Examensarbete

Improved implementation of a 1K FFT with low power consumption

Petter Näslund, Mikael Åkesson

LITH-ISY-EX--05/3737--SE
Improved implementation of a 1K FFT with low power consumption

Division of Electronics Systems, Linköpings Universitet

Petter Näslund, Mikael Åkesson

LITH-ISY-EX--05/3737--SE

Examensarbete: 20 p

Level: D

Supervisor: Kenny Johansson,
Division of Electronics Systems, Linköpings Universitet

Examiner: Oscar Gustafson,
Division of Electronics Systems, Linköpings Universitet

Linköping: April 2005
Improved implementation of a 1K FFT with low power consumption

In this master thesis, a behavioral VHDL model of a 1k Fast Fourier Transform (FFT) algorithm has been improved, first to make it synthesizable and second to obtain a low power consumption. The purpose of the thesis has not been to focus on the FFT algorithm itself or the theory behind it. Instead the aim has been to document and motivate the necessary modifications, to reach the stated requirements, and to discuss the results. The thesis is divided into sections so that the design flow closely can be followed from the initial FFT, down to the final architecture. The two major design steps covered are synthesis and power simulation.

The synthesis process has been the most time consuming part of the thesis. The synthesis tool Cadence Ambit PKS was used. Throughout the synthesis, the modifications and solutions will be discussed and comparisons are continuously made between the different solutions and the initial FFT. The best solution will then be the starting point in the next design step, which is simulation of the design with respect to power consumption. This is done by using a simulation tool from Synopsys called NanoSim. Also here, every solution is tested and compared to each other, followed by a concluding discussion. The technology used to implement the design is a $0.35\mu m$ CMOS process.

FFT, synthesis, low power, VHDL.
Abstract

In this master thesis, a behavioral VHDL model of a 1k Fast Fourier Transform (FFT) algorithm has been improved, first to make it synthesizable and second to obtain a low power consumption. The purpose of the thesis has not been to focus on the FFT algorithm itself or the theory behind it. Instead the aim has been to document and motivate the necessary modifications, to reach the stated requirements, and to discuss the results. The thesis is divided into sections so that the design flow closely can be followed from the initial FFT, down to the final architecture. The two major design steps covered are synthesis and power simulation.

The synthesis process has been the most time consuming part of the thesis. The synthesis tool Cadence Ambit PKS was used. Throughout the synthesis, the modifications and solutions will be discussed and comparisons are continuously made between the different solutions and the initial FFT. The best solution will then be the starting point in the next design step, which is simulation of the design with respect to power consumption. This is done by using a simulation tool from Synopsys called NanoSim. Also here, every solution is tested and compared to each other, followed by a concluding discussion. The technology used to implement the design is a 0.35 µm CMOS process.

Keywords: FFT, synthesis, low power, VHDL.
Acknowledgements

We would like to thank our examiner Oscar Gustafsson for giving us the opportunity to do our master thesis here at Electronics Systems, Linköping University. And of course for his ideas, confusing us even more along the way. We thank our supervisor Kenny Johansson for all his bright ideas and for borrowing his binder including the NanoSim users guide. Emil Hjalmarsson helped us a great deal in the beginning with Cadence Layout. Henrik Ohlsson has helped us getting started with Cadence Ambit PKS and has given us many valuable hints along the way. We thank our class mate Markus Åkerman, who also made his master thesis at Electronics Systems, for giving good support, to have someone to have lunch with and for car pooling to and from school. We would also like to thank the authors of \LaTeX, \TeX and the team behind \LaTeX{} in which this report was written. Last but not least we would like to thank Arnold Schwarzenegger for being such an inspiration source. Here are some well suited quotes:

*Strength does not come from winning. Your struggles develop your strengths. When you go through hardships and decide not to surrender, that is strength.*

Arnold Schwarzenegger

*The mind is the limit. As long as the mind can envision the fact that you can do something, you can do it, as long as you really believe 100 percent.*

Arnold Schwarzenegger
# Contents

<table>
<thead>
<tr>
<th>1</th>
<th>Introduction</th>
<th>1</th>
</tr>
</thead>
<tbody>
<tr>
<td>1.1</td>
<td>Purpose</td>
<td>1</td>
</tr>
<tr>
<td>1.2</td>
<td>Background</td>
<td>2</td>
</tr>
<tr>
<td>1.3</td>
<td>Limitations</td>
<td>2</td>
</tr>
<tr>
<td>1.4</td>
<td>Time planing</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>Initial FFT</td>
<td>5</td>
</tr>
<tr>
<td>2.1</td>
<td>Theory</td>
<td>6</td>
</tr>
<tr>
<td>2.2</td>
<td>Block functions</td>
<td>7</td>
</tr>
<tr>
<td>2.3</td>
<td>Specification</td>
<td>8</td>
</tr>
<tr>
<td>3</td>
<td>Work flow and software</td>
<td>9</td>
</tr>
<tr>
<td>3.1</td>
<td>Software</td>
<td>9</td>
</tr>
<tr>
<td>3.1.1</td>
<td>Version Control</td>
<td>9</td>
</tr>
<tr>
<td>3.1.2</td>
<td>VHDL compiler</td>
<td>10</td>
</tr>
<tr>
<td>3.1.3</td>
<td>VHDL simulator</td>
<td>10</td>
</tr>
<tr>
<td>3.1.4</td>
<td>Synthesis</td>
<td>10</td>
</tr>
<tr>
<td>3.1.5</td>
<td>Power simulation</td>
<td>10</td>
</tr>
<tr>
<td>3.2</td>
<td>Work flow</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>Controller</td>
<td>13</td>
</tr>
<tr>
<td>4.1</td>
<td>Background</td>
<td>13</td>
</tr>
<tr>
<td>4.1.1</td>
<td>Implementation</td>
<td>13</td>
</tr>
<tr>
<td>5</td>
<td>Synthesis</td>
<td>15</td>
</tr>
<tr>
<td>5.1</td>
<td>Goal</td>
<td>15</td>
</tr>
<tr>
<td>5.2</td>
<td>Controller</td>
<td>16</td>
</tr>
<tr>
<td>5.2.1</td>
<td>Problems</td>
<td>18</td>
</tr>
<tr>
<td>5.3</td>
<td>Butterfly</td>
<td>18</td>
</tr>
<tr>
<td>5.3.1</td>
<td>Initial implementation</td>
<td>18</td>
</tr>
<tr>
<td>5.3.2</td>
<td>Serial interface</td>
<td>20</td>
</tr>
<tr>
<td>5.3.3</td>
<td>Synthesis of initial model</td>
<td>20</td>
</tr>
</tbody>
</table>
6 Power simulation

6.1 Testing the FFT
   6.1.1 Test case 1 (Black box RAM and ROM table)
   6.1.2 Test case 2 (Fake RAM and ROM table)
   6.1.3 Test case 3 (Fake RAM and fake ROM table)
   6.1.4 Test case summary

6.2 Simulating the complete FFT

6.3 Conclusion

7 Conclusions and discussion

7.1 Synthesis

7.2 Modified FFT vs Initial FFT

7.3 Miscellaneous

7.4 Further development

A Cadence PKS and Ambit BuildGates

A.1 Introduction

A.2 Notation and TCL script

A.3 Synthesis flow
   A.3.1 Libraries
   A.3.2 Register Transfer Level (RTL) synthesis
   A.3.3 Timing constraints
   A.3.4 Floorplaning
   A.3.5 Optimization
   A.3.6 Post-Placement optimization
   A.3.7 do_optimize
      A.3.7.1 Pre placement optimization
      A.3.7.2 Path groups
      A.3.7.3 Generic logic optimization
      A.3.7.4 Dissolving (break up) Hierarchy
      A.3.7.5 Preserve design
List of Figures

2.1 Schematic of the initial FFT ........................................ 5
3.1 Thesis work flow .................................................... 11
5.1 Butterfly initial computation for ResultA .................. 19
5.2 Butterfly initial computation for ResultB ................ 19
# List of Tables

<table>
<thead>
<tr>
<th>Table</th>
<th>Description</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>5.1</td>
<td>Synthesis result of the Controller</td>
<td>17</td>
</tr>
<tr>
<td>5.2</td>
<td>Initial Butterfly operation</td>
<td>18</td>
</tr>
<tr>
<td>5.3</td>
<td>Synthesis area result for initial Butterfly with goal frequency 1 MHz</td>
<td>20</td>
</tr>
<tr>
<td>5.4</td>
<td>Synthesis timing result for initial Butterfly</td>
<td>21</td>
</tr>
<tr>
<td>5.5</td>
<td>Improvement I - rearrangement</td>
<td>22</td>
</tr>
<tr>
<td>5.6</td>
<td>Synthesis timing result for rearranged Butterfly</td>
<td>22</td>
</tr>
<tr>
<td>5.7</td>
<td>Area result for Butterfly with resource sharing (1 MHz)</td>
<td>23</td>
</tr>
<tr>
<td>5.8</td>
<td>Improvement II - resource sharing cycles</td>
<td>23</td>
</tr>
<tr>
<td>5.9</td>
<td>Synthesis result for Butterfly with resource sharing</td>
<td>23</td>
</tr>
<tr>
<td>5.10</td>
<td>Serial interface</td>
<td>24</td>
</tr>
<tr>
<td>5.11</td>
<td>Final synthesis result for Butterfly</td>
<td>25</td>
</tr>
<tr>
<td>5.12</td>
<td>Synthesis result of BaseIndexGen</td>
<td>26</td>
</tr>
<tr>
<td>5.13</td>
<td>Synthesis result of WPGen</td>
<td>27</td>
</tr>
<tr>
<td>5.14</td>
<td>Synthesis result of AddressGen</td>
<td>28</td>
</tr>
<tr>
<td>5.15</td>
<td>Synthesis result of OutputCtrl</td>
<td>28</td>
</tr>
<tr>
<td>5.16</td>
<td>Synthesis result of DigitReversal</td>
<td>28</td>
</tr>
<tr>
<td>6.1</td>
<td>Summary of the top module of the three test cases</td>
<td>31</td>
</tr>
<tr>
<td>6.2</td>
<td>Summary of the complete FFT processor</td>
<td>32</td>
</tr>
<tr>
<td>A.1</td>
<td>Synthesis result of WPGen with SOP = false</td>
<td>46</td>
</tr>
<tr>
<td>A.2</td>
<td>Synthesis summary for addition/subtraction using 24 bits signed</td>
<td>51</td>
</tr>
<tr>
<td>A.3</td>
<td>Synthesis summary for subtraction using 44 bits signed</td>
<td>51</td>
</tr>
<tr>
<td>A.4</td>
<td>Synthesis summary for multiplication using 24 bits signed</td>
<td>52</td>
</tr>
<tr>
<td>A.5</td>
<td>LPS synthesis result</td>
<td>52</td>
</tr>
<tr>
<td>B.1</td>
<td>Results of test case 1</td>
<td>55</td>
</tr>
<tr>
<td>B.2</td>
<td>Results of test case 2</td>
<td>56</td>
</tr>
<tr>
<td>B.3</td>
<td>Results of test case 3</td>
<td>56</td>
</tr>
<tr>
<td>B.4</td>
<td>Results of complete FFT</td>
<td>57</td>
</tr>
</tbody>
</table>
Nomenclature

Abbreviations

<table>
<thead>
<tr>
<th>Acronym</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CAD</td>
<td>Computer Aided Design</td>
</tr>
<tr>
<td>CMOS</td>
<td>Complementary Metal Oxide Semiconductor</td>
</tr>
<tr>
<td>CVS</td>
<td>Concurrent Versions System</td>
</tr>
<tr>
<td>DFT</td>
<td>Discrete Fourier Transform</td>
</tr>
<tr>
<td>DSP</td>
<td>Digital Signal Processing</td>
</tr>
<tr>
<td>FFT</td>
<td>Fast Fourier Transform</td>
</tr>
<tr>
<td>FIR</td>
<td>Finite Impulse Response</td>
</tr>
<tr>
<td>FSM</td>
<td>Finite State Machine</td>
</tr>
<tr>
<td>I/O</td>
<td>Input/Output</td>
</tr>
<tr>
<td>ISY</td>
<td>Department of Electrical Engineering</td>
</tr>
<tr>
<td>PKS</td>
<td>Physically Knowledgeable Synthesis</td>
</tr>
<tr>
<td>PLL</td>
<td>Phase Looked Loop</td>
</tr>
<tr>
<td>RAM</td>
<td>Random Access Memory</td>
</tr>
<tr>
<td>ROM</td>
<td>Read Only Memory</td>
</tr>
<tr>
<td>TF</td>
<td>Twiddle Factor</td>
</tr>
<tr>
<td>VHDL</td>
<td>VHSIC Hardware Description Language</td>
</tr>
<tr>
<td>VHSIC</td>
<td>Very High Speed Integrated Circuit</td>
</tr>
</tbody>
</table>

Notations

The synthesis program used is developed by Cadence Design Systems Inc. As described in [1] Cadence bought Ambit in 1999, including the tool *Ambit BuildGates* (AmbitBG). Moreover, Cadence has developed a more physical aware synthesis tool, named *Cadence PKS* (CPKS). But to confuse even more CPKS uses AmbitBG for synthesis and activates only when more accurate physical timing is required. Therefore some parts are performed by AmbitBG and other by CPKS. So when referring to CPKS both tools are included.
1.1 Purpose

The starting point of this master thesis is to analyze an existing design of an FFT processor. The design in question is made by Rongzeng Mu in his master thesis [2]. This design was supposed to be built upon the FFT case study in Lars Wanhammars book [3]. This case study is made in a 0.35 $\mu$m process and our main goal is to implement the same design but in a 0.13 $\mu$m process and examine the behavior. Our work is meant to be considered as background material to the FFT case study in the next addition of the book DSP Integrated Circuits. Our first objective is to analyze the VHDL model of the FFT and modify the part of the code that is not synthesizable. The synthesis tools to be used are Leonardo and CPKS. At this point some interesting questions are:

- What can the synthesis tools do and not do?
- Which operations are critical?
- How is the implementation done?
- Can it be done in a better way?

These questions are for us to consider and answer. There are two ways in improving the design: rewrite the code, or to use full custom design of the critical parts. This instead of using the standard cells that are available for the process. A full custom design can be done in the Cadence Virtuoso Layout environment.

When the synthesis has reached satisfying results our next objective will be to simulate the design with respect to power consumption. The aim is to reach as low power levels as possible. Also in this step there might be critical parts needed to be modified in the model. This will also be done by either rewriting the VHDL code or to improve the full custom designs.

Our last objective is to convert the design from the 0.35 $\mu$m process (this process will be used until the model is fully functional) to the intended 0.13 $\mu$m process. Can the transfer be done without problems or are there effects that need to be considered? This is for us to discover. Through the design the voltage will be set to 3.3 V and the ambient temperature will be set to 105 degrees Celsius.
1.2 Background

What is an FFT processor and in what applications are they used? FFT is an abbreviation for Fast Fourier Transform and the algorithm is often connected to Digital Signal Processors (DSP’s). This report will not deeply show the theory behind the FFT algorithm but only shortly explain the basic concepts.

Basically the FFT is a fast algorithm transforming information from the time domain to the frequency domain, and vice versa. This feature is a frequent requirement in signal processing for the detection and analysis of frequency components that are of interest. The algorithm is widely used in many areas, but to mention a few applications that is related to the field of engineering and signal processing, communication and filtering can be included. In communications theory for example, the signal processed is usually a voltage or a current. In order to understand how these signals behave when they pass through a filter, amplifier or communication channel, the FFT can be utilized to analyze the frequency components. Focusing more on digital signals with discrete values of “0” and “1” they also include frequency contents which can be computed with the FFT. When it comes to filtering signals the FFT is a good tool as for transforming the signal from the time domain to the frequency domain. This gives the opportunity to view the characteristics of the signal before and after the filtering. The FFT is often used to improve the performance of FIR filters [4].

The idea of this master thesis came up when we contacted the div. of E.S. Linköping University. The person who was to become our examiner, Oscar Gustafsson, wanted us to continue the work on a master thesis previously done. This master thesis was made by Rongzeng Mu a few years earlier at ISY [2]. His assignment was to implement a FFT processor according to the FFT case study in Lars Wanhammars book “DSP Integrated Circuits” [3]. The status of this project was uncertain. Our master thesis was stated to continue where Mu left of.

1.3 Limitations

After reaching a comprehensive overview on the assignment in this master thesis, two initial limitations were stated. The first one is, not to include the RAM and ROM in the synthesis flow, but only in the simulation flow. The motivation to this is, since implementing our own effective memories can be considered as a master thesis itself (at a minimum). Therefore, the decision was made to use existing memory models for simulation purpose and a black box memory for synthesis purpose. The second limitation is to be considered during the synthesis flow. The attentions, are only to redesign what we think is necessary to complete the synthesis, and not to rewrite the basic algorithm. Hence the algorithm will be the same but implemented differently.
1.4 Time planing

Since the status of Mu’s work (which was the starting point of our master thesis) was uncertain, it was hard to come up with a time plan for our work. Our time is despite limited to 20 weeks. The design step which is going to be the most prominent and time consuming is hard to predict. However, the goal is to at least touch and acquire an appraisal on how the different design steps are executed.
Initial FFT

In this chapter a brief introduction to the theory behind the FFT algorithm will be presented. The idea is not to be comprehensive but only to explain the basic concepts and theory behind FFT algorithm. The readers who are interested in a deeper theory study behind the FFT algorithm are referred to [2] and [3].

Figure 2.1: Schematic of the initial FFT

In order for the reader to get an understanding in what has been accomplished during the master thesis, also a brief description of the structure and function of the initial FFT (starting point) will be presented in this chapter. In Fig. 2.1 a
schematic of the initial FFT processor can be seen. As mentioned earlier the basic algorithm of the FFT will not be changed. Therefore, only the major function of each building block will be explained. Later chapters will emphasize on what is changed and/or improved in each block.

2.1 Theory

The discrete Fourier transform (DFT) is the mathematical method used when analyzing the frequency components of a sampled signal [5]. The Fast Fourier Transform (FFT) is a more efficient algorithm to compute the DFT. The $N$-point DFT (in this project $N = 1024$) is defined by Eq. (2.1):

$$X(k) = \sum_{n=0}^{N-1} x(n) \cdot W_N^{kn}, \quad \left\{ \begin{array}{ll} k & = 0, 1, 2, \ldots, N-1 \\ W_N^{kn} & = e^{-j2\pi/N} \end{array} \right. \quad (2.1)$$

If the DFT algorithm is used directly when computing, the number of additions and multiplications are in the order $N^2$. On the other hand if the FFT algorithm is used to compute the DFT, the order of additions and multiplications are reduced to $(N/2) \cdot \log_2(N)$. When using the FFT algorithm it is possible to express the DFT as a weighted sum of two $N/2$-point DFT’s (one for the odd and one for the even index, respectively). The new expression is defined by Eq. (2.2):

$$X(k) = \sum_{n=0}^{(N/2)-1} x(2n) \cdot W_{N/2}^{kn} + W_N^k \sum_{n=0}^{(N/2)-1} x(2n+1) \cdot W_{N/2}^{kn} \quad (2.2)$$

Where $k = 0, 1, 2, \ldots, N-1$, and since $W_{N/2}^k$ is periodic with the period $N/2$, each $N/2$-point DFT only have to be evaluated for $k = 0, 1, 2, \ldots, (N/2) - 1$. The complex weighting factors are called Twiddle Factors (TF). By using the definition of the TF and Euler’s formula, it can be shown that cosines and sines operations are required when computing the TF’s, see Eq. (2.3):

$$W_N^k = e^{-j2\pi \cdot m/N} = \cos(2 \cdot \pi \cdot m/N) - j \cdot \sin(2 \cdot \pi \cdot m/N) \quad (2.3)$$

For efficiency the TFs are computed prior or “outside” the FFT and stored in a look-up table, i.e. a ROM.
2.2 Block functions

- **AddresGen**
  This block generates the address and the read and write signals to the two RAM's.

- **StagePE**
  This block keeps track of the current stage of the FFT: *input*, *compute*, or *output* and the different substages in *compute*. It basically increments a counter and compares it with predefined values.

- **BaseIndexGen**
  During *input* and *output* stage, this block generates the index to the RAM. In the *output* stage it also generates a signal which decides from which RAM to read. The index is generated according to the algorithm defined in [3].

- **Butterfly(0/1)**
  The Butterfly's are the process blocks that performs the FFT computations. Several arithmetic operations such as multiplications, additions, and subtractions are included. These blocks reads the TF from *WPGen*, data from RAM, executes the computations and sends the result back to RAM.

- **CacheCtrl(0/1)**
  The *CacheCtrl's* can be considered as buffers. Their only purpose is to be a middle hand between the RAM and the *Butterfly's*.

- **DigitReversal**
  This block reverses the address in the *output stage* so that the data is read from RAM in the correct order.

- **OutputCtrl**
  The *OutputCtrl* is activated when the *compute stage* is finished. The *outputLen* signal is set and the resulting data is read from either RAM0 or RAM1 to the output port of the FFT.

- **WPGen**
  *WPGen* computes an address and then reads the raw TF from a ROM. The raw TF is then processed in a special way and sent to the *Butterfly*. The TF is a 42 bit complex number where 21 bits are used for the real imaginary part, respectively [3].
2.3 Specification

These are the specifications for the initial FFT model described in [2]:

- Only behavioral description (ideal).
- Two *Butterfly’s*, two ideal RAM’s.
- Three stages in the FFT: *input*, *compute*, and *output*.
- A parallel interface between the *Butterfly’s* and the *CacheCtrl’s*.
- 1024 point FFT (16 *real* + 16 *img*).
- The *compute* stage converts the 32 bit input to 48 bits for higher accuracy.
- One computation per four memory cycles.
- The intended RAM speed is 32 MHz.
- Requires 12425 clock cycles to do the three stages.
- Throughput (FFT computations per second):
  \[
  \frac{1}{32 \cdot 10^6} \cdot 12425 \approx 2500 \text{ (FFT's/s)}
  \]
Work flow and software

To be able to realize a complex electronic circuit in today’s rapidly developing market, the use of modern software tools is a must. The reason why this is mentioned in the report, is to highlight that a lot of time has been spent during the project to learn the different tools.

3.1 Software

The initial FFT processor was designed in Mentor Graphics HDL Designer (MHDL). MHDL is a graphical CAD tool, which makes it easier to survey the hierarchy and structure of the circuit. But when the complexity of the circuit increase even a nice schematic might confuse and the comprehensibility of the design will fade. This tool was considered hard to work with since it did not behave as expected. The solution to this was to rewrite the design to structural VHDL code using XEmacs. When this was done, it became easier to experiment and make changes to the design. Another gain by writing structural VHDL code instead of using a graphical CAD tool is that the design becomes platform independent (which in some cases may be preferable).

3.1.1 Version Control

After rewriting of the VHDL code the possibility to use a version control system was discussed. This because when two or more persons works on the same project and edit the same files it will often lead to confusion about which version is the most recent, what changes were made, etc. The conclusion was to use a version control system which keeps track of versions, author, changes, dates etc and the ability to revert to older versions. The basic principle of operation is that the files is saved in a repository from which you can checkout projects and get a local copy, in which the editing is done. When the user has changed something and want to update the files you do a commit which will synchronize the repository with the local copy. This provides great freedom in projects where many files and authors are involved. The decided program to use was Concurrent version systems (CVS) and PCL-CVS on XEmacs (see Appendix E).
3.1.2 VHDL compiler

The compiler used was *Mentor Graphics ModelSim SE (VCOM)* (see Appendix D).

3.1.3 VHDL simulator

For simulating the design, *Mentor Graphics ModelSim SE (VSIM)* has been used, which is well suited for its purpose. The tool is easy to use and nothing was encountered to strengthen anything negative about it.

3.1.4 Synthesis

The next step in the design was to synthesize the VHDL code. In this step two different tools were tested. The first of the two was *Mentor Graphics Leonardo*. Many major weaknesses was found (using this tool):

- It can only handle technology processes down to 0.35 µm.
- It is rarely used in the industry.
- It was showing strange and inconsistent results.

Thus it was decided to stop using it, instead CPKS was used. This tool is supposed to be a very powerful tool and broadly used in the industry. This tool has the power to both synthesize, floorplan and auto route the design. The tool was on the other hand rather hard to learn because of all its complex features and functions. Quite an effort had to be done in order to make progress with the tool, see Appendix A for more information. Afterwards we think the right choice was made by using this tool, both in the manner to obtain a better and more accurate result, and to achieve useful experience for future work in the industry.

3.1.5 Power simulation

The last step in the design flow covered in this master thesis is to simulate the design with respect to power. The tool used in this step is Synopsys NanoSim. NanoSim is, compared to CPKS, a script based tool and also rather time consuming to learn. In dead it is a good tool with a lot of useful features. For more information about NanoSim, see Appendix B.
3.2 Work flow
To get a fast understanding in how the work flow has been performed during this master thesis, see the flow chart in Fig. 3.1 where the flow is summarized. By following the flow from the beginning to end, it can be seen which design steps that are included and it is clarified which tool that has been used in each process. Every major design step will be more explained throughout the report.
This chapter will discuss the background and the reason why a control unit was designed. It will also cover how the block was implemented and tested to meet the required function. The problems encountered through the design and the synthesis results will be discussed in Chapter 5.

4.1 Background

The initial model of the FFT processor was more or less written in behavioral VHDL code [2]. The model was running at one frequency and StagePE was the heart of the design working as a control unit. Based upon a counter which was increased every clock cycle the FFT started by reading the input data, process the data, and finally output the result. The RAM used was implemented by a two dimensional array of ideal D flip-flops and no care were taken to neither setup times nor hold times. This would likely cause problems if a real RAM was to be integrated in the model, which was our intention to make the project more realistic. In order for this to work, and to adapt the FFT processor to our modifications it was decided that a new control unit had to be designed.

The main idea with the control unit is not only that it should substitute StagePE (which is to be removed from the design) but it should also become the new heart of the modified FFT processor. Beside the major function of controlling the global stages, it will also include control signals to the new (real) RAM model to meet the required setup and hold times.

Due to modifications of the FFT the control unit will also generate the two extra clocks required, and control the added serial communication between the RAM and the Butterfly. The goal of this control unit is not to move all the controlling logic from the other blocks, but only the major parts. Hence every block still contains controlling logic that communicates with the control unit via the added control signals.

4.1.1 Implementation

The new control unit was decided to be implemented as a Finite State Machine (FSM). This is an often used methodology in constructing control units. There are two different kinds of FSM: Moore and Mealy. The outputs of a Moore machine is only dependent of the state vector (current state) and is delayed one clock
cycle. The outputs of a *Mealy* machine are on the other hand both dependent on the current state and all the inputs\[6\]. The decision was made to use a *Mealy* machine. Furthermore to control the *global stage* five states were introduced:

- *idle*
- *ramInt*
- *inputFFT*
- *computeFFT*
- *outputFFT*

The first state, *idle*, is a reset state where the FFT processor does nothing. When the *enable* signal is true the FFT processor starts working. The *ramInt* stage initialize and synchronize the process with the RAM. The three consecutive states correspond to the initial FFT stages. The difference here is how they shift. Several control signals have been added, which are used to communicate between the blocks and the control unit. Hence a handshake between the blocks is obtained. The control unit signals a block when to start, and when that certain process is finished, the control unit verifies it by changing state. Not all control functions have been moved to the control unit, but only the shift of *global stage*. Just as in the initial FFT, all control functions are located in the different blocks, modified to communicate with the new control unit.

Another task performed by the control unit is to make sure that the hold times and setup times are met for the RAM. Every time the RAM is accessed, the requirements have to be fulfilled. If, for example, a write operation is to be done the data must be placed on the bus, and the write signal enabled prior to the rising edge of the clock (setup time). Furthermore it must also be stable for a certain time (hold time). This was (as mentioned) not considered in the initial FFT model. This operation is done by a control signal, trigged by a faster clock (*ClkFFT*) than the RAM clock (*ClkRAM*).

As mentioned earlier, the modified FFT require three clocks. In order to overcome the problem with multiple clocks in the synthesis, the FFT was fed with the fastest clock of the three and then generated the other two clocks in the control unit. This was implemented using shift registers.
The first objective with the master thesis was to synthesize the VHDL model of the initial FFT [2]. Since the status of this VHDL model was uncertain it was not known how much work that needed to be done to make this work. With the key at hand it can be stated that it took much longer than first expected. Without any hesitation this design step has dominated the work of this master thesis.

The first attempt of synthesizing the VHDL code was made in the synthesis tool Leonardo, which could directly be invoked from HDL Designer, the platform the initial FFT was designed on (see Chapter 3). After modifying the code and removing the RAM’s the model passed the synthesis. When it was decided to write the design in structural VHDL code instead of using HDL Designer (see Chapter 3), Leonardo started to behave awkward. Therefore it was decided to use the more powerful synthesis tool CPKS. This tool has then consistently been used through the synthesis process. Since CPKS is more complicated than Leonardo, it took quite some time to get familiar with its syntax. Because this has been a big part of the thesis the decision was to summarize the work flow in an appendix. This appendix is to be considered as a simplified users guide, including the necessary and most important commands to carry out a complete synthesis flow (see Appendix A).

Further on in this chapter, the focus will be on the work of redesigning and improving the individual blocks of the FFT. In order to reach the synthesis goal, necessary modifications will be explained and motivated.

5.1 Goal

The main goal of the synthesis process has, as mentioned earlier, been to modify the initial FFT to a synthesizable model. A second goal has been to work towards and fulfilling certain requirements and constraints (see Appendix A). The requirements have in particular been to design the FFT to reach the demands of the different clock frequencies. This means that after a complete synthesis flow of the circuit, there can be no reports indicating negative slack on any physical path. Negative slack means that the signal is arriving late to an endpoint. In other words the propagation delay through the circuitry (from start point to endpoint) is too large. This problem can either be solved by improving the design or to decrease the clock frequency, giving the signal more time to propagate through the circuit.
In the modified FFT processor there are three clock frequencies feeding the different blocks instead of the single one in the initial FFT. The addition of these clocks has been necessary in order to implement the model according to the modifications we have made. The clocks and their frequencies are listed below.

- **ClkFast → 256 MHz**
  This is the main input clock to the FFT. Its main purpose is to feed the bit serial communication between the butterfly and RAM.

- **ClkFFT → 128 MHz**
  This clock is generated in the Controller by dividing ClkFast by two. It feeds most of the processes in the FFT and is set so that setup and hold times are met due to the RAM.

- **ClkRAM → 32 MHz**
  This clock is generated in the Controller by dividing ClkFast by eight. It feeds the RAM and the I/O processes reading from and writing to the RAM, respectively.

To reach a successful synthesis of the FFT, based on the requirements stated above, has more or less been our main goal. If these requirements, for any reason, not can be fulfilled within a reasonable time frame, the synthesis flow will be run at the lowest possible frequency, to see if the design, what so ever, is realizable. In this case the frequencies will be decreased to 8 MHz, 4 MHz and 1 MHz, respectively. Hence, the relationship will be equivalent to the original frequencies. The outcome of potential problems and how they were solved will together with the achieved results be further discussed throughout this chapter and highlighted in the concluding discussion. It should be mentioned that the optimization in the synthesis have consistently been set to optimize the design due to worst slack and not due to area (see Appendix A).

### 5.2 Controller

Since the control unit is designed from scratch, it can not be compared to a reference design as the other blocks in the FFT. Instead this block was designed and redesigned until the results were satisfying the requirements.

The first version of the control unit mainly comprised the functions of generating and receiving the control signals, running the FFT through its global stages. It also handles the read/write signals to the RAM with its setup and hold times. This early version of the control unit was aimed to handle the frequency 128 MHz. At first it was hard to reach this goal but after rewriting the state machine some times and letting CPKS optimize the design the time constraint were
finally met with a few nano seconds to spare. So far the results were good and the control unit was put aside, waiting for the next part to be integrated (the process controlling the serial communication as mentioned in Chapter 4).

This process is also built upon a state machine that consists of nine states. The main function is to control the serial communication between the Butterfly and RAM. In order to get the serial communication to work, a faster clock was required. Therefore the control unit was modified towards the next goal, and the decision was made to generate the two extra clocks (see Section 5.1), in a separate process in the control unit. The faster clock was set to twice the frequency of the current clock.

The control unit is now synthesized consisting three processes and $ClkFast$ feeding the block. The results can be viewed in Table 5.1 below. It can be seen that the timing constraint is not reached, as the worst slack is $-5.7 \text{ns}$. When this result was achieved, quite some time had been spent on this control unit, trying to reach the synthesis goal. It was realized that the initial timing constraints could not be met. Thus, instead of further time demanding improvement of the design it was now decided to run the synthesis flow with the lower frequency (easier time constraint). This result is also shown in Table 5.1. As seen, the worst slack is now $111 \text{ ns}$. The slack is computed by subtracting the arrival time from the phase shift ($T = \frac{1}{f} = 125 \text{ns}$). It is clear that the area increase with increasing frequency. This can be explained due to reaching the tighter timing demands, the synthesis tool automatically inserts buffers (repeaters) along the physical paths.

<table>
<thead>
<tr>
<th>Controller</th>
<th>$f = 256 \text{MHz}$</th>
<th>$f = 8 \text{MHz}$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst slack ($\text{ns}$)</td>
<td>-5.7/104</td>
<td>111/71</td>
</tr>
<tr>
<td>Max freq ($\text{MHz}$)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Area ($\mu \text{m}^2$)</td>
<td>4211</td>
<td>3334</td>
</tr>
</tbody>
</table>

**Table 5.1: Synthesis result of the Controller**

To bring an end to the discussion about the control unit, these achieved results are the final results. Because of the time limitation of the project there has been no more time improving the design towards the initial time constraint. Somewhere a boundary has to be set to be able to move on in the design flow. More about this will be discussed in the conclusions.

### 5.2.1 Problems

The need for multiple clocks became one of the concerns early in the design. How should these clocks be implemented? The first idea was to feed the Controller with one intermediate frequency and then generate the other clocks from this
unit. And as mentioned this is almost how it was done. But this solution is only satisfying the synthesis. After some research it was discovered that this is not how it actually should be done. The clock is supposed to be delivered externally to our chip by a Phase Locked Loop (PLL) or several PLL’s. In order to simulate and synthesize our design we let the synthesis tool create ideal clocks and build clock trees with specified constraints (see Appendix A.3.9.1).

Another problem encountered when synthesizing the control unit were if the state encoding should be defined as one-hot in the VHDL code, or to let the “built in” function in CPKS define the state encoding, as one-hot. The first choice was to let CPKS take care of that part, for two reasons. First, because we thought the synthesis result would become more optimized since less combinational logic is required, leading to a smaller area. And second, if for some reason an extra state would be needed in the design, the VHDL code would not have to be extensively rewritten. However, after several hours of “trial-and-error”, the idea of letting CPKS do this part was canceled. It turned out that CPKS was extremely fastidious, in how the VHDL code was written. Hence, it was hard to get CPKS to extract a fully functional FSM. Consequently this lead to bad synthesis results. To overcome and solve the problem, the state vector was instead encoded manually.

5.3 Butterfly

5.3.1 Initial implementation

Here is the initial Butterfly model as described in [2] without any modifications. The flow of operations for the Butterfly’s can be seen in Table 5.2.

<table>
<thead>
<tr>
<th>Cycle</th>
<th>Butterfly0</th>
<th>Butterfly1</th>
</tr>
</thead>
<tbody>
<tr>
<td>Cycle1</td>
<td>read from RAM</td>
<td>calculate</td>
</tr>
<tr>
<td>Cycle2</td>
<td>write previous result</td>
<td>idle</td>
</tr>
<tr>
<td>Cycle3</td>
<td>calculate</td>
<td>read from RAM</td>
</tr>
<tr>
<td>Cycle4</td>
<td>idle</td>
<td>write previous result</td>
</tr>
</tbody>
</table>

Table 5.2: Initial Butterfly operation

In the compute step the actual computation is performed. To get a better understanding of the modifications, a flow of the computation cycle is shown in Figs. 5.1 and 5.2 (note that the division operator shown is only for clarification purpose, it is implemented using a shift operation). All arithmetic operations are carried out as signed operations, an explanation of the input names is listed below:
The theoretical model in [3] describes that the FFT should have a serial interface between the Butterfly’s and the CacheCtrl. It was then realized that this interface
did not exist. In the initial design, there is a parallel interface between the memories and CacheCtrl. Each CacheCtrl have a separate bus to each Butterfly (4 buses) and each Butterfly has a separate bus to each CacheCtrl (4 buses), which all resulted in $8 \times 48$ bits. The connection between the CacheCtrl and RAM's where parallel.

### 5.3.3 Synthesis of initial model

The first action was to synthesize the initial model without any modifications. As that design was not well done, it was uncertain if it would pass the synthesis step at all. At first the timing constraints were set way too optimistic, which resulted in long run times for CPKS. Thus it was necessary to reduce the goal frequency. The result with the goal frequency 1 MHz can be seen in Table 5.3.

As seen the multiplications are the area consuming unit. Because of the low timing constraints set, the selected architecture naturally becomes ripple-carry adders. This is the architecture which occupies the smallest area, but is also the slowest. For further comparisons higher constraint goals has also been set, this to be able to compare with the improved versions. As seen in Table 5.4 the area increases with increasing constraint goals. Hence also the running time for CPKS increases.

<table>
<thead>
<tr>
<th>Module</th>
<th>Area ($\mu m^2$)</th>
<th>Size (bits, signed)</th>
<th>Architecture</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>32782</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>Nets</td>
<td>1550</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ADD1</td>
<td>349</td>
<td>24x24</td>
<td>ripple</td>
</tr>
<tr>
<td>ADD2</td>
<td>653</td>
<td>45x45</td>
<td>ripple</td>
</tr>
<tr>
<td>ADD3</td>
<td>349</td>
<td>24x24</td>
<td>ripple</td>
</tr>
<tr>
<td>MULT1</td>
<td>6940</td>
<td>24x21</td>
<td>ripple/booth</td>
</tr>
<tr>
<td>MULT2</td>
<td>6957</td>
<td>24x21</td>
<td>ripple/booth</td>
</tr>
<tr>
<td>MULT3</td>
<td>6967</td>
<td>24x21</td>
<td>ripple/booth</td>
</tr>
<tr>
<td>MULT4</td>
<td>6973</td>
<td>24x21</td>
<td>ripple/booth</td>
</tr>
<tr>
<td>SUB1</td>
<td>398</td>
<td>24x24</td>
<td>ripple</td>
</tr>
<tr>
<td>SUB2</td>
<td>738</td>
<td>45x45</td>
<td>ripple</td>
</tr>
<tr>
<td>SUB3</td>
<td>398</td>
<td>24x24</td>
<td>ripple</td>
</tr>
</tbody>
</table>

**Table 5.3:** Synthesis area result for initial Butterfly with goal frequency 1 MHz

### 5.3.4 Summary of initial model

As seen, not much effort has been put into designing the Butterfly. Moreover the VHDL model is not written as it should have been done. After discussion
5.3.5 Synthesis result for common arithmetic

Before trying to rewrite the design, it is recommended to synthesize some simple arithmetic tests, this to know where the constraint limits are for a given technology and synthesis tool. In Section A.7 a summary of arithmetic operations for a 0.35 µm process in CPKS can be seen. The different bit sizes chosen is actual sizes used in the Butterfly’s compute step. When studying the result it is clear that CPKS does a good job to balance area and timing constraints effectively. When the timing constraints increase, a faster architecture is selected. As described in Appendix A.4.3 it is possible to choose which implementation to use. However, this is not recommended. As it is better to let the tool do the decision on which architecture to use.

5.3.6 Improvement I - Rearrange operations

The first improvement would be to perform the most critical operations in separate clock cycles. The most critical operation is the multiplication, therefore the multiplication should be performed in a separate clock cycle (in comparison with the initial model where all arithmetic operations were performed in a single clock cycle). This will of course require more clock cycles to perform the computations. However, this is not a problem as the design is implemented so that the Butterfly’s have five clock cycles to perform the operations. Thus, as long as the required operations are performed within five memory clock cycles (at 32 MHz) the computations can be performed at any frequency.

When looking at Figs. 5.1 and 5.2 it illustrates the possibility to rearrange the Butterfly’s computations. The result can be seen in Table 5.5.

<table>
<thead>
<tr>
<th>Constraints goal</th>
<th>Area</th>
<th>Maximum frequency</th>
</tr>
</thead>
<tbody>
<tr>
<td>(MHz)</td>
<td>(µm²)</td>
<td>(MHz)</td>
</tr>
<tr>
<td>1</td>
<td>32782</td>
<td>24</td>
</tr>
<tr>
<td>32</td>
<td>36195</td>
<td>31</td>
</tr>
<tr>
<td>64</td>
<td>57140</td>
<td>56</td>
</tr>
<tr>
<td>96</td>
<td>61932</td>
<td>57</td>
</tr>
</tbody>
</table>

Table 5.4: Synthesis timing result for initial Butterfly

with the examiner, the conclusion was that it may not be possible to reach the specified goals [3].
When the rearrangement is done, a new run in the synthesis tool gives the results shown in Table 5.6. As seen the area is larger than in the initial model (compare with Table 5.4) this is due to ineffective resource sharing (see Appendix A.5.1).

<table>
<thead>
<tr>
<th>Constraint goals (MHz)</th>
<th>Area (µm²)</th>
<th>Maximum frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>39939</td>
<td>26</td>
</tr>
<tr>
<td>32</td>
<td>42579</td>
<td>31</td>
</tr>
<tr>
<td>64</td>
<td>49256</td>
<td>58</td>
</tr>
<tr>
<td>96</td>
<td>69644</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 5.6: *Synthesis timing result for rearranged Butterfly*

### 5.3.7 Improvement II - Resource Sharing

To decrease the area it is possible to use resource sharing (Appendix A.5.1). As seen in Table 5.3 the two adders and subtractors are the same length and could therefore be shared. But most important would be to share the multipliers. To make it possible to share resources, the synthesis tool must be aware that the different operations are not performed at the same time. To do that one may design a state machine (FSM) to ensure the correct flow. Although it sounds simple there where problems in getting CPKS to understand that it was a FSM. See [7, 221] in how to write proper FSM syntax. After a bit of struggling CPKS understood that it could share the resources. The result are shown in Table 5.7. As seen, the area is reduced with approximately 50%. The module type “PARTITION” is shown when (to reduce area) CPKS divides common arithmetic into partitions, in this case the subtraction and addition operations have been divided. An overview of the different cycles can bee seen in Table 5.8.
<table>
<thead>
<tr>
<th>Module</th>
<th>Area (µm²)</th>
<th>Size (bits, signed)</th>
<th>Architecture (type)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complete</td>
<td>19478</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>ADD</td>
<td>349</td>
<td>24x24</td>
<td>ripple</td>
</tr>
<tr>
<td>MULT</td>
<td>6956</td>
<td>24x21</td>
<td>ripple/booth</td>
</tr>
<tr>
<td>SUB</td>
<td>398</td>
<td>24x24</td>
<td>ripple</td>
</tr>
<tr>
<td>PARTITION0</td>
<td>1950</td>
<td>-</td>
<td></td>
</tr>
<tr>
<td>PARTITION1</td>
<td>663</td>
<td>-</td>
<td></td>
</tr>
</tbody>
</table>

*Table 5.7: Area result for Butterfly with resource sharing (1 MHz)*

<table>
<thead>
<tr>
<th>Cycle 1</th>
<th>Cycle 2</th>
<th>Cycle 3</th>
<th>Cycle 4</th>
<th>Cycle 5</th>
<th>Cycle 6</th>
</tr>
</thead>
<tbody>
<tr>
<td>ResultB</td>
<td>MULT</td>
<td>MULT</td>
<td>MULT</td>
<td>MULT</td>
<td>ADD[2]</td>
</tr>
</tbody>
</table>

*Table 5.8: Improvement II - resource sharing cycles*

<table>
<thead>
<tr>
<th>Constraints goals (MHz)</th>
<th>Area (µm²)</th>
<th>Maximum frequency (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>19478</td>
<td>25</td>
</tr>
<tr>
<td>32</td>
<td>20973</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>25339</td>
<td>60</td>
</tr>
<tr>
<td>96</td>
<td>30785</td>
<td>70</td>
</tr>
</tbody>
</table>

*Table 5.9: Synthesis result for Butterfly with resource sharing*

### 5.3.8 Improvement III - Serial communication

As mentioned, the initial FFT has a parallel communication, which may cause synthesis problems when routing. Therefore a serial interface was evaluated and implemented. As mentioned in Chapter 4 a fast serial clock is then needed. The serial interface as shown in Table 5.10 is complicated and need some clarifications. The *state* may consist of one or more clock cycles. The character after the state number describes special actions taken:

---

5.3. Butterfly 23
A | The previously computed result from the calculation process is loaded.
B | Data from one Butterfly is transferred to both CacheCtrl’s (multicycle process).
C | New data from RAM is loaded to CacheCtrl.
D | Performs B and transfer the new data obtained in C to the Butterfly.
E | Now the result is transferred to CacheCtrl’s, so only the new data from RAM is transferred.
F | The shift is done and the previous result is written to RAM.

<table>
<thead>
<tr>
<th>State</th>
<th>BF0</th>
<th>BF1</th>
<th>CCs</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>-</td>
<td>compute</td>
<td>-</td>
</tr>
<tr>
<td>2-A</td>
<td>load prev result</td>
<td>compute</td>
<td>-</td>
</tr>
<tr>
<td>3-B</td>
<td>shift to CC’s</td>
<td>compute</td>
<td>shift from BF0</td>
</tr>
<tr>
<td>4-C</td>
<td>shift to CC’s</td>
<td>compute</td>
<td>read RAM</td>
</tr>
<tr>
<td>5-D</td>
<td>shift both</td>
<td>compute</td>
<td>shift both</td>
</tr>
<tr>
<td>6-E</td>
<td>shift from CC’s</td>
<td>compute</td>
<td>shift to BF0</td>
</tr>
<tr>
<td>7-F</td>
<td>shift done</td>
<td>compute done</td>
<td>write RAM</td>
</tr>
<tr>
<td>8</td>
<td>compute</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>9-A</td>
<td>compute</td>
<td>load prev result</td>
<td>-</td>
</tr>
<tr>
<td>10-B</td>
<td>compute</td>
<td>shift to CC’s</td>
<td>shift from BF1</td>
</tr>
<tr>
<td>11-C</td>
<td>compute</td>
<td>shift to CC’s</td>
<td>read RAM</td>
</tr>
<tr>
<td>12-D</td>
<td>compute</td>
<td>shift both</td>
<td>shift both</td>
</tr>
<tr>
<td>13-E</td>
<td>compute</td>
<td>shift from CC’s</td>
<td>shift to BF1</td>
</tr>
<tr>
<td>14-F</td>
<td>compute done</td>
<td>shift done</td>
<td>write RAM</td>
</tr>
</tbody>
</table>

Table 5.10: Serial interface

5.3.9 Result

As shown in Table 5.11 both the timing and area can be improved using operation rearrangement and resource sharing. However, this comes with a cost in terms of increasing design time. This can probably be decreased if knowledge about the design process exists. When it comes to the serial implementation there is no concrete answer, if it is better to have serial implementation or not. This because of the possible increasing power consumption for the faster clock. This has not been investigated thoroughly as the time limit has been reached.
<table>
<thead>
<tr>
<th>Constraints</th>
<th>Initial Area (µm²)</th>
<th>Initial Freq. (MHz)</th>
<th>Improvement I Area (µm²)</th>
<th>Improvement I Freq. (MHz)</th>
<th>Improvement II Area (µm²)</th>
<th>Improvement II Freq. (MHz)</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>32782</td>
<td>24</td>
<td>39939</td>
<td>26</td>
<td>19478</td>
<td>25</td>
</tr>
<tr>
<td>32</td>
<td>36195</td>
<td>31</td>
<td>42579</td>
<td>31</td>
<td>20973</td>
<td>32</td>
</tr>
<tr>
<td>64</td>
<td>57140</td>
<td>56</td>
<td>49256</td>
<td>58</td>
<td>25339</td>
<td>60</td>
</tr>
<tr>
<td>96</td>
<td>61932</td>
<td>57</td>
<td>69644</td>
<td>70</td>
<td>30785</td>
<td>70</td>
</tr>
</tbody>
</table>

Table 5.11: Final synthesis result for Butterfly

5.4 CacheCtrl

The CacheCtrl is responsible for storing the data between the read and write instructions from the RAM. In the initial design it was a bidirectional memory. This was later changed to a separate read and write bus. As the CacheCtrl simply stores the data received from either RAM or Butterfly not much has been done to it. The thing that was added is the serial interface which is described in Section 5.3.8.

5.5 BaseIndexGen and WPGen

In order to reach good synthesis results multiplications and division operators should be avoided in the VHDL code. Multiplications can be performed but divisions and modulus operations are not supported. One way to solve this problem is to use shift operations. A multiplication corresponds to shifting a vector to the left while division corresponds shifting the vector to the right. In the process of BaseIndexGen Eq. (5.1) and WPGen Eq. (5.2) there are two expressions including multiplication, division and modulus [3].

\[
k_1 = 4 \cdot N_{s_n} \cdot \left( \frac{m_n}{N_{s_n}} \right) \mod N_{s_n}, \text{ where } k_1 \text{ is a RAM index} \tag{5.1}
\]

\[
p = \left( k_1 \cdot \frac{512}{N_{s_n}} \right) \mod 512, \text{ where } p \text{ is a ROM address} \tag{5.2}
\]

If these expressions are directly written in the VHDL code there will be no problem simulating the code but as mentioned the synthesis will cause problems. In our case \( N_{s_n} \) is a 10 bit vector, “hot encoded”, and can only take the value \( 2^n \) where \( n \) ranges from 0 to 9. This is something which can be used since it is known, depending on \( N_{s_n} \), how many steps the vector is to be shifted left or right (multiplied or divided). The modulus operator also causes problems in the synthesis. Since the value of the argument (\( N_{s_n} \) and 512) always is known, it can
easily be solved. The result of the modulus operator will take the value of the
\( n - 1 \) downto 0 bits of the left hand side vector \((N_{S_n})\).

\( WPGen \) is a small process which computes the ROM address stated above
and receive the TF in a ROM table \((512 \times 42 \times 2)\) defined in a VHDL package.
To synthesize this process is a waste of time since there is no chance in meeting
the timing constraints required. The synthesis tool creates a large combinational
logic network to implement the ROM. This ROM table should be substituted
with a real ROM, in the way the D-flip-flops which instantiated the RAM was
exchanged with a real RAM.

5.5.1 Results

In Table 5.12 and Table 5.13 the results are displayed to get an overview of
how the modifications affected the blocks. \( BaseIndexGen \) show an improvement
in speed. This improvement is gained due to implementing the multiplication
operation in the VHDL code using shift operators instead of the straight forward
multiplication operator. The multiplication in question is parallel. When the
multiplication operator is used, CPKS implements the operation as a \texttt{fcla/non-
booth} multiplier (fast carry lookahead adder). How this algorithm is implemented
is described in [8]. On the contrary if the shift operator is used, only an \texttt{n-bit}
shift register is required, which was discovered to be faster (see Table 5.12).
So if it is possible to substitute multiplication operators with shift operators, it
should be done. On the other hand the area was slightly increased. This can be
explained due to reaching the timing demands, the synthesis tool automatically
inserts buffers along the physical paths. Even though the modifications done,
the desired timing demand \((128 \, MHz)\) could not be reached, although close.
This is the turning point in this work (as mentioned earlier) and the frequency is
decreased to \( 4 \, MHz \). The result is now satisfying with plenty of time at hand.
The reduced area is the opposite of that mentioned above. Because of the timing
demand not so strict, there is no need of inserting as many buffers.

<table>
<thead>
<tr>
<th>BaseIndexGen</th>
<th>Before mod</th>
<th>After mod ( f = 128 , MHz )</th>
<th>After mod ( f = 128 , MHz )</th>
<th>After mod ( f = 4 , MHz )</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst slack (\textit{ns}) / Max freq (\textit{MHz})</td>
<td>-8.61/60</td>
<td>-1.44/108</td>
<td>205/22</td>
<td></td>
</tr>
<tr>
<td>Area (( \mu m^2 ))</td>
<td>10445</td>
<td>12190</td>
<td>8226</td>
<td></td>
</tr>
</tbody>
</table>

Table 5.12: \textit{Synthesis result of BaseIndexGen}

As mentioned earlier \( WPGen \) is mainly comprised of a ROM table and there-
fore not synthesizable. Or more accurate: not meaningful to synthesize since it is
not recommended to implement such large ROM tables in VHDL code [6]. How-
ever, along the way a curiosity arose how bad the results really would become if the ROM table was synthesized. The result can be considered as disastrous. It took days to finish the synthesis process and the physical area required to implement the combinational logic was huge. If this ROM table would have been used in the FFT design, it would have comprehended more than 50% of the total area. Moreover, because of the large combinational logic network the worst slack, as anticipated, also turned out bad. As discovered earlier, there was no chance to reach the timing demands first stated. The ROM table does on the other hand reach the timing demands, with a large area, running at the frequency $1 \text{ MHz}$.

By concluding this it was decided that the ROM table should be removed from the FFT design and used for simulation purposes only.

<table>
<thead>
<tr>
<th>WPGen</th>
<th>Before mod</th>
<th>After mod</th>
<th>After mod</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>$f = 32 \text{ MHz}$</td>
<td>$f = 32 \text{ MHz}$</td>
<td>$f = 1 \text{ MHz}$</td>
</tr>
<tr>
<td>Worst slack (ns)</td>
<td>-</td>
<td>-41/13</td>
<td>914/11</td>
</tr>
<tr>
<td>Max freq (MHz)</td>
<td>-</td>
<td>114082</td>
<td>91046</td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>-</td>
<td>114082</td>
<td>91046</td>
</tr>
</tbody>
</table>

Table 5.13: Synthesis result of WPGen

Note: Late in the report writing, a new function was discovered in CPKS which improved the result of WPGen quite much. This will not be brought up here (to confuse) but instead in Appendix A.4.4 where the function is explained further and the result is presented.

5.6 AddressGen, OutputCtrl and DigitReversal

In order to synthesize these blocks, nothing worth mentioning needed to be rewritten in the VHDL code. However, they have been modified in order to communicate with the control unit and the control signals added to the design. Nothing is critical within these blocks, hence the result will only be mentioned shortly. They can be viewed in Tables 5.14, 5.15 and 5.16 below. As seen, all these blocks pass the timing requirements as specified in our goal, except for AddressGen (Table 5.14) which has a slightly negative slack. No effort in fixing this slack has been done, since the goal had to be compromised, but it is surely fixable. The area difference is likely to depend on the same reason as stated earlier, buffer insertion, caused by the tighter timing constraint.
<table>
<thead>
<tr>
<th></th>
<th>$f = 128 \text{ MHz}$</th>
<th>$f = 4 \text{ MHz}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>AddressGen</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst slack (ns)</td>
<td>-0.0035 / ~ 128</td>
<td>216 / 29</td>
</tr>
<tr>
<td>Max freq (MHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>12629</td>
<td>8128</td>
</tr>
</tbody>
</table>

**Table 5.14: Synthesis result of AddressGen**

<table>
<thead>
<tr>
<th></th>
<th>$f = 32 \text{ MHz}$</th>
<th>$f = 1 \text{ MHz}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>OutputCtrl</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst slack (ns)</td>
<td>18.9 / 80</td>
<td>495.8 / 2</td>
</tr>
<tr>
<td>Max freq (MHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>1298</td>
<td>1297</td>
</tr>
</tbody>
</table>

**Table 5.15: Synthesis result of OutputCtrl**

<table>
<thead>
<tr>
<th></th>
<th>$f = 128 \text{ MHz}$</th>
<th>$f = 4 \text{ MHz}$</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>DigitReversal</strong></td>
<td></td>
<td></td>
</tr>
<tr>
<td>Worst slack (ns)</td>
<td>5.36 / 407</td>
<td>247.5 / 400</td>
</tr>
<tr>
<td>Max freq (MHz)</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>282</td>
<td>282</td>
</tr>
</tbody>
</table>

**Table 5.16: Synthesis result of DigitReversal**

`OutputCtrl` and `DigitReversal` are small blocks (see Chapter 2), which can be concluded by viewing the results in Tables 5.15 and 5.16. They both consume small areas (compared to the other blocks) and there is no problem in meeting the initial timing constraints. The attentive reader might notice that there is no difference in the area between the two frequency goals. The explanation to this is that the synthesized designs are exactly the same. Hence, the simplicity of the blocks there is no need of further optimization.
After reaching the requirements in the synthesis process of the design, the next step in the design flow is to simulate power consumption. Designing a low power FFT processor was one of the aims in this project. The idea with the thesis from the beginning was to find and identify critical parts of the FFT processor. Then to compare the performance of these critical building blocks with alternative solutions, and either rewrite the VHDL code or make a full custom design of that block. In this aspect both speed performance of the block, in order to reach the timing demands for the required clock frequency, as well as the power consumption of the block was to be considered. Since the synthesis process occupied most of the time, the main focus has not been to optimize or improve individual blocks considering power consumption. However, the complete design and some of the sub blocks have been investigated. Comparisons have been made and some conclusions have been reached. The software tool used to simulate power is Synopsys NanoSim (see Appendix B).

6.1 Testing the FFT

In order to estimate the power consumption for the complete FFT processor, a simulation was run over a fixed time interval where the processor works the most (processes the most data). Three simulations have been made over the same time interval with three different versions of the FFT processor where different parts were excluded:

- FFT with RAM as a black box and ROM implemented in the VHDL code.
- FFT with fake RAM and the ROM implemented in the VHDL code.
- FFT with fake RAM and fake ROM.

To clarify the word fake, it is meant that the block in question is not described with a functional VHDL model. Instead the block is stimulated with accurate data from a vector file. The data has been assembled from previous simulations in VSIM.

\footnote{Notice that a complete simulation run is not done, since it is too time consuming. For testing purpose only approximate numbers are needed.}
6.1.1 Test case 1 (Black box RAM and ROM table)

In the first test run the FFT was simulated with the RAM memory as a black box. This was the initial approach since the missing of a synthesizable behavioral model or a C language description of the RAM. The results were suspected to be deceiving and that was also how it turned out. Since the RAM was connected to the model and did nothing, the output vector feeding the Butterfly’s with data was always set to logic zeros. Consequently the Butterfly’s will always make its computations (multiplications and additions) with zero vectors. In other words the power consumption generated in the Butterfly’s will be inaccurate since there will be no toggles within the circuit. The result of this simulation is a wasted current of 95% (see Appendix B) and a total power consumption of 43 mW. The sub blocks overall have large wasted currents except for the CacheCtrl. This result can not be considered as satisfying and the outcome can be explained because of the reason mentioned above.

6.1.2 Test case 2 (Fake RAM and ROM table)

When the problem with the RAM as a black box was discovered, a reassessment had to be made in order to obtain a power estimation for the FFT. The second approach was to fake the output data from the RAM and resimulate. By doing this, the idea was to fool NanoSim by removing the RAM from the design. Now the data that was supposed to be read from the RAM was instead read from a stimuli file as mentioned earlier. The outcome of this simulation improved the total wasted current from 95% to 60% of the FFT processor, while the total power consumption dropped to 6.8 mW. This was a satisfying result and overall every sub block was improved by this action.

6.1.3 Test case 3 (Fake RAM and fake ROM table)

After the positive results of test case 2 it was speculated, if the results would improve even more by removing the ROM table from the design (see Section 5.5). The only interest is to see how the data from it effects the FFT. So the decision was made to fake this ROM table equally as the RAM was faked. But still WPGen is not excluded from the design, only the ROM table.

The result by this action improved as expected. The total wasted current was reduced to 16% and the total power consumption measured 2 mW. However, it seemed as if something had happened to BaseIndexGen and DigitReversal. The wasted current in these two blocks now reached 100%, even though the simulation ran over the same time interval as in the two previous test cases. This problem did on the other hand not arise when the complete FFT processor was simulated (see Section 6.2). The cause of this problem is not known, but something can either be wrong with the generated Verilog file from CPKS or with the vector file.
containing the stimulus (or is it a phenomenon of something being overlooked). Because the time frame was beginning to exceed its limit, there has been no time, able to locate and fix this problem.

6.1.4 Test case summary

In Table 6.1 the results are summarized. The conclusion made by these simulations (test cases) is that the result is continuously improved. By looking a little closer into each test case, it is as expected the Butterfly that is the most power consuming block. Note that these test cases have been made to estimate the power consumption and for comparison purpose only. The result can not be considered as a final result.

<table>
<thead>
<tr>
<th>Results</th>
<th>Test case 1</th>
<th>Test case 2</th>
<th>Test case 3</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average supply current ($\mu$A)</td>
<td>12972</td>
<td>2047</td>
<td>602</td>
</tr>
<tr>
<td>Average wasted current ($\mu$A)</td>
<td>12357</td>
<td>1229</td>
<td>97</td>
</tr>
<tr>
<td>Wasted current percentage (%)</td>
<td>95</td>
<td>60</td>
<td>16</td>
</tr>
<tr>
<td>Average power (mW)</td>
<td>42.8</td>
<td>6.8</td>
<td>2.0</td>
</tr>
</tbody>
</table>

Table 6.1: Summary of the top module of the three test cases

6.2 Simulating the complete FFT

After establishing that test case 3 must be the most accurate, excluding the memories (RAM and ROM) from the design, it was decided to simulate the complete FFT flow with this test case. In the previous test cases only parts of the compute stage and output stage were simulated. In this simulation the complete input stage, compute stage and output stage will be included. Simulating such a large design is time consuming and takes 2-3 days to complete. This is the reason why testing different versions of the FFT, prior to this simulation to determine which design to focus on.

Now let us discuss the outcome of this simulation. In Table 6.2 the result is listed. As can be seen the wasted current is 50%. This can probably be improved by analyzing the design and redesign the critical parts. The average power is 8.5 mW.
### Table 6.2: Summary of the complete FFT processor

<table>
<thead>
<tr>
<th>Results</th>
<th>Complete FFT</th>
</tr>
</thead>
<tbody>
<tr>
<td>Average supply current ($\mu A$)</td>
<td>2567</td>
</tr>
<tr>
<td>Average wasted current ($\mu A$)</td>
<td>1291</td>
</tr>
<tr>
<td>Wasted current percentage (%)</td>
<td>50</td>
</tr>
<tr>
<td>Average power (mW)</td>
<td>8.5</td>
</tr>
</tbody>
</table>

#### 6.3 Conclusion

As told, *NanoSim* is a good simulation tool and as can be seen through the simulations, the results are improved by changing the design and comparing. But personally we think it is hard to determine if the results achieved can be considered as accurate or reasonable in the real world. We feel this way since neither of us have had any previous experience within this area. A good way of verifying the results would have been to have a reference model, considered to be developed for low power consumption. This would at least have given us some guidelines in the process. Of course there are plenty of FFT models designed in different literature to compare with, but they are still all implemented differently. The time frame we have been working in has not given us the time to find an equivalent FFT design to compare the results with. But still improvements can be seen in the design flow and that is what counts. The rest we leave to *NanoSim* and hopefully the results are reliable. In the discussion above about the different test cases, only parts of the results are mentioned. The complete result reports can be found in Appendix B, where they more thoroughly can be compared.
7.1 Synthesis

In general divisions are not synthesizable. The exception is if the division is made with a constant value and the expression is not to complicated (nested). However, if the value is constant it should rather bee implemented with shifting and additions. This applies also to multiplications with constant values. As shifting can be implemented fast the timing gain is large.

Multiplications are implemented relatively good. CPKS first uses the slowest multiplication architecture (with the smallest area) and if the constraints not are met, a faster architecture is selected. There is hardly any need of telling CPKS which architecture to use, as it does a good job in balancing between area and timing. This also applies to addition and subtraction. However, both the area and run time increase with higher constraint goals and wordlength.

CPKS also does a good job in finding resources which can be shared. This reduce the area since only one hardware implementation is required. It was though quite hard to get CPKS to understand that certain operations not could occur at the same time. CPKS was fastidious in how the code was written. This led to some problems as the compiler/simulator and CPKS interpreted the FSM differently. This resulted in that some parts still had to be implemented by hand.

The modulus (mod) operator can, if the used value is on the form $2^n$, be implemented with the logical “and” operator. In the aspects of building clock trees due to large fanout CPKS does a good job, it balance the tree until the required goal is achieved. Also, if some control signals feeds many blocks the ability to build a physical tree is well developed. If an optional clock tree is designed (which in practice is impossible), no clock skew will occur.

Summary:

- The most critical part of a design (if included) is arithmetic operations.
- Implementation of arithmetic with small wordlengths is generally not a problem.
- Increased arithmetic wordlengths leads to long run times and large area. Studies about the maximum reachable constraint goals for different arithmetics should be performed prior designing.
Multiplications and divisions with constant coefficients should be implemented using shift- and add operations.

If doing multiplications with large wordlengths, try to divide it into smaller pieces.

If many arithmetic operations are performed, see if they can be performed after each other, to achieve good resource sharing.

Bad VHDL code will produce bad synthesis results and run times.

7.2 Modified FFT vs Initial FFT

To get an overview on what has been done in this master thesis we here compare the modified FFT with the initial FFT (see 2) and summarize the results. The most important difference is that the modified FFT can be synthesized. As the initial FFT model was written in behavioral VHDL code the major challenge during the thesis has been to rewrite the code towards certain synthesis goals. Beside reaching a certain goal also some block designs has been added and modified. Among these we can include the design of the new control unit and the adaption toward using a real RAM model (setup/hold times). Another thing added to the initial design is the partly serial interface between the Butterfly’s and RAM. This can on the other hand be questioned if it is required since the multiplications in the Butterfly still are parallel. This is though something that should be further investigated and developed.

When it comes to the synthesis results and the performance limitations of the modified FFT, it can clearly be stated that the initial goals were not reached. In the beginning we had high expectations and did not think the synthesis process would be such a vast process. Because of this the initial goals were maybe set a little optimistic. Even though this mistake the design is fully functional. Since the critical parts in the design is the control unit and the computations in the Butterfly this has to be considered as the bottleneck. The control unit can handle a frequency of nearly 104 MHz and the idea was that even the Butterfly should be fed with this frequency. But this is not possible since the computations not can be done faster than 70 MHz (according to the synthesis). Since the design is adapted to multiples of the feeding frequency this has to be taken into consideration and the result is as follows. The maximum frequency of ClkFFT is 70 MHz, leading ClkRAM to a maximum of 17.5 MHz and ClkFast to 140 MHz. Considering this the maximum in terms of FFT computations per second (FFT’s/s) will reach approximately 1400. Comparing this result with the initial FFT, which computed ∼ 2500 FFT’s/s, the performance has decreased.
7.3 Miscellaneous

After discussing the specific conclusions for this master thesis in previous chapters we would also like to mention a few general conclusions made along the way. For a starter there were some question marks when the control unit was designed. Should all the controlling logic be placed in the control unit or not? The conclusion was to keep as much control logic as possible in each block and activate them with control signals from the control unit. When the action in the block is performed the control unit is notified and can move on in the flow. This can be seen as a handshake instruction. By doing this each block will become more independent and easier to modify.

Further on we would also like to mention a few things about the software tools. To make it short the conclusion is that all the tools have its pros and cons. An overall con is that they all take lots of time to learn. This time is though necessary since it is hard to get a relevant result out of a tool if the setup is not right. It can be said that lots of time during this master thesis has been devoted to learn new tools, especially CPKS and NanoSim. It has though been concluded that if the tools are used in the correct way they are very useful (and some not). Unfortunately because of the complexity it is hard to say that you will ever get fully learned. It can’t be said that we are. Through the master thesis many new functions have been encountered that affects the result in different ways. Therefore it is hard to say that our results are the best results since there might be something overlooked. This can only be further developed by personal experience.

Another thing we think is recommendable at an early stage in a project is to find an agreement on how the documentation and file managing should be done. Especially if there are two or more people involved. In this thesis hundreds of files and formats have been processed including different results, input files, stimuli etc. It is always confusing if these are not all documented in the same way. This is at least something we have experienced. To summarize this we say ”don’t think that you will remember what you did the other day, because you will not”.

7.4 Further development

Since the 20 weeks flew by quite quickly it can’t be said that all the initial goals has been reached in this master thesis. There are several things that can be done and further developed such as:

- Serial multiplication and a complete serial interface between Butterfly and RAM.
- Integration of a ROM memory.
• Further power simulations.
• Convert from the $0.35 \mu m$ process to a $0.13 \mu m$ process.
• Add I/O pads to the floorplan and adjust the design towards production.

Since there was a misunderstanding in how the interface between the Butterfly and RAM should be implemented, it is now only partial serial. It was late discovered that this interface should be completely serial together with serial multiplications in the Butterfly. This could be implemented and simulated towards low power consumption.

Another thing that could be done is to integrate a real ROM in the design and adapt it to its setup/hold times, similar to what was done with the RAM. As mentioned in Appendix A.8 a way to design for low power was discovered parallel to the report writing. Because lack of time this method has not extensively been tested and could therefore be further investigated.

One of the main goals in the master thesis was to implement the FFT in a $0.13 \mu m$ process. Since the synthesis occupied most of the thesis time followed by lack of knowledge about the process at the department (ISY), this conversion has unfortunately not been done. We are sure that this conversion is an interesting task and an origin to many new challenging problems in the design flow.

When all problems are solved the last thing to do is to adjust the design to a producable layout including I/O pads and all that is necessary. We did not spend much time doing this so it is hard to say how much work that is required.
Although all information needed about *Ambit BuildGates®* and *Cadence PKS* is found in *Cadence®* manuals they are not easy to overview. Because of this master thesis has focused primarily on synthesis, there is a need for a tutorial. The layout of the appendix follows:

1. Introduction
2. Notation and TCL script
3. Synthesis flow
4. VHDL pragmas
5. High level optimization
6. Tutorial
7. Synthesis of common arithmetic
8. Low power synthesis (LPS)

### A.1 Introduction

*Cadence®* Physically Knowledgeable Synthesis (CPKS) is a product which features: synthesis, timing, placement (*Qplace®*), and global routing (*Wroute technology*). According to *Cadence* the timing delivered will differ only 3% from final routed timing [9, 17]. CPKS uses *BuildGates®* Synthesis as a base for synthesis, with enhancement/additional physical knowledge. This is because today the physical state of a design must be taken into account and evaluated when computing delays [9, 17]. The key components in the PKS technology are [9, 18]:

- **Timing using physical placement**

  CPKS will add a physical timing model to the already existing model made by AmbitBG. Hence, allowing the optimizing process to take physical timing in consideration at an early stage. It uses Steiner routes to estimate interconnects which gives an accurate approximation.
- **Incremental placement of logical changes**

  CPKS uses an incremental placement engine which makes its possible during netlist optimizing to do many transforms such as:
  
  - Repeater insertion (buffers)
  - Gate-cloning
  - Full restructuring

- **Timing and congestion-driven placement**

  It keep tracks of placed cells and free space, thus avoiding congesting of signals and cells.

### A.2 Notation and TCL script

The curly brackets {} denotes an argument to be provided, the or operator | denotes different choices, for example:

\[
\text{do\_optimize \{-priority area / time\}}
\]

Which means that the command name is `do_optimize` and may take the argument `-priority` and that it will either optimize for `time` or for `area`. The script syntax used is TCL script syntax (Tool command language), it should be mention that the dollar sign ($) denotes a variable which is initialized by the `set <variable name> <value>` command.

### A.3 Synthesis flow

Here a brief summary of the synthesis flow is presented.

#### A.3.1 Libraries

To be able to map your design to a particular process, you will need library’s provided by your ASIC vendor. They contain information about: functionality, timing, and power. First you need to load the timing library (TLF) in which the timing information exists. Second to load is the physical library (LEF) which includes information such as wire unit parasitic information, physical footprints, pin locations, detailed cell size, routing level information, etc. Probably your ASIC vendor has already provided you with a TCL file from which the required libraries can be read in from. The file name should be `read_libs.tcl` and exist under a directory called ambit, i.e,
env(VENDOR_DIR)/ambit/<processes>/read_libs.tcl

The commands used here are:

```tcl
read_tlf
read_symbol
read_symbol_update
read_lef
read_lef_update
```

The *update* alternative syntax is used to update an existing LEF file, so that corrections can be made without generating a new file [9, 55].

### A.3.2 Register Transfer Level (RTL) synthesis

Although CPKS supports other formats such as: EDIF and Verilog the focus in this appendix is on VHDL. In this step the design will be transformed to a generic netlist that contains units such as flip-flops, arithmetic logic units (ALU’s), etc. It is recommended to read in your files to a library that is mapped to a directory. Hence, your synthesized files will be stored in a known location [9, 59]. After specifying a library you can begin to read in your files:

```tcl
set_vhdl_library <library name> <library directory>
read_vhdl -library <library name> <vhdl files>
```

### A.3.3 Timing constraints

Timing constraints are placed on signals arriving or departing from the design. Depending on selected cells and process, the timing may differ [9, 65].

### A.3.4 Floorplanning

In floorplanning it is decided where to place blocks and logic on your chip design. CPKS can handle initial floorplanning. However for more advance floorplanning *Cadence* recommends to use *Silicon Ensemble*® or other tools [9, 67]. The `set_floorplan_parameters` command recommended parameters are:

- **row_utilization_initial 65**
  
  Set the initial row utilization to 65% of the row.

- **max_row_utilization 90**
  
  Set the maximum row utilization to 90% of the row.
• abut_row_pairs
  Informs PKS that pairs of rows must be abutted.

• flip_alternate_rows
  Instructs CPKS to flip every other row. The first (bottom) row will always
  be set in the north position.

### A.3.5 Optimization

Theses are the different steps involved in the optimization [9, 102]:

• Pre Optimization
  Uniquification, constant propagation, structuring, dissolving, and redund-
  dancy removal.

• Timing Optimizations
  Resizing, buffering, cloning, pin swapping, restructuring, area reclaim.

• Final Optimizations
  Design-rule fixing and multiport net fixing.

### A.3.6 Post-Placement optimization

After the initial optimization the next step is placement. During this phase CPKS
extracts capacitance and resistance information, and the delay in the nets can
now be more accurately computed [9, 104]. These are the steps involved:

1. Placement Initialization and Setup
   In this step CPKS divides the design in a grid net, thus making it easier
to overview utilization and congestion problems. The problems are then
analyzed and resolved (if possible) before going any further.

2. Incremental Placement
   The incremental placer in CPKS use a proprietary algorithm to find the
optimal locations for new instances in the design, which has been added in
the optimization step in PKS.

3. Congestion and Utilization Limit Optimization

4. Routing Estimation and Timing
   CPKS uses a Steiner routing algorithm to generate the net routes.

5. Congestion Analysis
6. Measuring Supply Rail Resources

7. Make Routable
   If there are any congestion problems these cells will be moved to other locations in this step. This is commonly referred to as *smoothing* the design according to *Cadence*.

8. Adjusting Placement for Optimizability
   In this step all over utilization caused by optimization will be taken care of.

9. Legalization
   The *smoothing* and the incremental optimization may generate illegal placement. Therefore this step will verify that the instances have correct placement after physical optimization. It also performs cleanup of the layout. After this step the design is ready for routing.

A.3.7 do\_optimize

It should be noted that if you either load a floorplan file or use the command *set_floorplan\_parameters* PKS based optimization is performed.

A.3.7.1 Pre placement optimization

The following commands are useful in incremental optimization:

\[
\text{do\_optimize} \ -\text{incremental}
\]

A.3.7.2 Path groups

To set a path group, use the following command: *set_path\_group*. Groups are a set of paths that is used for timing analysis and timing optimization. This command can be used in conjunction with the *set_path\_group\_options*.

- Path groups are useful when setting different timing constraints on different signals, which are to be individually optimized.

- If you only want to optimize flip-flops etc. group all inputs and outputs in a path group:

\[
\begin{align*}
\text{set\_path\_group} & \ -\text{name} \ IO \\
& \ -\text{from} \ [\text{find} \ -\text{inputs} \ -\text{no\_clock}] \\
\text{set\_path\_group} & \ -\text{name} \ IO \\
& \ -\text{to} \ [\text{find} \ -\text{outputs}]
\end{align*}
\]

- If you have different clocks in your design you can set the path groups like:
set_path_group -name 200MHZ -clock_to slow_clk1
set_path_group -name 800MHZ -clock_to fast_clk1
set_path_group -name 800MHZ -clock_to fast_clk2

- It should however be noted that adding path groups will increase the memory consumption for CPKS. So it is recommended to have clear (non overlapping) path groups. Carefully evaluations and by setting the right path groups will reduce run time.

- To help you identify the nets that have fanout’s exceeding a given threshold, use the following command [9, 121]:

```plaintext
report_net -hier
    -min_fanout fanout_threshold
    -summary
```

### A.3.7.3 Generic logic optimization

CPKS will use the `get_state_of_design` to find out in which state the design is, and therefore there is no need to specify from where to optimize [9, 122].

### A.3.7.4 Dissolving (break up) Hierarchy

If you have many hierarchies in your design this may prevent/restrict the ability to combine/share common logic. The Quality of result (QOR) may be improved if you dissolve the hierarchies. However it should be noted that dissolving into too small units will probably produce worse results. If you notice that AmbitWare hierarchies are involved in your most critical paths then you should consider dissolving them. Based on data collected from different design cases Cadence has set these default values to:

```
aware_dissolve_width 4
aware_mux_dissolve_size 8
```

However, these globals affect all aware components which may not be wished. Hence it is often better to dissolve specific instances or modules:

```
do_dissolve_hierarchy [-hierarchical] [instance_list]
```

If hierarchical is used the instance/module will be recursively dissolved [7, 55].
A.3.7.5 Preserve design

Use the \texttt{set\_dont\_modify} command to preserve the optimization applied to modules, instances, or nets. The command syntax is:

\begin{verbatim}
set_dont_modify [-network]
[-hierarchical]
list_of_object_id
\end{verbatim}

This is useful if you for example have a custom made RAM that you do not want to be touched in the optimization process.

A.3.7.6 Tasks: Post-placement Optimization

- Set current module and top module. The first step is to process large fanout nets, CPKS will try to buffer these nets and if unable to do so mark them as \textit{ideal nets}, thus avoiding these nets to affect timing (these are dealt with later).

- CPKS will look over the complete design and try to do redesign it: downsize cells, remove buffers, and clone logic cells.

- It will insert clock buffers to avoid fanout problems \cite{9, 135}.

A.3.8 Placement (\texttt{do\_placement})

The command \texttt{do\_placement} is usually called from \texttt{do\_optimize} in the initial placement and legalize placement flow.

A.3.9 Clock tree synthesis

This chapter describes clock tree synthesis (CTPKS) in the CPKS. CTPKS will place buffers or inverters to create a clock tree, and also make the tree balanced, ie, group leaf nodes \cite{9, 162}. Arguments for \texttt{set\_clock\_tree\_constraints} are:

- \texttt{-min\_delay 0.1}
- \texttt{-max\_delay 1.0}
- \texttt{-max\_skew 0.3}
- \texttt{-max\_tree\_transition 0.3 -pin [find\ -port clk]}
- \texttt{-physical\_tree}

Specifies that the constraints are for the \texttt{do\_build\_physical\_tree} command for non-clock signals.
A.3.9.1 Frequency Dividers

CTPKS can build a clock tree through a register used as clock frequency divider:

```
set_generated_clock -from ClkRAM_t_reg/C
  -divide_by 8
  -name gramclk ClkRAM_t_reg/Q

set_generated_clock -from ClkFFT_t_reg/C
  -divide_by 2
  -name gfftclk ClkFFT_t_reg/Q
```

To ensure that this register is not treated as a leaf, set the `ct_no_leaf` attribute for its clock input. Set the value of the attribute to the name of the register output, through which the clock tree should be built:

```
set_attribute [find -pin DIV1/CP] ct_no_leaf true Q
```

A.3.9.2 Reoptimize clocks

It is possible to reoptimize the clock tree to achieve better results, by using the `do_optimize_clock_tree` command. This will be done on an already existing clock tree [9, 175]. When issued it will try to minimize:

1. Number of design rule violations
2. Worst maximum latency violation
3. Worst skew violation
4. Worst minimum latency violation

There are some argument that should be considered to use: `symmetric`, `asymmetric`, and `rebalance_subtree`.

A.3.10 Global routing (**do_route**)

It should be noted that the final routing is not included in CPKS, and must therefore use an external tool, either `wroute` or *Silicon Ensemble*. The tool is capable of `timing_driven` routing which will focus on timing critical paths. The `do_route` command will store the routing information in the netlist.

**Note:** After running `do_optimize` you must reroute your design.

At this point you should only do (in optimization): resizing or buffering and not area reclamation. This can be achieved by the argument `-ipo` for `do_optimize`. The recommended flow for routing and final (detail) routing is:
do_optimize -ipo
do_route -timing_driven true
do_optimize -ipo
do_route -timing_driven true -output_db_name route.wdb
do_wroute -inputDbName route.wdb
    -outputDbName groute.wdb
    -outputDefName groute.def
    -run

A.4 VHDL Pragmas

A.4.1 Synthesis on/off
The synthesis on and off directive is used to tell CPKS not to synthesize the code between the directives. This is useful when only simulation or debugging code exist in the design [7, 175].

    -- ambit synthesis off
    :
    -- ambit synthesis on

A.4.2 Translation on/off
If desired CPKS can completely ignore the VHDL code, even if it contains errors [7, 175]:

    -- translate on
    :
    -- translate off

A.4.3 Architecture selection
Normally CPKS select the best adder/multiplier architectures regarding timing and area, but it is possible to select a specific architecture [7, 176]. The possible types of adders are:

- fcla (fast carry lookahead)
- cla (carry lookahead)
- csel (carry select)
- ripple (ripple carry)

And for multiplier encoding:
- non-booth
- booth

Example;

```plaintext
-- uses a cla and a fcla
x <= a + -- ambit synthesis architecture = cla
b + c;  -- ambit synthesis architecture = fcla
```

### A.4.4 Sum of product logic (SOP)

If the assigned variable in a case statement purely depends on constant values, CPKS will extract/create a boolean expression for the output. To avoid this use;

```plaintext
set_global hdl_extract_sum_of_products_logic
{true | false}
```

The default value is `true`. This is useful when your module have many expressions that should not be treated as boolean expression this to avoid long rung times [7, 54].

#### A.4.4.1 WPGen

In Section 5.5 it was stated that when `WPGen` is synthesized a large network of logic is created, making the block area consuming and timing demanding. Late in the report writing this new function mentioned above was discovered in CPKS. A curiosity arose how this function would affect `WPGen` since this block was dominated by SOP networks. A new synthesis was made and the results can be seen (Table A.1). It is clear that the results are improved compared to those in Table 5.13, both concerning area and timing. In this case the maximum possible frequency reaches 30 $MHz$ compared to 13 $MHz$ in the previous case. In other words, the initial goal frequency of 32 $MHz$ is almost reached. And to mention the area it is decreased to nearly 1/3. This conclusion will not change the discussion in Section 5.5 but it is noted that this function is useful in cases where unwanted logic appears.

<table>
<thead>
<tr>
<th></th>
<th>SOP = false $f = 32 MHz$</th>
<th>SOP = false $f = 1 MHz$</th>
</tr>
</thead>
<tbody>
<tr>
<td>Worst slack (ns) / Max freq (MHz)</td>
<td>-1/30</td>
<td>948/19</td>
</tr>
<tr>
<td>Area ($\mu m^2$)</td>
<td>35668</td>
<td>31704</td>
</tr>
</tbody>
</table>

**Table A.1:** Synthesis result of WPGen with SOP = false
A.5 High level optimization

CPKS and AmbitBG performs many high level optimization techniques by default, that (probably) produces better netlist when turned on:

- Implicit Constant Propagation (ICP)
  The tool tries to find values that are constant in the flow.

- Common Sub-Expression Elimination (CSE)
  Removes redundant arithmetic expressions.

A.5.1 Resource sharing

If the same arithmetic operations are performed, it may be possible to only generate one hardware implementation of it. This can drastically reduce the required area if many complex arithmetic operations are performed. CPKS will however never violate any given timing constraints when trying to do resource sharing. Because no special directives are required lightens the burden of the designer.

```
set_global hdl_resource_sharing {true | false}
Default: true
```

Resource sharing is only possible when the expressions are mutually exclusive. Example:

```
if (sel = '1') then
  x <= a + b;
else
  y <= c + d;
end if;
```

CPKS [9, 123] is also capable of sharing adders/subtractors that share common logic. However, it should be noted that if a complex FSM is designed, care must be taken when synthesizing. If CPKS not understands that the design is a FSM, resource sharing is impossible.
A.6 Tutorial

Here follows a complete set of commands to use when synthesizing. However, this is not a tutorial for TCL scripting (only the most basic commands are shown). More complicated scripts have been used in this thesis.

1. This is a process (function, method) in which the necessary reports are generated. The character > means that it should write (pipe) its output to a file. So if the process is called with genReports (ambit/reports) the output from check_netlist will be stored in the file “ambit/reports/check_netlist”.

   ```tcl
   proc genReports {foo} {
      check_netlist -verbose
      report_analysis_c -verbose violated
      report_timing
      report_timing -uncon
      report_path_exceptions
      report_area
      report_resources
      report_fsm -stat
      report_path_group_timing
      report_power
   }
   ```

2. Read the necessary libraries and clear out previous designs:

   ```tcl
   source tcl/read_libs.tcl
   do_remove_design -all
   ```

3. Here the clock name and speed (in MHz) are set, design (a variable) is the name of the VHDL file to read:

   ```tcl
   set clk_name Clk
   set clk_cyc 32.0
   set design FFT
   ```

4. Read the VHDL files:

   ```tcl
   set_vhdl_library <lib name> <directory>
   read_vhdl -library <lib name> $design.vhd
   ```

5. Set operation parameters:

   ```tcl
   set_operating_parameter -voltage 3.3
   set_operating_parameter -temperature 105.0
   ```
6. Store the current date and time in the variables date and time.

```
set date [exec date "+%Y.%m.%d"]
set time [exec date "+%H:%M"]
```

7. Here different output directories are set. A "$" means a variable, so for example $date will be replaced with the current date. Thus $design (as declared above) will be replaced with the design name. This is convenient because $dir_design will then (as set here) change for each run, thus avoiding overwriting of previous runs:

```
set dir_design db/$design/$clk_cycMHz/$date-$time
set dir_adb $dir_route/$design.done.adb
set dir_report $dir_route/reports
set dir_verilog $dir_route/$design.v
mkdir -p $dir_report
```

8. This builds the generic netlist and also find (and extracts) any possible FSM. The option -sleep_mode is used to explore low power synthesis (LPS) available modules (see Appendix A.8):

```
do_build_generic -module $design
    -extract_fsm
    -sleep_mode
```

9. Set a top and current module:

```
set_current_module $design
set_top_timing_module $design
set_floorplan_parameters
    -row_utilization_initial 65
    -max_row_utilization 90
    -abut_row_pairs
    -flip_alternate_rows
```

10. Set the ideal clock:

```
set foo [expr 1.0 / $clk_cyc * 1000.0]
set_clock vclk0 -period $foo
    -waveform "0 [expr $foo / 2]"
set_clock_root -clock vclk0
    [find -port $clk_name ]
    -pos
set_clock_insertion_delay -source
    -clock vclk0
    -rise 0.2
    -pin $clk_name
```
11. You **must** set input and external delay and also drive resistance to avoid violations:

```tcl
set_input_delay -clock vclk0 0.5
    [get_names [ find -inputs -no_clocks *]]
set_external_delay -clock vclk0 0.5
    [ get_names [ find -outputs *]]
set_drive_resistance -clock vclk0
    -rise
    -slew_res 0.03
    -slew_intrinsic 0.08 0.9 *
set_drive_resistance -clock vclk0
    -fall
    -slew_res 0.03
    -slew_intrinsic 0.08 0.9 *
set_drive_resistance -clock vclk0
    -pos-slew_res 0.03
    -slew_intrinsic 0.08 0.9 *
```

12. Set the path group:

```tcl
set_path_group -name vclk0.group
    -clock_from vclk0
set_path_group_options vclk0.group
    -target_slack 0.0
    -all_end_points
set_clock_tree_constraints -pin $clk_name
    -min_delay 0.1
    -max_delay 0.3
    -max_skew 0.1
```

13. First optimize run:

```tcl
do_optimize -power
    -insert_clock_tree
    -priority {time | area}
```

14. Here the routing is performed:

```tcl
do_optimize -ipo
do_route -timing_driven true
do_optimize -ipo
do_route -timing_driven true
    -output_db_name $dir_route/route.wdb
do_wroute -inputDbName $dir_route/route.wdb
    -outputDbName $dir_route/groute.wdb
```

---

A-Cadence PKS and
Ambit BuildGates
15. Save the database and the Verilog file, and generate reports:

```plaintext
write_adb $dir_adb
write_verilog -hierarchical $dir_verilog
genReports $dir_report
```

### A.7 Synthesis of common arithmetic

As mentioned in Chapter 5.3 it is good to synthesize some common arithmetic operations. This to see if some arithmetic operations may be critical. The notation 24s means 24 bits signed, and the “module” (in the area column) means only the arithmetic block, the rest of the area consists of the net.

<table>
<thead>
<tr>
<th>Operation</th>
<th>Goal f. ((MHz))</th>
<th>Max f. ((MHz))</th>
<th>Area ([\text{tot}/\text{mod}] ,(\mu\text{m}^2))</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Addition</td>
<td>32</td>
<td>66</td>
<td>721/349</td>
<td>ripple</td>
</tr>
<tr>
<td>24s + 24s</td>
<td>64</td>
<td>66</td>
<td>721/349</td>
<td>ripple</td>
</tr>
<tr>
<td>-</td>
<td>96</td>
<td>222</td>
<td>1345/973</td>
<td>fcla</td>
</tr>
<tr>
<td>-</td>
<td>256</td>
<td>225</td>
<td>1360/988</td>
<td>fcla</td>
</tr>
<tr>
<td>Subtraction</td>
<td>32</td>
<td>66</td>
<td>770/398</td>
<td>ripple</td>
</tr>
<tr>
<td>24s – 24s</td>
<td>64</td>
<td>66</td>
<td>770/398</td>
<td>ripple</td>
</tr>
<tr>
<td>-</td>
<td>96</td>
<td>208</td>
<td>1412/1040</td>
<td>fcla</td>
</tr>
<tr>
<td>-</td>
<td>256</td>
<td>226</td>
<td>1475/1103</td>
<td>fcla</td>
</tr>
</tbody>
</table>

**Table A.2:** Synthesis summary for addition/subtraction using 24 bits signed

<table>
<thead>
<tr>
<th>Operation</th>
<th>Goal f. ((MHz))</th>
<th>Max f. ((MHz))</th>
<th>Area ([\text{tot}/\text{mod}] ,(\mu\text{m}^2))</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Subtraction</td>
<td>32</td>
<td>36</td>
<td>1410/738</td>
<td>ripple</td>
</tr>
<tr>
<td>44s – 44s</td>
<td>64</td>
<td>157</td>
<td>2842/2170</td>
<td>fcla</td>
</tr>
<tr>
<td>-</td>
<td>96</td>
<td>157</td>
<td>2842/2170</td>
<td>fcla</td>
</tr>
<tr>
<td>-</td>
<td>256</td>
<td>214</td>
<td>3304/2629</td>
<td>fcla</td>
</tr>
</tbody>
</table>

**Table A.3:** Synthesis summary for subtraction using 44 bits signed
<table>
<thead>
<tr>
<th>Operation</th>
<th>Goal f. (MHz)</th>
<th>Max f. (MHz)</th>
<th>Area [tot/mod] (µm²)</th>
<th>Arch.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Multiplication</td>
<td>1</td>
<td>30</td>
<td>7644/6957</td>
<td>ripple/bootth</td>
</tr>
<tr>
<td>24s × 21s</td>
<td>32</td>
<td>49</td>
<td>8240/7553</td>
<td>csel/bootth</td>
</tr>
<tr>
<td>-</td>
<td>64</td>
<td>63</td>
<td>9095/8405</td>
<td>fcla/bootth</td>
</tr>
<tr>
<td>-</td>
<td>128</td>
<td>90</td>
<td>14875/14025</td>
<td>fcla/non_booth</td>
</tr>
</tbody>
</table>

Table A.4: Synthesis summary for multiplication using 24 bits signed

A.8 Low Power Synthesis (LPS)

CPKS and AmbitBG are capable of saving power by using different techniques:

- Clock gating (CG)
- Sharing of clock gate logic

As this was discovered late in the project, not much efforts has been put in evaluating this. However, it can be seen in Table A.5 that the use of LPS decrease the power consumption [10].

<table>
<thead>
<tr>
<th>Operation</th>
<th>Area (µm²)</th>
<th>Constraint goals (MHz)</th>
<th>Power (µW)</th>
<th>Wasted power (µW)</th>
<th>Static power (µW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>OutputCtrl</td>
<td>888</td>
<td>1</td>
<td>469</td>
<td>99</td>
<td>7</td>
</tr>
<tr>
<td>OutputCtrl (CG)</td>
<td>1068</td>
<td>1</td>
<td>262</td>
<td>89</td>
<td>8</td>
</tr>
<tr>
<td>OutputCtrl</td>
<td>888</td>
<td>32</td>
<td>2499</td>
<td>250</td>
<td>7</td>
</tr>
<tr>
<td>OutputCtrl (CG)</td>
<td>796</td>
<td>32</td>
<td>1627</td>
<td>504</td>
<td>6</td>
</tr>
<tr>
<td>Butterfly</td>
<td>27920</td>
<td>8/4</td>
<td>12219</td>
<td>1344</td>
<td>53</td>
</tr>
<tr>
<td>Butterfly (CG)</td>
<td>24513</td>
<td>8/4</td>
<td>3422</td>
<td>787</td>
<td>58</td>
</tr>
<tr>
<td>Butterfly</td>
<td>41863</td>
<td>256/128</td>
<td>66421</td>
<td>9963</td>
<td>1822</td>
</tr>
<tr>
<td>Butterfly (CG)</td>
<td>56108</td>
<td>256/128</td>
<td>37665</td>
<td>9792</td>
<td>1533</td>
</tr>
</tbody>
</table>

Table A.5: LPS synthesis result
NanoSim appendix

The last part in the design flow of the FFT was to find critical parts from the aspect of power consumption. In order to do this, the simulation tool NanoSim from Synopsys has been used. NanoSim required several input files to work correctly.

- Stimulus file (vector file)
- Optimized Verilog file of the design
- SPICE netlist of all the instances in the Verilog file
- Configuration, command and run script files

The stimuli file is a vector file containing stimulus (random binary data) to all the input pins of the design. To create the vector files, the command “write” to file in the VHDL code has been used. The Verilog file of the optimized design is generated from the synthesis tool (Leonardo or CPKS). An optimized design is achieved when the results from the synthesis tool reach the demands stated by the specification of the design and after place and route. The generated Verilog file includes all the physical instances used in the design (e.g gates, adders etc.). These instances can be found as spice netlists in the core library of the desired process (in this thesis a 0.35\,\mu m process). If NanoSim can not find a netlist for an instance in the design the simulation will fail. You will then have to generate that particular netlist, which can easily be done using Cadence and save it to your core library.

The configuration file includes the commands you wish NanoSim to perform during the simulation, and what to report when the simulation is finished. The command file declares what sort of stimuli that is used, where the file is located and what input signals to include. Finally, a run script is created, which starts the NanoSim simulator, and declares the input files (and location) required to run a simulation.
B.1 Specifics

*NanoSim* is a powerful tool when it comes to simulating a circuit and there are many options in how the result is presented. It is recommended to do power simulation on the top level or hierarchy. This means that the power consumption is computed for the complete design and for all sub blocks separately. By doing this, the results can be compared between different test cases. Further, if a hot spot is traced in any sub block, which needs further examination you can move down in the hierarchy of the block locating and isolating the source. The results compared in the simulations are primary “wasted current percentage” and “average block power”. The average block power works as described above. In order to clarify what wasted current means, a further explanation in how *NanoSim* works is acquired.

*NanoSim* defines wasted current (or power), as any DC current consumed by the circuit block, that is not used to charge block capacitances or capacitances driven by the block. The total current consumed by the circuit is divided in two categories, capacitive current and wasted current respectively. The capacitive current is essential in carrying out the switching functions of the circuit, while the wasted current is any current, for which both ends of the conducting path are voltage sources. Usually this wasted power either consist of DC current flowing from power to ground, floating inputs or unintended DC paths resulting in current leakage. The wasted current can be split into even smaller parts, static and dynamic current. Static wasted current is the part of current wasted when the circuit is inactive (i.e. current leakage) while dynamic wasted current is the current wasted when the circuit is active (i.e. short-circuit) [11]. By having these expressions clear it will be easier for the reader to understand and follow the discussion in Chapter 6.
B.2 Results

Here are the complete results shown from the different test cases mentioned in Chapter 6, in order to make thoroughly comparisons. A list of notations for the results follows:

<table>
<thead>
<tr>
<th>Av</th>
<th>Average</th>
<th>pwr</th>
<th>Power</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>Current</td>
<td></td>
<td></td>
</tr>
<tr>
<td>C</td>
<td>Controller</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BFn</td>
<td>Butterfly n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>OC</td>
<td>OutputCtrl</td>
<td></td>
<td></td>
</tr>
<tr>
<td>CCn</td>
<td>CacheCtrl n</td>
<td></td>
<td></td>
</tr>
<tr>
<td>DR</td>
<td>DigitReversal</td>
<td></td>
<td></td>
</tr>
<tr>
<td>AG</td>
<td>AddressGen</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BIG</td>
<td>BaseIndexGen</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

B.2.1 Test case 1

<table>
<thead>
<tr>
<th>Block</th>
<th>Av supply I (µA)</th>
<th>Av cap I (µA)</th>
<th>Av wasted I (µA)</th>
<th>Av wasted I (%)</th>
<th>Av block pwr (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>12972</td>
<td>615</td>
<td>12357</td>
<td>95</td>
<td>43</td>
</tr>
<tr>
<td>C</td>
<td>104</td>
<td>73</td>
<td>31</td>
<td>30</td>
<td>0.34</td>
</tr>
<tr>
<td>WPGen</td>
<td>3361</td>
<td>10</td>
<td>3350</td>
<td>99</td>
<td>11</td>
</tr>
<tr>
<td>BF1</td>
<td>602</td>
<td>169</td>
<td>433</td>
<td>72</td>
<td>2.0</td>
</tr>
<tr>
<td>BF0</td>
<td>601</td>
<td>168</td>
<td>433</td>
<td>72</td>
<td>2.0</td>
</tr>
<tr>
<td>OC</td>
<td>8</td>
<td>2.4</td>
<td>5.6</td>
<td>70</td>
<td>0.026</td>
</tr>
<tr>
<td>CC1</td>
<td>72</td>
<td>68</td>
<td>4.3</td>
<td>6</td>
<td>0.24</td>
</tr>
<tr>
<td>CC0</td>
<td>72</td>
<td>68</td>
<td>4.3</td>
<td>6</td>
<td>0.24</td>
</tr>
<tr>
<td>AG</td>
<td>800</td>
<td>33</td>
<td>767</td>
<td>96</td>
<td>2.6</td>
</tr>
<tr>
<td>DR</td>
<td>15</td>
<td>4</td>
<td>11</td>
<td>73</td>
<td>0.05</td>
</tr>
<tr>
<td>BIG</td>
<td>7336</td>
<td>20</td>
<td>7316</td>
<td>99</td>
<td>24</td>
</tr>
</tbody>
</table>

Table B.1: Results of test case 1
### B.2.2 Test case 2

<table>
<thead>
<tr>
<th>Block</th>
<th>Av supply I (µA)</th>
<th>Av cap I (µA)</th>
<th>Av wasted I (µA)</th>
<th>Av wasted I (%)</th>
<th>Av block pwr (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>2047</td>
<td>818</td>
<td>1229</td>
<td>60</td>
<td>6.8</td>
</tr>
<tr>
<td>C</td>
<td>112</td>
<td>80</td>
<td>32</td>
<td>28</td>
<td>0.4</td>
</tr>
<tr>
<td>WPGen</td>
<td>57</td>
<td>10</td>
<td>46</td>
<td>82</td>
<td>0.2</td>
</tr>
<tr>
<td>BF1</td>
<td>643</td>
<td>205</td>
<td>438</td>
<td>68</td>
<td>2.1</td>
</tr>
<tr>
<td>BF0</td>
<td>643</td>
<td>205</td>
<td>438</td>
<td>68</td>
<td>2.1</td>
</tr>
<tr>
<td>OC</td>
<td>8</td>
<td>2</td>
<td>6</td>
<td>71</td>
<td>0.026</td>
</tr>
<tr>
<td>CC1</td>
<td>100</td>
<td>92</td>
<td>9</td>
<td>9</td>
<td>0.33</td>
</tr>
<tr>
<td>CC0</td>
<td>110</td>
<td>100</td>
<td>9</td>
<td>8</td>
<td>0.36</td>
</tr>
<tr>
<td>AG</td>
<td>181</td>
<td>33</td>
<td>148</td>
<td>82</td>
<td>0.6</td>
</tr>
<tr>
<td>DR</td>
<td>14</td>
<td>3</td>
<td>-11</td>
<td>76</td>
<td>0.05</td>
</tr>
<tr>
<td>BIG</td>
<td>111</td>
<td>22</td>
<td>-89</td>
<td>80</td>
<td>0.37</td>
</tr>
</tbody>
</table>

Table B.2: Results of test case 2

### B.2.3 Test case 3

<table>
<thead>
<tr>
<th>Block</th>
<th>Av supply I (µA)</th>
<th>Av cap I (µA)</th>
<th>Av wasted I (µA)</th>
<th>Av wasted I (%)</th>
<th>Av block pwr (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>602</td>
<td>505</td>
<td>96</td>
<td>16</td>
<td>2.0</td>
</tr>
<tr>
<td>C</td>
<td>18</td>
<td>17</td>
<td>1</td>
<td>7</td>
<td>0.06</td>
</tr>
<tr>
<td>WPGen</td>
<td>76</td>
<td>24</td>
<td>52</td>
<td>72</td>
<td>0.25</td>
</tr>
<tr>
<td>BF1</td>
<td>116</td>
<td>108</td>
<td>8</td>
<td>7</td>
<td>0.38</td>
</tr>
<tr>
<td>BF0</td>
<td>117</td>
<td>108</td>
<td>9</td>
<td>7</td>
<td>0.39</td>
</tr>
<tr>
<td>OC</td>
<td>9</td>
<td>3</td>
<td>6</td>
<td>71</td>
<td>0.03</td>
</tr>
<tr>
<td>CC1</td>
<td>108</td>
<td>101</td>
<td>8</td>
<td>7</td>
<td>0.36</td>
</tr>
<tr>
<td>CC0</td>
<td>108</td>
<td>100</td>
<td>8</td>
<td>7</td>
<td>0.36</td>
</tr>
<tr>
<td>AG</td>
<td>4</td>
<td>0.69</td>
<td>3</td>
<td>81</td>
<td>0.01</td>
</tr>
<tr>
<td>DR</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
<tr>
<td>BIG</td>
<td>0</td>
<td>0</td>
<td>0</td>
<td>100</td>
<td>0</td>
</tr>
</tbody>
</table>

Table B.3: Results of test case 3
## B.2.4 Complete FFT

<table>
<thead>
<tr>
<th>Block</th>
<th>Av supply I ($\mu A$)</th>
<th>Av cap I ($\mu A$)</th>
<th>Av wasted I ($\mu A$)</th>
<th>Av wasted I (%)</th>
<th>Av block pwr (mW)</th>
</tr>
</thead>
<tbody>
<tr>
<td>top</td>
<td>2567</td>
<td>1276</td>
<td>1291</td>
<td>50</td>
<td>8.5</td>
</tr>
<tr>
<td>C</td>
<td>127</td>
<td>93</td>
<td>34</td>
<td>27</td>
<td>0.4</td>
</tr>
<tr>
<td>WPGen</td>
<td>81</td>
<td>29</td>
<td>33</td>
<td>65</td>
<td>0.27</td>
</tr>
<tr>
<td>BF1</td>
<td>819</td>
<td>362</td>
<td>457</td>
<td>56</td>
<td>2.7</td>
</tr>
<tr>
<td>BF0</td>
<td>819</td>
<td>363</td>
<td>457</td>
<td>56</td>
<td>2.7</td>
</tr>
<tr>
<td>OC</td>
<td>9</td>
<td>2</td>
<td>6</td>
<td>71</td>
<td>0.03</td>
</tr>
<tr>
<td>CC1</td>
<td>164</td>
<td>150</td>
<td>13</td>
<td>8</td>
<td>0.54</td>
</tr>
<tr>
<td>CC0</td>
<td>163</td>
<td>149</td>
<td>14</td>
<td>8</td>
<td>0.54</td>
</tr>
<tr>
<td>AG</td>
<td>201</td>
<td>50</td>
<td>151</td>
<td>75</td>
<td>0.66</td>
</tr>
<tr>
<td>DR</td>
<td>14</td>
<td>4</td>
<td>11</td>
<td>76</td>
<td>0.05</td>
</tr>
<tr>
<td>BIG</td>
<td>115</td>
<td>26</td>
<td>89</td>
<td>78</td>
<td>0.38</td>
</tr>
</tbody>
</table>

**Table B.4:** Results of complete FFT
Correctness of produced Verilog files

To be able to verify the correctness of the CPKS generated files after routing, in terms of:

- Not too much delay through blocks, cells, gates.
- Clock tree.
- Simulation and synthesis mismatch.

Although CPKS should generate/fix the first two without interaction the third is more vital, because it is harder to find. This should not be (in theory) a problem. The routed files can be opened in *Silicon Ensemble* and there you are able to see wires, blocks, registers etc. but you can not do simulations. Different possibilities were then evaluated to verify the Verilog files. It might be possible to use *NanoSim* because all required input files have already been made. Further investigations showed that it is possible to look at any signals logic level/voltage. So it was decided to evaluate the possibilities to use *NanoSim* for verification.

C.1 NanoSim output file formats

As described in [11, 347] there are different output formats available:

- EPIC
  
  ASCII text based format.
- FSDB (Fast Signal Database)
  
  Compressed binary format, which can be displayed in the *turbowave* waveform tool.
- ISDB
  
  Proprietary format which only can be displayed in the *SimWave* waveform viewer.

The format can be selected by the following command in the configuration file:

```
set_print_format [for=epic|fsdb|isdb]
```
Instead of using a new tool, the possibilities to use an already well known tool was evaluated. *ModelSim* has a waveform viewer with waveform compare abilities. The accepted format is only WLF (Waveform Library File), but there exist a utility called *vcd2wlf* which converts from the IEEE 1000-1364 standard VCD (Value Change Dump) format to WLF. Unfortunately, as stated, *NanoSim* can not output VCD files so the problem lies in converting EPIC format to VCD format.

### C.2 Decision

The decision was made to transform the EPIC output file to a VCD file. The syntax of the two file formats are similar, and quite easy to transform. The program decided to be used was *gawk* which is a *pattern scanning and processing language* featuring both regular expression match, line record search and *C* like programming syntax. The basic concept of *gawk* is that it scans the input file for patterns which are specified in the program, and perform an associated action. Another method is to scan each line and break them up in fields [12].

### C.3 EPIC file format

**EPIC file format**

**Header:**

- `.time_resolution [resolution]`

  Tells which time unit that is used and need to considered.

- `.bus_notation [left] - [right]`

  Which character that is used as bus character, default [ and ].

- `.index <variable name> <node index> <type=l(logic)|v(voltage)>`

  A node declaration with index declaration and type.

**Simulation data:**

- `<time>`

  Tells that the current time is `<time>`.

- `<node index> <value>`

  Indicates that the given node index has changed value.
C.3.1 Program flow

Conversion:

init

Write out VCD file header: date, version

Readlines from file:

Header

Print out correct time resolution and variable names

Simulation data:

1 col   Save time
2 col   A node has changed voltage value -> search node and print node, and time

It should be noted that even though you simulate a pure digital design it is necessary to use analogism to get a correct (improved) time resolution:

```
set_sim_level level=[1|2|3|4]
```

This sets the simulation level. It should be noted that the simulation time increases rapidly if you do a pure analog simulation for a digital circuit. However, it should be sufficient to use level 1 (level 4 is pure analog simulation).
ModelSim and makefile generation

If using the VHDL mode in *XEmacs* it is possible to generate a Makefile for the VHDL files automatically:

```
xemacs -batch
   -l ~/.xemacs/custom.el
   -l vhdl-mode
   -f vhdl-generate-makefile
```

The compiler options used are:

<table>
<thead>
<tr>
<th>Compiler Option</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>-check_synthesis</td>
<td>Use VHDL 93 standard</td>
</tr>
<tr>
<td>-93</td>
<td>Use VHDL 93 standard</td>
</tr>
<tr>
<td>-O4</td>
<td>Optimize more</td>
</tr>
<tr>
<td>-pedantic errors</td>
<td>Be more pedantic about errors</td>
</tr>
<tr>
<td>-nologo</td>
<td>Do not show logo</td>
</tr>
<tr>
<td>-source</td>
<td>Be more verbose when reporting errors</td>
</tr>
<tr>
<td>-quiet</td>
<td>Suppress unwanted information</td>
</tr>
</tbody>
</table>
This is a short introduction on how to use CVS in XEmacs. For a more comprehensive guide see [13, 14]. First you need to install CVS. Then you need to decide where to store the repository (can also be on a remote machine), which is done by setting the environment variable CVSROOT. For example:

```bash
$ export CVSROOT=/home/user/CVSROOT (bash)
$ setenv CVSROOT=/home/user/CVSROOT (other)
```

To initialize the CVSROOT for the first time:

```bash
$ cvs init
```

If you already have files that you would like to put under CVS control you can simply import them using the argument `import`. But before importing the files it should be noted that you may store some information within your files by using predefined keywords which will be updated when committing. The recommended keywords to use are:

- **$Source$**
  Get replaced with source file name
- **$Revision$**
  Get replaced with current revision of file
- **$Date$**
  Get replaced with last changed
- **$Log$**
  Each time the file is committed this log will be extended and contain: revision number, date, who modified it and the log message.

After adding the desired keywords, run:

```bash
cd directory
cvs import -m [message] <directory> <vendor tag> <module>
```

The `<module>` is used to refer to a specific project. Now the files are in the repository and to make sure it worked checkout the module:
When you have verified that the files has been put in the repository you can (should) remove the original source directory. Now it is time to start XEmacs, change directory to where you want to have your project files. In the menu Tools ⊳ PCL ⊳ CVS, are the main functions. To get a local copy of your project you need to do a `cvs checkout`. Then it appears (in XEmacs) “Modules(s):” write in your module name and select directory to save the files. Then some text about checked out files and modules appears.

Now you are ready to do some seriously editing. You are now located in the CVS buffer which is used to run most of the commands. Mark one of the files with the arrow keys or the mouse and press enter to open. In the lower window of XEmacs you will now note that it is aware that the files are under version control. Make some changes and save it, the version number will then be highlighted and mark the modified file. If you change back to the CVS buffer you will notice it says modified for the file. So far the file is only modified at your local directory.

To make the changes available to others you must commit the changes. Mark the desired file(s) by pressing the “m” key on the and to commit the file(s) press the “c” key. Then a buffer appears in which a message about what has been changed should be written. When you are done exit with ”C-c C-c”. Notice how the version number will increase and it states “up-to-date”. If you right click on a file, a menu with available commands will appear. Here are some useful commands available within the CVS buffer:

- **M-s**
  This will display if there are some files that need to be updated or committed

- **M-u**
  This will update the current directory.

You may also need to add this to your configuration file for XEmacs [13]:

```
(requires 'vc)
(requires 'vc-hooks)
```
References


Copyright

Svenska

Detta dokument hålls tillgängligt på Internet - eller dess framtida ersättare - under en längre tid från publiceringsdatum under förutsättning att inga extra-ordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns det lösningar av teknisk och administrativ art.

Uppphovsmannens ideella rätt innefattar rätt att bli nämnt som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är krävande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se fölagets hemsida: http://www.ep.liu.se/

English

The publishers will keep this document online on the Internet - or its possible replacement - for a considerable time from the date of publication barring exceptional circumstances.

The online availability of the document implies a permanent permission for anyone to read, to download, to print out single copies for your own use and to use it unchanged for any non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional on the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its WWW home page: http://www.ep.liu.se/

© Petter Näslund, Mikael Åkesson
Linköping, April 2005