Cosinus-modulerade filterbankar

Examensarbete utfört i *elektroniksystem*

av

Magnus Nord

LiTH-ISY-EX-3360-2003
Linköping 2003
COSINE MODULATED FILTER BANKS
IMPLEMENTATION AND COMPARISON

Master thesis in electronics systems,
Linköping Institute of Technology

By Magnus Nord

LiTH-ISY-EX-3360-2003

Supervisor: Linnéa Rosenbaum
Examinator: Håkan Johansson

Linköping, 2003-02-28
### Sammanfattning

Abstract

The initial goal of this report was to implement and compare cosine modulated filter banks. Because of time limitations, focus shifted towards the implementation. Filter banks and multirate systems are important in a vast range of signal processing systems. When implementing a design, there are several considerations to be taken into account. Some examples are word length, number systems and type of components. The filter banks were implemented using a custom made software, especially designed to generate configurable gate level code. The generated code was then synthesized and the results were compared. Some of the results were a bit curious. For example, considerable effort was put into implementing graph multipliers, as these were expected to be smaller and faster than their CSDC (Canonic Signed Digit Code) counterparts. However, with one exception, they turned out to generate larger designs. Another conclusion drawn is that the choice of FPGA is important. There are several things left to investigate, though. For example, a more thorough comparison between CSDC and graph multipliers should be carried out, and other DCT (Discrete Cosine Transform) implementations should be investigated.

### Nyckelord

Keyword

filter banks, DCT, implementation, FPGA, VHDL, DIT, DIF
Abstract

The initial goal of this report was to implement and compare cosine modulated filter banks. Because of time limitations, focus shifted towards the implementation. Filter banks and multirate systems are important in a vast range of signal processing systems. When implementing a design, there are several considerations to be taken into account. Some examples are word length, number systems and type of components. The filter banks were implemented using a custom made software, especially designed to generate configurable gate level code. The generated code was then synthesized and the results were compared. Some of the results were a bit curious. For example, considerable effort was put into implementing graph multipliers, as these were expected to be smaller and faster than their CSDC (Canonic Signed Digit Code) counterparts. However, with one exception, they turned out to generate larger designs. Another conclusion drawn is that the choice of FPGA is important. There are several things left to investigate, though. For example, a more thorough comparison between CSDC and graph multipliers should be carried out, and other DCT (Discrete Cosine Transform) implementations should be investigated.
# Table of Content

## 1 INTRODUCTION
1.1 Background ................................................................. 3  
1.2 Purpose ........................................................................... 3  
1.3 Intended Audience ....................................................... 3  
1.4 Restrictions ................................................................. 3  

## 2 THEORY
2.1 Filter Banks ....................................................................... 5  
  2.1.1 Multirate Systems ......................................................... 5  
  2.1.2 Polyphase Representation ........................................... 6  
2.2 The Discrete Cosine Transform ................................ ....... 7  
  2.2.1 DCT-IV ................................................................. 8  
  2.2.2 Fast DCTs ............................................................. 10  
2.3 Cosine Modulated Filter Banks ................................ ....... 11  
2.4 VHDL ............................................................................... 12  
  2.4.1 Signals vs Variables .................................................... 12  
  2.4.2 Simulation ............................................................... 12  
  2.4.3 Components ............................................................ 13  
  2.4.4 Testbenches ............................................................. 14  
2.5 FPGAs ............................................................................... 14  

## 3 IMPLEMENTATION
3.1 Arithmetics ....................................................................... 17  
  3.1.1 Two's-Complement ..................................................... 17  
  3.1.2 Fixed-Point Fractional ............................................... 18  
  3.1.3 CSDC ........................................................................... 18  
  3.1.4 Graph Multipliers ...................................................... 19  
3.2 Error Sources .................................................................... 20  
  3.2.1 Word Length ............................................................ 20  
  3.2.2 Number Representation .......................................... 20  
  3.2.3 Scaling ................................................................. 20  
  3.2.4 Coefficient Quantization ........................................ 21  
3.3 Adders ............................................................................... 21  
3.4 Multipliers ......................................................................... 21  
  3.4.1 Graph Multipliers ...................................................... 22  
3.5 Butterflies ......................................................................... 22  
3.6 Multiplier Blocks ............................................................. 23  
3.7 DCT Block ........................................................................ 23  

## 4 RESULTS AND CONCLUSIONS
4.1 Number of Channels ....................................................... 25  
4.2 Word Length ..................................................................... 26  
4.3 Type of DCT ...................................................................... 27  
4.4 Choice of Multipliers ....................................................... 28  
4.5 Comparison between DCTs and filters ............................... 29  
4.6 Future Work ................................................................. 30
<table>
<thead>
<tr>
<th>Section</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>A    DIGITAL FILTERS</td>
<td>31</td>
</tr>
<tr>
<td>B    DCT SIGNAL FLOW GRAPHS</td>
<td>33</td>
</tr>
<tr>
<td>C    TABLES AND GRAPHS</td>
<td>35</td>
</tr>
<tr>
<td>REFERENCES</td>
<td>37</td>
</tr>
</tbody>
</table>
Table of Figures

**Figure 2-1.** Example of a filter bank. It is divided into an analysis filter bank and a synthesis filter bank. .......................................................... 5

**Figure 2-2.** Operation of a decimator followed by an expander. The expander expands the signal \( y_\text{in}(n) \) and fills the gaps with zeros. An interpolation filter is then used to restore the original signal \( x(n) \). .......................................................... 6

**Figure 2-3.** Decimation circuit modified using polyphase representation and noble identities. In this case, \( M = 2 \). .......................................................... 6

**Figure 2-4.** Cosine modulated M-channel analysis filter bank. .......................................................... 11

**Figure 2-5.** Cosine-modulation block. \( C \) is the DCT, \( I \) and \( J \) are permutation matrices and \( \Lambda_\alpha \) is a diagonal matrix with elements \( \pm 1 \). ................................................. 12

**Figure 2-6.** VHDL simulation time queue. .......................................................... 13

**Figure 2-7.** General testbench structure. .......................................................... 14

**Figure 2-8.** Schematic of a Virtex-II slice. .......................................................... 15

**Figure 3-1.** Example input file generating a DCT block with word length 16. ............. 17

**Figure 3-2.** 0.875 in CSDC format. .......................................................... 19

**Figure 3-3.** (a) CSDC representation of multiplication by 45 (b) Graph multiplier representation of multiplication by 45. .......................................................... 19

**Figure 3-4.** Signed division by two implemented using shifting........................................... 22

**Figure 3-5.** Example of a scaled butterfly. \( \alpha \) is an arbitrary constant. ..................... 22

**Figure 3-6.** Optimized DCT-IV butterfly ........................................................................ 23

**Figure 3-7.** Filter implemented using transposed direct form and multiplier block. 23

**Figure 3-8.** A two-channel DCT implemented as a butterfly ............................................. 24

**Figure 3-9.** A four-channel DIF DCT-II structure. The shaded areas are two-channel DCTs. .......................................................... 24

**Figure 4-1.** Number of function generators in a Virtex-II FPGA for a DCT block with CSDC multipliers and two, four, eight and 16 channels respectively. Word length 12.......................................................... 25

**Figure 4-2.** The number of function generators and packed CLBs for different word lengths. (a) Two channels (b) Four channels (c) Eight channels. ........................ 26

**Figure 4-3.** Four-channel analysis filters with order 16 and 32........................................ 27

**Figure 4-4.** Differences between DIT and sparse-matrix DCT blocks with word length 12.......................................................... 27

**Figure 4-5.** Difference in number of FG- and H-function generators and packed CLBs between CSDC and graph multipliers. The word length is 12 for all designs... 28

**Figure 4-6.** The left graph shows FG function generators for the 4085 and the right graph shows function generators in the Virtex-II. In both cases, eight-channel analysis filters with order = 32 were generated........................................ 29

**Figure 4-7.** DCT block and filters with 2 channels. Wordlength is 12........................................ 29

**Figure 4-8.** DCT block and filters with (a) 4 channels (b) 8 channels. The word length is 12 in both cases. The filter with order 16 did not work for eight channels.......................................................... 30

**Figure A-1.** Lowpass filter specifications.......................................................... 31

**Figure B-1.** Eight-channel DIT DCT-I. \( C_j = (2\cos(j\pi/8))^{-1} \) ........................................ 33

**Figure B-2.** Eight-channel DIT DCT-II. \( C_j = (2\cos(j\pi/32))^{-1} \) ........................................ 33

**Figure B-3.** Eight-channel DIT DCT-III. \( C_j = (2\cos(j\pi/16))^{-1} \) ........................................ 34

**Figure B-4.** Eight-channel DIT DCT-IV structure. \( C_j = (2\cos(j\pi/32))^{-1} \) and \( S_j = (2\sin(j\pi/32))^{-1} \) ........................................ 34
1 Introduction

In this chapter, a short background is presented, as well as purpose, intended audience and restrictions that have been necessary. Because of time limitations, a thorough comparison was not possible.

1.1 Background
From advanced image recognition systems and compression to high speed AC/DC converters, subband coding plays an integral part in signal processing, and new research is constantly adding to the applications of filter banks. One new area for subband coding is adaptive and statistical signal processing. Other applications include transmultiplexing and equalization.

1.2 Purpose
The initial purpose of this master thesis was to compare implementation complexity between different parts of cosine modulated filter banks. As work progressed, it became evident that there would not be enough time to make a thorough comparison between all filter banks of interest. Instead, focus shifted towards finishing the computer program, called DCTgen, used to generate the filters and DCT (Discrete Cosine Transform) blocks. The goal has been to make a software that lets users add their own components, thus facilitating further comparisons later on.

1.3 Intended Audience
The goal is to make this report available for undergraduate students. Some prerequisites in electrical engineering and transforms make things easier, but should not be necessary.

1.4 Restrictions
Because of time limitations, several restrictions were necessary. There are numerous ways to implement DCT structures in hardware. In this report, only a couple of these have been covered. Some of the others are mentioned briefly, others not at all. As one of the initial goals was to compare size between different parts of the filter bank, only isomorphic mappings have been studied. This entails another restriction – the number of channels. An isomorphic mapping uses considerable area and thus, only a limited number of channels are practically possible.
2 Theory

In this chapter, the theory of filter banks and DCTs is discussed. For a more in-depth coverage, see [1], [2] or [3]. VHDL and FPGAs are also discussed.

2.1 Filter Banks

A filter bank is a collection of filters with a common input, or output, signal. A distinction is made between filter banks dividing a signal into subbands and filter banks merging subbands into one signal. The former is called analysis filter banks and the latter synthesis filter banks (see Figure 2-1). When a signal has been divided into subbands, it is possible to process each subband individually.

![Figure 2-1. Example of a filter bank. It is divided into an analysis filter bank and a synthesis filter bank.](image)

A simple example of an analysis filter bank is an audio system. An hi-fi audio system separates the sound signal into bass and treble and outputs them using two different loudspeakers, i.e. it divides the signal into subbands and treats each subband individually. Each speaker is adjusted to best handle its respective frequency band.

2.1.1 Multirate Systems

Multirate systems operate at several frequencies. Conversion between frequencies is accommodated by decimators and expanders. The former forms an output signal from every \( n \)th input sample. The latter uses the input samples every \( n \)th time, and fills out the rest of the output samples with zeros (see Figure 2-2).
Figure 2-2. Operation of a decimator followed by an expander. The expander expands the signal $y_D(n)$ and fills the gaps with zeros. An interpolation filter is then used to restore the original signal $x(n)$.

Usually decimators and expanders work together with filters. The decimation filter together with a decimator forms a decimator circuit. The filter is usually a lowpass filter to avoid aliasing [1]. The expander is usually followed by an interpolation filter forming an interpolation circuit, used to restore the original signal.

2.1.2 Polyphase Representation

The polyphase representation was a great breakthrough in multirate systems. It made it possible to lower the speed of processors by carrying out each operation at the minimal sample rate. The idea is to separate a signal into several functions $E_i(z^M)$ where $M$ is related to how much the signal is decimated. The transfer function can now be written as

$$H(z) = \sum_{i=0}^{M-1} z^{-i} E_i(z^M)$$  \hspace{1cm} (2.1)

In the special case when $M = 2$ the transfer function is separated into two functions $E_0(z^2)$ and $E_1(z^2)$ containing even and odd numbered coefficients, respectively. The transfer function is thus written

$$H(z) = E_0(z^2) + z^{-1}E_1(z^2)$$  \hspace{1cm} (2.2)

As mentioned earlier, a decimation circuit consists of a decimation filter followed by a decimator. It can now be modified using the polyphase representation and a noble identity [1] as in Figure 2-3.

Figure 2-3. Decimation circuit modified using polyphase representation and noble identities.

In this case, $M = 2$. 
2.2 The Discrete Cosine Transform

The discrete cosine transform consists of a kernel

\[ K_c(m, n) = \cos \left( \frac{\pi mn}{N} \right) \]  \hspace{1cm} (2.3)

The best way to obtain the DCT is to regard the kernel as a matrix \( M \) where

\[ M_{mn} = \cos \left( \frac{\pi mn}{N} \right), \quad m, n = 0, 1, \ldots, N \]  \hspace{1cm} (2.4)

The DCT is obtained as \( X = Mx \), and thus

\[ X(m) = \sum_{n=0}^{N} \cos \left( \frac{\pi mn}{N} \right) x(n) \quad m = 0, 1, \ldots, N \]  \hspace{1cm} (2.5)

The DCT is divided into four types, each having different properties [2]. The properties of the DCT-IV makes it suitable for cosine modulated filter banks, therefore it will be discussed in more detail in the next section. When constructing fast algorithms for the DCT-IV, other types are used as well. There is also another transform, the DST (Discrete Sine Transform) closely related to the DCT, which is also used in fast DCT algorithms. The four types of DCTs are defined by the following equations:

- **DCT-I**
  \[ c_{N+1}^I_{mn} = \sqrt{\frac{2}{N}} \left[ k_m k_n \cos \left( \frac{mn\pi}{N} \right) \right] \quad m, n = 0, 1, \ldots, N \]

- **DCT-II**
  \[ c_{N}^{II}_{mn} = \sqrt{\frac{2}{N}} \left[ k_m \cos \left( \frac{m(2n+1)\pi}{2N} \right) \right] \quad m, n = 0, 1, \ldots, N - 1 \]

- **DCT-III**
  \[ c_{N}^{III}_{mn} = \sqrt{\frac{2}{N}} \left[ k_n \cos \left( \frac{(2m+1)n\pi}{2N} \right) \right] \quad m, n = 0, 1, \ldots, N - 1 \]

- **DCT-IV**
  \[ c_{N}^{IV}_{mn} = \sqrt{\frac{2}{N}} \cos \left( \frac{(2m+1)(2n+1)\pi}{4N} \right) \quad m, n = 0, 1, \ldots, N - 1 \]

where
\[ k_j = 1 \quad \text{if } j \neq 0 \text{ or } N \quad \text{and} \quad k_j = \sqrt{\frac{1}{2}} \quad \text{if } j = 0 \text{ or } N \]

and the corresponding four types of DSTs by:

- **DST-I**
  \[
  [S_{N-1}^I]_{mn} = \frac{2}{N} \left[ \sin \left( \frac{mn\pi}{N} \right) \right] \quad m, n = 1, 2, \ldots, N - 1
  \]

- **DST-II**
  \[
  [S_{N}^II]_{mn} = \sqrt{\frac{2}{N}} k_m \sin \left( \frac{m(2n-1)\pi}{2N} \right) \quad m, n = 1, 2, \ldots, N
  \]

- **DST-III**
  \[
  [S_{N}^III]_{mn} = \sqrt{\frac{2}{N}} k_n \sin \left( \frac{(2m-1)n\pi}{2N} \right) \quad m, n = 1, 2, \ldots, N
  \]

- **DST-IV**
  \[
  [S_{N}^IV]_{mn} = \sqrt{\frac{2}{N}} \sin \left( \frac{(2m+1)(2n+1)\pi}{4N} \right) \quad m, n = 0, 1, \ldots, N - 1
  \]

where

\[ k_j = 1 \quad \text{if } j \neq N \quad \text{and} \quad k_j = \sqrt{\frac{1}{2}} \quad \text{if } j = N \]

For more information about the DCTs and DSTs, see [2].

### 2.2.1 DCT-IV

The DCT-IV can be decomposed into a product of \(2J + 1\) sparse matrices, where \(J = \log_2 N\) and \(N\) is the number of channels [9]:

\[
[C_{N}^A] = [Q_N][V_N(J)][U_N(J-1)][V_N(J-1)]\cdots[U_N(1)][V_N(1)][H_N] \tag{2.6}
\]

There are five different types of matrices in (2.11). The first matrix is a permutation matrix that reverses the odd-numbered components of the vector:
Note that the first component, with index zero, is considered to be even. The last matrix is a permutation matrix as well. It changes the increasing index into a Hadamard index:

$$[Q_N] = \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & \cdot & 0 & 1 & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdot & \cdots & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdots & 1 & 0 \\ 1 & 0 & \cdots & 0 \end{bmatrix} \quad (2.7)$$

$$\begin{align*}
[Q_N] &= \begin{bmatrix} 1 & 0 & \cdots & 0 \\ 0 & \cdot & 0 & 1 & 0 \\ 0 & 0 & 1 & \cdots & 0 \\ \cdot & \cdots & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdots & 1 & 0 \\ 1 & 0 & \cdots & 0 \end{bmatrix} \\
\text{Note that the first component, with index zero, is considered to be even. The last matrix is a permutation matrix as well. It changes the increasing index into a Hadamard index:}
\end{align*}

$$[H_N] = [P_N] \begin{bmatrix} P_{N/2} & \bar{P}_{N/4} \\ \bar{P}_{N/2} & P_{N/4} \end{bmatrix} \cdots$$

$$[P_{N/2}] = \begin{bmatrix} P_4 & \bar{P}_4 \\ \bar{P}_4 & P_4 \end{bmatrix}$$

$$[\bar{P}] = [I][P][\bar{I}] = \begin{bmatrix} \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ \cdot & \cdot & \cdot & \cdot \\ 1 & 1 & \cdot & \cdot \end{bmatrix}, N \geq 4 \text{ and}$$

$$[P_N] = \begin{bmatrix} 1 & 0 & \cdots & \cdots & \cdots & \cdots \\ 0 & \cdot & \cdots & \cdots & \cdot & 1 \\ 0 & 1 & 0 & \cdots & \cdots & \cdot \\ 0 & \cdot & \cdots & 1 & \cdot & 0 \\ \cdot & \cdots & \cdot & \cdots & \cdot & \cdot \\ \cdot & \cdots & \cdot & \cdot & \cdot & \cdot \\ 0 & \cdot & \cdots & 1 & 0 & \cdots \\ 0 & \cdot & 1 & 0 & \cdots & \cdot \\ 0 & \cdot & \cdots & 1 & 0 & \cdots \end{bmatrix} \quad (2.9)$$
The matrices $[U_N(j)]$, $j = 1, 2, \ldots, J - 1$ are block diagonal binary matrices.

$$[U_N(j)] = \frac{1}{\sqrt{2}} \text{block diag} \{B(j), B(j), \ldots, B(j)\} \quad (2.10)$$

where

$$B(j) = \begin{bmatrix} I_{2^j} & I_{2^j} \\ I_{2^j} & -I_{2^j} \end{bmatrix} \quad (2.11)$$

The matrices $[V_M(j)]$, $j = 1, 2, \ldots, J$, are also a block diagonal matrices. The first matrix is formed by

$$[V_N(J)] = \frac{1}{\sqrt{2}} \text{block diag} \{T_{1/4N}, T_{5/4N}, \ldots, T_{(2N-3)/4N}\} \quad (2.12)$$

where

$$T_r = \begin{bmatrix} \cos r\pi & \sin r\pi \\ \sin r\pi & -\cos r\pi \end{bmatrix} \quad (2.13)$$

The matrices $T_r$ and $B$ can also be called butterfly matrices, as they are implemented as butterflies in hardware (see Figure 3-5 and Figure 3-6). The remaining matrices $[V_M(j)]$, $j = 1, 2, \ldots, J - 1$, are formed by

$$[V_M(J)] = \text{block diag} \{I_{2^j}, E(j), I_{2^j}, \ldots, E(j)\} \quad (2.14)$$

where

$$E(j) = \text{block diag} \{T_{1/2^j}, T_{5/2^j}, \ldots, T_{(2^j-3)/2^j}\} \quad (2.15)$$

and $T_r$ are butterfly matrices of the form (2.13). The sparse-matrix decomposition of DCTs is extensively covered in [9].

### 2.2.2 Fast DCTs

Just like the FFT (Fast Fourier Transform), FDCTs (Fast Discrete Cosine Transforms) are not new transforms, but rather algorithms used to optimize the
implementation of DCTs. There are many different approaches, all with their specific advantages and disadvantages. Which one to use depends on the situation and personal preferences.

A complete comparison of cosine modulated filter banks should include cases where the number of channels are not fixed to power-of-twos. One reason is that there are other, potentially more efficient, methods of implementing power-of-two channel filter banks. However, as mentioned earlier, because of time limitations only the power-of-two case has been covered in this report.

2.3 Cosine Modulated Filter Banks

The cosine modulated filter bank consists of a number of filters and the cosine-modulation block (see Figure 2-4). The cosine-modulation block itself consists of permutation matrices, a diagonal matrix and the DCT block (see Figure 2-5).

The reason for grouping the polyphase components as pairs is that they can be optimized if they are implemented as one multiplier block. This has to do with the fact that the same value is multiplied with several coefficients (see Section 3.6). One of the advantages with cosine modulated filter banks is that all analysis filters $H_k(z)$ are obtained from a single prototype filter with real coefficients. Other advantages of cosine modulated systems are explained in [1].
Figure 2-5. Cosine-modulation block. $C$ is the DCT, $I$ and $J$ are permutation matrices and $\Lambda_C$ is a diagonal matrix with elements ±1.

2.4 VHDL

HDLs (Hardware Description Languages) are specialized at describing hardware. VHDL, an abbreviation of VHSIC (Very High Speed Integrated Circuit) Hardware Description Language, is, together with Verilog, one of the most commonly used HDLs. Numerous hardware related properties are not adequately covered in common software programming languages. Some examples are propagation of time and parallelism. Synthesizing code is the process where VHDL code is translated into a structured gate-level circuit that is either put onto an FPGA or silicon.

2.4.1 Signals vs Variables

Variables, common in all programming languages, do not sufficiently describe hardware signals. In VHDL this is handled by introducing an alternative variable, a signal. Signals are better equipped to describe hardware as they also have a time dimension.

2.4.2 Simulation

VHDL code can be run in a simulator to verify its functionality. There are, however, further complications when synthesizing the code. It is possible to write VHDL code that is not synthesizable, or does not behave the way intended. There are several applications for VHDL code that can only be run in software, though. It might be useful to start by implementing a component at a behavioral, or algorithmic, level, to make sure the idea is correct, before implementing more complex and abstract synthesizable code. It is common to implement code at several hierarchical levels and compare them with each other to minimize the risk of errors. Behavioral code is also useful in testbenches (see Section 2.4.4).
Figure 2-6. VHDL simulation time queue.

The simulator consists of a time queue where each time element is associated with one or several events. The simulation runs until there are no more entries in the time queue.

In VHDL, all events outside sequential processes occur simultaneously. When simulating code, it is of course nice to get the same result irrespective of the simulator, and of how the code is written. Consider the code below:

<table>
<thead>
<tr>
<th>Case A</th>
<th>Case B</th>
</tr>
</thead>
<tbody>
<tr>
<td>b &lt;= a</td>
<td>a &lt;= '0'</td>
</tr>
<tr>
<td>a &lt;= '0'</td>
<td>b &lt;= a</td>
</tr>
</tbody>
</table>

In case A, a sequential programming language would assign $b$ the value $a$ has before it is changed, while in case B, zero would be assigned to $b$. In VHDL, however, as the two statements occur simultaneously, different simulators could possibly treat case A and B differently. The solution is delta delays, a key concept in VHDL. A delta delay is a time period longer than zero, but shorter than any time period possible to specify by the user. Signals are assigned their new values after the delta delay while the old value always is used to assign values to other signals (see Figure 2-6).

2.4.3 Components

One strength of VHDL is its component based structure. Hardware is often described as black boxes, i.e. with a number of in- and outputs and a description of the system. The inside of a black box, however, is not known. Most of the time, this is sufficient for using the system as a component in your own design. In VHDL, this is accomplished by entities and architectures. Entities describe a component by specifying in- and outputs. It is then possible to connect one or several architectures, e.g. with different levels of abstraction, to the entity. The architecture contains the implementation of a component.
When using a system as a sub component in a larger design, it is declared using the keyword component and in case several architectures have been constructed, possibly with an indication to which one of these to use.

Entities and architectures facilitate code reusability and a component based design, thus simplifying verification. DCTgen relies heavily on this features of VHDL to accomplish configurable designs.

2.4.4 Testbenches

Testbenches are used when verifying a system. They are not really a VHDL feature, they exist both in other HDLs and as hardware implementations.

Testbenches provide the DUT (Design Under Test) with input patterns and records the output, hence the testbench is usually divided into two parts, an input testbench and an output testbench (see Figure 2-7).

![Figure 2-7. General testbench structure.](image)

Input patterns are usually generated using pattern generators that also create a correct output pattern that can be compared with the one provided by the DUT. A distinction is made between different test methods as well: A complete test pattern contains all possible input combinations, in a random pattern a partial input set is chosen randomly and corner testing involves testing of special cases, e.g. addition by the largest or smallest possible value.

2.5 FPGAs

Manufacturing chips are expensive and only economically justified on an industrial scale. A popular alternative is using FPGAs (Field Programmable Gate Arrays). FPGAs consist of thousands of CLBs (Complex Logic Blocks). These are designed in different ways, depending on the FPGA model. Two different FPGAs have been considered when synthesizing code in this report. One from the Xilinx 4085XL series and one from the Xilinx Virtex-II series. The 4085XL CLB consists of FG- and H-function generators. These take four and three input signals, respectively, and produce one output signal (see Table 2-1).
Table 2-1. Truth table definitions of FG- and H-function generators.

The Virtex-II FPGA consists of slices. One slice, in turn, consists of two 4-input function generators (as the 4085 FG-function generators), arithmetic logic gates, carry logic etc (see Figure 2-8). More detailed information about how FPGAs are constructed can be found in [8] and specific information about the Virtex-II family in [10].

There are numerous tools to aid the conversion from HDLs to hardware. These convert code to structural gate-level circuits using available primitives. The user can choose whether to optimize for area, power consumption, speed etc.
3 Implementation

This chapter covers implementation of the cosine modulated filter banks. Different implementation choices are discussed and components of the system are described.

There are several choices to be made when implementing a system. Power consumption, performance constraints and size are some of the factors that influence the final design specification. The primary goal of DCTgen is to make many of the choices configurable.

DCTgen takes a text file as input and generates synthesizable, gate-level VHDL code based on the given specification (see the technical reference for details). The text file does not only control the design specification. Other information, e.g. whether to use a log file, how to comment the code and where to place the generated files, is also configurable. Figure 3-1 shows an example input file that generates a DCT block with word length 16, creating a log file of the generation process and using the file extension .vhdl for the generated files.

![Program]
Extension=.vhdl
LogFile=Yes

![Design]
Top=Dct
Dct=DITDCTIV
WordLength=16

Figure 3-1. Example input file generating a DCT block with word length 16.

When the VHDL code has been generated, it is synthesized using a special tool called *Leonardo*. This tool converts VHDL code into a structured gate-level design that is written to the FPGA. Fortunately, the tool produces extensive statistics about the design, e.g. how many CLBs (Complex Logic Blocks) it uses, thus it is not necessary to physically copy the design to the FPGA. This also meant it was possible to synthesize the design for different types of FPGAs, even if they were not available in the lab.

3.1 Arithmetics

Several number representations are used in digital hardware. DCTgen generally works with two's-complement representation, one of the easiest and most straight-forward number systems. In fixed multipliers, however, another representation called CSDC (Canonic Signed Digit Code) is used. These, and other, number systems are covered extensively in [3].
To understand the arithmetics it is important to understand how fractional numbers are represented in the binary number system. It should not be too difficult, though, as it is analogous to the decimal case. It is best illustrated with an example.

The decimal number \((0.625)_{10}\) consists of three decimals; 6, 2 and 5. These represent \(6 \cdot 10^{-1}\), \(2 \cdot 10^{-2}\) and \(5 \cdot 10^{-3}\) respectively. The same number binary would be \((0.101)_2 = 1 \cdot 2^{-1} + 0 \cdot 2^{-2} + 1 \cdot 2^{-3} = 0.5 + 0.125 = (0.625)_{10}\). Each digit represents the value \(b^x\) where \(x\) is the position and \(b\) the base, i.e. ten in the decimal number system and two in the binary number system.

### 3.1.1 Two's-Complement

A positive number in two's-complement looks just like an ordinary binary number, e.g. \((0.25)_{10}\) is represented by \((0.010000)_2\). To negate a number, it is first inverted and then 1 is added at the LSB (Least Significant Bit) position. Thus, \((-0.25)_{10}\) is represented by \((1.110000)_2\). The range will be \([-1, 1]\), i.e. \(-1\) can be represented but not \(+1\). This has to do with the fact that the MSB is a \textit{sign} bit and tells whether or not the number is negative. The largest possible is thus \((0.11111111)_2\) which is smaller than 1. However, the smallest possible number is \((1.00000000)_2 = (-1.0)_{10}\).

### 3.1.2 Fixed-Point Fractional

In a fixed-point fractional number system, a predefined part of the total number of bits represent integers, and the rest fractions, hence fixed-point. Considering a fixed number of total bits, the position where the fractional point is placed determines the number range covered by the system (see Table 3-1).

<table>
<thead>
<tr>
<th>Fractional Point Position</th>
<th>Numbers represented in an unsigned system</th>
<th>Numbers represented in a signed system</th>
</tr>
</thead>
<tbody>
<tr>
<td>0.000</td>
<td>0.0.125, 0.25, 0.375, 0.625, 0.75, 0.875, 1.0, 1.25, 1.5, 1.625, 1.75, 1.875</td>
<td>-1.0, -0.875, -0.75, -0.625, -0.5, -0.375, -0.25, -0.125, 0.125, 0.25, 0.375, 0.5, 0.625, 0.75, 0.875</td>
</tr>
<tr>
<td>00.00</td>
<td>0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75, 2.0, 2.25, 2.5, 2.75, 3.0, 3.25, 3.5, 3.75</td>
<td>-2.0, -1.75, -1.5, -1.25, -1.0, -0.75, -0.5, -0.25, 0.25, 0.5, 0.75, 1.0, 1.25, 1.5, 1.75</td>
</tr>
<tr>
<td>000.0</td>
<td>0.0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5, 4.0, 4.5, 5.0, 5.5, 6.0, 6.5, 7.0, 7.5</td>
<td>-4.0, -3.5, -3.0, -2.5, -2.0, -1.5, -1.0, -0.5, 0, 0.5, 1.0, 1.5, 2.0, 2.5, 3.0, 3.5</td>
</tr>
</tbody>
</table>

Table 3-1. Numbers represented for some signed and unsigned fixed-point numbers systems using four bits.

Depending on the application, different placements of the fractional point is preferred. In our case, numbers are considered to be in the range \([-1, 1]\] which means the fractional point is placed immediate to the right of the sign bit.

### 3.1.3 CSDC

CSDC is a number system where each position is assigned either +1, −1 or 0. Also, there are never two positions next to each other that contain nonzero
values. The goal of CSDC is to minimize the number of ones. As an example, consider \((0.875)_{10}\) which in two's-complement is represented by \((0.1110000)_{2C}\) and in CSDC by \((+.00-)_\text{CSDC}\), i.e. \((+1) + (-0.0125) = 1 - 0.0125\). When implementing a fixed multiplier, you want to minimize the number of required adders and subtractors, which is equal to minimizing the number of ones in the word, that is why CSDC is well suitable for implementing fixed multipliers. If the multiplier was to be implemented using two's-complement, two adders would be required. In CSDC, however, only one subtractor is needed. One illustrative way to present constants in CSDC graphs. Figure 3-2 shows the graph for the decimal number 0.875.

![Figure 3-2. 0.875 in CSDC format.](image)

The numbers on the vertices are what the number at the source node is multiplied by (alternatively, they may show the number of shifts). At the destination node, the two vertices are added or subtracted (indicated by a minus sign at the edge). In the example above, \(1 \cdot 1 - 1 \cdot 2^{-3} = 1 - 0.125 = 0.875\). This means the multiplier can be implemented using only one subtractor, instead of two adders when using two's-complement representation. Of course, the potential gain increases with increased word length and more complex constants.

### 3.1.4 Graph Multipliers

An enhancement of fixed multipliers implemented using CSDC is the graph multiplier. The basic idea is the same, but graph multipliers also take advantage of partial sums. As an example, \((45)_{10}\) is represented by \((00+0++0+)_\text{CSDC}\) which is \(32+8+4+1\), requiring three additions. However, \(9 \cdot 5\) also equals 45 and the partial sum \((9)_{10}\) equals \((0000+00+)\_\text{CSDC}\) and \((5)_{10}\) equals \((00000+0+)\_\text{CSDC}\). This means only two adders are needed to represent \((45)_{10}\) (see Figure 3-3).

![Figure 3-3. (a) CSDC representation of multiplication by 45 (b) Graph multiplier representation of multiplication by 45.](image)

Graph multipliers can be up to 20-30% smaller than CSDC multipliers [6].
3.2 Error Sources

One of the greatest challenges when designing digital hardware is coping with the numerous error sources. Handling output noise is often a trade-off between a complex implementation and larger errors.

3.2.1 Word Length

Output noise decreases with increasing word length. It is up to the designer to decide the necessary word length, depending on system specifications.

3.2.2 Number Representation

When using a fractional fixed-point number system, errors will occur because of overflow and quantization. An alternative would be to use floating point arithmetics. The advantage of a fixed-point scheme, however, is its superior simplicity and speed. Some of the errors may be counter measured by increasing the word length.

3.2.3 Scaling

When using fixed point arithmetics, there is always a risk of overflow. Consider a fixed point number system, using two's-complement arithmetics, where only the sign bit represents integers. An addition between 0.75 \((0.110)_2\) and 0.50 \((0.100)_2\) becomes \(-0.75\) \((1.001)_2\) instead of 1.25, which cannot be represented using this fixed point number system. There are two approaches of dealing with overflow: Saturation and scaling.

When using saturation, extra bits, called guard bits, are added to arithmetic operations. A saturation circuit is constructed that compares MSBs in the result (including the guard bits) to determine whether an overflow has occurred. In case it has, the largest, or smallest in case of an underflow, possible value is chosen instead.

Scaling, or safe scaling, does not require extra hardware or longer word lengths. The idea is that when both inputs to an addition or subtraction is in the region \([-0.5, -0.5]\) no overflow will occur, as the result is going to be within the valid region, i.e. \([-1, 1]\). The easiest way to implement this is shifting the outputs one step. Of course, the inputs to the DCT-block have to be scaled in the same manner. Scaling causes additional round-off noise, as the LSB is lost. To make sure multipliers are not overflown, either, it is preferable to have a design where all multiplications are by numbers in the range \([-1, 1]\). Note that this is not possible in some designs.

When scaling, it is important to remember that the result has to be scaled back. Thus, as a last measure in the design, the output signals from the DCT are
shifted to the left. This scaling may be considered to take place outside the system, but it is, however, important not to forget about it.

3.2.4 Coefficient Quantization
When using fixed point fractional arithmetic, there are several approaches to rounding numbers. As there is only a certain number of bits available (the word length), numbers are going to have to be either rounded, magnitude-rounded or truncated. Truncation is by far the easiest scheme to implement in hardware and the noise produced is often acceptable. See Figure 3-4 for an example of how quantization arises.

3.3 Adders
Depending on performance constraints, different adder implementations are preferred. There is a trade-off between size and speed and it is up to the designer to decide what adder to use.

Ripple-carry adders are small, but also the slowest. In this report, speed has not been a primary issue, thus ripple-carry adders are the best choice. It should be mentioned, however, that isomorphic mappings like the ones considered, are mostly used in high-performance applications where cost is not an issue, and hence, a faster adder might be a more appropriate choice, even if they generate larger designs.

3.4 Multipliers
A general multiplier is expensive to implement, as it requires a large area. There are several schemes to minimize the cost of multipliers. In the case of DCTs, all multiplications are by fixed coefficients. This enables considerable simplifications of the multipliers.

Multiplications by $2^x$ where $x = \pm 1, \pm 2, \pm 3, \ldots$, can be implemented with virtually no cost, as they can be hardwired. Figure 3-4 shows a division by two implemented through shifting. It also illustrates the quantization effect. The MSB is copied, and the LSB is lost. If the MSB is lost, overflow occurs. This can be avoided by proper scaling, described in Section 3.2.3.
3.4.1 Graph Multipliers
Graph multipliers were described in Section 3.1.4. The graphs are found using an algorithm called MAG (Minimum Adder Graph) [6]. As the name suggests, it finds the optimal graph for each coefficient.

One of the toughest challenges when generating the graph multipliers was handling negative numbers efficiently. The implemented functions do not always choose the optimal solution, but it is not a serious problem because of two things: First, in the DCT-block, there are very few negative numbers. In butterflies, the negation is carried out in the subsequent addition (by implementing a subtractor instead). This won’t work, of course, if both edges need to be negated. Second, when implementing filters through multiplier blocks, the negation can be moved to the adders following the taps from the multiplier block. By doing this, only the first output from the multiplier block may need to be negated.

3.5 Butterflies
A common structure that encapsulates one pair of additions and multiplications (a 2×2 matrix) is the butterfly. It is called so because of its shape that somewhat resembles a butterfly.

Figure 3-4. Signed division by two implemented using shifting.

![Figure 3-4. Signed division by two implemented using shifting.](image)

Figure 3-5. Example of a scaled butterfly. α is an arbitrary constant.
When implementing the DCT-IV, it is possible to optimize the butterflies made up by (2.13) and hence reducing the number of multiplications to three. The cost is an extra adder, but these are much easier and cheaper to implement.

\[
\begin{align*}
\cos j\pi + \sin j\pi \\
u \\
sin j\pi \\
+ \\
v \\
- \cos j\pi + \sin j\pi \\
+ \\
x \\
y
\end{align*}
\]

*Figure 3-6. Optimized DCT-IV butterfly.*

### 3.6 Multiplier Blocks
Implementing the filter using transposed direct form (see Figure 3-7) makes it possible to minimize size by using graph multipliers and the RAG (Reduced Adder Graph) algorithm. In transposed direct form, all multiplications of a sample are carried out simultaneously, enabling further optimization of the multipliers. The idea is to represent the single-coefficient multipliers as a multiplier block and reusing partial sums of the multiplications.

*Figure 3-7. Filter implemented using transposed direct form and multiplier block.*

### 3.7 DCT Block
The DCT-IV generates a more complex design than the DCT-II, commonly used for image compression, which is one of the reasons why it is so much easier to find documentation about the DCT-II. However, the DCT-IV possesses properties useful in filter banks.

DCTgen is capable of generating DIF (Decimation-In-Frequency) and DIT (Decimation-In-Time) DCTs (see Appendix B), as well as a sparse-matrix DCT-IV. There are other approaches to implementing DCTs, e.g. lifting schemes, but they are not considered in this report because of time limitations. The DCT block is a matrix multiplication and hence implemented using adders and multipliers. As all multiplications are by fixed coefficients, these may be
optimized using fixed multipliers built of shifts and adders, as described in the sections above.

The advantage of both the DIF and DIT algorithms are that they are recursive. An eight-channel DIF DCT-IV can be built using two four-channel DIF DCT-IVs. They, in turn, are composed of two two-channel DCTs each. And a two-channel DCT is implemented as a butterfly, i.e. a $2 \times 2$ matrix (see Figure 3-8 and Figure 3-9). The signal-flow graphs for other DIT and DIF DCTs can be found in Appendix B.

![Figure 3-8. A two-channel DCT implemented as a butterfly.](image)

![Figure 3-9. A four-channel DIF DCT-II structure. The shaded areas are two-channel DCTs.](image)
4 Results and Conclusions

In this chapter, the results of the synthesizing is presented and discussed. The complete results can be found as tables in Appendix C.

The designs were synthesized with different word lengths, number of channels and multiplier types. They were also synthesized using two different FPGAs: One in the Xilinx 4085XL series and one in the Xilinx Virtex-II series, abbreviated 4085 and Virtex-II, respectively, from now on. The synthesize tool used was Leonardo from Mentor Graphics. As it takes considerate time to synthesize one design, only some of the possible configurations were synthesized – and with the fastest possible synthesize time. This means that smaller designs might be achieved, but for the sake of comparison, it is not that important.

On the 4085, FG function generators, H function generators and packed CLBs have been compared, on the Virtex-II function generators and CLB slices were compared. In the tables in Appendix C, there are also other parameters listed, e.g. IOs (Input/Outputs) and speed. IOs are pretty straight-forward, and not really interesting to compare. Speed may, of course, be interesting, but not when the design is synthesized for size.

Even though verification was done for some designs, some designs generated strange results or reported erroneous VHDL code. This is, of course, because of errors in the computer program generating the code. The following sections discuss different comparisons, and some conclusions that can be drawn.

4.1 Number of Channels

The number of channels naturally affect size. The most interesting aspect when considering channels, though (and this has not been investigated in this report), is the combination between channels and word length, and the resulting errors.

![Figure 4-1. Number of function generators in a Virtex-II FPGA for a DCT block with CSDC multipliers and two, four, eight and 16 channels respectively. Word length 12.](image)
As the design is scaled after each stage (after each arithmetic operation), the error increases with the number of steps. As a result, a design with more channels but with the same word length generates larger errors. A fair comparison on how channels affect size should consider this, and compensate with longer word lengths as well, further increasing the size of the design.

4.2 Word Length

As suggested in Figure 4-2 and Figure 4-3, size is directly proportional to word length. This is what one would expect. It might seem a bit curious, though, that the increase between 8 and 10 bits is greater than between 10 and 12. It could possibly be explained by the multipliers. A very short word length makes the multipliers very simple as they will use few bit adders. Increasing the word length does not necessarily mean more complex multipliers, depending on the coefficients.

![Figure 4-2](image.png)

*Figure 4-2. The number of function generators and packed CLBs for different word lengths. (a) Two channels (b) Four channels (c) Eight channels.*
4.3 Type of DCT

There are two main reasons why fast algorithms are used: They are, obviously, faster, but they also generate smaller designs. In Figure 4-4, the DIT algorithm is compared with a sparse-matrix implementation. The DIT algorithm is advantageous, but not by much. In the two-channel case, the designs are actually exactly the same, as a $2 \times 2$ matrix is implemented as a butterfly, independent of the algorithm used. The gain is going to be even greater with more channels.

Figure 4-3. Four-channel analysis filters with order 16 and 32.

Figure 4-4. Differences between DIT and sparse-matrix DCT blocks with word length 12.
4.4 Choice of Multipliers

The main reason for implementing graph multipliers, which is much more difficult than implementing CSDC multipliers, was that they were expected to generate a smaller design. The result when synthesizing, however, was the opposite (see Figure 4-5).

This might depend on several things, but the most probable is that they have actually been implemented differently – maybe more bits have been considered in the graph multipliers. This, of course, was not the intention. Another possible reason is how the synthesize tool has dealt with the designs. It performs many optimizations and might be able to optimize the CSDC multipliers better. Graph multipliers, however, generate faster designs.

![Figure 4-5. Difference in number of FG- and H-function generators and packed CLBs between CSDC and graph multipliers. The word length is 12 for all designs.](image)

Something interesting is the results obtained when comparing CSDC and graph multipliers for filters, implemented as separate multiplier blocks (see Figure 4-6). When synthesizing on the 4085, the graph multipliers generate larger designs but when implementing on the Virtex-II, the CSDC multipliers generate the larger designs. It seems like the choice of FPGA is very important.
4.5 Comparison between DCTs and filters

The initial goal of this master thesis was to compare the DCT block with the filters in a cosine modulated analysis filter bank. Figure 4-8 shows the CLBs for a DIT DCT-IV and filters with order 16 and 32. It is evident that the filters take up much more space than the DCT, and also that the order of the filter is of great importance. However, as the number of channels increase the DCT block will grow larger while the filter part won't (assuming the order of the filter does not change).

Using RAG multiplier blocks instead of, as in Figure 4-7 and Figure 4-8, a multiplier block with separate multipliers, should decrease the size of the filters and also make a higher order filter take up less space, as parts of the multipliers are reused.

Figure 4-7. DCT block and filters with 2 channels. Wordlength is 12.
Figure 4-8. DCT block and filters with (a) 4 channels (b) 8 channels. The word length is 12 in both cases. The filter with order 16 did not work for eight channels.

### 4.6 Future Work

As mentioned earlier, a thorough comparison has not been possible. There are several things that could be examined further. Here are some examples:

- **Non power-of-two channels.** It would be interesting to compare a DCT implementation with power-of-two channels with a DCT block with an arbitrary number of channels.

- **Other DCT algorithms.** There are many different DCT algorithms and it would be interesting to investigate, and compare, different schemes, e.g. lifting schemes.

- **Correct bugs.** Of course, as it is evident some bugs have managed to avoid detection, these should be corrected. Also, the MAG and RAG-n algorithms should be examined further.

- **Implement the whole design.** It might be interesting to synthesize the whole design to see if the sum is greater than the parts and maybe see how many channels can be accomplished on different FPGAs.
A Digital Filters

This appendix gives a brief review of digital filters. A detailed description can be found in [3].

Filters are divided into two main categories: FIR (Finite-length Impulse Response) and IIR (Infinite-length Impulse Response) filters. IIR filters may provide smaller designs, using less memory and fewer algorithmic operations, but they are difficult to design. Instead, FIR filters are often preferred, as they are always stable. The transfer function of a FIR filter is

\[ H(z) = \sum_{n=0}^{N-1} h(n) z^{-n} \]  

where \( N - 1 \) is the order of the filter. Linear-phased FIR filters are designed using either window techniques or numeric optimization techniques. The former is simple but usually generates higher order filters and is thus not preferred. Optimization techniques are usually implemented as a computer program. A filter is divided into three regions called passband, transition band and stopband respectively (see Figure A-1).

![Figure A-1. Lowpass filter specifications.](image)

Filters can be implemented in different ways. Some of the most common are direct form and transposed direct form, but there are other forms as well, for example linear-phase structures and structures adapted to complementary FIR structures [3]. One advantage of the transposed direct form, at least in isomorphic implementations, is that it is possible to optimize the multiplier block considerably (see Figure 3-7).
B  DCT Signal Flow Graphs

In this appendix, signal flow graphs for eight-channel DIT DCTs are listed. Signal-flow graphs for DIT DSTs can be found in [4] and signal-flow graphs for DIF DCTs and DIF DSTs in [5]. Note that there is an error in [4] where $[C^{II}]$ and $[C^{III}]$ has been interchanged.

**Figure B-1.** Eight-channel DIT DCT-I. $C_j = (2\cos(j\pi/8))^{-1}$.

**Figure B-2.** Eight-channel DIT DCT-II. $C_j = (2\cos(j\pi/32))^{-1}$. 
Figure B-3. Eight-channel DIT DCT-III. $C_j = (2\cos (j\pi/16))^{-1}$.

Figure B-4. Eight-channel DIT DCT-IV structure. $C_j = (2\cos(j\pi/32))^{-1}$ and $S_j = (2\sin(j\pi/32))^{-1}$. 
Here the results from the synthesize runs are presented in tables and as graphs.

### DCT Properties

<table>
<thead>
<tr>
<th>Channels</th>
<th>Wordlength</th>
<th>Mult. Type</th>
<th>Flip-Flops and IOs</th>
<th>4085XLBG560</th>
<th>Virtex-II 2V8000ff1517</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FG (6272)</td>
<td>H (3136)</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>c</td>
<td>16 17 16</td>
<td>90 24 57 16</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>c</td>
<td>20 21 20</td>
<td>116 35 75 18</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>10</td>
<td>g</td>
<td>20 21 20</td>
<td>139 42 93 11</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>c</td>
<td>24 25 24</td>
<td>176 60 121 13</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>g</td>
<td>24 25 24</td>
<td>193 62 129 10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>c</td>
<td>56 33 32</td>
<td>211 71 143 11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>g</td>
<td>56 33 32</td>
<td>267 78 170 10</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>c</td>
<td>72 41 40</td>
<td>319 99 204 11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>10</td>
<td>g</td>
<td>72 41 40</td>
<td>429 128 281 9</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>c</td>
<td>88 49 48</td>
<td>494 167 328 11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>g</td>
<td>88 49 48</td>
<td>564 175 372 12</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>c</td>
<td>131 65 64</td>
<td>707 197 444 7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>8</td>
<td>g</td>
<td>132 65 64</td>
<td>819 248 519 9</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>c</td>
<td>174 81 80</td>
<td>1108 343 729 6</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>10</td>
<td>g</td>
<td>175 81 80</td>
<td>1237 417 825 7</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>c</td>
<td>219 97 96</td>
<td>1469 469 969 8</td>
<td></td>
</tr>
<tr>
<td>8</td>
<td>12</td>
<td>g</td>
<td>220 97 96</td>
<td>1549 526 1027 6</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>c</td>
<td>589 193 192</td>
<td>3534 1153 2321 4</td>
<td></td>
</tr>
<tr>
<td>16</td>
<td>12</td>
<td>g</td>
<td>596 193 192</td>
<td>3734 1268 2472 4</td>
<td></td>
</tr>
</tbody>
</table>

### Sparse-Matrix DCT-IV

<table>
<thead>
<tr>
<th>Channels</th>
<th>Wordlength</th>
<th>Mult. Type</th>
<th>Flip-Flops and IOs</th>
<th>4085XLBG560</th>
<th>Virtex-II 2V8000ff1517</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td>FG (6272)</td>
<td>H (3136)</td>
</tr>
<tr>
<td>2</td>
<td>8</td>
<td>g</td>
<td>16 17 16</td>
<td>116 35 75 18</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>12</td>
<td>g</td>
<td>24 25 24</td>
<td>211 71 143 11</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>8</td>
<td>g</td>
<td>64 31 32</td>
<td>338 142 235 13</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>12</td>
<td>g</td>
<td>90 47 48</td>
<td>684 236 456 10</td>
<td></td>
</tr>
<tr>
<td>Channels</td>
<td>Order</td>
<td>Wordlength</td>
<td>Mult. Type</td>
<td>DFFs</td>
<td>Inputs</td>
</tr>
<tr>
<td>----------</td>
<td>-------</td>
<td>------------</td>
<td>------------</td>
<td>------</td>
<td>--------</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>8</td>
<td>g</td>
<td>109</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>10</td>
<td>c</td>
<td>135</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>10</td>
<td>g</td>
<td>135</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>12</td>
<td>c</td>
<td>162</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>16</td>
<td>12</td>
<td>g</td>
<td>163</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>8</td>
<td>g</td>
<td>240</td>
<td>9</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>10</td>
<td>c</td>
<td>300</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>10</td>
<td>g</td>
<td>300</td>
<td>11</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>12</td>
<td>c</td>
<td>360</td>
<td>13</td>
</tr>
<tr>
<td>2</td>
<td>32</td>
<td>12</td>
<td>g</td>
<td>360</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>8</td>
<td>g</td>
<td>91</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>10</td>
<td>c</td>
<td>112</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>10</td>
<td>g</td>
<td>114</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>12</td>
<td>c</td>
<td>135</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>16</td>
<td>12</td>
<td>g</td>
<td>139</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>8</td>
<td>g</td>
<td>224</td>
<td>9</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>10</td>
<td>c</td>
<td>280</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>10</td>
<td>g</td>
<td>280</td>
<td>11</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>12</td>
<td>c</td>
<td>336</td>
<td>13</td>
</tr>
<tr>
<td>4</td>
<td>32</td>
<td>12</td>
<td>g</td>
<td>336</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>8</td>
<td>g</td>
<td>189</td>
<td>9</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>10</td>
<td>c</td>
<td>234</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>10</td>
<td>g</td>
<td>238</td>
<td>11</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>12</td>
<td>c</td>
<td>282</td>
<td>13</td>
</tr>
<tr>
<td>8</td>
<td>32</td>
<td>12</td>
<td>g</td>
<td>287</td>
<td>13</td>
</tr>
</tbody>
</table>

**RAG-n Multiplier Blocks**

| 4       | 16    | 12 | -       | 144  | 13     | 96      | 1129      | 578      | 754         | 11 | 1671             | 836                | 44 |
| 8       | 32    | 12 | -       | 287  | 13     | 192     | 2476      | 839      | 1662        | 9  | 4157             | 2079               | 41 |
| 8       | 32    | 8  | -       | 191  | 9      | 128     | 1105      | 353      | 733         | 14 | 1860             | 930                | 69 |
References


TECHNICAL REFERENCE
Table of Content

1  OVERVIEW AND STRUCTURE .............................................................9

1.1 The Program ......................................................................................... 9
  1.1.1 Generated VHDL Code ...................................................................... 9
  1.1.2 Files and Folders .............................................................................. 9

2  COMPONENTS .......................................................................................11
  2.1.1 Creating New Components .............................................................. 11

3  GLOBAL FUNCTIONS ............................................................................13

  3.1 Report .................................................................................................... 13
  3.2 double2twos .......................................................................................... 13
  3.3 double2csdc .......................................................................................... 13
  3.4 int2twos ................................................................................................ 13
  3.5 toString .................................................................................................. 13
  3.6 double2string ........................................................................................ 14
  3.7 trunc ....................................................................................................... 14
  3.8 rest ......................................................................................................... 14
  3.9 drawChar ............................................................................................... 14
  3.10 getArgs ............................................................................................... 14
  3.11 readGenFile ........................................................................................ 14
  3.12 toLower ............................................................................................... 14
  3.13 int2csdc .............................................................................................. 15
  3.14 mark ..................................................................................................... 15
  3.15 reduceToFund ..................................................................................... 15

4  CONSTANTS AND DEFINITIONS ......................................................17
4.1 Constants .................................................................17
4.2 Enumerations .........................................................18
4.3 Type Definitions ....................................................18
4.4 Structures .............................................................18

5   VHDL SUPPORT FUNCTIONS .........................19
5.1 Scope .................................................................20
5.2 vhdl.................................................................20
5.3 vhdlLibraries ......................................................20
5.4 vhdlBegin ..........................................................21
5.5 vhdlSignal ..........................................................21
5.6 vhdlArch and vhdlEndArch .........................21
5.7 vhdlEntity and vhdlEndEntity ......................21
5.8 vhdlIO ...............................................................21
5.9 vhdlPortMap and vhdlEndPortMap ..............21
5.10 vhdlMap ..........................................................22
5.11 vhdlScaleDown and vhdlScaleUp ..............23
5.12 vhdlHeader and vhdlEndHeader ..............23
5.13 vhdlComment ...................................................23

6   GENERATION FILES .............................................25
6.1 Program Section ...............................................25
   6.1.1 Log File .....................................................25
   6.1.2 Batch Files .................................................25
   6.1.3 Makefile ....................................................25
   6.1.4 Extension ..................................................25
   6.1.5 Feedback ..................................................26
6.2 Design Section ...................................................26
   6.2.1 Round Method ..........................................26
6.2.2 Constants ................................................................. 26
6.2.3 Word Length .......................................................... 26
6.2.4 Comments .............................................................. 27
6.2.5 Top Design ............................................................ 27
6.2.6 Name ................................................................. 27
6.2.7 File Path .............................................................. 27
6.2.8 Order, Outputs and Coefficients ............................... 27
6.2.9 Channels .............................................................. 28
6.2.10 Pipelines .............................................................. 28
6.2.11 Components ........................................................ 28

6.3 MAG Section ............................................................. 28
6.3.1 Use Files .............................................................. 28
6.3.2 Secondary ............................................................ 28

6.4 Coefficient section ..................................................... 29

6.5 Example Files .......................................................... 29

APPENDIX A - CLASSES ............................................. 31
CDesign ........................................................................ 31
CComponent .................................................................. 33
CMag ............................................................................ 35
CSkeleton ...................................................................... 38
CSubSkel ...................................................................... 38
CDct ............................................................................. 39

APPENDIX B – EXISTING COMPONENTS ......................... 41
B.1 Basic Blocks ........................................................... 41
  B.1.1 Bit Adders ......................................................... 41
  B.1.2 D Flip-Flop ......................................................... 41

B.2 Adders ................................................................. 41
  B.2.1 Ripple-Carry Adder ............................................. 41

B.3 Butterflies ............................................................... 41
  B.3.1 General Butterfly ............................................... 42
  B.3.2 Sande-Tukey Butterfly ........................................ 42
  B.3.3 Optimized DCT-IV Butterfly ............................... 42
  B.3.4 Multiply-First Butterfly ....................................... 42
B.4 Fixed Multipliers.................................................................42
  B.4.1 CSDC Multiplier .........................................................42
  B.4.2 Graph Multipliers......................................................42
  B.4.3 Negate .................................................................43

B.5 DCT Blocks .................................................................43

B.6 Registers .................................................................43

B.7 Multiplier Blocks .......................................................43

B.8 Filters .................................................................43

B.9 Analysis Filter Bank ...................................................43
1 Overview and Structure

This chapter describes the purpose and general functionality of the program, as well as files and folders. The program was designed with further development in mind, thus it relies heavily on a component based, object oriented framework.

1.1 The Program

This program was primarily developed to facilitate generation of DCT-modulated filter banks. However, it was designed with expansion in mind. The object oriented structure and elaborate framework should make it fairly straightforward to add new components. VHDL support functions simplify writing code to generate VHDL structures (see Chapter 3). The idea of the component based architecture is to enable component development without knowledge of how the sub blocks are implemented, i.e. similar to the entity based component structure of VHDL programs.

1.1.1 Generated VHDL Code

The VHDL code is generated at the gate level, and uses components (that’s why the base class is called CComponent, as well) and port maps extensively. One particular class, CBasicBlock, contains the lowest-level blocks, e.g. full adders and half adders. The reason for generating gate level code is the increased control of how the design is synthesized, thus allowing more detailed comparisons.

1.1.2 Files and Folders

The main functionality of the program is contained in a number of files, described here. The other files are components, derived from CComponent. A couple of template files facilitate creation of new component classes. These are found in the folder named Templates.

VHDLgenerator

The header file contains constants, global function declarations, structures, enumerations and the CProperties class definitions. The source files contain the main() function and definition of global functions and functionality to interpret command line options and text files.

CProperties

This is the file where definitions for the CProperties class are stored.
CDesign
This is the main class of the program, instantiated once, globally, to be available from all classes and functions. The CDesign class encapsulates design related functionality. It keeps track of default settings, the design file path, and it is used to add new components to the design. The components call CDesign to find out default values etc.

CComponent
This abstract class is the base class for all components. It contains information all components have in common, e.g. id and name. It also keeps track of the number of components. There are also several functions to simplify writing code to generate VHDL statements, from now on called VHDL support functions (see Chapter 3).

CMag
The CMag files encapsulate the MAG (Minimum Adder Graph) and RAG (Reduced Adder Graph) algorithms. The CMag class contain functions for finding, storing, drawing and generating code for MAGs and RAG multiplier blocks.

CTable
This is a template file, containing the CTable template. CTable is used in the MAG and RAG classes.
2 Components

The program is based on components used to build the system. In this chapter, the component classes and creation of new components are discussed.

Each component is encapsulated in its own class, derived from CComponent. Different implementations are derived from the component classes (see Figure 2-1). The former will from here on be called the category class and the latter the implementation class. All implementations of a category can be described with the same properties, e.g. all adders have two inputs (A and B) and two outputs (SUM and COUT). This means the entities are the same for all implementations. An exception to this rule is the CBasicBlock category class where the entities for different implementations differ.

![Figure 2-1. Class structure](image_url)

2.1.1 Creating New Components

When creating a new component, one has to modify the existing files and, of course, add new component class files, i.e. a category class and at least one implementation class. The easiest way to do this is to create macros that automate the process. These preferably take advantage of the VHDLGEN comments to find the appropriate places in the code, and the skeleton files to add the new category and implementation class files. There are macros available for the Microsoft® .NET environment. These might give some ideas of how to design macros for other IDEs as well. After having added the new class sets to the project and modified the existing files, it should be possible to compile the code directly.
Adding Category and Implementation Classes

There are skeleton files available in the Templates folder to facilitate creation of new class sets. Using these files is a three-step process:

- **Add copies of all four files to the project and rename them.**

- **In the CSkeleton files, replace all instances of Skeleton with your own category class name and all instances of SKELETON with your own category class name in upper case. Match case, but not whole words.**

- **In the CSubSkeleton files, replace all instances of Skelton and SKELETON as above, and also replace SubSkel and SUBSKEL with the implementation class name in the same way.**

Modifying Existing Files

The places where the files need to be modified are marked with comments starting with VHDLGEN. Just add your new component in the same manner as the already existing ones. Below is an example from CDesign.cpp where the code needs to be altered. When adding a new implementation class, a new case statement needs to be added for the appropriate category class (right above the VHDLGEN_FIND_SUBTYPE comment). If a new category class is added, it should be added in the end of the outer switch statement. Mimic the existing classes!

```cpp
// VHDLGEN_FIND_LIST_BEGIN -- Do not remove
switch (cmpType)
{
    case Adder:
        nStructSize = sizeof(CAdderProp);
        switch (nSubType)
        {
            case RIPPLECARRY:
                strTypeName="CRippleCarry";
                break;
            case NEGATE:
                strTypeName="CNegate";
                break;
            // VHDLGEN_FIND_SUBTYPE_Adder -- Do not remove
            } break;
    ...
    ...
    ...
} // VHDLGEN_FIND_LIST_END -- Do not remove
```
3 Global Functions

This chapter describes the global functions that can be used to control user feedback and conversion between number systems etc.

3.1 Report
The report function is used to give the user feedback at runtime. It may also add text to the log file.

```c
void report(string strText, EFeedbackLevel severity = Note)
```

The first argument, `strText`, is the text to present to the user. The second argument decides the severity level of the text. The text is only displayed if the severity is greater or equal to the feedback level chosen by the user. By default, the severity is considered to be `Note`.

3.2 double2twos
Converts a double to a string representing a two’s-complement number.

```c
string double2twos(double dValue, int nLength = DEFAULT)
```

The second argument is the word length and thus, the number of characters in the returned string. If no length is specified, the default word length is used.

3.3 double2csdc
Converts a double to a string representing a CSDC number.

```c
string double2csdc(double dValue, int nLength = DEFAULT)
```

Here, too, the second argument is the word length. The resulting string contains +, - and 0's.

3.4 int2twos
This function converts an integer to a two’s-complement number.

```c
string int2twos(int nValue, int nLength = DEFAULT)
```

3.5 toString
Converts an integer to a string.

```c
string toString(int nVal)
```
3.6 double2string
Converts a double to a string.

```
string double2string(double dVal)
```

3.7 trunc
This function truncates a number to the specified word length.

```
double trunc(double dValue, int nWordLength = DEFAULT)
```

As an example, if the word length is 4 the number 0.53125 would be truncated to 0.5 as 0.03125 cannot be represented. Note that it is assumed that the fractional point is placed immediately to the right of the sign bit.

3.8 rest
This function returns the rest when a number is truncated.

```
double rest(double dValue, int nWordLength = DEFAULT)
```

The returned value is the original value (dValue) minus the truncated value (which can be obtained by trunc()). As an example, if the word length is 4 the number 0.53125 would be truncated as above, and the rest would be 0.03125, returned by this function.

3.9 drawChar
This function outputs a character to an output stream the specified number of times.

```
void drawChar(int nCnt, char cChar = ' ', ostream &out = cout)
```

3.10 getArgs
Used to handle command line options. Returns a SCmdLineInfo structure which contains the command line information.

```
SCmdLineInfo getArgs(int argc, char *argv[])
```

3.11 readGenFile
 Reads the generation file, and returns a structure with generation file information.

```
SDesignInfo readGenFile()
```

3.12 toLower
Converts a string to lowercase.
3.13 int2csdc
Converts an integer to a CSDC number.

string int2csdc(int nVal, int nLen)

3.14 mark
Used for debugging. Outputs text to the screen with an associated counter. The counter can be reset by setting bReset to true.

void mark(bool bReset = false)

3.15 reduceToFund
Reduces a number to its fundamental, i.e. shifts it to the right as far as possible.

pair<uint, byte> reduceToFund(int nValue)

The function returns a pair where the first component is the fundamental and the second the number of times the number was shifted.
4 Constants and Definitions

This chapter describes structures, constants, type definitions and enumerations. These are all located in the vhdlgenerator.h file.

4.1 Constants

Some of these constants may be considered global while others are only used in certain parts of the code. In addition to the constants listed here, each category and implementation has its own set of constants.

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>COMPONENTS</td>
<td>9</td>
<td>Number of implemented category classes</td>
</tr>
<tr>
<td>APP_NAME</td>
<td>&quot;vhdlgen&quot;</td>
<td>The application name</td>
</tr>
<tr>
<td>DEFAULT</td>
<td>0</td>
<td>Used to specify that the default argument should be used</td>
</tr>
<tr>
<td>pi</td>
<td>3.141592654</td>
<td></td>
</tr>
<tr>
<td>MAX_CHANNELS</td>
<td>64</td>
<td>The maximal number of channels</td>
</tr>
<tr>
<td>SUBCOMP</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>TOPCOMP</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>SCALE</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>NOSCALE</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>LINE</td>
<td>(char)196</td>
<td>The character for a horizontal line</td>
</tr>
<tr>
<td>DOUBLE_LINE</td>
<td>(char)205</td>
<td>Character for a double horizontal line</td>
</tr>
<tr>
<td>ADD</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>SUB</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>TOP</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>NOTOP</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>WLEN</td>
<td>PROP-&gt;nLength</td>
<td>Word length</td>
</tr>
<tr>
<td>N</td>
<td>PROP-&gt;nChannels</td>
<td>Number of channels</td>
</tr>
<tr>
<td>NO_PIPELINE</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>TRUNCATE</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>ROUND</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>CO</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>CI</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>NO_CI</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>NO_CO</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>BITADDERS</td>
<td>2</td>
<td>MAG secondary optimization</td>
</tr>
<tr>
<td>PROPTIME</td>
<td>3</td>
<td>MAG secondary optimization</td>
</tr>
<tr>
<td>STD_LOGIC</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>IN</td>
<td>1</td>
<td></td>
</tr>
<tr>
<td>OUT</td>
<td>0</td>
<td></td>
</tr>
<tr>
<td>NONE</td>
<td>1</td>
<td>VHDL scope</td>
</tr>
<tr>
<td>ENTITY</td>
<td>2</td>
<td>VHDL scope</td>
</tr>
<tr>
<td>ARCH_HEAD</td>
<td>4</td>
<td>VHDL scope</td>
</tr>
<tr>
<td>ARCH_BODY</td>
<td>8</td>
<td>VHDL scope</td>
</tr>
<tr>
<td>PROC_HEAD</td>
<td>16</td>
<td>VHDL scope</td>
</tr>
<tr>
<td>PROC_BODY</td>
<td>32</td>
<td>VHDL scope</td>
</tr>
<tr>
<td>COMPONENT</td>
<td>64</td>
<td>VHDL scope</td>
</tr>
</tbody>
</table>
### Constant

<table>
<thead>
<tr>
<th>Constant</th>
<th>Value</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>PORTMAP</td>
<td>128</td>
<td>VHDL scope</td>
</tr>
<tr>
<td>NEWLINE</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>CLOCKED</td>
<td>true</td>
<td></td>
</tr>
<tr>
<td>NOT_CLOCKED</td>
<td>false</td>
<td></td>
</tr>
<tr>
<td>PROGRAM</td>
<td>1</td>
<td>Generation file section</td>
</tr>
<tr>
<td>DESIGN</td>
<td>2</td>
<td>Generation file section</td>
</tr>
<tr>
<td>COEFFS</td>
<td>3</td>
<td>Generation file section</td>
</tr>
<tr>
<td>MAG</td>
<td>4</td>
<td>Generation file section</td>
</tr>
</tbody>
</table>

### 4.2 Enumerations

<table>
<thead>
<tr>
<th>Name</th>
<th>Content</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>EComponent</td>
<td>NotDefined Adder Butterfly FixedMult BasicBlock Dct Filter Register MultBlock AnalysisFilterBank</td>
<td>This enumeration contains the category classes. If a new category class is created, an entry should be added here</td>
</tr>
<tr>
<td>EFeedbackLevel</td>
<td>Debug Detailed Note Warning Error</td>
<td>The feedback level to the user.</td>
</tr>
<tr>
<td>ECommentStyle</td>
<td>Extensive Sparse None</td>
<td>How the VHDL code should be commented</td>
</tr>
</tbody>
</table>

### 4.3 Type Definitions

<table>
<thead>
<tr>
<th>name</th>
<th>type definition</th>
</tr>
</thead>
<tbody>
<tr>
<td>uint</td>
<td>unsigned int</td>
</tr>
<tr>
<td>byte</td>
<td>unsigned char</td>
</tr>
<tr>
<td>stringpair</td>
<td>pair&lt;string, string&gt;</td>
</tr>
<tr>
<td>intpair</td>
<td>pair&lt;int, int&gt;</td>
</tr>
</tbody>
</table>

### 4.4 Structures

<table>
<thead>
<tr>
<th>name</th>
<th>description</th>
</tr>
</thead>
<tbody>
<tr>
<td>SComponentList</td>
<td>List of implemented components</td>
</tr>
<tr>
<td>SCmdLineInfo</td>
<td>Contains information about command line options</td>
</tr>
<tr>
<td>SImplementInfo</td>
<td>Used when implementing multipliers</td>
</tr>
<tr>
<td>SDesignInfo</td>
<td>Information about the design</td>
</tr>
<tr>
<td>SOpInfo</td>
<td>Also used with multipliers</td>
</tr>
</tbody>
</table>
In order to facilitate the generation of VHDL code, several support functions are available in the CComponent class. This chapter serves as a reference to the VHDL support functions.

The VHDL support functions are supposed to facilitate VHDL generation, and to make the source code more readable. It is, of course, possible to write the VHDL code without these functions as well. Here is an example of how some code generation might look:

```vhdl
vhdlArch();

nHA = vhdlComponent( BasicBlock, new CBasicBlockProp(), HALFADDER);
nFA = vhdlComponent( BasicBlock, new CBasicBlockProp(), FULLADDER);
vhdlSignal("c", WLEN);
vhdlBegin();

vhdlComment("This is the first bit adder", NEWLINE);
vhdlPortMap(nHA);
vhdlMap( "A", "A", 0);
vhdlMap( "B", "B", 0);
vhdlMap( "SUM", "SUM", 0);
vhdlMap( "CO", "C", 0);
vhdlEndPortMap();

vhdlComment("These are the following bit adders", NEWLINE);
vhdlPortMap(nFA, NOT_CLOCKED, WLEN-2);
vhdlMap( "A", "A(X+1)" );
vhdlMap( "B", "B(X+1)" );
vhdlMap( "CI", "C(X)" );
vhdlMap( "SUM", "SUM(X+1)" );
vhdlMap( "CO", "C(X+1)" );
vhdlEndPortMap();

vhdlMap( "COUT", "C", WLEN-1 );
vhdlEndArch();
```

This excerpt uses several of the VHDL support functions. It generates an n-bit adder. The generated VHDL code is going to be:

```vhdl
architecture GATE of RC8_ADD is
    component HA2 is port(
        A, B : in STD_LOGIC;
        SUM, CO : out STD_LOGIC );
    end component;
    component FA3 is port(
        A, B, CI : in STD_LOGIC;
        SUM, CO : out STD_LOGIC );
    end component;
```
signal C         : STD_LOGIC_VECTOR(7 downto 0);
begin
  pm1: HA2 port map(A(0), B(0), SUM(0), C(0));
  pm2: for X in 1 to 7 generate
    pms: FA3 port map(A(X), B(X), C(X-1), SUM(X),
                        C(X));
  end generate;
  COUT <= C(7);
end GATE;

5.1 Scope
The program keeps track of the current scope of two reasons. First, it provides
some minimalistic error checking. Second, it is used to determine what code to
generate for certain support functions, e.g. vhdlMap, that work for several
scopes. The defined scopes are:

- **NONE.** Means the current position is outside any block.
- **ENTITY.** Inside an Entity-definition.
- **ARCH_HEAD.** Inside the header of an architecture, i.e. where sub
  components, signals etc. are defined.
- **ARCH_BODY.** Architecture body.
- **PROC_HEAD.** Process header.
- **PROC_BODY.** Process body.
- **PORTMAP.** Inside a port map block.

The scope is kept in the variable vhdlScope and is checked by calling the
function checkScope(int nScope) where nScope is the correct scope. If
the current scope is incorrect, the program outputs the error message "Illegal
VHDL scope" and terminates.

5.2 vhdl
vhdl is an output stream defined separately for each component, pointing to
the VHDL file where the code should be placed.

5.3 vhdlLibraries
Adds the standard IEEE libraries std_logic_1164, std_logic_arith and
std_logic_unsigned as well as the package generated by the program.
5.4 vhdlBegin
First, vhdlBegin makes sure the current scope is ARCH_HEAD. If it is, vhdlBegin adds a VHDL begin-statement and changes the scope to ARCH_BODY. It also adds a comment unless the comment level is set to none by the user.

5.5 vhdlSignal

```vhdl
vhdlSignal(string strSignal, int nSigType)
```

Adds a signal with name strSignal to an architecture header. If nSigType is one, the signal will be of type STD_LOGIC. If nSigType is greater than one, it will be of type STD_LOGIC_VECTOR(nSigType-1 downto 0).

5.6 vhdlArch and vhdlEndArch

```vhdl
vhdlArch(string strArch = "GATE", string strEntity = "")
vhdlEndArch();
```

These functions initiate and end an architecture declaration, respectively. In case no arguments are specified, the architecture name will be GATE and the component name will be used as the entity reference.

5.7 vhdlEntity and vhdlEndEntity

```vhdl
vhdlEntity(string strName = "")
vhdlEndEntity(string strName = "")
```

Begins and ends an entity declaration, respectively. If no arguments are specified, the entity name will be the same as the component name.

5.8 vhdlIO

```vhdl
vhdlIO(string strName, int nType, int nLen)
vhdlIO(string strName, int nType, string strType)
```

Generates an in-/output used in either entities or components. nType should be either IN or OUT. nLen is the word length of the signal. If it equals one, the signal type is going to be STD_LOGIC, otherwise it will be an STD_LOGIC_VECTOR(nLen-1 downto 0).

5.9 vhdlPortMap and vhdlEndPortMap

```vhdl
vhdlPortMap(string strName, bool bClocked = false, int nVal = 1)
vhdlPortMap(int nId, bool bClocked = false, int nVal = 1)
vhdlEndPortMap()
```
Generates a port map statement by using either strName or the name of the component with id nId. If nVal is greater than one, an array of port maps will be created by the use of the VHDL statement generate. As an example

```
vhdlPortMap("MYCOMP", 7)
```

generates the following VHDL code

```
pl: for X in 1 to 7 generate
pms: port map MYCOMP is ( where

```

where pm1 is a unique identifier for this port map. A static integer in CComponent is increased every time a new port map is generated. The variable X is always used when creating multiple port maps, keep this in mind when using the vhdlMap function.

### 5.10 vhdlMap

- `vhdlMap(string strPort, string strSignal)`
- `vhdlMap(string strPort, int nPos, string strSignal)`
- `vhdlMap(string strPort, string strSignal, int nPos)`
- `vhdlMap(string strPort, int nPos1, string strSignal, int nPos2)`
- `vhdlMap(string strPort, int nStart, int nEnd, string strSignal)`
- `vhdlMap(string strPort, string strSignal, int nStart, int nEnd)`
- `vhdlMap(string strPort, int nStart1, int nEnd1, string strSignal, int nStart2, int nEnd2)`
- `vhdlMap(string strPort, int nStart, int nEnd, string strSignal, int nPos)`

`vhdlMap` may be used both inside and outside `vhdlPortMap` blocks. They work a bit different however, but the key functionality is the same. They are used to map ports and signals in a port map, or to map a signal in the architecture body, outside a port map, i.e.

```
vhdlMap( "S0", WLEN-2, 0, "I", WLEN-1, 1 );
```

translates to, if `WLEN` equals 32

```
S0(30 downto 0) <= I(31 downto 1);
```

if outside a port map, and

```
S0(30 downto 0) => I(31 downto 1)
```

if inside a port map. There are eight overloaded versions of this function, to cover all cases, e.g. if the five MSBs of the destination signal should be mapped to the sign bit of the source signal, one could write

```
vhdlMap( "S0", WLEN-1, WLEN-5, "I", WLEN-1 );
```

which would generate the following code:
It is also possible to use 0 or 1 instead of a source signal. The vhdlMap functions check if the start and end value of a range is equal, and adapts the generated code accordingly.

### 5.11 vhdlScaleDown and vhdlScaleUp

```vhdl
vhdlScaleDown(string strDest, string strSrc, int nSteps = 1,
               int nWL = DEFAULT)
vhdlScaleUp(string strDest, string strSrc, int nSteps = 1,
             int nWL = DEFAULT)
```

scaleDown scales the nWL-bit source signal nSteps to the right, and extends the sign-bit to the empty positions. scaleUp scales a signal to the right in the same way, inserting trailing zeros.

### 5.12 vhdlHeader and vhdlEndHeader

```vhdl
vhdlHeader()
vhdlHeader(string strTitle, string strText)
vhdlHeader(string strTitle, double dValue)
vhdlEndHeader()
```

These functions are used when generating file headers, i.e. text (comments) in the beginning of a file – describing its content.

### 5.13 vhdlComment

```vhdl
vhdlComment(string strText, bool bNewLine,
             ECommentStyle comment = Sparse)
vhdlComment(string strText = "", ECommentStyle comment = Sparse)
```

These two functions are used to generate comments in the code.
6 Generation Files

The program communicates with the user through text files, called generation files. These are described in this chapter.

The generation files (with file extension .GEN) are ordinary text files where the user can set design and runtime options. The files are divided into sections that control different aspects of the program. Spaces are considered to be white characters, and are thus ignored. It is also possible to add comments. These are indicated by semi colons, and are effective to the end of the line.

6.1 Program Section

This section controls program related information, such as file extensions and whether or not to use a log file. The section header is encapsulated in square brackets.

[MAG] ; Beginning of MAG section
[DESIGN] ; Beginning of Design section
[PROGRAM] ; Beginning of Program section
[COEFFS] ; Beginning of Coefficient section

6.1.1 Log File

This option lets the user decide whether or not to use a log file.

LogFile = Yes
LogFile = No

6.1.2 Batch Files

If this option is set to yes, batch files will be created. The batch files probably need to be modified to fit the user's needs.

BatchFiles = Yes
BatchFiles = No

6.1.3 Makefile

This options control the creation of a make file.

Makefile = Yes
Makefile = No

6.1.4 Extension

Controls the file extension of the generated VHDL files. An optional dot may be placed in front of the extension. Note that because the generation file is not case sensitive, the file extensions will always be in lower case.
6.1.5 Feedback
This controls the user feedback at runtime.

- Feedback = silent
- Feedback = normal
- Feedback = detailed
- Feedback = debug
- Feedback = warnings

The user is going to get feedback about all events with the same, and higher, priority as the level specified.

6.2 Design Section
This section controls design options. This is where the user specifies what design to generate.

6.2.1 Round Method
This option allows the user to decide what kind of rounding mechanism to use in different components. When creating own implementation classes, one could check this option to generate different designs. Note that this option is not implemented in any components.

- RoundMethod = Truncate
- RoundMethod = Round

6.2.2 Constants
In addition to using the coefficients section (see Section 6.4), the user can set coefficient values here.

- Constant = 0.232342
- Coeff1 = 0.2523424
- Coeff2 = 0.7898732
- Coeff3 = 0.9182329
- Coeff4 = -0.2452234

It is possible to set up to four different constants. Note that the keyword constant is equivalent to coeff1.

6.2.3 Word Length
The word length to use as a default in the design.

- WordLength = 12
6.2.4 Comments
Controls how extensive the generated VHDL code will be commented.

| Comments = Extensive |
| Comments = Sparse   |
| Comments = None     |

6.2.5 Top Design
This option sets the top design, i.e. the design to generate. The available options are the names of the category classes. When a category class is added to the program, it should be added here as well.

| Top = Adder |
| Top = Butterfly |
| Top = FixedMult |
| Top = BasicBlock |
| Top = DCT |
| Top = Register |
| Top = MultBlock |
| Top = AnalysisFilterBank |
| Top = Filter |

6.2.6 Name
This option is used to set the name of the design. The file path will be set to automatically if the path is not set specifically.

| Name=MyDesignName |

6.2.7 File Path
File path where the design should be placed. Note that the program is not capable of creating the path itself, i.e. the folder must exist beforehand.

| Path=d:/temp/dct/MyDesign/ |

Note also that the file path should end with a slash.

6.2.8 Order, Outputs and Coefficients
This option is used to specify the filter order, number of outputs or number of coefficients to use for the top design. Note that these are equivalent, rather than three different options.

| Order=32 |
| Outputs=16 |
| Coeffs=4 |
6.2.9 Channels
Used to specify how many channels to use.

Channels=16

6.2.10 Pipelines
Specifies how many pipeline stages to add in a design, such as a DCT block. The option specifies the number of stages between each pipeline stage. If no pipeline stages should be used, 0 should be chosen. It is also possible to enter max and min, which is equivalent to 1 and 0, respectively.

Pipelines=2
Pipelines=0
Pipelines=max
Pipelines=min
Pipelines=1

6.2.11 Components
Each category class has its own option as well, determining what implementation to use for that specific category. If no implementation is specified, the first implementation class will be used to generate the component.

Adder=RippleCarry
Butterfly=SandeTukey
FixedMult=GraphMult
DCT=SparseMatrix_DCTIV
Filter=TransposedDirect
MultBlock=RagBlock
AnalysisFilterBank=DCTAnalysisBank

6.3 MAG Section
This section is used to control the MAG and RAG-n algorithms.

6.3.1 Use Files
Used to determine whether or not to use files to save the generated cost and fundamentals tables.

UseFiles = Yes
UseFiles = No

6.3.2 Secondary
Determines the secondary optimization constraint.

Secondary = PropTime  ; Optimize for shortest propagation time
Secondary = BitAdders ; Optimize for least # of bit adders
Secondary = None      ; Ignore secondary optimization
6.4 Coefficient section

Here the user can specify coefficients to use with, for example, filters. It is easy to generate coefficients in an external program, e.g. Matlab, and then copy and paste them to this section.

```
[Coeffs]
0.44509643228
-0.93181457846
0.46599434167
0.41864946772
-0.84622141782
0.52515249630
0.20264735765
```

6.5 Example Files

Below are two example files that illustrate how different designs can be accomplished through generation files. The first example creates a multiplier block with word length eight and four outputs. As no multiplier block type is specified, the default is used. The second example generates a DCT with 16 channels and word length 16. In this example, no log file is generated, and only errors and warnings will be displayed.

Example 1

```
; Program settings. This example generates
; a log file and generates detailed on-screen feedback

[PROGRAM]
LogFile=Yes
Feedback=Detailed

[DESIGN]
Top=MultBlock
Name=MULTBLOCK
WordLength=8
Comments=Extensive
Path=d:/temp/dct/
Outputs=4

; Multiplier block coefficients. The number of coefficients are
; controlled by Outputs in the design section. Any superfluous
; coefficients will be ignored!

[COEFFS]
0.1353245
0.4323411
0.6783423
0.9892342
```
Example 2

; Program settings
[PROGRAM]
LogFile = No ; Do not use a log file
Feedback = Silent ; No output to user. Only errors

; Design specifications
[DESIGN]
Top = Dct ; Generate a DCT
Dct = DITDCTIV ; Set DCT type
WordLength = 16
Comments = Sparse ; How to comment the code
Channels = 16
Appendix A - Classes

In this appendix, all classes are listed. As all component classes contains the same basic member functions, the template classes are described instead of each component unless the component class diverges considerably. This should also help when adding new component class sets to the program.

<table>
<thead>
<tr>
<th>Class</th>
<th>CDesign</th>
</tr>
</thead>
<tbody>
<tr>
<td>Base Class</td>
<td>-</td>
</tr>
</tbody>
</table>

**Description**
This is the main class which contains design related information and functionality. It keeps track of created components, default design specifications etc.

**Properties**
- **SComponentList *pFirst, *pLast**
  Pointers to the first and last item in the component list.
- **int cmpDefault[getSizeof(EComponent)]**
  An array with the default subtype for each component.
- **string strDesignName**
  The name of the design. Used mainly for naming files, folders etc.
- **string strFilePath**
  The path to the location where the generated VHDL code is saved.
- **string strExtension**
  File extension used on generated files, e.g. vhdl
- **string strGenFile**
  Name of the configuration file to use
- **ECommentStyle commentLevel**
  This controls how much the code should be commented
- **bool bLogFile**
  Whether or not to use a log file
- **ofstream logFile**
  An output stream to the log file
- **int nWordLength**
  The default word length
- **int nRoundMethod**
  Rounding method in, for example, multipliers. (Not implemented.)

**Public Member Functions**
- **CDesign()**
  Constructor
- **~CDesign()**
  Destructor
- **int add(EComponent cmpType, CProperties *propComponent, int nSubType = DEFAULT)**
  Adds the specified component. The VHDL code is generated in a separate file
- **int add(EComponent cmpType, CProperties *propComponent, ofstream out, int nSubType = DEFAULT)**
  Adds the specified component. The VHDL code is appended to the file pointed to by out
- **int find(EComponent cmpType, CProperties *propComponent, int nSubType)**
  Looks for a specific component in the component list. If it is found, its ID is returned.
- **void openLogFile()**
  Opens the log file
- **void log(string strText)**
  Adds an entry to the log file if the file is opened and should be used
- **string getDesignName()**
  Returns the design name
- **int getWordLength()**
  Returns the word length
- **string getFilePath()**
  Returns the file path
<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>ECommentStyle getCommentLevel()</code></td>
<td>Returns the comments level</td>
</tr>
<tr>
<td><code>bool getLogFile()</code></td>
<td>Returns true if a log file is used</td>
</tr>
<tr>
<td><code>string getFileExtension()</code></td>
<td>Returns the file extension</td>
</tr>
<tr>
<td><code>string getGenFile()</code></td>
<td>Returns the configuration file name</td>
</tr>
<tr>
<td><code>int trunc()</code></td>
<td>Returns the used round method (not implemented)</td>
</tr>
<tr>
<td><code>string getCompName(int nId)</code></td>
<td>Returns the name of the component with the specified id</td>
</tr>
<tr>
<td><code>void setCommentLevel(ECommentStyle comm)</code></td>
<td>Sets the comment level</td>
</tr>
<tr>
<td><code>void setLogFile(bool bLog)</code></td>
<td>Turn log file usage on/off</td>
</tr>
<tr>
<td><code>void setFileExtension(string strExt)</code></td>
<td>Sets the file extension</td>
</tr>
<tr>
<td><code>void setDefault(EComponent cmpType, int nSubType)</code></td>
<td>Sets the default subtype for a component</td>
</tr>
<tr>
<td><code>void setDesignName(string strName)</code></td>
<td>Sets the design name</td>
</tr>
<tr>
<td><code>void setLength(int nLength, int nGuard)</code></td>
<td>Sets both the default word length and the number of guard bits</td>
</tr>
<tr>
<td><code>void setFilePath(string strPath)</code></td>
<td>Sets the file path</td>
</tr>
<tr>
<td><code>void setGenFile(string strFile)</code></td>
<td>Sets the name of the configuration file to use</td>
</tr>
<tr>
<td><code>void generatePackage()</code></td>
<td>Generate a package file containing some libraries and STAGE_SIGNALS</td>
</tr>
<tr>
<td><code>void generateComp(int nId, ofstream &amp;out)</code></td>
<td>Generates a component declaration in an architecture header, and is used by vhdlComponent when adding sub components</td>
</tr>
<tr>
<td><code>void createBatch()</code></td>
<td>Creates two DOS batch files that can be run to compile the code. These probably need to be modified to work with the compiler used</td>
</tr>
<tr>
<td><code>void createMakeFile()</code></td>
<td>Creates a makefile to be used when compiling the VHDL files. Probably needs to be modified to work with the environment at hand</td>
</tr>
<tr>
<td><code>void setRoundMethod</code></td>
<td>Sets the round method to use (not implemented)</td>
</tr>
</tbody>
</table>

**Private/Protected Member Functions**

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>CComponent *find(int nId)</code></td>
<td>Returns a pointer to the component with the specified id</td>
</tr>
</tbody>
</table>
### Class CComponent

#### Base Class
- CComponent

#### Description
This is an abstract class for all components. It does contain some vital information in common for all components. For example, each component is associated with a unique id and has a name and a subtype description.

#### Properties

<table>
<thead>
<tr>
<th>Type</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>nId</td>
<td>The component’s unique id</td>
</tr>
<tr>
<td>string</td>
<td>strCompName</td>
<td>The name of the component</td>
</tr>
<tr>
<td>string</td>
<td>strType</td>
<td>The subtype description</td>
</tr>
<tr>
<td>static int</td>
<td>nCount</td>
<td>Keeps track of the number of components in the design. Primarily used to give each component a unique id</td>
</tr>
<tr>
<td>static int</td>
<td>nPorts</td>
<td>Keeps track of the number of instantiated port maps in the design</td>
</tr>
<tr>
<td>static int</td>
<td>nPipeCnt</td>
<td>The number of stages until next pipeline (used in DCT blocks)</td>
</tr>
<tr>
<td>static int</td>
<td>nPipeStep</td>
<td>The number of stages between each pipeline (used in DCT blocks)</td>
</tr>
<tr>
<td>ofstream</td>
<td>vhdl</td>
<td>An output stream pointing at the file used for the component</td>
</tr>
<tr>
<td>string</td>
<td>vhdlArchName</td>
<td>Keeps track of the architecture name, used by the VHDL support functions</td>
</tr>
<tr>
<td>int</td>
<td>vhdlCount</td>
<td>Used by VHDL support functions</td>
</tr>
<tr>
<td>int</td>
<td>vhdlArg</td>
<td>Used by VHDL support functions</td>
</tr>
<tr>
<td>int</td>
<td>vhdlCompCnt</td>
<td>Used to keep track of duplicate sub component declarations in an architecture header</td>
</tr>
<tr>
<td>int</td>
<td>vhdlSubComps[100]</td>
<td>Same as above</td>
</tr>
<tr>
<td>int</td>
<td>vhdlScope</td>
<td>Scope in VHDL code, used by VHDL support functions</td>
</tr>
<tr>
<td>CProperties *</td>
<td>pProperties</td>
<td>A pointer to the properties class for the component</td>
</tr>
</tbody>
</table>

#### Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CComponent()</td>
<td>Constructor</td>
</tr>
<tr>
<td>~CComponent()</td>
<td>Destructor</td>
</tr>
<tr>
<td>string getName()</td>
<td>Gets the component name</td>
</tr>
<tr>
<td>int getId()</td>
<td>Gets the component id</td>
</tr>
<tr>
<td>CProperties *getProperties()</td>
<td>Returns a pointer to the properties class</td>
</tr>
<tr>
<td>virtual void generateComponent(ofstream &amp;out) = 0</td>
<td>Abstract function that generates the VHDL code that goes in the architecture header</td>
</tr>
<tr>
<td>static int getCount()</td>
<td>Returns the number of components in the design</td>
</tr>
</tbody>
</table>
### Private/Protected Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>virtual void generateName() = 0</td>
<td>Abstract function, generates the name of the component and puts it in the property \texttt{strCompName}</td>
</tr>
<tr>
<td>void checkScope(int nScope)</td>
<td>Compares the current scope with \texttt{nScope} and generates an error message if they don’t match</td>
</tr>
</tbody>
</table>

### VHDL Support Functions (see Appendix A for descriptions)

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>void vhdlLibraries()</td>
<td></td>
</tr>
<tr>
<td>void vhdlBegin()</td>
<td></td>
</tr>
<tr>
<td>void vhdlSignal(string strSignal, int nSigType)</td>
<td></td>
</tr>
<tr>
<td>void vhdlArch(string strArch = &quot;GATE&quot;, string strEntity = &quot;&quot;)</td>
<td></td>
</tr>
<tr>
<td>void vhdlEntity(string strName = &quot;&quot;)</td>
<td></td>
</tr>
<tr>
<td>int vhdlComponent(EComponent cmpType, CProperties *propComponent, int nSubType = DEFAULT)</td>
<td></td>
</tr>
<tr>
<td>int vhdlComponent(EComponent cmpType, CProperties *propComponent, string strAppend, int nSubType = DEFAULT)</td>
<td></td>
</tr>
<tr>
<td>void vhdlEndEntity(string strName = &quot;)</td>
<td></td>
</tr>
<tr>
<td>void vhdlEndArch()</td>
<td></td>
</tr>
<tr>
<td>void vhdlIO(string strName, int nType, int nLen)</td>
<td></td>
</tr>
<tr>
<td>void vhdlIO(string strName, int nType, string strType)</td>
<td></td>
</tr>
<tr>
<td>void vhdlPortMap(string strName, bool bClocked = false, int nVal = 1)</td>
<td></td>
</tr>
<tr>
<td>void vhdlPortMap(int nId, bool bClocked = false, int nVal = 1)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, string strSignal)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, int nPos, string strSignal)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, string strSignal, int nPos)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, int nPos1, string strSignal, int nPos2)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, int nStart, int nEnd, string strSignal)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, string strSignal, int nStart, int nEnd)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, int nStart1, int nEnd1, string strSignal, int nStart2, int nEnd2)</td>
<td></td>
</tr>
<tr>
<td>void vhdlMap(string strPort, int nStart, int nEnd, string strSignal, int nPos)</td>
<td></td>
</tr>
<tr>
<td>void vhdlEndPortMap()</td>
<td></td>
</tr>
<tr>
<td>void vhdlProcess()</td>
<td></td>
</tr>
<tr>
<td>void vhdlHeader()</td>
<td></td>
</tr>
<tr>
<td>void vhdlHeader(string strTitle, string strText)</td>
<td></td>
</tr>
<tr>
<td>void vhdlHeader(string strTitle, double dValue)</td>
<td></td>
</tr>
<tr>
<td>void vhdlEndHeader()</td>
<td></td>
</tr>
<tr>
<td>void vhdlComment(string strText, bool bNewLine, ECommentStyle comment = Sparse)</td>
<td></td>
</tr>
<tr>
<td>void vhdlComment(string strText = &quot;, ECommentStyle comment = Sparse)</td>
<td></td>
</tr>
<tr>
<td>void vhdlScaleDown(string strDest, string strSrc, int nSteps = 1, int nWL = DEFAULT)</td>
<td></td>
</tr>
<tr>
<td>void vhdlScaleUp(string strDest, string strSrc, int nSteps = 1, int nWL = DEFAULT)</td>
<td></td>
</tr>
</tbody>
</table>
### Class **CMag**

**Base Class**
- 

**Description**
This class encapsulates the MAG and RAG algorithms. The CMag class is instantiated globally in order to spare time by not having to initialize it more than once. The CMag class utilizes the template class CTable to create the two necessary tables: The cost table and the fundamentals’ table. The CMag class has a number of type definitions and structures associated with it, described in their respective sections.

<table>
<thead>
<tr>
<th>Properties</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>int nSecondary</strong></td>
</tr>
<tr>
<td><strong>costIter iOne</strong></td>
</tr>
<tr>
<td>*<em>SFund <em>pOne</em></em></td>
</tr>
<tr>
<td><strong>int articulationPoint</strong></td>
</tr>
<tr>
<td><strong>bool bMagInitialized</strong></td>
</tr>
<tr>
<td><strong>int nInfo</strong></td>
</tr>
<tr>
<td><strong>SImplementInfo rgInfo[20]</strong></td>
</tr>
<tr>
<td><strong>bool bSubUsed</strong></td>
</tr>
<tr>
<td><strong>int nLen</strong></td>
</tr>
<tr>
<td><strong>uint nMax</strong></td>
</tr>
<tr>
<td><strong>int nRagCoeffs</strong></td>
</tr>
<tr>
<td><strong>list&lt;incompPair&gt; incompleteSet</strong></td>
</tr>
<tr>
<td><strong>list&lt;incompPair&gt; outputSet</strong></td>
</tr>
<tr>
<td><strong>list&lt;SRagGraph&gt; graphSet</strong></td>
</tr>
<tr>
<td><strong>TCost costTable</strong></td>
</tr>
<tr>
<td><strong>TFund fundTable</strong></td>
</tr>
<tr>
<td><em><em>list&lt;SFund</em>&gt; convList</em>*</td>
</tr>
<tr>
<td><strong>bool bUseFiles</strong></td>
</tr>
<tr>
<td>Variable</td>
</tr>
<tr>
<td>---------------</td>
</tr>
<tr>
<td>byte nShiftFirst</td>
</tr>
<tr>
<td>bool bNeg</td>
</tr>
</tbody>
</table>

### Public Member Functions

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>CMag()</td>
<td>Constructor</td>
</tr>
<tr>
<td>void initMag(int nWordLen = DEFAULT)</td>
<td>Initializes the MAG tables</td>
</tr>
<tr>
<td>void drawGraphs(uint unVal, ostream &amp;out = cout)</td>
<td>Draws all graphs for a coefficient</td>
</tr>
<tr>
<td>void drawGraph(SFund *pFund, ostream &amp;out = cout)</td>
<td>Draws the graph for the coefficient pointed to by pFund</td>
</tr>
<tr>
<td>void drawGraph(uint unVal, ostream &amp;out = cout)</td>
<td>Draws the graph for the coefficient</td>
</tr>
<tr>
<td>SImplementInfo *generateMult(ostream &amp;out, double dValue, int nOptMethod)</td>
<td>Generates implementation info for a graph multiplier. The third argument, optimization method, is the secondary optimization, either BITADDERS or PROPTIME</td>
</tr>
<tr>
<td>bool isSubUsed()</td>
<td>Returns true if a subtractor is used for the latest generated graphmult</td>
</tr>
<tr>
<td>int getUsedSignals()</td>
<td>Returns the number of internal signals needed for the latest generated multiplier</td>
</tr>
<tr>
<td>int getInitScale()</td>
<td>Returns how many steps the coefficient was scaled initially</td>
</tr>
<tr>
<td>int getPropTime(SFund *pFund)</td>
<td>Returns the propagation time</td>
</tr>
<tr>
<td>SFund* getBestGraph(uint unValue)</td>
<td>Returns a pointer to the best graph using the chosen secondary optimization method</td>
</tr>
<tr>
<td>int getSecondary()</td>
<td>Returns the secondary optimization method</td>
</tr>
<tr>
<td>void setSecondary(int nSec)</td>
<td>Sets the secondary optimization method</td>
</tr>
<tr>
<td>void setUseFiles(bool bUF)</td>
<td>Used to decide whether or not to use files</td>
</tr>
<tr>
<td>int getBitAdders(int nStep)</td>
<td>Returns the number of bit adders used for step nStep</td>
</tr>
<tr>
<td>SImplementInfo *ragn(int nCoeffs, double *pCoeff)</td>
<td>Generates a multiplier block using the RAG-n algorithm</td>
</tr>
<tr>
<td>Function Description</td>
<td></td>
</tr>
<tr>
<td>----------------------</td>
<td></td>
</tr>
<tr>
<td>Adds a node</td>
<td></td>
</tr>
<tr>
<td>Generates graphs with cost ucCost</td>
<td></td>
</tr>
<tr>
<td>Generates an additive graph</td>
<td></td>
</tr>
<tr>
<td>Adds nodes</td>
<td></td>
</tr>
<tr>
<td>Creates a node</td>
<td></td>
</tr>
<tr>
<td>Generates nodes</td>
<td></td>
</tr>
<tr>
<td>Expands fundamentals with cost ucCost to shifted versions as well</td>
<td></td>
</tr>
<tr>
<td>Returns true if the value is in range [0, nMax]</td>
<td></td>
</tr>
<tr>
<td>Outputs statistics of the generated graphs</td>
<td></td>
</tr>
<tr>
<td>Returns information about the node</td>
<td></td>
</tr>
<tr>
<td>Stores implementation information about the node</td>
<td></td>
</tr>
<tr>
<td>Used to remove duplicate coefficients in a RAG-n coefficient set</td>
<td></td>
</tr>
<tr>
<td>Removes ones in a RAG-n coeff. set</td>
<td></td>
</tr>
<tr>
<td>Removes zeros in a RAG-n coeff. set</td>
<td></td>
</tr>
<tr>
<td>Looks for the coefficient in the RAG-n list of incomplete coefficients</td>
<td></td>
</tr>
<tr>
<td>Looks for the graph in the RAG-n graph set</td>
<td></td>
</tr>
<tr>
<td>Looks for the coefficient in the RAG-n output set</td>
<td></td>
</tr>
<tr>
<td>Returns true if the coefficient exists in the RAG-n incomplete set</td>
<td></td>
</tr>
<tr>
<td>Returns true if the coefficient exists in the RAG-n graph set</td>
<td></td>
</tr>
<tr>
<td>Returns true if the coefficient exists in the RAG-n output set</td>
<td></td>
</tr>
<tr>
<td>Returns true if unValue is a fundamental</td>
<td></td>
</tr>
<tr>
<td>Returns the cost of the coefficient</td>
<td></td>
</tr>
<tr>
<td>Adds nodes to the RAG-n graph set</td>
<td></td>
</tr>
<tr>
<td>Displays a rotating cursor to indicate that the algorithm is working</td>
<td></td>
</tr>
<tr>
<td>Displays the RAG-n coefficient sets</td>
<td></td>
</tr>
<tr>
<td>Displays the RAG-n output set</td>
<td></td>
</tr>
<tr>
<td>Displays the RAG-n graph set</td>
<td></td>
</tr>
<tr>
<td>Generates a RAG-n multiplier block with nCoeffs coefficients</td>
<td></td>
</tr>
<tr>
<td>Function</td>
<td>Description</td>
</tr>
<tr>
<td>----------</td>
<td>-------------</td>
</tr>
<tr>
<td><code>void pushGraph(uint unValue, uint unLeft, uint unRight, byte ucShift)</code></td>
<td>Adds a graph to the graph set</td>
</tr>
<tr>
<td><code>void saveTables()</code></td>
<td>Saves the cost and fundamental tables to file</td>
</tr>
<tr>
<td><code>void readTables()</code></td>
<td>Reads the cost and fundamental tables from file</td>
</tr>
<tr>
<td><code>SFund *convert(int nIndex)</code></td>
<td>Used when reading and saving tables</td>
</tr>
<tr>
<td><code>int drawTraverse(SDrawInfo *pInfo, SFund *pFund, int nDep = 0, int nNode = 0)</code></td>
<td>Used when drawing graphs</td>
</tr>
<tr>
<td><code>void negImplInfo()</code></td>
<td>Negates the implementation info</td>
</tr>
<tr>
<td><code>bool nodeDone(int nNode, bool bAddNode = false)</code></td>
<td>Used by <code>negImplInfo()</code> to determine whether a node has been negated</td>
</tr>
</tbody>
</table>

### Class: `CSkeleton`<br>**Base Class:** `CComponent`<br>**Description:** This is the skeleton for the component category classes. Some of the components might have additional properties and functions, but this class includes the basic functionality.<br><br>**Public Member Functions**<br>```cpp
CSkeleton(CSkeletonProp *propComp) Constructor. Takes a derived CProperties class as its only argument.
virtual void generateComponent(ofstream &out) = 0 (see CComponent for details)
```

**Private/Protected Member Functions**<br>```cpp
virtual void generateName() = 0 (see CComponent for details)
void generateHeader() Generates the file header for the .vhdl file. Also includes IEEE libraries.
virtual void generateEntity()```

### Class: `CSubSkel`<br>**Base Class:** `CSkeleton`<br>**Description:** This is the main class which contains design related information and functionality. It keeps track of created components, default design specifications etc.<br><br>**Public Member Functions**<br>```cpp
CSubSkel(CSkeletonProp *propComp, string strAppend = "") Constructor. Passes propComp to the base class. If strAppend is not empty (the default value), the generated VHDL code is placed in the specified file rather than in a separate file. The file header is only generated if the component is placed in a separate file.
```

**Private/Protected Member Functions**<br>```cpp
void generateName() (see CComponent for details)
void generateArchitecture() (see CComponent for details)```
<table>
<thead>
<tr>
<th>Class Base Class</th>
<th>CDct</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Description</strong></td>
<td></td>
</tr>
<tr>
<td>This is the component base class for DCT blocks. DCT blocks consist of a number of stages, usually containing butterflies or similar operations. To enable pipelining, the generation is controlled by means of a common function, <code>archBody</code>, in the base class, keeping track of pipelining etc. Each derived class has its own <code>stage</code> function with code for the specific DCT.</td>
<td></td>
</tr>
</tbody>
</table>

**Public Member Functions**

- `CDct(CDctProp *propComp)`
  - Constructor. Takes a derived `CProperties` class as its only argument.
- `virtual void generateComponent(ofstream &out) = 0;`  
  - (see `CComponent` for details)

**Private/Protected Member Functions**

- `virtual void generateName() = 0`  
  - (see `CComponent` for details)
- `void generateHeader()`  
  - (see `CSkeleton` for details)
- `virtual void generateEntity()`  
  - Generates an entity for this component.
- `void dct2()`  
  - Generates the VHDL body for an arbitrary type 2-channel DCT block.
- `void dct3()`  
  - Generates VHDL body for an arbitrary type 3-channel DCT. Not implemented!
- `void archBody(int nStages)`  
  - All DCTs consist of a number of stages. This function controls what stage to generate and whether the stage should be followed by registers (pipelining).
- `virtual void stage(int &nStage) = 0`  
  - This abstract function contains the code to generate all stages in the DCT.
- `void internalSignals(int nStages)`  
  - Adds internal signals for `nStages` stages.
- `string S(int nStage, int nChannel, bool bPipe = false)`  
  - Generates a string for the specified signal.
- `string S(int nStage, string strIndex, bool bPipe = false)`  
  - Generates a string for the specified signal.
- `string I(int nChannel)`  
  - Generates a string for an input signal.
- `string O(int nChannel)`  
  - Generates a string for an output signal.
- `double COS(int i, int j)`  
  - Calculates \(\cos(i \pi / j)\).
- `double SIN(int i, int j)`  
  - Calculates \(\sin(i \pi / j)\).
Appendix B – Existing Components

In this appendix, all existing component categories and implementations are listed. Some implementations behave a little different from the general case as they are most likely optimized with a certain design in mind.

B.1 Basic Blocks
Basic blocks are small, elementary units used in virtually every design. Full adders and D flip-flops are examples of basic blocks. Basic blocks do not consist of components, but rather and, or and inverter gates.

B.1.1 Bit Adders
A full adder adds three bits: A, B and C\textsubscript{IN}, generating two outputs: SUM and C\textsubscript{OUT}. A half adds two bits, A and B. One output bit is thus generated. The last bit adder is the “last adder”, which is a full adder but with no carry out.

B.1.2 D Flip-Flop
Implements a simple D flip-flop.

B.2 Adders
This component category represents adders. To optimize the final implementation, general adders are not used.

B.2.1 Ripple-Carry Adder
Implements a ripple-carry adder. The first block is a half adder, and the consequent blocks full adders. A subtractor is implemented by first inverting one of the inputs (B) and use a full adder as the first block, with C\textsubscript{IN} tied to 1.

B.3 Butterflies
Butterflies are used extensively in DSPs. They are the hardware equivalence of a 2\times2 matrix. Many times, the signals are scaled to avoid overflow. This is facilitated in the butterflies implemented in this program as well. Both the input and output signals may be scaled.
B.3.1 General Butterfly
The general butterfly consists of four multiplications and two additions, as illustrated by the figure to the right.

B.3.2 Sande-Tukey Butterfly
This butterfly is used, for example, in DCTs. The inputs are added/subtracted cross-wise, i.e. \( U + V \) and \( U - V \). The sum, and the difference, is then multiplied with constants.

B.3.3 Optimized DCT-IV Butterfly
Instead of using the general butterfly when implementing a DCT-IV, it is possible to take advantage of the fact that the outputs are sums of \( \cos(j\pi) \) and \( \sin(j\pi) \), i.e.
\[
\begin{align*}
    x &= u \cdot \cos(j\pi) + v \cdot \sin(j\pi) \\
    y &= -u \cdot \cos(j\pi) + v \cdot \sin(j\pi)
\end{align*}
\]
The sums can be calculated in advance, and the structure in the figure to the right is obtained.

B.3.4 Multiply-First Butterfly
This is really just a different version of the general butterfly, but where the multipliers have been placed before the adders.

B.4 Fixed Multipliers
Multiplications are complex operations and an unoptimized general multiplicator uses a lot of space. One way to optimize multiplications is to use special multipliers when multiplying with a constant. By ignoring zeros and add shifted versions of the positions where the constant contains ones, very efficient multipliers can be accomplished.

B.4.1 CSDC Multiplier
CSDC multipliers use the CSDC number system to obtain what positions to add and subtract with. For example, 0.875 is represented by 01110000 in binary format. A multiplication would require two additions: 0.5 + 0.25 + 0.125. However, in CSDC format 0.875 is represented by +00-0000, i.e. 1 – 0.125, and thus only one subtraction is necessary.

B.4.2 Graph Multipliers
Graph multipliers are based on CSDC multipliers. The idea is that CSDC multiplications can be optimized further by using shifted versions of the produced sums and differences.
B.4.3 Negate
Negates a two-complement number by inverting the input and adding one. Implemented by using a series of halfadders, and a "last adder".

B.5 DCT Blocks
DCT blocks consist of $N$ inputs and $N$ outputs, where $N$ is the number of channels. There are numerous ways of implementing DCTs in hardware, and there are also different versions of the DCT. So far, only some of the DIT (Decimation In Time) and DIF (Decimation In Frequency) DCTs have been implemented, as well as a sparse-matrix DCT-IV. The main advantage of the DIT and DIF algorithms are their recursive nature. It makes it easy to implement a DCT block with arbitrary channels. However, and this is the major drawback, the algorithms are locked to power-of-two channels.

B.6 Registers
Registers consist of $N$ inputs and $N$ outputs. The only register implemented is a D register, built of D flip-flops.

B.7 Multiplier Blocks
Multiplier blocks consist of one input and $N$ outputs. The input signal is multiplied with different constants. Two different multiplier blocks have been implemented. One using separate fixed multipliers, e.g. CSDC multipliers or graph multipliers. The other uses the RAG-n algorithm.

B.8 Filters
The only implemented filter uses the transposed direct form to fully take advantage of the potential benefits of multiplier blocks. A filter has one input and one output.

B.9 Analysis Filter Bank
Analysis filter banks have one input and $N$ outputs. In this case, analysis filter banks is a somewhat misleading name, as it is really just the filter part. For example, in case of cosine-modulated filter banks, the cosine-modulation block is also part of the filter bank.
På 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 ick-ekommersiell 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.

Upphovsmannens 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 kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

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

In 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/

© Magnus Nord