print("{", end = '')
                #else:
                #    if lastchild: print("}", end = '')
            else:
                print(self.cell_lib_list[c].matching_key+" {", end = '')
                depth.i = depth.i +1
        #if self.cell_lib_list != []:
        #    print("}", end = '')

        #elif (len(self.children))
        #if self.children != []:
        #    print("{", end='')
        #childbracketcount = len()

        for child in range(len(self.children)):
            #print(self.represented_object_handle.id+", ",end = '')
            if child+1 == len(self.children):
                self.children[child].print(depth, depth.i)
            else:
                self.children[child].print(depth, depth.i)
            #if self.children[child].children == []:
            #    print("}",end='')
                #childbracketcount = childbracketcount-1
            #else:
            #    print(", ", end= '')

        #if lastchild: print("}",end='')
        #if self.children == []: print("}",end='')
        #if self.children != []: print("}", end='')
        #if self.cell_lib_list != [] and self.children == []:
        #    print("}", end = '')
        #    depth.i = depth.i -1

        while depth.i != startlvl:
            print("}", end = '')
            depth.i = depth.i-1
        #if self.children != []: print("}", end='')
        #if self.cell_lib_list == []:
        #    print("}", end = '')
        #print( )
        #print("}", end = '')
            #for c in child.cell_lib_list:
            #    print(c.matching_key+" ",end = '')

    def __repr__(self, level=0):
        value = ''
        for v in self.cell_lib_list:
            value = value+v.matching_key+" "
        ret = "\t"*level+repr(value)+"\n"
        if level < 11:
            for child in self.children:
                ret += child.__repr__(level+1)
        return ret
class value:
    i = 0
        #if self.children == []: print("}", end = '')

nots    = 0
logic   = 0
mux     = 0
arithm  = 0
comp    = 0
regs     = 0
def count_powerStructure(s):
    notstruct = {'not'}
    logicstruct = {'and5', 'nor5', 'or5', 'xor5', 'xnor5', '
    nand5', 'oa221', 'ao221', 'andor22' ,'andori31' ,'andor31' ,
    'ind4', 'and4','nor4','nand4','xnor4','xor4','orand211','or4
    ','iinor4','andor211', 'and3','nand3','inand3','nor3','inor3
    ','iao21','or3','xor3','andori21' ,'orand21','xnor3','
    iorand21' ,'andori222','and2' ,'ind2' ,'nand2','nor2' ,'
    xnor2','or2'  ,'xor2' ,'inor2'  }
    regstruct = {'reg'}
    muxstruct = {'mux', 'mux2n'}
    arithmstruct = {'adder', 'mult'}
    compstruct = {'comp'}
    global nots
    global logic
    global mux
    global arithm
    global comp
    global regs
```

```
    #print(s.cell_lib_list)
    str_rep_offset = 0
    strlist2 = []
    removestruct = {'input', 'output', 'gtech_buf'}
    if s.countedBool == False:
        for obj in s.structural_rep_list:
            h = obj.represented_object_handle
            if h.id in removestruct:
                pass
            else:
                strlist2.append(obj)
        s.countedBool = True
        for cellindex in range(0,len(s.cell_lib_list)):

            cell = s.cell_lib_list[cellindex]

            structure = strlist2[cellindex+str_rep_offset]
            handle = structure.represented_object_handle

            handle = structure.represented_object_handle

            str_rep_offset = str_rep_offset + len(cell.
    synthetic_gate_list)-1

            if handle.id == 'reg' :

                if handle.has_parent:
                    if cell.matching_key in regstruct and
    cellindex < 1:
                        regs = regs +1
                        print("added reg")
            elif handle.id != 'input' and handle.id != 'output'
        and structure.structure_connection_characteristic != '
        control':
                if handle.output_nodes[0] != None:
                    if cell.matching_key in notstruct:
                        nots = nots+1
                        print("added not")
                    elif cell.matching_key in logicstruct:
                        logic = logic+1
                        print("added logic")
                    elif cell.matching_key in muxstruct:
                        mux = mux+1
                        print("added mux")
                    elif cell in arithmstruct:
                        arithm = arithm +1
                        print("added arithm")
                    elif cell.matching_key in compstruct:
                        comp = comp+1
                        print("added comp")

#need to find a good way to count so not multiplied....
    structure... make objects?
def print_powercount():
    print("nots:\t"+str(nots))
    print("regs:\t"+str(regs))
    print("logic:\t"+str(logic))
    print("mux:\t"+str(mux))
    print("arithm:\t"+str(arithm))
    print("comp:\t"+str(comp))
    print("total:\t"+str(nots+logic+mux+arithm+comp+regs))
go_through_structures()
```



Embla Trasti Bygland

Power Modeling of Complex Designs

NTNU
Norwegian University of
Science and Technology