UWB Imaging Reconstruction Unit for Breast Cancer Detection

BY

FRANCESCO COlonna
B.S., Politecnico di Torino, Turin, Italy, 2010
M.S., Politecnico di Torino, Turin, Italy, 2012

Thesis

Submitted as partial fulfillment of the requirements
for the degree of Master of Science in Electrical and Computer Engineering
in the Graduate College of the
University of Illinois at Chicago, 2013

Chicago, Illinois

Defense Committee:
Zhichun Zhu, Chair and Advisor
Dan Schonfeld
Mariagrazia Graziano, Politecnico di Torino
ACKNOWLEDGMENTS

As it usually happens when working on projects of the size and complexity as the one presented in this paper, even if I performed most of the work by myself there is a number of people whose help was critical to me to complete it over the relatively short period of time I needed without a nervous breakdown (at least not a major one).

Of course I would not have been able to obtain a result of this quality without prof. Mariagrazia Graziano and prof. Mario Casu, who offered me this opportunity in the first place and also challenged me to improve further and further my work with enlightening advice.

I need to thank my parents too, who gave me the opportunity to have such a high level education by supporting my in many matters (the economical one being not the least of them).

Finally I thank my girlfriend Francesca, who proved to be able of a tremendous amount of patience and tolerance as I have never seen before.

FC
TABLE OF CONTENTS

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>1 BREAST CANCER DETECTION</strong></td>
<td>1</td>
</tr>
<tr>
<td>1.1 Breast cancer facts</td>
<td>1</td>
</tr>
<tr>
<td>1.1.1 CBE</td>
<td>2</td>
</tr>
<tr>
<td>1.1.2 Mammography</td>
<td>2</td>
</tr>
<tr>
<td>1.1.3 Magnetic Resonance Imaging</td>
<td>3</td>
</tr>
<tr>
<td>1.1.4 Ultrasonography</td>
<td>4</td>
</tr>
<tr>
<td>1.2 Microwave imaging and the APIBUS project</td>
<td>4</td>
</tr>
<tr>
<td><strong>2 SYSTEM OVERVIEW</strong></td>
<td>9</td>
</tr>
<tr>
<td>2.1 Image Reconstruction algorithms</td>
<td>9</td>
</tr>
<tr>
<td>2.1.1 Skin Artifact Removal</td>
<td>10</td>
</tr>
<tr>
<td>2.1.2 MIST Beamforming</td>
<td>15</td>
</tr>
<tr>
<td>2.2 Algorithms implementation</td>
<td>21</td>
</tr>
<tr>
<td>2.2.1 Overall execution scheme</td>
<td>21</td>
</tr>
<tr>
<td>2.2.2 Hardware acceleration</td>
<td>24</td>
</tr>
<tr>
<td><strong>3 SOFTWARE OPTIMIZED IMPLEMENTATION</strong></td>
<td>27</td>
</tr>
<tr>
<td>3.1 SKAR</td>
<td>27</td>
</tr>
<tr>
<td>3.2 Beamforming</td>
<td>33</td>
</tr>
<tr>
<td><strong>4 SOFTWARE PERFORMANCE</strong></td>
<td>40</td>
</tr>
<tr>
<td>4.1 Parallelization</td>
<td>41</td>
</tr>
<tr>
<td>4.2 Other trends</td>
<td>47</td>
</tr>
<tr>
<td><strong>5 DESIGN LAYOUT</strong></td>
<td>51</td>
</tr>
<tr>
<td>5.1 Memories</td>
<td>53</td>
</tr>
<tr>
<td>5.2 FSM and counters</td>
<td>55</td>
</tr>
<tr>
<td>5.3 Computation Block</td>
<td>60</td>
</tr>
<tr>
<td>5.3.1 Delay</td>
<td>62</td>
</tr>
<tr>
<td>5.3.2 Window 1</td>
<td>63</td>
</tr>
<tr>
<td>5.3.3 FIR</td>
<td>64</td>
</tr>
<tr>
<td>5.3.4 Window 2</td>
<td>65</td>
</tr>
<tr>
<td>5.3.5 Energy Computation</td>
<td>66</td>
</tr>
<tr>
<td><strong>6 USB COMMUNICATION INTERFACE</strong></td>
<td>67</td>
</tr>
<tr>
<td>6.1 Hardware</td>
<td>68</td>
</tr>
<tr>
<td>6.2 Software</td>
<td>76</td>
</tr>
</tbody>
</table>
# TABLE OF CONTENTS (continued)

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>ACCURACY ESTIMATION</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>7</td>
<td>Quality factors</td>
<td>83</td>
</tr>
<tr>
<td>7.1</td>
<td>Contour conditions and parameters</td>
<td>83</td>
</tr>
<tr>
<td>7.2</td>
<td>Programmable support</td>
<td>85</td>
</tr>
<tr>
<td>7.2.1</td>
<td>Transmitters properties and tumor positions</td>
<td>87</td>
</tr>
<tr>
<td>7.2.2</td>
<td>SKAR</td>
<td>87</td>
</tr>
<tr>
<td>7.2.3</td>
<td>BEAF</td>
<td>90</td>
</tr>
<tr>
<td>7.2.4</td>
<td>Quantization</td>
<td>91</td>
</tr>
<tr>
<td>7.3</td>
<td>Accuracy performance</td>
<td>92</td>
</tr>
<tr>
<td>7.3.1</td>
<td>Ideal energy computation</td>
<td>93</td>
</tr>
<tr>
<td>7.3.2</td>
<td>Low resources approximation</td>
<td>98</td>
</tr>
<tr>
<td>7.3.3</td>
<td>Medium resources approximation</td>
<td>110</td>
</tr>
<tr>
<td>7.3.4</td>
<td>Full accuracy approximation</td>
<td>118</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>DESIGN FLEXIBILITY AND PERFORMANCE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
<td>Parametric implementation</td>
<td>124</td>
</tr>
<tr>
<td>8.1</td>
<td>Iterative energy computation concept</td>
<td>125</td>
</tr>
<tr>
<td>8.1.2</td>
<td>BEAF modifications</td>
<td>126</td>
</tr>
<tr>
<td>8.1.3</td>
<td>Memories</td>
<td>128</td>
</tr>
<tr>
<td>8.2</td>
<td>FPGA vs serial software</td>
<td>134</td>
</tr>
<tr>
<td>8.2.1</td>
<td>Direct measurement</td>
<td>136</td>
</tr>
<tr>
<td>8.2.2</td>
<td>Extrapolated timing</td>
<td>138</td>
</tr>
<tr>
<td>8.2.3</td>
<td>Acceleration trend</td>
<td>139</td>
</tr>
<tr>
<td>8.3</td>
<td>ASIC vs parallel software</td>
<td>142</td>
</tr>
<tr>
<td>8.3.1</td>
<td>Single accelerator</td>
<td>145</td>
</tr>
<tr>
<td>8.3.1.1</td>
<td>Masked memory access</td>
<td>145</td>
</tr>
<tr>
<td>8.3.2</td>
<td>Multiple accelerators</td>
<td>148</td>
</tr>
<tr>
<td>8.3.2.1</td>
<td>Multiple accelerators with masked memory access</td>
<td>152</td>
</tr>
<tr>
<td></td>
<td></td>
<td>154</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CHAPTER</th>
<th>CONCLUSIONS</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>9</td>
<td>Possible improvements and future work</td>
<td>162</td>
</tr>
<tr>
<td>9.1</td>
<td></td>
<td>163</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>APPENDICES</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>Appendix A</td>
<td>166</td>
</tr>
<tr>
<td>Appendix B</td>
<td>301</td>
</tr>
<tr>
<td>Appendix C</td>
<td>323</td>
</tr>
<tr>
<td>Appendix D</td>
<td>353</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>CITED LITERATURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>355</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>VITA</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>358</td>
</tr>
</tbody>
</table>
# LIST OF TABLES

<table>
<thead>
<tr>
<th>TABLE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>I</td>
<td>25</td>
</tr>
<tr>
<td>II</td>
<td>28</td>
</tr>
<tr>
<td>III</td>
<td>50</td>
</tr>
<tr>
<td>IV</td>
<td>88</td>
</tr>
<tr>
<td>V</td>
<td>88</td>
</tr>
<tr>
<td>VI</td>
<td>95</td>
</tr>
<tr>
<td>VII</td>
<td>99</td>
</tr>
<tr>
<td>VIII</td>
<td>99</td>
</tr>
<tr>
<td>IX</td>
<td>100</td>
</tr>
<tr>
<td>X</td>
<td>100</td>
</tr>
<tr>
<td>XI</td>
<td>105</td>
</tr>
<tr>
<td>XII</td>
<td>107</td>
</tr>
<tr>
<td>XIII</td>
<td>108</td>
</tr>
<tr>
<td>XIV</td>
<td>108</td>
</tr>
<tr>
<td>XV</td>
<td>112</td>
</tr>
<tr>
<td>TABLE</td>
<td>Description</td>
</tr>
<tr>
<td>--------</td>
<td>-----------------------------------------------------------------------------</td>
</tr>
<tr>
<td>XVI</td>
<td>SCR AND SMR COMPARISON BETWEEN IDEAL AND SECOND APPROXIMATED COMPUTATION</td>
</tr>
<tr>
<td>XVII</td>
<td>MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE SECOND APPROXIMATION STRATEGY</td>
</tr>
<tr>
<td>XVIII</td>
<td>SCR AND SMR COMPARISON BETWEEN IDEAL AND APPROXIMATED COMPUTATION, REAL BREAST MODEL</td>
</tr>
<tr>
<td>XIX</td>
<td>MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE SECOND APPROXIMATION STRATEGY</td>
</tr>
<tr>
<td>XX</td>
<td>TUMOR DETECTION OUTCOMES, SECOND APPROXIMATION</td>
</tr>
<tr>
<td>XXI</td>
<td>SCR AND SMR COMPARISON BETWEEN IDEAL AND MAXIMUM ACCURACY APPROXIMATED COMPUTATION, IDEAL BREAST MODEL</td>
</tr>
<tr>
<td>XXII</td>
<td>SCR AND SMR COMPARISON BETWEEN IDEAL AND MAXIMUM ACCURACY APPROXIMATED COMPUTATION, REAL BREAST MODEL</td>
</tr>
<tr>
<td>XXIII</td>
<td>MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE THIRD APPROXIMATION STRATEGY</td>
</tr>
<tr>
<td>XXIV</td>
<td>ASIC SPEED-UP OVER 4.2M VOXELS, WITH 50 AND 43 ANTENNAS AND 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916.25S AND 74945S RESPECTIVELY</td>
</tr>
<tr>
<td>XXV</td>
<td>ASIC SPEED-UP OVER 4.2M VOXELS WITH MASKED MEMORY ACCESS, WITH 50 ANTENNAS AND 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916.25S</td>
</tr>
<tr>
<td>XXVI</td>
<td>ASIC SPEED-UP OVER 4.2M VOXELS WITH MASKED MEMORY ACCESS, WITH 43 ANTENNAS AND 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 74945S</td>
</tr>
<tr>
<td>XXVII</td>
<td>ASIC SPEED-UP OVER 4.2M VOXELS WITH MULTIPLE ACCELERATORS, FOR 50 AND 43 ANTENNAS SYSTEMS AND WITH 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916S AND 74945S RESPECTIVELY</td>
</tr>
<tr>
<td>TABLE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>XXVIII</td>
<td>ASIC SPEED-UP OVER 4.2M VOXELS WITH MULTIPLE ACCELERATORS AND MASKED MEMORY ACCESS, FOR A 50 ANTENNAS SYSTEM AND WITH 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916.25S</td>
</tr>
<tr>
<td>XXIX</td>
<td>ASIC SPEED-UP OVER 4.2M VOXELS WITH MULTIPLE ACCELERATORS AND MASKED MEMORY ACCESS, FOR A 43 ANTENNAS SYSTEM AND WITH 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 74945S</td>
</tr>
</tbody>
</table>
## LIST OF FIGURES

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>2</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>7</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
</tr>
<tr>
<td>5</td>
<td>12</td>
</tr>
<tr>
<td>6</td>
<td>22</td>
</tr>
<tr>
<td>7</td>
<td>42</td>
</tr>
<tr>
<td>8</td>
<td>43</td>
</tr>
<tr>
<td>9</td>
<td>44</td>
</tr>
<tr>
<td>10</td>
<td>45</td>
</tr>
<tr>
<td>11</td>
<td>46</td>
</tr>
<tr>
<td>12</td>
<td>48</td>
</tr>
<tr>
<td>13</td>
<td>49</td>
</tr>
<tr>
<td>14</td>
<td>50</td>
</tr>
<tr>
<td>15</td>
<td>52</td>
</tr>
</tbody>
</table>

viii
### LIST OF FIGURES (continued)

<table>
<thead>
<tr>
<th>FIGURE</th>
<th>PAGE</th>
</tr>
</thead>
<tbody>
<tr>
<td>16</td>
<td>56</td>
</tr>
<tr>
<td>17</td>
<td>61</td>
</tr>
<tr>
<td>18</td>
<td>62</td>
</tr>
<tr>
<td>19</td>
<td>64</td>
</tr>
<tr>
<td>20</td>
<td>65</td>
</tr>
<tr>
<td>21</td>
<td>69</td>
</tr>
<tr>
<td>22</td>
<td>86</td>
</tr>
<tr>
<td>23</td>
<td>89</td>
</tr>
<tr>
<td>24</td>
<td>93</td>
</tr>
<tr>
<td>25</td>
<td>94</td>
</tr>
<tr>
<td>26</td>
<td>101</td>
</tr>
<tr>
<td>27</td>
<td>102</td>
</tr>
<tr>
<td>28</td>
<td>103</td>
</tr>
<tr>
<td>29</td>
<td>105</td>
</tr>
<tr>
<td>30</td>
<td>109</td>
</tr>
<tr>
<td>31</td>
<td>111</td>
</tr>
<tr>
<td>32</td>
<td>114</td>
</tr>
<tr>
<td>33</td>
<td>116</td>
</tr>
<tr>
<td>34</td>
<td>119</td>
</tr>
<tr>
<td>35</td>
<td>121</td>
</tr>
<tr>
<td>FIGURE</td>
<td>PAGE</td>
</tr>
<tr>
<td>--------</td>
<td>------</td>
</tr>
<tr>
<td>36</td>
<td>127</td>
</tr>
<tr>
<td>37</td>
<td>128</td>
</tr>
<tr>
<td>38</td>
<td>133</td>
</tr>
<tr>
<td>39</td>
<td>143</td>
</tr>
<tr>
<td>40</td>
<td>148</td>
</tr>
<tr>
<td>41</td>
<td>149</td>
</tr>
<tr>
<td>42</td>
<td>150</td>
</tr>
<tr>
<td>43</td>
<td>153</td>
</tr>
<tr>
<td>44</td>
<td>154</td>
</tr>
<tr>
<td>45</td>
<td>156</td>
</tr>
<tr>
<td>46</td>
<td>157</td>
</tr>
<tr>
<td>47</td>
<td>160</td>
</tr>
<tr>
<td>48</td>
<td>161</td>
</tr>
</tbody>
</table>
# LIST OF ABBREVIATIONS

<table>
<thead>
<tr>
<th>Abbreviation</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>BW</td>
<td>Bandwidth</td>
</tr>
<tr>
<td>BEAF</td>
<td>Mist Beamforming</td>
</tr>
<tr>
<td>SKAR</td>
<td>Skin ARtifact removal</td>
</tr>
</tbody>
</table>
SUMMARY

The Ultra-Wideband (UWB) imaging technique for breast cancer detection is based on the fact that cancerous cells have different dielectric characteristics than healthy tissues. When an UWB pulse in the microwave range strikes a cancerous region, the reflected signal is more intense than the back-scatter originating from the surrounding fat tissue. An UWB imaging system consists of transmitters, receivers and antennas for the RF part, and of a digital back-end for processing the received signals. In this paper we focus on the Imaging Unit, which elaborates the acquired data and produces 2D or 3D maps of reflected energies. We show that one of the processing tasks, Beamforming, is the most timing-critical and cannot be executed in software by a standard microprocessor in a reasonable time. We thus propose a specialized hardware accelerator for it. We optimize and parallelize the software implementation for a more even comparison. We design the accelerator in VHDL and test it in an FPGA-based prototype. We also evaluate its performance when implemented on a CMOS 45 nm ASIC technology. The speed-up with respect to a software implementation depends on the degree of parallelism permitted by the target technology.
CHAPTER 1

BREAST CANCER DETECTION

1.1 Breast cancer facts

Together with skin tumor, the breast cancer is the most common cancer pathology and the leading cause of cancer death among the female world population. During the last 30 years the incidence of this pathology rate has seen an overall increasing. However, the number of deaths has been decreasing thanks to the efficiency improvements of diagnosis and treatment techniques(3).

During its early stages, this pathology does not show significant symptoms making it very difficult to diagnose. At the same time, statistics show that when detected soon enough breast cancer can be treated with a very high success rate. This is why it is important to have efficient and flexible techniques to detect it, even in its first stages. In the last decade the awareness of the importance of a periodical screening among women, especially after the 40th year, has been improved through sensitization campaigns. Screening consist in a breast analysis performed when there is no sign of disease yet: according to the U.S. National Cancer Institute, regular use of screening mammograms, followed by timely treatment in case of diagnosed cancer, can help reduce the mortality of this pathology. For women between the ages of 50 and 69 screening lowers this risk by 30 percent. For women in their 40s, the risk can be reduced by about 17 percent(4).
The current most common screening technique is the mammography, that uses low energy X-Ray emission to obtain a high definition picture of the breast. Together with mammography various detection techniques are available, each having its pros and cons:

1.1.1 **CBE**

Clinical Breast Examination or CBE is the most basic screening approach. It consist in the physician’s tactile examination of the breast. This technique does not require any expensive equipment and can be performed anywhere. Of course, it is also very inaccurate giving very short information about the detected lumps, that can be identified only if they are reasonably big. This procedure is a basic routine that should be performed as part of general check-ups and in case of very young females that are at very low risk for breast cancer, but should not be considered a valid substitute to a periodic screening.

1.1.2 **Mammography**

This is the most widely used screening technology. By irradiating the patient’s breast with low energy X-Ray beams it is possible to obtain a detailed image with great resolution and contrast. The amount of radiation required (about 0.7mSv) is very low and does not present a risk per se. However, the risk can increase for very young women or for individuals whose job or life conditions expose them to radiations and in addition this method is not suitable for frequent controls since for each individual the maximum radiation exposure per year is strictly limited by the law (in the USA it is of 1mSv for members of the public and 50mSv for adults working with radioactive material(5)). The accuracy is quite good, and also tiny calcium deposit can be identified that usually indicate the presence of forming cancer tissue. Mammography is very
popular because its relatively affordable if compared with MRI but guarantees high image detail and definition. It has thus been promoted as the best screening technique currently available. This technique can however easily produce false positives, especially in young women, and there is a chance of false negative when the cancer is hidden by other dense tissue. As most of existing breast analysing technologies the mammography requires some expensive equipment. Last but not least, mammography is extremely uncomfortable for the patient, since it requires to apply strong pressure to the breast so that its volume is reduced as much as possible to increase the penetration factor.

1.1.3 Magnetic Resonance Imaging

Using magnetic field variation and radio frequency pulses this technique takes advantage of the nuclear magnetic resonance phenomenon. This is a step further in accuracy with respect to other techniques, producing cross sectional maps of the breast with higher reliability and definition. To do so, a small quantity of contrast material must be injected to increase accuracy. This exposes the patient to a very small chance of allergic reaction.

MRI requires very large and expensive equipment and because of that many hospitals and imaging centers do not have dedicated breast MRI equipment available. Also, MRI screenings should be done at facilities that are capable of performing an MRI-guided breast biopsy at the time of the exam if abnormalities are found. Otherwise, the scan must be repeated at another facility at the time of the biopsy. It is an expensive procedure, and sometimes it is not fully covered by insurance companies.

As a screening tool, it is suggested for women with high risk of breast tumor (based on pa-
rameters like age, ethnicity, medical history etc.) but should supplement and not replace the mammogram screening.

1.1.4 Ultrasonography

Using ultrasound waves in the 7-18 MHz range and analysing the breast’s reflections characteristics it is possible to obtain information on suspicious areas usually identified with a previous mammography. Even if supplying high contrast resolution, sometimes it is not possible to distinguish between cancer cells and fat tissue because of similar acoustic properties. In addition, ultrasonography does not image microcalcifications and tiny calcium deposit, that often are the first signs of breast cancer.

This approach is less accurate than mammography, and even if more suitable for young individuals since in does not impose any risk it should not be considered as a generally reliable screening procedure.

1.2 Microwave imaging and the APIBUS project

As we have seen, every existing technology for breast cancer screening has pros and cons, making it more or less suitable depending on the case we take in consideration. The APIBUS project, acronym of Accurate, Pervasive and Early, Integrated Breast Cancer Detection Ultra Wide Band Screening System, has as goal the development of a new technique to detect breast cancer by using microwave technology. This kind of technology represents a good compromise between resolution and penetration depth. As we will see, it has an accuracy that is comparable to the mammography, but does not imply any relevant risk for the patient, has the potential to be implemented with very low cost technology and does not cause any discomfort in the
Figure 1. APIBUS system diagram

Figure 2. APIBUS block scheme
patient. All this facts together make this new approach a valid candidate as leading breast screening approach in the future.

Generally speaking, various microwave-based possible solutions may be applied to investigate properties of the human body. There are both passive (thermography) and active techniques, and the active ones can be based either on the tissues abilities to scatter or absorb the waves. For the aim of this project, a Beam-Steering Radar Technique has been implemented: the basic concept consists in irradiating the breast with Ultra-Wideband electromagnetic impulses and analysing the backscattered signals in order to obtain information about the breast. Ultra Wideband signals are very short radio pulses with a very large bandwidth (greater than 500 MHz), hence the name. This kind of signals has good penetration ability without being anyway dangerous for the people nearby the emitter. Even if the penetration of this pulses is not very high, they are suitable for applications like breast analysis for which the required penetration is relatively low.

The reason why the breast interaction with radio signals is so interesting is that the dielectric properties of the normal tissue are different with respect to cancer tissue, as shown in Figure 3 obtained from(1). Because of that when in the breast there is a lump of tumor cells this creates a discontinuity between two substances with different dielectric values. When the ultra wideband pulse reaches this discontinuity it is partially backscattered, generating a reflected pulse that can be detected by receiving antennas. Once the reflected pulse has been sampled it is possible to extrapolate information about the position of the tumor.

Depending on how the antennas are subdivided into emitters and receivers, there can be three
kind of configurations: monostatic, multistatic and bistatic. In the monostatic configuration, a single antenna is used both as emitter and receiver. In the bistatic case, there is an emitter and a set of separate receivers, while in the multistatic case also the emitter is used to sample the reflected signal. Multistatic and bistatic configurations offer better Signal to Clutter Ratio (tumor over normal tissue response, as explained later) at the expense of increased complexity.

The project structure developed so far can be organized in blocks, identified by their role in the breast analysis process:

- Transmitter(6; 7; 8);
- Receiver(9; 10);
- Communication Bus(9);
• FDTD (electromagnetic simulation of breast model)(11);

• Image Reconstruction Unit(12).

This elements have been studied and designed in previous papers. This specific paper describes the hardware design process of the Image Reconstruction Unit. Note that, since a complete functioning system has not yet been implemented, the Image Reconstruction Unit has been tested over repository data(13) that include breasts dielectric characteristics used to simulate the breast behaviour when irradiate with microwave pulses(12).
CHAPTER 2

SYSTEM OVERVIEW

As previously explained, this breast cancer detection system is based on a Beam-Steering Radar Technique. In this scenario the goal of the image reconstruction unit is, starting from the sampled backscattered signals, to recombine this values in order to obtain an energy map representing the amount of energy reflected from each point of the breast. This paper largely relies on the analysis performed in a previous work(12) to confirm the validity of this design concept. The possible design choices for this kind of device were all compared there, and some of them were identified to be the most suitable for the case in analysis. Throughout this paper some design parameters will be given with brief motivation: the interested reader is invited to verify the detailed analysis of the benefits of this choices in (12). Some significant portions of the following algorithms derive from another paper(2), whose main concepts are here presented.

2.1 Image Reconstruction algorithms

In literature there is a variety of possible algorithms to extrapolate images from radar-like sampled signals. First of all, let’s try to understand the context in which the Image Reconstruction Unit is used.

Once the UWB pulse has been emitted and a number of samples has been collected from the receivers, there will be a set of scattered signals each of which associated with one of the receivers.
Assuming there is some kind of discontinuity in the space in front of the emitter this sampled signals will be similar to the original one, except for some modest but significant alterations. If we consider the peak of the pulse, the time at which its reflection reaches one of the receivers depends on the distance the signal has to travel from the emitter to the dielectric discontinuity and from this back to the receiver. This gives us the possibility to compute the distance of this discontinuity from each of the receivers and to triangulate its position by combining this information. Moreover, since we know the distance of each point of the breast from transmitter and receiver, we can compute the overall energy reflected (if any) from each point of the breast by processing and adding every sampled signal.

In this way, a complete map of the energy reflected by the breast can be produced. This map is computed as a series of values associated to points in the space (or "voxels"). This procedure can be applied to obtain either two or even three dimensional maps, at the expense of increasing algorithm complexity.

To correctly process the samples a series of algorithms must be applied, having different purposes. Here we define $N$ as the number of receivers in the design, and $b_i[n]$ as the sampled waveform associated with the $i^{th}$ receiver.

2.1.1 Skin Artifact Removal

As said, the possibility of identifying the tumor position derives from the fact that discontinuity between materials with different dielectric characteristics causes the pulses to be partially reflected. This highlights an issue: in the space the pulse has to cross there is another discontinuity besides from the cancer tissue, that is the breast skin. When reaching the skin,
part of the wave is backscattered because of the difference of dielectric constant between the air and the skin. This unwanted reflections distort the signal adding components that are orders of magnitude larger than the portion of signal we are interested in as shown in Figure 5 and need thus to be removed before the data is furtherly processed. In addition, since we only want to consider the portion of signal reflected from the tumor we should exclude signals travelling through the emitter-receiver direct path. This undesired contributions are orders of magnitude larger than the response from any other source.

For the purpose of this project, the Skin Artifact Removal (SKAR) algorithm was implemented(2). If the skin-breast artifacts were identical for all the N channels, they could be removed by subtracting the average of the signals across the channels. Unfortunately artifacts
are similar but not identical due to local variations in skin thickness and breast heterogeneity. This algorithm generalizes the previous idea and computes for each channel the skin-breast artifact by filtering and combining the signals coming from all the other channels, compensating thus for channel-to-channel variation (see Figure 4).

Let’s now analyse the procedure that allows to compute the filters’ weights. For the sake of simplicity, let us consider just the filter of the first channel. For any $n_{th}$ sample the skin-breast response is computed from a window of $2J + 1$ samples centered on the $n_{th}$ one coming from
each of the other \( N - 1 \) channels. For each of this channels, we define a vector containing this set of samples:

\[
\mathbf{b}_i[n] = [b_i[n - J], ..., b_i[n], ..., b_i[n + J]]^T, 2 \leq i \leq N
\]  

(2.1)

and then a concatenation of this vectors \( \mathbf{b}_{2N}[n] = [\mathbf{b}_2^T[n], ..., \mathbf{b}_N^T[n]]^T \). In the same way, we define also \( \mathbf{q}_i \) to be a vector of coefficients for the FIR filter of size \((2J + 1)\) associated with the \( i \)th channel and concatenate this vectors from channel 2 to \( N \) producing vector \( \mathbf{q} = [\mathbf{q}_2^T, ..., \mathbf{q}_N^T]^T \). This coefficients have to be selected -squared error. From this consideration, we deduce that the coefficients have to satisfy the following condition:

\[
\mathbf{q} = \arg\min_{\mathbf{q}} \sum_{n=n_0}^{n_0+m-1} |b_1[n] - \mathbf{q}^T \mathbf{b}_{2N}[n]|^2
\]  

(2.2)

where the \( n_0 \leq n \leq n_0 + m - 1 \) interval is the initial portion of the data record containing unwanted artifacts and no backscattered signals from lesions. Solution of this problem is:

\[
\mathbf{q} = \mathbf{R}^{-1} \mathbf{p} \quad (2.3)
\]

\[
\mathbf{R} = \frac{1}{m} \sum_{n=n_0}^{n_0+m-1} \mathbf{b}_{2N}[n] \mathbf{b}_{2N}^T[n] \quad (2.4)
\]

\[
\mathbf{p} = \frac{1}{m} \sum_{n=n_0}^{n_0+m-1} \mathbf{b}_{2N}[n] \quad (2.5)
\]

Since there is high similarity between the \( N \) channels, the variance matrix \( \mathbf{B} \) is likely to be ill-conditioned, so that the matrix inversion could produce a \( \mathbf{q} \) that eventually amplifies noise
because of a large norm. We then have to replace \( R \) with the approximated \( R_p \). The matrix inversion operation becomes:

\[
R_p = \sum_{i=1}^{p} \lambda_i u_i u_i^T \quad (2.6)
\]

\[
R_p^{-1} = \sum_{i=1}^{p} \frac{1}{\lambda_i} u_i u_i^T \quad (2.7)
\]

where \( \lambda_i \), \((1 \leq i \leq p)\) are the \( p \) significant eigenvalues and \( u_i \), \((1 \leq i \leq p)\) are the corresponding eigenvectors.

To remove artifacts from the sampled data, we perform the following operation:

\[
x_1[n] = b_1[n] - q^T b_{2N}[n] \quad (2.8)
\]

If we consider the sampled signal \( b_1[n] \) as the sum of the skin-breast artifact component \( s_1[n] \) and the residual component \( d_1[n] \) we see that a distortion occurs in the computation of \( x_1[n] \):

\[
x_1[n] = s_1[n] - q^T s_{2N}[n] + d_1[n] - q^T d_{2N}[n] \approx d_1[n] + q^T d_{2N}[n] \quad (2.9)
\]

Even if this distortion is usually small, we could reduce it if the residual \( d_{2N} \) were available by filtering it with \( q \) and adding to \( x_1[n] \). We can approximate this operation by using the data from all the other \( N-1 \) channels after the first skin-breast artifact removal operation \( x_{2N}[n] \) instead of \( d_{2N} \):

\[
\tilde{x}_1[n] = x_1[n] + q^T x_{2N}[n] \quad (2.10)
\]
The algorithm so far has been presented as in (2) for a monostatic application, and in that case it behaves as expected. Extending it to a multistatic configuration, however, requires some expedients. First of all, good performance can be obtained by increasing the $J$ parameter. The drawback is that, as explained in next section, increasing $J$ (that is increasing the number of filters’ weights) means increasing the complexity of the weights finder with a ratio that is $O(J^3)$. Moreover, the $m$ parameter needs to be large enough to exclude unwanted components also from signals sampled from antennas that are far from the transmitter. Unfortunately this could lead to a loss of precious information regarding tissue right beneath the skin from other signals. A simple but useful workaround is to group together the signals that have similar early-stage artifacts, as proposed by our workgroup in (14). A suitable value of $m$ is than assigned to each of this groups.

2.1.2 MIST Beamforming

Once cleaned the signal from unwanted reflections, we use the MIST algorithm to obtain an image of backscattered energy. This algorithm applies a space-time beamformer specific for each scan location (voxel) to the signals sampled from each antenna. The aim is to keep the contribution to the signal due to reflections coming from that specific location while minimizing contributions from any other location.

Let’s now consider the contribution from a single location $r_0$ assuming the $i_{th}$ channel’s signal only contains backscatter from that source for simplicity. We denote this signal as $x_i[n]$ and its Fourier transform as:

$$X_i(\omega) = I(\omega)S_{ii}(r_0, \omega), 1 \leq i \leq N$$  (2.11)
Here \( i(t) \) is the signal used to irradiate the breast, \( I(\omega) \) is its Fourier transform and \( S_{ii}(r_0, \omega) \) is an expression of the monostatic frequency response that characterize the round trip path that goes from the \( i_{th} \) antenna to \( r_0 \) and back. Remember that the distances of each antenna from \( r_0 \) may not be the same. We then need to time-align the signals by delaying them by an integer number of samples \( n_i(r_0) = n_a - \tau_i(r_0) \) where \( \tau_i \) is the round trip propagation delay from the \( i_{th} \) antenna and location \( r_0 \), obtained by considering an average propagation speed and the round trip path length which is known, and \( n_a \) is chosen to be the worst case delay considering all the possible combinations of scan location and antennas and is used as reference to align all the signals: \( n_a \geq \text{round}(\max_i, r_0 \tau_i(r_0)) \). The signal is windowed since the samples before \( n_a \) are not of our interest and do not contain useful informations:

\[
g[n] = \begin{cases} 1, & n \geq n_a \\ 0, & \text{otherwise} \end{cases} \tag{2.12}
\]

After this coarse time alignment, residual fractional delays could persist. To compensate for them and also to account for dispersion and attenuation related to the path length and to filter the signal’s band, the FIR filters are needed. A vector of weights \( w_i = [w_{i0}, w_{i1}, ..., w_{i(L-1)}] \) is associated with each of this filters. Increasing the number of weights improves the performance but increases complexity too, so this number must be chosen by carefully exploring this trade-
off. If we define \( w_{i t} \) as the \( t^{th} \) weight of the \( i^{th} \) filter and \( n_i(r_0) \) as the delay value associated with the \( i^{th} \) filter, we can obtain the sum of each filter as:

\[
z[n, r_0] = \sum_{i=1}^{N} \sum_{l=0}^{L} w_{i t} \cdot \tilde{x}_i[n - l - n_i(r_0)]
\]  \hspace{1cm} (2.13)

We then define a window:

\[
h[n, r_0] = \begin{cases} 
1, & n_h \leq n \leq n_h + l_h \\
0, & \text{otherwise}
\end{cases}
\]  \hspace{1cm} (2.14)

The interval between \( n_h \) and \( n_h + l_h \) identifies the window of samples in which we expect to find the pulse. Even if we know exactly the shape of the emitted signal, we should consider that scattering from the tumor is frequency-dependent, so the backscattered signal is a distorted version of the transmitted pulse. The choice of these values is suggested in (2) to maximize the contrast for small lesions. We can now compute the energy as the square of the windowed values:

\[
p(r_0) = \sum_n |z[n]h[n, r_0]|^2
\]  \hspace{1cm} (2.15)
The computation of the BEAF beamforming FIR filters could be done either in time(2) or frequency(15; ?) domain. The last one is to be preferred since it does not involve matrix inversion operations. The frequency response of the filters can be expressed as:

$$W_i(\omega) = \sum_{l=0}^{L-1} w_i e^{-j\omega l T_s} = w_i^T d(\omega)$$

(2.16)

where $d(\omega) = [1, e^{-j\omega T_s}, ..., e^{-j\omega(L-1)T_s}]^T$ and $T_s$ is the sampling time interval. Frequency resolution for weights design is limited by maximum length of FIR filter L:

$$\Delta \omega = \frac{2\pi}{LT_s}$$

(2.17)

Considering that the filter is designed for UWB signals, we can define a set of $K$ frequencies of interest delimited by a lower and an upper pulsation bounds:

$$\omega_l = \omega_0 \leq \omega_k \leq \omega_{K-1} = \omega_h, \quad 0 \leq k \leq K-1$$

(2.18)

If we define $\omega_l$ and $\omega_h$ as integer multiples of $\Delta \omega$ we get that:

$$K = \frac{\omega_h - \omega_l}{\Delta \omega} + 1$$

(2.19)

Supposing a monostatic configuration, we can express the frequency transformation of the backscattered signal as:

$$B_i(\omega_k) = I(\omega_k) S_{ii}(\omega_k, r_0)$$

(2.20)
Each value $S_{ii}(\omega_k, r_0)$ corresponds to the ideal scattering value in frequency domain for each voxel, while $I(\omega_k)$ is the portion of the transmitted signal belonging to the $k^{th}$ frequency in the defined frequency interval. In a monostatic configuration the frequency response can be formulated considering radial spreading, path loss and phase shift. Depending on if the scanning is performed in two or three dimensions formulas are:

$$3D : S_{ii}(\omega_k, r_0) = \left[ \frac{1}{|r_0 - r_i|} e^{\alpha(\omega_k)|r_0 - r_i|} e^{\beta(\omega_k)|r_0 - r_i|} \right]^2$$  \hspace{1cm} (2.21)

$$2D : S_{ii}(\omega_k, r_0) = \left[ \frac{1}{|r_0 - r_i|^{0.5}} e^{\alpha(\omega_k)|r_0 - r_i|} e^{\beta(\omega_k)|r_0 - r_i|} \right]^2$$  \hspace{1cm} (2.22)

Attenuation factor $\alpha(\omega)$ and phase constant $\beta(\omega)$ are function of dielectric characteristics of the medium(16).

Since weights are designed to have unit gain and linear phase response, we get that:

$$I(\omega_k) \sum_{i=1}^{M} S_{ii}(\omega_k, r_0) e^{j\omega_k|\tau_i(r_0/T_s)|T_s} W^*[k, r_0] = e^{j\omega_k\tau_0}$$  \hspace{1cm} (2.23)

$e^{j\omega_k|\tau_i(r_0/T_s)|T_s}$ represents the coarse time alignment, while $\tau_0 = (L - 1)T_s/2$ is the average time delay introduced by the filters. The formula above can be split in two contributes: $V(\omega_k, r_0)$ that includes effects of pulse shape, coarse time alignment and propagation, and $W[k, r_0]$ that is the vector of weights associated to each signal at a given frequency.

$$\tilde{S}_{ii}(\omega_k, r_0) = S_{ii}(\omega_k, r_0) e^{j\omega_k|\tau_i(r_0)/T_s|T_s}$$  \hspace{1cm} (2.24)
\[ V(\omega_k, r_0) = I(\omega_k)\{\hat{S}_{ii}(\omega_k, r_0)\}_{i=1}^N \]  
(2.25)

\[ W[k, r_0] = \{[W_i[k, r_0]]\}_{i=1}^N \]  
(2.26)

\[ W[k, r_0] = \frac{V(\omega_k, r_0)e^{j\omega_k\tau_0}}{V^H(\omega_k, r_0)V(\omega_k, r_0)} \]  
(2.27)

Note that this solution can lead to errors because of amplified noise when the denominator is small. Higher frequencies and deeper scan location are prone to this kind of misbehaviour. To have an estimate of the noise performance of the algorithm, we can consider the white noise gain values: white noise gain should be low to avoid noise amplification and errors between real case scenario and modelled effects. Through a penalized least squares approach(15) this design error can be hold down. The new equation obtained is:

\[ W[k, r_0] = \arg\min_{W[k, r_0]} \{|W^H[k, r_0]V(\omega_k, r_0) - e^{-j\omega_k\tau_0}|^2 + \xi W^H[k, r_0]\Lambda[k, r_0]W[k, r_0]\} \]  
(2.28)

The matrix \( \Lambda[k, r_0] \) contains the structure of penalty. Various kind of matrix can be chosen to perform this operation. As discussed in (15) and verified in (12), in a comparison between identity matrix, diagonal matrix and square diagonal matrix, the second one resulted to be the best choice for the case in analysis.

\[ W_i[k, r_0] = \frac{V_i(\omega_k, r_0)e^{j\omega_k\tau_0}}{|V_i(\omega_k, r_0)|\left(\xi + \sum_{j=1}^N |V_j(\omega_k, r_0)|\right)}, \quad \Lambda_{ii}[k, r_0] = |V_i[k, r_0]| \]  
(2.29)
With this choice all antennas give the same contributions in the analysis of a single voxel and gain of deeper points is increased. As a last step, the weights in time domain are computed through the IFFT producing a set of vectors of size $L$.

Examples in literature show the BEAF algorithm application to backscattered signal in a case of 4\textit{mm} in a constant breast tissue and shape obtained from repository(17), with a monostatic configuration of 9 antennas and 1\textit{mm} resolution. This solution can be easily extended to multistatic configuration by simply considering the round-trip distance as composed of two contributes: the distance between emitting antenna and scanning point, and the distance between scanning point and receiving antenna.

$$d_{(i,j)} = d_{lat} + d_{tar} \tag{2.30}$$

2.2 Algorithms implementation

A study on the possible implementation choices to design a complete scanning system has been carried on in previous papers(12). Here a synthetic summary of the results of this analysis is proposed.

2.2.1 Overall execution scheme

The complete scanning process can be schematized as in Figure 6. Let’s now see in detail the purpose of each portion of the execution flow:

1. **Acquisition:**

   In this phase the breast is irradiated with pulses and the backscattered signals are gath-
Figure 6. Block scheme of the breast scan operation

erred from the antennas into memory. Considering the UWB frequency band (1GHz to 10GHz) it is possible to deduce the required sampling frequency. According to literature(2) a sampling frequency of $f_c = 50GHz$ was considered. To have good noise performance, a 500MHz ADC was chosen that ensures good SNDR (about 60dB). The difference between the desired sampling frequency and actual ADC frequency forces us to use Coherent Equivalent Time Sampling(9; 18). Sampling need thus to be repeated a number of times equal to:

$$M = \frac{f_c}{f_{ADC}} = 100$$  \hspace{1cm} (2.31)

To obtain better precision, we iterate this process an arbitrary number of times $N_{ave} = 64$ and then compute an average between the results. To obtain the complete sampled signals from each antenna the scan process has thus to be repeated $N_{TX}' = M \ast N_{ave} = 6400$ times.
2. **Weights-SKAR:**

We define with this name the process that produces the SKAR algorithm’s weights in a way similar to the one previously cited for BEAF weights. This operation needs to be performed only once for each scan.

3. **SKAR:**

This identifies the actual application of SKAR algorithm with the computed weights to obtain signals without unwanted skin artifacts.

4. **Weights-BEAF:**

In a similar way to the Weights-SKAR step, this is the phase in which the weights for the BEAF FIR filters are computed. This operation needs to be performed only una tantum during the device calibration.

5. **Weights-BEAF access**

We separate the memory access operation to retrieve the BEAF weights from the actual BEAF algorithm. This is because this phase is quite time expensive and can become a bottleneck in some circumstances. If we consider a scenario with \( L = 55 \) weights of 16\textit{bit} per filter, 50 antennas and a worst case number of voxels(13) \( N_{\text{vox}} = 4.25 \cdot 10^6 \) we deduce that we need an amount of memory equal to:

\[
W_{BEAF} = L \cdot N \cdot 16\text{bit} \cdot N_{\text{vox}} = 24\text{GB}yte
\]  
\( (2.32) \)
This means that we would probably need an hard disk to store the BEAF weights, making the access slow and thus critical.

6. **BEAF** In this operation the actual BEAF algorithm is applied, producing the energy evaluation for each voxel.

While the other operations must be executed only once, Weights-BEAF access and BEAF must be repeated for each voxel.

### 2.2.2 Hardware acceleration

All the operations and algorithms mentioned were analysed to identify the implementation technique most suitable to our case of study. Operations could be either executed via software by a general purpose machine, or could be implemented on a custom device obtained via FPGA programming or ASIC implementation.

Complete software simulation of the Image Reconstruction Unit was performed in (12) using Matlab/Octave software. Together with this software simulation, some reasonable estimates of the possible hardware performance were made based on a very essential preliminary VHDL implementation and on an analysis regarding the amount of computation needed to perform the discussed algorithms.

Table I presents this results. For hardware implementation of SKAR and BEAF operations’ two cases were identified: a parallel one in which every antenna channel is associated to a dedicated filter and than all of the signals can be processed in parallel, and a serial one in which a single filter is used repeatedly to process all of the signals one after the other. Of course this two possibilities represent a trade-off between computation speed and required resources (this
subject is explored more deeply in Chapter 8).

The hardware execution times shown in Table I were obtained by roughly estimating the memory access time and the amount of algebraic operations required to perform the image reconstruction and considering the typical frequency available on the selected hardware supports. We see that hardware implementation can boost timing performance of some of the algorithms by some orders of magnitude. At the same time, implementing every part of the system on a dedicated hardware would require more resources increasing both design time and required resources for a speed up that may not match the effort required.

<table>
<thead>
<tr>
<th></th>
<th>SW</th>
<th>FPGA 200MHz</th>
<th>ASIC 500MHz</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>ACQUISITION</strong></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>W-SKAR</td>
<td>100ms</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>SKAR</td>
<td></td>
<td>0.25 s</td>
<td>1.31µs</td>
</tr>
<tr>
<td></td>
<td></td>
<td>65.5µs</td>
<td>0.56µs</td>
</tr>
<tr>
<td>W-BEAF</td>
<td>2.17ms/Vox</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td></td>
<td>2.6h</td>
<td>-</td>
<td>-</td>
</tr>
<tr>
<td>W-BEAF</td>
<td>7.32µs/Vox</td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACCESS</td>
<td>31 s</td>
<td></td>
<td></td>
</tr>
<tr>
<td>BEAF</td>
<td>4.88ms</td>
<td>2.52µs</td>
<td>1.01µs</td>
</tr>
<tr>
<td></td>
<td>126µs</td>
<td>50.5µs</td>
<td></td>
</tr>
<tr>
<td></td>
<td>10.7s</td>
<td>4.3s</td>
<td></td>
</tr>
<tr>
<td></td>
<td>9min</td>
<td>3.6min</td>
<td></td>
</tr>
</tbody>
</table>
For example, both the W-SKAR and SKAR operations are performed only once at the beginning of the scan operation, and thus impact lightly on the total execution time. W-BEAF operation is quite time consuming, but it is executed only una tantum for system calibration and so it does not impact the scan time.

On the other hand, BEAF algorithm is the one that would more benefit from an optimization: overall it is the most time consuming and being repeated many times (one for each voxel) even a small acceleration would multiply proportionally.

This considerations suggest that implementing this design in an optimized form would be very interesting, since it would give use more accurate informations for a real life application of this system. Since the software implementation we have been using as a reference in the previous table is not really an optimal one, it is important to produce a high performance version so that we can understand the actual advantages of a custom hardware design (if any). For this purpose in Chapter 3 an optimized C language program that performs the image reconstruction operations is presented, while 4 shows its performance.

A full VHDL hardware design of the most time critical portion of the image reconstruction, which is the BEAF as verified in 4, is then outlined in Chapter 5. Finally, a full comparison of software and hardware performance is presented in Chapter 8 so that we can present some conclusive consideration about the validity each of the explored designs.
CHAPTER 3

SOFTWARE OPTIMIZED IMPLEMENTATION

To evenly compare the hardware and software implementations of the Beamforming algorithm, we should consider that the software performances considered so far refer to a Octave/-Matlab implementation. Matlab is an interpreted language: this means that the scripts’ code is executed by an interpreter program that interprets it at runtime. This kind of languages is much slower than compiled languages such as the C language, that are converted to machine instructions at compile time. To evenly compare the performance of a hardware implementation versus a software one, we should implement the software program in the most efficient way so that we can really estimate the advantages of one implementation over the other.

Moreover as we will see the algorithms we are talking about have a good degree of parallelism: the energy computation procedure for each of the voxel is completely independent from all the other voxels. This means that we could improve the software execution time by executing the algorithm in parallel over multiple processors. In this chapter we discuss a C language implementation of the whole energy computation process. In Table II a list of the constants and parameters used in the program is presented.

3.1 SKAR

Let’s see how the SKAR calibration of the samples was implemented. At the beginning of the execution we suppose to have all the reflected samples available for the computation. In
TABLE II
CONSTANTS FOR SOFTWARE BREAST MAPPING

<table>
<thead>
<tr>
<th>CONSTANTS</th>
<th>NCAMP_fDTD</th>
<th>4696</th>
</tr>
</thead>
<tbody>
<tr>
<td>number of samples of input signals</td>
<td>NCAMP_fDTD</td>
<td>4696</td>
</tr>
<tr>
<td>Space increment of square lattice (cm)</td>
<td>DX_FDTD</td>
<td>0.05</td>
</tr>
<tr>
<td>beamforming sampling frequency (GHz)</td>
<td>FC</td>
<td>50</td>
</tr>
<tr>
<td>beamforming resolution (cm)</td>
<td>RES</td>
<td>0.1</td>
</tr>
<tr>
<td>light speed in space (cm/s)</td>
<td>C0</td>
<td>2.99792458E10</td>
</tr>
<tr>
<td>light speed in breast (cm/s)</td>
<td>VF</td>
<td>0.95E10</td>
</tr>
<tr>
<td>Time step FDTD</td>
<td>DT_FDTD</td>
<td>DX_FDTD/(2.0 · C0)</td>
</tr>
<tr>
<td>vacuum permeability (Vs/Am)</td>
<td>MU0</td>
<td>4.0 · M_PI · 1.0E−7</td>
</tr>
<tr>
<td>vacuum permittivity (F/m)</td>
<td>EPS0</td>
<td>1.0/(1E−4 · C0 · C0 · MU0)</td>
</tr>
<tr>
<td>phantom space resolution (cm)</td>
<td>RES_PHANTOM</td>
<td>0.05</td>
</tr>
<tr>
<td>decimation constant</td>
<td>DC</td>
<td>1/(DT_FDTD · FC · 1E9)</td>
</tr>
<tr>
<td>sampling frequency after decimation</td>
<td>FC_REAL</td>
<td>1/(DC · DT_FDTD)</td>
</tr>
<tr>
<td>Time step beamforming</td>
<td>TS</td>
<td>DT_FDTD · DC</td>
</tr>
<tr>
<td>Number of samples after decimation</td>
<td>NCAMP</td>
<td>floor(NCAMP_FDTD/DC)</td>
</tr>
<tr>
<td>Minimum UWB frequency</td>
<td>MINF UWB</td>
<td>0.5e9</td>
</tr>
<tr>
<td>Maximum UWB frequency</td>
<td>MAXF UWB</td>
<td>12e9</td>
</tr>
</tbody>
</table>

SKAR parameters

| First sample used in the weights evaluation | N0 | 30 |
| Last sample used in the weights evaluation | NFIN | 60 |

length of FIR filter (2J + 1)

| J | 3 |

BEAF parameters

| Number of weights | L | 55 |
| Start of samples window | NH | 252 |
| Size of samples window | LH | 5 |
| Start of pulse | LH | 5 |
| Penalty factor | EPSI | 4 |

our particular case, this samples were obtained by a FDTD simulation of the breast based on phantom repository(17). After this samples have been loaded to memory into a matrix called b, the weights computation for the SKAR algorithm is performed.

As explained in Chapter 2, a matrix here called R is obtained that contains a set of samples for each of the antennas in the system, each of which is a combination of the samples coming
from all the other antennas. Also a $p$ vector is computed that contains a set of coefficients.

Both matrix and vector are then normalized. This operations are shown in listing 3.1.

After matrix $R$ has been obtained, it needs to be inverted. To do so a popular function library named **CLAPACK** was included in the design. This library provides a set of functions that are useful to perform matrix operations, that were developed for Fortran language and then ported to C. To perform the inversion a function named **dsyevx** provides eigenvalues and eigenvectors
of matrix $R$ as shown in listing 3.2. As we see in listing 3.3 once obtained the inverse matrix $R_{p\text{inv}}$, by combining it with the $p$ coefficients we obtain all the weights needed to perform the calibration.

Listing 3.2. SKAR Weights computation

```c
jobz='V';
range = 'A';
uplo = 'U';
matorder = (2*J+1)*(nrx-1);
lda= (W)*(nrx-1);
ldz=(W)*(nrx-1);
il=1;
\text{iu=P}_EIGH;
\text{abstol = 0;}
ispec=1;
strcpy (tmp , "dsyevx");
strcpy (opts , "U");
v=20000;
work = (\text{double*}) malloc (\text{sizeof} (\text{double*})*v);
dsyevx(&jobz, &range, &uplo, &matorder, R, &lda, NULL, NULL, &il, &iu, &abstol, &neig, eigv, 
z, &ldz, work, &v, iwork, ifail, &info);
```

This way of performing the matrix inversion is probably not the most time efficient, since the CLAPACK library was not fully optimized for the test machine we are considering. In any case, we choose this solution because it was both based on open source code and easy to implement.

Once the weights are ready, the calibration itself begins: every sampled signal associated to an antenna is filtered by subtracting values that are the combination of the signals associated to all the other antennas with the computed weights, as explained in Chapter 2. This operation
Listing 3.3. SKAR Weights computation

```plaintext
for (i = 0; i < W * (nrx - 1); i++)
{
    for (j = 0; j < W * (nrx - 1); j++)
    {
        Rp_inv[i][j] = 0;
    }
}

for (n = W * (nrx - 1) - 1; n >= W * (nrx - 1) - PEIGH; n--)
{
    for (i = 0; i < W * (nrx - 1); i++)
    {
        for (j = 0; j < W * (nrx - 1); j++)
        {
            Rp_inv[i][j] += (z[n*W*(nrx - 1) + i] * z[n*W*(nrx - 1) + j]) / eigv[n];
        }
    }
}

for (i = 0; i < W * (nrx - 1); i++)
{
    qr[ant][i] = 0;
    for (j = 0; j < W * (nrx - 1); j++)
    {
        qr[ant][i] += Rp_inv[i][j] * p[j];
    }
}
```

is repeated a second time over the new calibrated signals in a slightly different way to reduce distortion. Listing 3.4 shows the instructions used to perform this calibration.

Listing 3.4. SKAR calibration instructions

```plaintext
for (ant = 0; ant < nrx; ant++)
{
    for (j = 0; j < NCAMPfin; j++)
    {
        if (j >= J)
        {
            start = -J;
        }
        else
        {
            start = -j;
        }
        x1[ant][j] = local[ant][j];
        for (k = 0; k < nrx - 1; k++)
```
{
    if (k < ant)
    {
        i_adj = 0;
    }
    else
    {
        i_adj = 1;
    }
    for (camp = start; camp < J && j + camp < NCAMPfin; camp++)
    {
        x1[ant][j] = local[k + i_adj][j + camp] * qr[ant][(k * W) + camp - start];
    }
}
}

for (ant = 0; ant < nx; ant++)
{
    for (j = 0; j < NCAMPfin; j++)
    {
        if (j > J)
        {
            start = -J;
        }
        else
        {
            start = -j;
        }
        x2[ant][j] = x1[ant][j];
        for (k = 0; k < nx - 1; k++)
        {
            if (k < ant)
            {
                i_adj = 0;
            }
            else
            {
                i_adj = 1;
            }
            }
\begin{align*}
&\text{for (camp=start; camp<J && j+camp<NCAMPfin; camp++)} \\
&\quad \{ \\
&\quad \quad x2[\text{ant}][j] += qr[\text{ant}][(k*W)+\text{camp-start}] \times x1[k+i_{\text{adj}}][j+\text{camp}]; \\
&\quad \}
&\}
\end{align*}

3.2 Beamforming

To be able to perform the BEAF, we need to execute a series of preliminary operations that do not depend on the results of the calibration, since the BEAF weights only depend on the properties of the scan system and not on the breast. This operations can be found in listing 3.5.

\textbf{Listing 3.5. BEAF weights and delay computation}

\begin{verbatim}
wstep = 1/(L*TS) * 2*M_PI; // Resolution
wl = round (MINFUWB * 2*M_PI / wstep) * wstep; // lower angular frequency evaluated
wu = round (MAXFUWB * 2*M_PI / wstep) * wstep; // upper angular frequency evaluated
i_wl = round (wl / wstep); // lower angular frequency index
i_wu = round (wu / wstep); // upper angular frequency index
K_tot = i_wu - i_wl + 1; // number of frequencies evaluated
Lmid = ceil(L/2);
tau0 = TS*(L-1)/2;

// Debye equation and absorption evaluation
epsilon_inf = 7;
delta_epsilon = 3;
t_debye = 7.00e-12;
cond_debye = 0.15;

p = fftw_plan_dft_r2c_1d(L, &pulse[PULSE_START-1], pulsefreq, FFTW_ESTIMATE);
fftw_execute(p);
fftw_destroy_plan(p);
\end{verbatim}
MAXpath=0;
for (ant=0; ant<NRX; ant++)
{
    for (i=0; i<i_index; i++)
    {
        for (j=0; j<j_index; j++)
        {
            if (Lpath[ant][i][j]==2*RES*sqrt((i-receivers_pos2D[ant][0])*(i-receivers_pos2D[ant][0])+(j-receivers_pos2D[ant][1])*(j-receivers_pos2D[ant][1])))>
                MAXpath);
            MAXpath=Lpath[ant][i][j];
        }
    }
}
MAXpath=MAXpath/VF;
na=ceil(MAXpath/TS);
num_vox=0;
for (i=0; i<i_index; i++)
{
    for (j=0; j<j_index; j++)
    {
        if (phantom2D[i][j]>=0)
        {
            num_vox++;
        }
    }
}
weights_mat=(double****)malloc(sizeof(*weights_mat)*num_vox);
delays_mat=(int****)malloc(sizeof(*delays_mat)*num_vox);
energy_mat=(double****)malloc(sizeof(*energy_mat)*num_vox);
for (i=0; i<num_vox; i++)
{
    delays_mat[i]=(int*)malloc(sizeof(**delays_mat)*NRX);
    weights_mat[i]=(double*)malloc(sizeof(**weights_mat)*NRX);
    for (j=0; j<NRX; j++)
    {
        weights_mat[i][j]=(double*)malloc(sizeof(**weights_mat)*L);
    }
}
To be able to compute the BEAF weights, we made the choice to move from the time to the frequency domain to simplify the operations. To be able to do so, the library \texttt{fftw} was included in the design: this library includes a series of functions that allow to perform Discrete Fast Fourier Transform operation. The prototype of the function included in the design is the following:

\begin{verbatim}
fftw_plan fftw_plan_dft_r2c_1d(int n0, double *in, fftw_complex *out, unsigned flags)
\end{verbatim}

which plan a real-input/complex-output discrete Fourier transform (DFT) in one dimension, returning an \texttt{fftw_plan}.

As a first step the round trip time required from a signal to travel from each antenna to any of the breast point and back is computed, and the maximum of this values is stored. Using this travel times we can obtain the delay value that will be used to align the signals during beamforming. Then the exact number of voxels that are inside the breast shape is computed so that the size of the problem is known. Once this operations have been performed, we can start to actually compute the weights.

The weights computation is looped for each voxel inside the breast shape and for each antenna channel as in listing 3.6. First the weights are computed in the frequency domain as explained in Chapter 2, with a series of operations over complex numbers. Then they are transformed back to the time domain and reduced to real numbers.
Listing 3.6. BEAF weights and delay computation

```plaintext
vox=0;
for (i=0; i<i_index; i++)
{
    for (j=0; j<j_index; j++)
    {
        if (phantom2D[i][j] >= 0)
        {
            for (ant=0; ant<NRX; ant++)
            {
                tau=Lpath[ant][i][j]/VF;
                delays_mat[vox][ant]=nารound(tau/TS);
                for (k=0; k<K_tot; k++)
                {
                    Sa=cexp(-I*((wl+wstep*k)*Lpath[ant][i][j]/2)/VF);
                    epsilon_d=epsilon_inf + (delta_epsilon/(1+(1*(wl+wstep*k)*
                        tdebye)) + (cond_debye/(1*(wl+wstep*k)*EPS0));
                    alfa_abs=cabs(cimag(epsilon_d))*(wl+wstep*k)*(VF/(C0*C0));
                    // absorption coefficient
                    Sb=cexp(-alfa_abs*Lpath[ant][i][j])/2;
                    S=Sa*Sb*(1/sqrt(Lpath[ant][i][j])/2);
                    tauw=cexp((1*(wl+k*wstep))*round(tau/TS)*TS);
                    V[ant][k]=S*tauw*pulsefreq[i_wl+k];
                }
            }
        }
    }
}
for (k=0; k<K_tot; k++)
{
    sumV[k]=0;
    for (ant=0; ant<NRX; ant++)
    {
        sumV[k]+=V[ant][k];
    }
}
for (ant=0; ant<NRX; ant++)
{
    for (k=0; k<L; k++)
    {
        WeightfreqL[ant*L + k]=0;
    }
}
```

Once the weights are ready, we can perform the beamforming. As discussed in previous chapters, we expect this to be the most time consuming portion of the image reconstruction process. Since we want to accelerate this section of the program by implementing a dedicated hardware, we should also try to improve its C language implementation. To do so we just need to remember that the beamforming process has several degrees of parallelism, one of which is what we will call **voxel parallelism**. If we consider the energy computation process as described, it's
easy to see that the computation of the energy value for each voxel is completely independent from all the others: each energy value is computed combining the calibrated signals with a set of weights that is specific for each voxel, and there is no correlation among the execution of this operation for all the voxels. This means that ideally all the energy values could be computed in parallel at the same time, assuming that all weights and calibrated signals are available.

Listing 3.7. BEAF computation

```c
int BEAF(int num_thread) {
    unsigned int i, vox, ant, w, j;
    int r;
    double x;
    char d;
    clock_t cstart, cfinish;
    char weights_string[5], line[35];
    struct timespec ts_start;
    struct timespec ts_end;
    pthread_t *pid;
    PARMS *inputs;
    FILE *energyFile;

    pid=(pthread_t*)malloc((num_thread)*sizeof(*pid));
    inputs=(PARMS*)malloc((num_thread)*sizeof(*inputs));

    // Create N thread
    for (i=0; i < num_thread; i++) {
        // Initialize parameters
        inputs[i].threadnum = i;
        inputs[i].num_thread = num_thread;
        r = pThreadCreate(&pid[i], NULL, child, (void *) &inputs[i]);

        if (r) {
            printf("Error_creating_Thread_%d!", i);
            joining_fn (pid, i);
            return -1;
        }
    }

    // wait for all threads to finish
    joining_fn (pid, num_thread);
    clock_gettime(CLOCK_MONOTONIC, &ts_end);

    energyFile=fopen("energy_par_swcomplete.bin","wb");
    fwrite(energy_mat, 8, numvox, energyFile);
    fclose(energyFile);
    free(pid);
    free(inputs);
    return 0;
}
```
Thanks to this property of the algorithm, the whole BEAF operation was implemented so that it could be run over multiple processors. To do so, the `pthread` library was used to evenly divide the computation load over multiple threads. Depending on the number of threads the user intend to use, the problem is automatically divided into subsets each of which includes a portion of the breast voxels. Listing 3.7 shows how the launch of the threads was formulated in C, while Listing 3.8 shows the child function executed by each processor.

```
void * child(void * arg) {
    int i, j, ant, w, voxel;
    double tmp;
    PARMS * inputs = (PARMS *) arg;
    for (voxel=inputs->threadnum; voxel<num_vox; voxel=voxel+inputs->num_thread) {
        result=0;
        for (j=0; j<LH; j++)
            for (ant=0; ant<nrx; ant++)
                for (w=0; w<L; w++)
                    if (NH+j+w-delays_mat[voxel][ant]>na-delays_mat[voxel][ant])
                        tmp+=ct[ant][NH-L+w-delays_mat[voxel][ant]+j]*weights_mat[voxel][ant][w];
        result+=tmp*tmp;
    }
    energy_mat[voxel]=result;
    pthread_exit(NULL);
}
```
CHAPTER 4

SOFTWARE PERFORMANCE

To test the performance of the described program it was compiled and launched over a server machine running Linux with sixteen Intel Xeon E7-4870 CPUs with 2.40GHz clock frequency and 16.3 GB of RAM memory. To do so, the Gnu C Compiler was used with first level optimization and adequate linking of the additional libraries included in the code which were described above.

There are three main degrees of freedom that we can use to exploit the performance trend when applying this software to different system designs: the dimension of the map (i.e. the number of voxels to be processed), the number of threads used to execute in parallel the beamforming, and the number of antennas in the system. A series of simulations was performed by combining different values for this parameters. Since the maximum number of CPUs available in the system is sixteen, the number of used threads ranges from 1 to 16. The numbers of antennas considered for the simulations ranged between 9 and 50, which are the reference values used to test the Octave implementation we have seen before. The map dimensions used were 180000, 360000 and 720000, which because of memory constraints was the maximum amount of voxels we could process in a single execution of the program. Consider that in a real life case to obtain a map with a resolution of 1 mm we would have a number of voxels that could be around 4 millions.
4.1 Parallelization

By looking at the performance of the software in a single threaded case we can confirm the hypothesis we made in earlier chapters based on the Octave implementation performance that the Mist beamforming was the most relevant portion of the execution in terms of time. Figure 7, which shows the relative portion of the BEAF over the complete execution, highlights how most of the computation time is spent for the beamforming. The total time considered for the comparison is obtained from the sum of the BEAF, the SKAR calibration and its weight computation but excluding the computation of the weights for the BEAF algorithm: since this operation only depends on the scan device characteristics and must be executed only once for system calibration, in a real case application it would not need to be executed during a random breast scan.

As we may have expected, the portion of the execution time due to the beamforming is higher for large breast maps since it is the only recurrent operation that must be repeated for every voxel in the map. On the other hand the signals calibration depends quadratically on the number of sampled signals i.e. on the number of antennas in the system, since each channel’s signal is calibrated using a combination of all the others. That’s why with higher number of antennas the BEAF impact on the overall execution is slightly smaller, since it only depends linearly on the number of sampled signals (an exception is the maximum size map for a 50 antennas design, the reason will be presented later). Anyway we can say that in a serial case the BEAF is responsible for almost all the computation time. Since the SKAR and its weights computation are not affected by the number of voxels in the map it is no surprise that with
Figure 7. Portion of the BEAF time over the total execution as a function of the map size, for various numbers of antennas.
larger maps the BEAF impact on the performance is larger. This is also confirmed by Figure 8 that shows how the serial portion of the execution remains constant even if we change the map size.

We can see from Figure 9 and Figure 10 that for this serial implementation of the image reconstruction the impact on timing of SKAR and W-SKAR algorithms is pretty low, with any map size. If we compare this values to the ones we used as reference in Chapter 2 we see that actually implementing the software in C has improved the BEAF portion of the algorithms more than the other two. In any case, the BEAF is still responsible for almost all the execution time in this
serial implementation. However, in Figure 9 and Figure 10 some data regarding the parallelization of the BEAF algorithm is anticipated: we see that with a high number of threads used for the BEAF, the impact of the other portions of the image reconstruction grows and especially the W-SKAR part can get quite significant for a small map. Increasing the number of voxels anyway reduces this effect since it has impact on the BEAF but not on the other two algorithms.
Figure 10. Portion of the WSKAR time over the total execution as a function of the map size, for various numbers of parallel BEAF threads.
We are now interested in seeing how far parallelizing the beamforming execution can actually improve the image reconstruction performance. If we look at Figure 11 that only considers the large size map we notice that the speedup appears to have a linear trend, which means it is basically ideal.

Some interesting behaviour can be observed in case of a small number of antennas (9 or 20) when having a very large parallelism: in this case the speedup trend is slightly sublinear. This means that there is a portion of the BEAF execution that is not parallelized and constitutes an
overhead; the origin of this behaviour can probably be found in some serial time required by the OS to handle the threads and their initialization. When having a small number of antennas the amount of computation needed to perform the beamforming is significantly reduced, and so the portion of execution time due to this serial overheads becomes sufficiently large to have some small impact on the performance. On the other hand with a larger amount of computation (due to many antennas or low degree of parallelism) the serial overhead becomes neglectable.

4.2 Other trends

Some interesting information could be obtained by observing how the BEAF time changes when applying maps of different size, given a fixed number of threads. In figure Figure 12 this trend is shown for the first four values in the range of the possible number of antennas in case of 16 parallel threads. As expected, the dependence of the time required by the BEAF scales almost linearly with the number of voxels. A higher number of voxels actually only implies that the beamforming algorithm needs to be looped identical (except for the input data) for a higher number of iterations. Again in case of small number of antennas we can see a sublinear trend, which confirms our previous considerations.

If we include in the discussion also the results obtained for 50 antennas as in Figure 14, we see a strong change. When passing to the biggest map size the 50 antennas case shows an increase in the order of magnitude of the timing performance. This trend is very different from what we have seen so far and occurs with any number of threads used, as shown in Figure 13. The explanation is actually quite simple and again is related to the memory access: if we look
Figure 12. Performance trend with parallel 16 threads as a function of the map size, for four antennas sizes
to Table III that shows the amount of dynamic memory allocated to perform the BEAF, we see that with 50 antennas and 720000 voxels this value exceeds the size of the RAM memory available on the server machine. We can deduce that when using that amount of memory part of it is virtualized by swapping it on the hard disk. This of course introduces a very high latency in the memory access, since the hard disk has a read/write memory bandwidth that is an order of magnitude greater than the RAM memory.
Figure 14. Performance trend with parallel 16 threads as a function of the map size, including the 50 antennas set

TABLE III

DYNAMIC MEMORY ALLOCATION IN MB

<table>
<thead>
<tr>
<th>voxel</th>
<th>9</th>
<th>20</th>
<th>35</th>
<th>43</th>
<th>50</th>
</tr>
</thead>
<tbody>
<tr>
<td>180000</td>
<td>749.611</td>
<td>1600.521</td>
<td>2902.671</td>
<td>3565.15</td>
<td>4144.820</td>
</tr>
<tr>
<td>360000</td>
<td>1499.131</td>
<td>3320.841</td>
<td>5804.991</td>
<td>7129.870</td>
<td>8289.140</td>
</tr>
<tr>
<td>720000</td>
<td>2998.214</td>
<td>6641.577</td>
<td>11609.799</td>
<td>14259.517</td>
<td>16578.020</td>
</tr>
</tbody>
</table>
CHAPTER 5

DESIGN LAYOUT

In this chapter we will see the layout and characteristics of the hardware that were described in VHDL to accelerate the breast scanning. As discussed above, this hardware only performs the BEAF algorithm’s portion of the image reconstruction process, and it is actually designed to be used as a high performance peripheral by a general purpose computer that will coordinate the image reconstruction.

To simplify the description of this device we can focus on an implementation that is supposed to be used in a nine antennas system with the same number of filters. Once understood how it works, we will be able to generalize the design so that in can be adapted to systems with any combination of antennas and filters. The support for which the system was designed is a AVNET Virtex-4 LX Development Kit, interfaced with the computer via a usb connection. The description of the usb interface control system developed by our workgroup can be found in the following chapters.

Figure 15 shows the top entity layout of the design. We can see how the portion of the system that actually performs the energy computation, here named DelBEAM, is surrounded by a group of additional blocks that are required to put it in the right conditions to perform its operations. Let’s analyse their purpose and behaviour. Note that The USB communication interface is not included in Figure 15 and will be discussed in later chapters.
Figure 15. High level layout of the design
5.1 Memories

In this hybrid hardware/software system the preliminary operations like the SKAR calibration are performed outside of the device we are considering. To be able to perform the BEAF we thus need to transfer the input data that will be combined to extrapolate the energy values to some internal memories connected to the computation block. As explained in the following chapters, the communication interface used in this design works with blocks of 32bit. This is why all the memories have 32bit parallelism, so that the access from outside the system is optimally simplified. The Virtex-4 FPGA family has a big number of BlockRAMs integrated into the FPGA. Each RAM has 18Kb memory and from one of these it is possible to make different RAM configurations, from 512x32 to 16Kx1. The best configuration for the signal memories is 512x32.

Samples Memories

Here is where we store the calibrated signals produced by applying the SKAR algorithm to the sampled signals. This data consist in a set of samples represented on 16bit for each of the antenna channels in the system. Since the memories have a width of 32bit but the samples are represented over 16bit values for the quantization issues explained in the later chapters, each memory location stores a sample related to one antenna channel in the 16 most significant bits and one related to another channel in the 16 least significant bits.

This memories can be controlled for write operations by the communication interface, so that they can be accessed one after the other and set up at the beginning of the BEAF process,
while in read mode they are controlled by the system’s Finite State Machine and their outputs are delivered simultaneously for each filter channel of the computation block.

**Weights memory and weights shift register**

The major part of this memory is used to store the weights values needed to perform the BEAF algorithm, while a very small amount contains the delays associated with each of the filters. We can consider the data stored in this memory as subdivided in a series of blocks of $L$ weights plus one delay value: each of this blocks constitute inputs for one of the filters, and a number of this blocks equal to the number of antennas is stored for each of the voxels. Since the amount of weights needed to compute all the voxels is way too large to be stored locally, we launch the energy computation iteratively over a limited number of voxels, depending on the amount of data that can be stored in this memory.

When performing the BEAF we need to combine samples and weights in a way very similar to convolution. The best way to do it is to be able to access all the weights values in parallel at the same time. For this purpose we need to extract them from the memory we mentioned and store them into a SIPO-like shift register, that is able to store the weights and delay values of one voxel for each of the filter channels, that is:

$$N_{reg} = (L + 1)N_{FIR}$$  \hspace{1cm} (5.1)
Energy Memory

This memory is where we store the energy values after the computation process has completed. In write mode this memory is addressed and fed by the computation block, while in read mode its outputs are connected to the communication interface to be collected by the external computer.

For reasons that are explained in the next chapters, we represent this values over 64-bit values. However, this memory has a 32-bit parallelism just like the others. Because of that, we split the energy values in two parts and store them in little-endian fashion. To correctly access the memory for write operation, we need two independent access ports with same clock signals so that two memory locations can be written at the same time. For the read operation we need only one since in any case the communication interface can handle only 32-bit packets.

5.2 FSM and counters

To control the execution flow, we use a Finite State Machine that is in charge of all the control signals. After being triggered from a signal coming from outside, it sets the control signals so that each step of the energy computation process in executed in the right order and timing. Since most of the operations need to be executed a specific number of times and for the right amount of clock cycles, we include a set of counters to support the FSM.

FSM

The FSM flow chart is represented in Figure 16. A brief explanation of each of the execution stages follows.
Figure 16. Energy computation execution flow
1. **Idle:**

   this is the state were the system stands waiting for execution to be triggered from the outside. Typically during this stage the data to be processed is stored in the memories, and the results in the Energy Memory (if any) are read. In this state all the memory and counter signals are disabled: this does not affect the memory access from the outside, since the communication interface is able to exclude this signals and directly control the memories. When the execution signal is activated, the FSM moves to the next state.

2. **Load A:**

   in this stage the weights memory is enabled to be later accessed for read operation. Memories need to be enabled one clock cycle earlier than the actual access to be sure that the address is correctly understood. We also use this state to check if the data in the weights memory have been completely "consumed" for processing; if so the computation process has completed and the FSM is brought back to Idle state, while a completion signal connected to the communication interface is triggered.

3. **Load B:**

   during this phase, the actual weights memory reading operation is performed. All the weights and delay value associated to a single voxel and coming from one of the receivers are read and stored in a shift register, so that later they can be accessed in parallel. When the desired amount of data has been read, this stage completes.

4. **Load C:**

   Stage 3 must be repeated a number of times equal to the FIR filters available. It is
resumed from this stage until all the samples and delays for all the filters have been
loaded in the shift register. The counter used in the previous stage to keep track of the
number of weights that have been read is reset during this stage. After this, execution
proceeds to the following stage.

5. **Load D**:

Here we reset the counter used to address the samples memory, and so is the BEAF block.
We do this in anticipation of the computation process start, that occurs in the next stage.

6. **Exe A**:

the samples memory is enabled to be later read for the energy computation

7. **Exe B**

During this stage the actual computation performed by the FIR filters occurs. The Sample
Memory is accessed and its data is passed to the BEAF block. This continues until a
finish signal is triggered by the BEAF block. After that, the system proceeds to store the
obtained energy value in the dedicated memory. Please note that a different behaviour
occurs if the computation is not fully parallel, that is to say if the number of FIR filters
does not match the number of antennas, as explained in chapter 6.

8. **Save A**

The Energy Memory is enabled for successive write of the result value.

9. **Count Iter**

The energy value is actually stored in the memory, and all the counters that keep track
of weights, filters, antennas and that address the sample memory are reset. A Iteration counter that keeps track of the processed voxel is enabled.

10. **Wait for Count Iter End**

Now that the processing of a voxel has reached completion, a signal will trigger from the Iteration counter if all the voxels associated to the data currently stored in the Weights Memory have been analysed. The execution will now go back to stage 2, where this signal will be checked. However, this final stage is needed since the Iteration end signal needs an additional clock cycle to be triggered by the counter.

**Counters**

This counters are controlled by the FSM. Most of them are used to keep count of the occurrence of specific events, so that the FSM knows which should be the next state. Two of them are used to address the memories when the stored data needs to be accessed.

- **Count w**

  This counter keeps track of the number of weights that have been loaded in the shift register. Its *end* signal is triggered once all the $L$ weights plus the delay value corresponding to one antenna for the given voxel have been stored.

- **Count ant** and **Count FIR**

  This two counters store respectively the number of antennas and of filters for which the weights have been extracted from Weights Memory and put in the Shift Register. The
need of two separate counters arise when the number of filters is lower than the number of antennas, case that is better explained in chapter 6.

- **Count smem**

  As in the previous case, this counter is needed only in case of a system in which the computation of the filters is not fully parallel. This value is used to connect the FIR inputs to the correct samples memories.

- **Count AddrWM and Count AddrSM**

  When the weights or the samples memory need to be accessed, this counters are used to generate the memory address, increasing sequentially at every clock cycle as long as new values need to be read.

- **Count Iter**

  To know the number of voxels that have been processed, this counter is enabled every time a new energy value is available. Since a limited number of voxels can be processed with the data stored in the Weights Memory, this counter is used to signal when all the available data has been processed and thus new values need to be loaded in the memory from outside the system. Moreover it is also used to generate the address signal for the Energy Memory when the computed values need to be stored.

5.3 **Computation Block**

Let’s now get inside of the portion of the hardware that actually computes the energy values that will be used for the image reconstruction. We can divide the inputs of this block in three
groups: the delay values that must be supplied in parallel determine how much each channel’s signal needs to be delayed depending on the considered voxel, the weights values that are also accessed in parallel, and the calibrated samples values that enter the block one per clock cycle for each of the filter channels.

The computation block takes care of delaying the samples of the right amount of clock cycles, combining them with the samples and store the resulting energy values in memory. We now analyse each of the sub-blocks, following the computation path. Figure 17 shows a Register Transfer Level view of the computation block.

![Figure 17. RTL scheme of the beamforming block](image-url)

Figure 17. RTL scheme of the beamforming block
5.3.1 Delay

This block, identified by the symbol "t" in Figure 17, is dedicated to delaying the signal of each antenna of the right amount of samples so that they are synchronized. We instantiate one of this blocks for each of the filter channels. Their internal design is shown in Figure 18.

To delay the samples, they are introduced into a SISO-like shift register. However, the amount of clock cycles they should be delayed changes depending on the voxel we are considering, so we cannot know how long this shift register should be at the time of the synthesis. To address this issue the shift register has a number of registers equal to the maximum possible round trip
delay considering all voxels and all antennas’ positions. Then, the input of each of the registers is multiplexed, so that it can receive the data coming from the previous register in the chain or directly the sample value coming from memory. We can instantiate a decoder such that it receives in input the delay value, and produces a number of outputs equal to the number of multiplexers. Depending on the desired delay, the correct multiplexer will be enabled such that it connects its register to the input samples, while all the others will act as pass through connections between each register and the following one. In this way the data will enter the chain at the right ”distance” from the block’s output, so that it will reach the end of the shift register at the desired time.

The Delay block also includes a small FSM that is not included in Figure 18, that takes care of keeping track of the amount of samples that have gone through the block and when there is no more to extract from memory stops and resets the shift register.

5.3.2  **Window 1**

Once the samples have been correctly delayed, they are windowed by this block. The samples will exit from the delay block ready to be processed after $n_a$ clock cycles that is the time reference point used to align all the signals, so what Window1 does is to let pass through only the samples coming after that time. All the samples before time $n_a$ are replaced with a zero value. At the same time this block triggers the beginning of the FIR computation after the first $n_a$ samples have been ignored. The same signal used to trigger the FIR is passed through a shift register to enable at the right time all the other blocks the data will travel through during
the computation. To better understand this see Figure 19 that shows an architectural view of the Computation Block.

5.3.3 FIR

In this block the actual sums and multiplications are performed. Its inputs consist of a set of weight that remains unchanged during a single voxel computation and a set of samples that enter the block one per clock cycle and travel through a shift register so that at each clock cycle a different window of samples is used for the computation. Figure 20 shows a RTL view of the FIR block.

This block mainly performs a combination of this data using a tree-like structure. In the first level, each weight-sample couple is multiplied. Then the results of this multiplications are summed together thanks to a tree adder whose depth is in the order of $\log_2(L)$, until a single
5.3.4 Window 2

After each of the FIRs output have been summed together thanks to a tree adder, the resulting values are fed to a window block. To compute the energy associated to a voxel, we want to consider only the samples that we expect will include the scattered signal’s peak. This set of samples is identified by the values $n_{h}$ and $l_{h}$. Window2 receives the filtered samples and lets only those that fall in this set to pass, while its output remains fixed to zero in all the other
cases.

To identify the desired set of samples, Window2 starts to count when the triggering signal produced by Window1 has passed through a series of registers, so that it has been delayed by the amount of time needed for the data to reach the end of the FIR sum tree. When it reaches the value $n_h$ it triggers a signal that initiate the energy computation, and ends it after $l_h$ clock cycles.

### 5.3.5 Energy Computation

When finally the set of significant samples that we want to use to compute the energy has been produced and identified, it is fed to a Energy Computation block. Here at each clock cycle the square of one of them is computed thanks to a multiplier. The square results are added together by using an accumulator whose output after all the samples have been processed will be the final energy value. This value is then redirected to the energy memory to be stored.
CHAPTER 6

USB COMMUNICATION INTERFACE

As previously explained our design choice was as a first step to implement in hardware only the Mist BEAF algorithm, maintaining on software the remaining parts of the computation process. To actually use the designed system a connection allowing hardware and software shares of the system to communicate data is required. A USB interface was programmed to accomplish this task.

The Cypress FX2 USB 2.0 chip integrated in the AVNET Virtex-4 LX Development Kit was used to communicate with the FPGA. FX2 was configured in Slave FIFO mode: the chip behaves simply as a conduit between the USB and the external data-processing logic. The FX2 clock was set to 48 MHz, the parallelism of data to 8 bits and packet size was set to 512 bytes. To implement a functional communication interface two components were needed: on the general purpose pc (software) side a C program was written that takes care of sending the data trough the USB connection to the development board; on the FPGA (hardware) side some control blocks were implemented to correctly handle the received data. As explained in this chapter, the data parallelism changes when passing from software to hardware and vice versa. To do so, some hardware components were added to the design.

The software running on the microprocessor is layered, and so the application part and the communication part are independent and portable. The communication part is also further partitioned in a set of low-level, hardware-specific USB primitives (we use the libusb library...
available in Linux) and a set of higher-level, technology-independent APIs for data organization and communication. When the hardware target changes, as it happens when moving from the prototype to the final target, the developers replace only the low-level communication primitives, and reuse both the application code and the communication APIs.

6.1 Hardware

The main components of the hardware communication block are two control blocks and two FIFOs. The first control block takes charge of receiving and sending the correct amount of data through the USB buffer depending on the received instructions. The second control block has the ability to access the memories and triggering the execution of the BEAF algorithm. The two FIFOs stand between this two blocks to adapt the data parallelism. An overview is presented in Figure 21. Purpose of this communication interface is to both analyse the commands sent from the PC and to act as a translator between the FX2 chip and the BEAF block, resizing the data with the correct parallelism and allowing two hardware with different clock periods to communicate.

RX and TX FIFOs

Two FIFOs were implemented using Xilinx Core Generator. The RXFIFO receives data from the Control 1 block and forwards it to the Control 2 block. It has a 8bit data parallelism at the input and 32bit parallelism at the output. The TXFIFO receives data from the Control 2 block and brings them to the Control 1, with a data parallelism of 32bit at the input and of 8bit at the output. Each of this FIFOs has two output signals to show when they are full or empty.
Figure 21. Overview of the communication interface

This two blocks allow two sections of the system with different properties to communicate: the FX2 section and the BEAF section work with different data parallelism and different clock periods, so this two blocks are essential to ensure that the data can pass from one side to the other without risk of being compromised.

Every command is formed by 4 bytes: the first one contains the op-code that identifies the command, while the remaining may contain a value that for read and write operations represents the number of packets to be transmitted. The FX2 on the other hand as a USB controller has a data parallelism of 8bit. Besides, the clock frequency of the FX2 can be higher than the one of the BEAF block, allowing to speed up the communication process. Both of the FIFOs have then two different clock signals, one for the input and one for the output so that the in-
put and output data are always correctly synchronized with the hardware they are connected to.

**Control 1**

This block is the first one the data find travelling from the FX2 chip to our system. It directly receives a series of signals coming from the FX2 chip that carry the transmitted data and some information about the state of the buffer. Its functionalities are to analyse the commands sent from the PC, read data from FX2 output FIFO and write it to RXFIFO and read data from TXFIFO and write it to FX2 input FIFO. Since it interfaces primarily with the FX2 chip, it works with its same clock signal. To more easily understand the functions of this block, let’s analyse its input and output connections:

- **Reset** and **Clock** are as usual, but note that the clock signal is the same used to control the FX2 chip, that is different from the one used for the BEAF block.

- **flagb** and **flage** carry information about the USB buffer, signalling respectively when it is empty or full.

- **sloe**, **slrd** and **slwr** are used to control the USB buffer by enabling it for either read (when receiving data from the USB interface) and write operations (when sending data to the PC through the USB interface).

- **pktend** is used during read operations, that’s to say when data is sent out to the PC, to communicate to the FX2 when the data packet have been completely sent.
• **fifoaddr** allows the Control 1 block to interact with the correct FX2 buffer: depending if the current operation implies data entering or exiting the FPGA, the Control 1 block may need to be connected either to the FX2 output FIFO buffer or the FX2 input FIFO buffer.

• **fd** carries the data coming from the FX2 buffer and have thus 8*bits* parallelism.

• **rxfifo-full** signals when the RXFIFO is full and cannot be feed with more data as long as some is consumed from Control 2 block.

• **rxfifo-wr-en** enables the RXFIFO for writing data coming from the USB output chip buffer.

• **txfifo-empty** signal when the data written in the TXFIFO by the Control 2 block have been completely processed.

• **txifo-rd-en** enables the writing of data coming from the TXFIFO to the USB chip input buffer.

• **read-data** is triggered when data needs to be read from the FX2 buffer: since the data connection of the FX2 chip is only one of type *inout* used for both incoming and outgoing data, this signal specifies to the chip when this connection should be used as input rather than as output.

The Control 1 block includes a Finite State Machine that is in charge of interpreting the commands coming from the external PC and of controlling the FIFOs according to the operation that must be performed. Even if the number of states of the FSM is very high and thus is not
suitable for a graphic representation, the phases in which it can work can be grouped in five main groups:

1. **IDLE**: this is the first phase from which the FSM start back every time an operation is concluded. No operation is performed in this phase.

2. **START**: in this phase the FX2 buffer is continuously checked for incoming data, until the first four bites are received. Since in every communication operation the first four bites include information about the operation to be performed, this data is analysed to proceed with the correct phase.

3. **WRITE** and **READ**: this operations do not differentiate depending on the kind of data is being transmitted (for example, if it is to be written in the Weights Memory or in the Samples Memory): this distinction in made in the Control 2 block. When a write or read command is received, its coding includes the amount of packets being transmitted. A program counter is thus set so that it keeps track of the amount of data transmitted and signals when there is no more. During the read and write operations both the RX and TX FIFOs and the FX2 buffer are consistently checked to determine when they are full or empty, so that none of the data packets is lost.

4. **HANDSHAKE**: is very similar to a read operation, except that only a single fixed value is transmitted, to trigger some event in the PC software. For example, when the BEAF completes the computation of the energy values a specific packet is sent to the PC to signal the completion of the operation.
5. **RST**: when a reset command is recognized, the Control 1 block does not have to perform any other operation except letting it pass through the RXFIFO since it is interpreted by the Control 2 block.

**Control 2**

The purpose of this block is to analyse the commands sent from the PC and set the BEAF’s block control signals accordingly so that all the operations are properly performed. In its interface we can find the following signals:

- **Clock** and **Reset**, where the clock signal is the same of the BEAF block so that this two blocks are synchronized.

- **rxfifo.dout** that carries the output data coming from the RXFIFO.

- **rxfifo.rd.en** which is used to enable the reading of data coming from the RXFIFO.

- **rxfifo.empty** signals when the RXFIFO is empty an thus no more data can be extracted until the Control 1 block fills it again.

- **txfifo.din** that carries the input data sent to the TXFIFO.

- **txfifo.wr.en** used to enable the TXFIFO fro writing new data in it.

- **txfifo.full** signals when the TXFIFO is full and thus no more data can be written in it until the Control 1 block consumes some of the data already stored in it.

- **bram16k control, data** and **address** signals that are used to access the Weights Memory for writing or reading values through the USB interface.
- **bram2k control, data and address** signals that allow to control and access both Samples and Energy Memories.

- **reset_cpu** allows to reset the BEAF block, clearing for example the values of the counters and of the registers it contains.

- **instrackF** is a very critical control signal that allows the Control 2 block to access the Weights Memory, bypassing the control, data and address signals coming from the FSM inside the BEAF block.

- **dataackM** very similarly allows this block to take control of the Samples and Energy Memory.

- **enmem_en** is used to enable the Energy Memory; this signal is needed since all the other control signals for the Energy memory are used also for the Samples memory. In this way the Energy Memory is enabled only when it is actually needed.

- **Start** is the signal that triggers the execution of the BEAF algorithm when all the memories have been filled with the data.

- **Nvox_cycle** indicates to the BEAF block the amount of voxels that should be processed with the available data when receiving the start command. This number is included in the 24 least significant bits of the start command sent from the PC.

- **data_processed** is triggered from the BEAF block when a computation iteration has been completed and thus new commands can be sent to the system.
The main purpose of this block is to analyse the commands and accordingly set the values in the memories so that they are ready when the energy computation starts. A dedicated FSM allows to perform properly the operations required to execute the received commands. The commands that it can process are shortly listed below as part of the FSM stages:

1. **Start**: in this first stage the RXFIFO is continuously checked for new data. When a new command is received, it is read from the FIFO and analysed to understand which is the operation the FSM needs to execute.

2. **Reset**: triggers the reset signal of the BEAF block, bringing it back to its starting condition.

3. **Write** and **Read**: for each of the memories in the design, a dedicated write and read process is implemented in the FSM. Depending on the memory that must be accessed the corresponding control signals are enabled. To address the memories a counter is implemented that counts up to the number of memory blocks specified in the command’s code three least significant bytes. During the write and read operations the signals coming from the FIFOs are continuously controlled, so that the data is transmitted only when there is no risk of errors due to empty or full buffers. During this operations the BEAF block is excluded from the control of the memories thanks to the dedicated signals. Even if some of the memories are accessed from outside only either for writing or reading, both options were implemented for all the memories to supply complete debug functionality.

4. **Computation Start**: this operation sets the correct values required for the energy computation (like the number of voxels to process) and triggers the execution. The FSM
than hangs waiting for the confirmation of the computation’s conclusion, and sends than a handshake packet to the PC to let it know that the energy values are ready to be read.

6.2 Software

In the brief description above, we referred to software portion of the design as consisting in a PC program that takes the data produced via software and sends them to the FPGA through USB connection. There is actually an aspect we have not mentioned yet: to be able to use the USB interface, the Cypress FX USB 2.0 chip needs to be programmed, so that it lets the data pass through the way we want it to. A firmware developed for a previous project was adapted to our case(19). Since the software share of the computation is performed on a Linux based platform, the FX2 chip programming could easily be performed by relying on some simple instructions already built in the Linux kernel (fxload). See Appendix 3 for more details.

The host application was written in C language with Linux compatibility. It relies on the libusb library to access the usb interface. It presents a series of commands to set up the communication with the FX2 chip. It includes one function for each of the possible operations that are needed to set up and execute the communication. At the beginning it sets up the Samples Memories, and then iterates a cycle in which the Weights Memory is loaded with new values and the computation is launched. At the end of each iteration, the corresponding energy values are read from the FPGA memory and stored in the PC. The input data is stored in text files, generated from the Matlab/Octave software that executes all the data processing besides the BEAF algorithm. Output energy values are written in a text file as well.

In each function, the appropriate command code is sent in the form of four consecutive bytes,
and then bulk transfers are used in case of read and write operations. The bulk transfer size was set to 512 bytes. Even if some of the memories are accessed only in write or read mode from the PC, all the possible read and write operations were implemented to allow debugging of every part of the computation process: since the FPGA state during execution cannot be monitored, the only relevant information can be obtained analysing the contents of the memories. The main function of the host application is presented in listing B.1.

Listing 6.1. Host application

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <libusb-1.0/libusb.h>

#define MAX_LEN 256
#define INTERFACE_NUMBER 0
#define BULK_OUT_ENDPOINT 0x02
#define BULK_IN_ENDPOINT 0x86
#define MAX_DATA_OUT_SIZE 512
#define MAX_DATA_IN_SIZE 512
#define TIMEOUT_MS 5000
#define NVOX_cycle 195
#define NWEIGHTS_PER_VOX 504
```
void reset_fpga(libusb_device_handle *mydevh);

void download_weights(libusb_device_handle *mydevh, int pktLen, FILE *weightsFile);

void read_weights(libusb_device_handle *mydevh, int pktLen);

void read_energy(libusb_device_handle *mydevh, int pktLen, FILE *receivedF);

void bin2asciihex(unsigned char input, char *asciibyte);

void download_samples(libusb_device_handle *mydevh, int file_num);

int start_FPGA(libusb_device_handle *mydevh, int pktLen);

void asciihex2bin(char *input, char *out);

void ascii2int(char *input, char *out, int length);

void read_samples(libusb_device_handle *mydevh, int pktLen);

int64_t timespecDiff(struct timespec *timeA_p, struct timespec *timeB_p)
{
    return ((timeA_p->tv_sec * 1000000000) + timeA_p->tv_nsec) -
            ((timeB_p->tv_sec * 1000000000) + timeB_p->tv_nsec);
}

int main(void)
{
    char cmd_buf[MAX_LEN], file_name[30];

    static const int VENDOR_ID = 0x04b4;
    static const int PRODUCT_ID = 0x8613;

    struct libusb_device_handle *mydevh = NULL;

    int device_ready = 0;
int result, pktLen, i;
int voxel, Nvox, weightsFileLen;
int err;

// time_t tempoInizio, tempoFine;

struct timespec tempoInizio, tempoFine;

double Tcycle;

FILE* weightsFile, *samplesFile[5], *energyFile, *energy_test;

err = 0;

// Initialize USB device
result = libusb_init(NULL);
if (result < 0)
{
    fprintf(stderr, "Failed to initialize libusb.\n")
    exit(1);
}
mydevh = libusb_open_device_with_vid_pid(NULL, VENDOR_ID, PRODUCT_ID);
if (mydevh == NULL)
{
    fprintf(stderr, "Unable to find the device.\n")
    exit(1);
}
libusb_detach_kernel_driver(mydevh, INTERFACE_NUMBER);
result = libusb_claim_interface(mydevh, INTERFACE_NUMBER);
if (result >= 0)
    device_ready = 1;
else
{
    fprintf(stderr, "libusb_claim_interface_error\n");
    exit(1);
}
if (device_ready)
{
    printf("Press ENTER to start");
    getchar();
    weightsFile = fopen("fpga_feed_hex.txt", "r");
    if (!weightsFile)
    {
        fprintf(stderr, "Cannot open fpga_feed_hex.txt\n");
    }
    else
    {
        energyFile = fopen("energy_FPGA.txt","w");
        if (!energyFile)
        {
            fprintf(stderr, "Cannot open energy_FPGA.txt\n", file_name);
            err=1;
        }
        if (err==0)
        {
            reset_fpga(mydevh);
            for (i=0; i<5; i++)
                }
download_samples(mydevh, i+1);

fseek(weightsFile, 0, SEEK_END);

weightsFileLen = ftell(weightsFile);

fseek(weightsFile, 0, SEEK_SET);

Nvox=(weightsFileLen /5) /NWEIGHTS_PER_VOX;

printf("total number of voxels: %d\n",Nvox);

voxel=0;

while(voxel<Nvox)
{
    printf(" voxel %d\n", voxel+1);

    if(Nvox-voxel<NVOX_cycle)
    {
        download_weights(mydevh,(Nvox-voxel)*NWEIGHTS_PER_VOX,weightsFile);

        if(start_FPGA(mydevh,Nvox-voxel)==0)
        {
            read_energy(mydevh,(Nvox-voxel)*2,energyFile);

            reset_fpga(mydevh);
        }
        else
        {
            printf(" error computing energy\n");
        }

        voxel=Nvox;
    }
    else
    {
    
    }
}
download_weights(mydevh, NVOX_cycle * NWEIGHTS_PER_VOX, weightsFile);
if (start_FPGA(mydevh, NVOX_cycle) == 0)
{
    read_energy(mydevh, NVOX_cycle * 2, energyFile);
    reset_fpga(mydevh);
}
else
{
    printf("error computing energy\n");
}
voxel+=NVOX_cycle;
}
}
fclose(energyFile);
for (i<5; i=0; i++)
{
    fclose(samplesFile[i]);
}
fclose(weightsFile);
}

return 0;
CHAPTER 7

ACCURACY ESTIMATION

Now that we have described how the system have been implemented, we should wonder about its effectiveness: would we actually be able to identify the presence of a tumor using this device?

An analysis of the effectiveness of the algorithms was performed in (12). We will now propose it again, together with a series of analysis that will show the actual accuracy of our specific design. Only the computation of two dimensional energy maps is considered here for the accuracy estimation, but information on 3D mapping can be found in (12).

7.1 Quality factors

To evaluate the results obtained by simulating the algorithms for energy estimation we cannot simply look at the energy maps. We need a way to express the accuracy of the algorithm. A series of values is now defined that condense information about the results obtained:

- **CBR**: Compensated waveform (with tumor) to Backscattered signal Ratio.

\[
\text{CBR} = 20 \cdot \log \frac{\max(C)}{\max(B)} \quad (7.1)
\]

This value shows how much the calibration stage attenuates the original signal. The lower it is, the better are the performance, as long as it is not smaller that the \(\text{CBR}_{\text{ideal}}\) obtained with ideal calibration.
- **TCR**: ratio between maximum value of tumor response with ideal calibration and maximum value of calibrated waveform without tumor.

\[
TCR = 20 \cdot \log \frac{\text{max}(C_{\text{ideal}})}{\text{max}(C)}
\]  

(7.2)

With a good calibration surely skin artifact and direct path contribution don’t overcome tumor value, and then this value is expected to be equal to positive dB value. It could be possible that a good calibration has negative value of TCR if difference between with-tumor and without-tumor results is less than backscattered contribution from normal tissue.

- **MAXPOS**: the position of the maximum value in the energy map. Of course, this value position should be near the tumor position if the reconstruction algorithm performs well.

- **SMR**: Ratio between the maximum value of the backscattered energy matrix in case of tumor \((rec_T)\) and average inbreast energy \(rec_{T,\text{ave}}\) of the same energy matrix.

\[
\text{SMR} = 10 \cdot \log \frac{\text{max}(rec_T)}{rec_{T,\text{ave}}}
\]

(7.3)

This value is a index of contrast, showing how much the maximum energy value is far from the average. Assuming that the maximum value is the tumor response, we want the SMR to be as high as possible. If otherwise the maximum value is not the tumor response than this factor becomes irrelevant.
• **SCR**: Ratio between maximum value of backscattered energy in case of tumor \((rec_T)\) and in case of no presence of tumor \((rec)\).

\[
SCR = 10 \cdot \log \frac{\text{max}(rec_T)}{\text{max}(rec)}
\]  \hspace{1cm} (7.4)

An high value for this parameter represents a good contrast of the tumor with respect to a case with no tumor. The maximum energy value is supposed to be the tumor response. If this is not true, this value becomes meaningless.

### 7.2 Contour conditions and parameters

A number of design choices and simulation conditions can impact the outcome of the tumor detection. Here we describe the design choices and the testing conditions that characterize this project.

#### 7.2.1 Programmable support

The aim of this work is to realize the Image Reconstruction Unit by programming the AVNET Virtex-4 LX Development Kit with a Xilinx Virtex-4 XC4VLX160-FF1513 FPGA. A picture of the board is in Figure 22.

This initial design is characterized by 9 FIR filters corresponding to an equal number of antennas, to replicate in hardware in the easiest and simplest way the simulation conditions proposed by the software implementation presented in (12). However, the required resources to implement this design exceed those available on the development board.

The V4-LX160 has a number of BlockRAMs equal to 288. It is possible to obtain the maximum
Figure 22. Picture of the AVNET Virtex-4 LX Development Kit
number of coefficient groups that can be stored in the remaining memory. With these conditions, 5 BlockRAMs are necessary as Signal Memories. Supposing that two Blocks are sufficient as Energy memory, the remaining blocks are 281. With simple calculations, it is possible to find that the maximum number of voxels that could be processed in the same cycle is 570. In our case the actual chosen number of voxels to be processed for each cycle was 195.

7.2.2 Transmitters properties and tumor positions

To correctly perform simulations and obtain consistent data to verify the performance of the algorithms we consider, we must strictly define the conditions in which we are going to test them. As a first step, we can describe conditions that determine the effects of the breast irradiation with microwaves. In table Table IV information regarding the position in space of the receivers used to sample the reflected signals are summarized. Table Table V shows the possible tumor positions in 2D maps. In this analysis the use of the Differential Gaussian pulse is considered, although the use of the Modulated and Modified Hermite Pulse is discussed in (12). The Differential Gaussian pulse used as UWB transmitted signal has a central frequency of $f_c = 6\text{GHz}$ and pulse width of $\tau = 110\text{ps}$. This type of signal is often used in literature combined to CMI and MIST beamforming.

Figure 23 shows the positions of the receivers and the three possible tumor locations that were used for testing.

7.2.3 SKAR

Let us now consider the two SKAR parameters $n_0$ and $n_{fin}$. They represent the samples interval considered for weights evaluation, and can be obtained via empirical process(2). Know-
TABLE IV

RECEIVERS POSITION IN 2D SIMULATION SPACE

<table>
<thead>
<tr>
<th>Name</th>
<th>$i$-(cm)</th>
<th>$j$-(cm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Rec. 1</td>
<td>15.75</td>
<td>1.95</td>
</tr>
<tr>
<td>Rec. 2</td>
<td>8.15</td>
<td>2.8</td>
</tr>
<tr>
<td>Rec. 3</td>
<td>5.35</td>
<td>4.4</td>
</tr>
<tr>
<td>Rec. 4</td>
<td>3.5</td>
<td>6.2</td>
</tr>
<tr>
<td>Rec. 5</td>
<td>1.95</td>
<td>9.0</td>
</tr>
<tr>
<td>Rec. 6</td>
<td>2.75</td>
<td>11.3</td>
</tr>
<tr>
<td>Rec. 7</td>
<td>4.65</td>
<td>13.3</td>
</tr>
<tr>
<td>Rec. 8</td>
<td>7.05</td>
<td>14.6</td>
</tr>
<tr>
<td>Rec. 9</td>
<td>10.05</td>
<td>15.0</td>
</tr>
</tbody>
</table>

TABLE V

POSSIBLE TUMOR POSITIONS IN 2D SIMULATION SPACE

<table>
<thead>
<tr>
<th>Tumor position</th>
<th>$i$-(cm)</th>
<th>$j$-(cm)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Low</td>
<td>4.4</td>
<td>8.8</td>
</tr>
<tr>
<td>Med</td>
<td>8.8</td>
<td>8.8</td>
</tr>
<tr>
<td>Deep</td>
<td>13.1</td>
<td>8.8</td>
</tr>
</tbody>
</table>

ing position of antennas and pulse width, it is possible to estimate these values. Their value depends on the pulse width, its starting time and the sampling frequency. In general, this interval of samples must include direct path contribution and skin reflections. The following constraints should be considered:
the \( n_{\text{fin}} \) value should be as close as possible to the region of the signal where direct path and skin reflection contributions have no more influence,

- \( n_{\text{fin}} \) value should not be so high that any part of the contributions of tissue near to the skin is deleted, otherwise low positioned tumors could remain undetected.

- the interval \( m = n_{\text{fin}} - n_0 \) should include only the portion of signal we want to minimize: including initial samples that have no relevance in this window would decrease the performance.

- \( n_0 \) must be greater or equal to \( J \) because of software limitations of the simulation program.
The condition that could affect the correct detection more significantly is the first one. The chosen values were:

\[ n_0 = 30, n_{\text{fin}} = 60 \]  \hspace{1cm} (7.5)

Other two parameters of SKAR are \( J \) and \( p_{\text{eigh}} \), that represent the width of the FIR filter and the low-rank approximation coefficient. In literature there is very short information about this values (2). They are dependent on the input pulse. Moreover, \( J \) should be as low as possible to reduce complexity of the weights computation algorithm. Through simulations the TCR values corresponding to different combinations of \( J \) and \( p_{\text{eigh}} \) were computed. The best combination, considering also the need of accuracy even in high depth locations, is suggested to be:

\[ J = 3, p_{\text{eigh}} = 20 \]  \hspace{1cm} (7.6)

### 7.2.4 BEAF

As in the SKAR algorithm, we need to define the starting point of the transmitted pulse, that in this case is defined as \( N_{\text{Start}} \). If this value is correct, it will be possible to perform the FFT of the input pulse in the right window. In our case the chosen value is \( N_{\text{Start}} = 25 \) to include also the pulse’s tail, as suggested in (2).

We then need to define the samples interval that contains the pulse. This window is identified by the values \( n_h \) and \( t_h \). The value of \( n_h \) can be computed knowing both time shift \( n_a \) and
input pulse shape. Remembering that $\tau_0 = (L - 1)/2$ is the delay induced by the FIR filter we get that:

$$n_h = N_{\text{Start}} + n_a + \tau_0$$  \hfill (7.7)

As in (2), the extent of the backscattered signal from a ill tissue is directly proportional to the tumor size. Since the beamformer must be able to detect also small lesions, the $l_h$ value is kept low. Although this produces a smaller SCR for large tumors, this is counterbalanced by the strong backscattered signal due to the tumor size. Optimal value is then suggested to be $l_h = 5$.

The parameter $L$ represents the length of FIR filters. In order to have a frequency resolution of $1GHz$ for the weights design, the minimum acceptable value is $L = 50$. This value should be kept as low as possible. An implementation problem is the great amount of space requested to memorize all these values if these are previously computed and loaded from storage devices. Moreover, in case of differential gaussian pulse the reconstruction is not improved by using a large value of $L$. The length of filters was chosen equal to $L = 55$.

Finally, the $\xi$ value was chosen as suggested in (15) and fixed at $\xi = 4$.

**7.2.5 Quantization**

To select a suitable data parallelism for our hardware design, we need to consider the quantization effect. Choosing a small parallelism can have multiple advantages: for example it means a lower complexity for the reconstruction device, and at the same time it would have a positive impact on the time required to transmit the data between PC and FPGA. At the same time, a small parallelism can be critical for the outcome of the analysis.
In (12) an analysis of the quantization effect is performed. The weights and samples values are converted in fixed point notation and then processed. The number of bits on which they are represented are defined as $bitQ$ for the weights values and $bitIN$ for the samples values. Ideally, we would like to use the smallest possible parallelism, so that the hardware design of the image reconstruction unit can be very simple, but performing this kind of quantization reduces heavily the amount of information available to reconstruct the breast image. Figure 24 shows the effect of the quantization error: as we see, it is not very significant if we choose to represent the values over at least 8 bit.

However, this analysis does not take into account an important factor: in the FIR filters there is a bit truncation after the initial multiplication. Depending on the data parallelism, this truncation can impact significantly on the results producing large errors. That’s why we should also consider this effect when performing the initial data quantization. The outcome of the analysis with 8bit and 16bit quantization are in Figure 25, that includes the outcomes with an ideal SKAR calibration and with a SKAR performed with bit truncation after the multiplication. As we can easily see from 25(c), in case of 8bit quantization the breast analysis is not consistent. A 16bit fixed point data representation was then chosen.

7.3 **Accuracy performance**

Let’s now consider the accuracy of the image reconstruction process. First of all, we will analyse the outcomes of the processing algorithms in ideal simulation conditions, that do not take care of quantization or approximation effects. We will then present the results of some
possible hardware configurations and compare them with the ideal case, to identify unwanted approximation effects.

7.3.1 Ideal energy computation

Here we describe the outcome of the simulation via Octave software of the image reconstruction. The operations are performed with Octave’s native data accuracy, without taking into account neither the approximations introduced by the hardware implementation which we will discuss later nor the quantization effect.

The only calibration option we will take into account is the SKAR, together with the Ideal one that will act as a basis for comparison. Both calibration types are performed together with the

Figure 24. Quantization error
Figure 25. Quantization effect comparison with ideal SKAR and bit truncation

(a) 8bit quantization, ideal SKAR
(b) 16bit quantization, ideal SKAR
(c) 8bit quantization, real SKAR
(d) 16bit quantization, real SKAR
BEAF algorithm on the three possible cases of tumor positions aforementioned. Moreover, the simulations are performed both with ideal and real breast models. Table Table VI shows some figures of merit for the two calibration algorithms.

Of course, this values are not fully comprehensive of all the aspects that determine the efficiency of a calibration algorithm. For example, no direct information about distortion is given. Anyway, they can give an approximate estimate on how the data processing accuracy is affected by calibration. Observing the CBR of SKAR calibration, it is lower than Ideal one. This leads us to the conclusion that a distortion of tumor contribution has occurred. TCR shows an evident downtrend moving from low to high depth in tumor position. This is no surprise, since the more far we move from the antennas the bigger the clutter contributions due to tissue discontinuity became. They could become predominant, since our algorithms are not able to delete them. On the other hand the CBR values are reasonably stable for SKAR calibration. This behaviours simply show us that the algorithms have removed the contributions from direct path and skin

<table>
<thead>
<tr>
<th></th>
<th>Low</th>
<th>Med</th>
<th>Deep</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>CBR(dB)</td>
<td>TCR(dB)</td>
<td>CBR(dB)</td>
</tr>
<tr>
<td></td>
<td>-49.5</td>
<td>0</td>
<td>-65.2</td>
</tr>
<tr>
<td>SKAR</td>
<td>-54.2</td>
<td>14.9</td>
<td>-64.5</td>
</tr>
</tbody>
</table>
artifact, but can’t remove the other reflections.

The reconstructed images of the energy reflected by the breast are represented in the following figures. In Figure 26 we can see outcome of the image reconstruction in case of an ideal breast model. Some quality factors corresponding to the same simulations are in table Table X. First of all, let’s spend a word on the SCR values for ideal calibration. As mentioned before, SCR is derived from the ratio between the maximum energy value in case of tumor and the one in case of no presence of tumor. While for other calibrations this value gives a good estimate on how much the tumor scan point stands out with respect to a healthy breast scan, in case of Ideal calibration this value does not actually represent any significant indication on the quality of the computation process. This is because in case of Ideal calibration there should be no unwanted skin artifact, and thus the energy matrix in case of absence of tumor is filled with zeroes: any ratio will give then an infinite value. Looking at the SCR value in case of SKAR calibration, we see how going deeper with the tumor position it gets lower. This is due to the fact that a deeper tumor produces backscattered signals that get to the receivers with lower intensity. As a consequence the contrast factor decreases. On the other hand observing the SMR values we can see how SKAR calibration produces a little degradation of the quality of the image in all the cases in analysis. Anyway, this loss in accuracy is small enough to not compromise significantly the tumor detection.

Simulations were also performed with real breast model, obtained from repository(17). The images of the reconstructed energy map are grouped in Figure 27. Looking at this images and comparing them with the previous ones, we notice right away that the quality of the detection
has decreased with this breast model. The contrast factor are lower, consequence of the presence of undesired reflections.

As the images show, the image reconstruction is pretty accurate when working with an ideal breast model. This fact is sustained by the data in table Table VII that shows the outcome of the detection process, comparing the maximum energy value position with the tumor position. Anyway we should keep in mind that the location of the maximum energy position is not a foolproof method to say if the tumor was correctly identified, since there are many others relevant factors like the shape or the number of lower energy points. This considerations should be made together with a deep analysis of the breast images obtained. When moving to a real breast model, things get a bit more difficult: the non continuous dielectric property of the same tissues generates unwanted reflections that cause distortion. The outcomes of the breast analysis in table Table VIII show a decline in performance. With Ideal calibration, the detection is still consistent even if the tumor is identified in a different location. When using SKAR calibration the detection is seriously compromised by this distortions, resulting in tumors detected in the wrong position or not detected at all. This is due to the high variation of the dielectric properties of the breast. In (2) study of robustness of the MIST algorithm was carried out and the authors have estimated the maximum variability of dielectric properties at
50% to obtain a correct detection. For our simulations we used breast phantom from repository (17), that is characterized by the following range of dielectric constant:

\[
\begin{align*}
\epsilon_{\text{min}} &= 2.4 & \epsilon_{\text{ave}} &= 12.8 & \epsilon_{\text{max}} &= 49.5
\end{align*}
\]  

(7.8)

In this model the percentage of fatty tissue is higher than the percentage of fibroconnective tissue, that has a higher permittivity value. This results in an average value that is near to the minimum one. In this case, the maximum variability is computed between average and maximum value of dielectric constant and in our case is equal to 400%.

This results show us that the image reconstruction process is accurate enough to allow tumor detection in an ideal case. When using the real breast model at our disposal, however, the use of a SKAR calibration produces inconsistent results. In the following sections we will not consider energy computation over real breast model with SKAR calibration, since the approximations derived by the hardware implementation are going to make it even more critical to correctly identify the tumor.

7.3.2 Low resources approximation

Up to now, we have discussed the results of the image reconstruction process as it is performed by the Matlab/Octave software. When moving to the hardware implementation we have to take into account how the data is going to be processed and represented. Previously an analysis on the quantization effect was discussed, concluding that the input data should be represented in 16bit fixed point notation. This consideration just tells us that this initial
### TABLE VII

TUMOR DETECTION OUTCOMES

<table>
<thead>
<tr>
<th>Actual Position</th>
<th>Calibration</th>
<th>Detected Position</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>44 ; 88</td>
<td>IDEAL</td>
<td>46 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>IDEAL</td>
<td>88 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>IDEAL</td>
<td>131 ; 89</td>
<td>YES</td>
</tr>
<tr>
<td>44 ; 88</td>
<td>SKAR</td>
<td>46 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>SKAR</td>
<td>88 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>SKAR</td>
<td>131 ; 89</td>
<td>YES</td>
</tr>
</tbody>
</table>

### TABLE VIII

TUMOR DETECTION OUTCOMES, REAL BREAST MODEL

<table>
<thead>
<tr>
<th>Actual Position</th>
<th>Calibration</th>
<th>Detected Position</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>44 ; 88</td>
<td>IDEAL</td>
<td>39 ; 95</td>
<td>YES</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>IDEAL</td>
<td>78 ; 92</td>
<td>NO</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>IDEAL</td>
<td>149 ; 85</td>
<td>NO</td>
</tr>
<tr>
<td>44 ; 88</td>
<td>SKAR</td>
<td>49 ; 78</td>
<td>NO</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>SKAR</td>
<td>91 ; 45</td>
<td>NO</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>SKAR</td>
<td>91 ; 45</td>
<td>NO</td>
</tr>
</tbody>
</table>
### TABLE IX

**QUALITY FACTORS FOR IDEAL SIMULATION WITH IDEAL BREAST MODEL**

<table>
<thead>
<tr>
<th>Calibration</th>
<th>Tumor Position</th>
<th>SCR</th>
<th>SMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>Low</td>
<td>Inf</td>
<td>18.781</td>
</tr>
<tr>
<td>SKAR</td>
<td></td>
<td>13.818</td>
<td>16.012</td>
</tr>
<tr>
<td>IDEAL</td>
<td>Med</td>
<td>Inf</td>
<td>18.557</td>
</tr>
<tr>
<td>SKAR</td>
<td></td>
<td>8.842</td>
<td>16.602</td>
</tr>
<tr>
<td>IDEAL</td>
<td>Deep</td>
<td>Inf</td>
<td>17.199</td>
</tr>
<tr>
<td>SKAR</td>
<td></td>
<td>1.719</td>
<td>12.506</td>
</tr>
</tbody>
</table>

### TABLE X

**QUALITY FACTORS FOR IDEAL SIMULATION WITH REAL BREAST MODEL**

<table>
<thead>
<tr>
<th>Calibration</th>
<th>Tumor Position</th>
<th>SCR</th>
<th>SMR</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>Low</td>
<td>Inf</td>
<td>14.920</td>
</tr>
<tr>
<td>SKAR</td>
<td></td>
<td>13.818</td>
<td>16.012</td>
</tr>
<tr>
<td>IDEAL</td>
<td>Med</td>
<td>Inf</td>
<td>11.136</td>
</tr>
<tr>
<td>SKAR</td>
<td></td>
<td>8.842</td>
<td>16.602</td>
</tr>
<tr>
<td>IDEAL</td>
<td>Deep</td>
<td>Inf</td>
<td>13.970</td>
</tr>
<tr>
<td>SKAR</td>
<td></td>
<td>1.719</td>
<td>12.506</td>
</tr>
</tbody>
</table>
Figure 26. Breast images for ideal breast model

(a) Low tumor, Ideal calibration
(b) Low tumor, SKAR calibration
(c) Med tumor, Ideal calibration
(d) Med tumor, SKAR calibration
(e) Deep tumor, Ideal calibration
(f) Deep tumor, SKAR calibration
Figure 27. Breast images for real breast model
Figure 28. Scheme of the low resources approximation operations in the image reconstruction representation maintains enough information to perform the image reconstruction. Following the data processing path however, we will need to perform some choices that will affect the data representation. The path the data has to go through is represented in Figure 28 to highlight the arithmetic operations that are performed on it. The main issue to be taken into consideration while designing this arithmetic blocks, is that we have to avoid overflow in computation and excessive approximation. At the same time, increasing the data parallelism would imply an increase in hardware complexity.

When an addition is performed between two binary values, the result may need to be represented with an additional bit with respect to the addends to avoid overflow. In a similar way, the multiplication operations need to double the number of bits to ensure a correct result. We
want to keep the data parallelism as low as possible while keeping this facts in consideration.

This is not an easy task if we consider that the dimension of the adder tree depends also on the number of FIR in our system: up to now we have considered a simple case with nine filters, but we want to keep this design as flexible as possible for other applications.

The first approximation strategy we are going to consider tries to keep the data parallelism to the lowest possible value. In the first multiplication, the 16-bit wide inputs are multiplied and a 32 bits output is produced. Before the addition, a reduction is performed in stage $B$ where the 16 least significant bits of the multiplication outcome are discarded. Then, to avoid possible overflow, the additions are performed over 32-bit values. This parallelism ensures no overflow for a tree sum up to 16 levels wide. Before the final multiplication (needed for computing the energy square) the 16 least significant bits are discarded once again in stage $D$, while the multiplication result is represented over 32 bits. Figure 29 shows a graphic representation of this approximation process. An analysis on the performance given by this configuration was performed by simulating it with Matlab/Octave. The simulation code from (12) was adapted and rewritten to emulate the effects of the bit manipulations described, as shown in listing 7.1.

This code includes not only the operations on the parallelism that we have described, but also the effects of the possible overflow that may occur during the sum operations. The results of this simulations are compared to the one previously performed in table Table XI and table Table XII for ideal breast model, and in table Table XIII and table Table XIV for real breast model.
Figure 29. Bit approximation flow for low resources

Table XI shows the variations in SCR and SMR values due to the data approximation for an ideal breast model. For the SMR value the decrease is noticeable both for ideal calibration and for SKAR calibration. We understand this fact remembering that for a tumor in deep position the contrast value is lower due to the weakening of the signal. This induces a lower contrast between the energy at the tumor position and the average value. With this...
Listing 7.1. Octave code for approximated energy computation with bit cut before energy square

```octave
td=single(zeros(NRX,NCAMP+na));
g=single(zeros(NRX,NCAMP+na));
zdiv=double(zeros(NRX,NCAMP+na+L-1));
z=zeros(1,NCAMP+na+L-1);

% Compute time delay
for ant=1:NRX
    %td(ant,delay(ant,ii,jj)+1:delay(ant,ii,jj)+NCAMP)=xfin(ant,:);
    td(ant,delay(ant)+1:delay(ant)+NCAMP)=xfin(ant,:);
end

% Compute window g
for instant=0:NCAMP-1
    for ant=1:NRX
        zdiv(ant,na+instant)=sum(fix(bitshift(int32(g(ant,na+instant-L+1:ant*L)).*double(reshape(w_conv((ant-1)*L+1:ant*L,1,L)),-16)));
    end
end

% % % % % % CUTTING 16 BITS BEFORE (energy)^2
z=floor((mod(double(sum(zdiv))+2^(bitQ*2-1),2^(bitQ*2))-2^(bitQ*2-1)).*2^32);
p=(mod(double(sum(z(nh:nh+lh).^2))+2^(bitQ*2-1),2^(bitQ*2))-2^(bitQ*2-1)).*2^32;
```

kind of approximation, this behaviour gets worse because of the 16bit cut before of the last multiplication: the higher values of energy are rounded downward, decreasing noticeably the maximum value and thus increasing even more the difference with the average value. The SCR value presents a similar trend with a moderate decrease for low depth tumor, while decreasing up to -59% for high depth tumor. The origin of this decrease is the same as explained before.
We can assume that this effects on SCR and SMR bring with them a loss in accuracy.

TABLE XII
MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE FIRST APPROXIMATION STRATEGY

<table>
<thead>
<tr>
<th>CALIBRATION</th>
<th>TUMOR POS.</th>
<th>MAX ERROR</th>
<th>AVG ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>low</td>
<td>26.78%</td>
<td>4.76%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>med</td>
<td>14.64%</td>
<td>1.75%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>deep</td>
<td>17.97%</td>
<td>1.42%</td>
</tr>
<tr>
<td>SKAR</td>
<td>low</td>
<td>33.60%</td>
<td>5.12%</td>
</tr>
<tr>
<td>SKAR</td>
<td>med</td>
<td>15.61%</td>
<td>2.05%</td>
</tr>
<tr>
<td>SKAR</td>
<td>deep</td>
<td>30.68%</td>
<td>5.98%</td>
</tr>
</tbody>
</table>

This is partially confirmed by table Table XII, in which we can see the maximum and average variations for the energy values computed. The maximum error is obtained with a weighted process as in the following equation, where \( rect \) represents the energy matrix obtained without approximations and \( rect_{fp} \) is the one obtained with the fixed point approximation we are considering. The error is measured by comparing the energy values normalized with the maximum values obtained with or without approximation respectively.

\[
MaxError = \max \left( \left| \frac{rect}{\max(rect)} - \frac{rect_{fp}}{\max(rect_{fp})} \right| \right) \quad (7.9)
\]
We see that the average error is consistent, and the maximum can get very high.

The performance degradation becomes clear when observing the images reconstructed during the simulation in Figure 30 and comparing them with the previous ones. The highpass filter effect due to the bitshift before the square computation causes a clear loss in the image accuracy, making it very difficult to extract useful information.

### TABLE XIII

SCR AND SMR COMPARISON BETWEEN IDEAL AND APPROXIMATED COMputation, REAL BREAST MODEL

<table>
<thead>
<tr>
<th>Calibration</th>
<th>Tumor pos.</th>
<th>SCR</th>
<th>SMR</th>
<th>variation</th>
<th>variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>low</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>14.920</td>
</tr>
<tr>
<td>IDEAL</td>
<td>med</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>11.136</td>
</tr>
<tr>
<td>IDEAL</td>
<td>deep</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>13.970</td>
</tr>
</tbody>
</table>

### TABLE XIV

MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE FIRST APPROXIMATION STRATEGY

<table>
<thead>
<tr>
<th>CALIBRATION</th>
<th>TUMOR POS.</th>
<th>MAX ERROR</th>
<th>AVG ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>low</td>
<td>84.12%</td>
<td>14.26%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>med</td>
<td>69.44%</td>
<td>6.56%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>deep</td>
<td>18.49%</td>
<td>4.33%</td>
</tr>
</tbody>
</table>
Figure 30. Breast images for ideal breast model
Moving to the real breast model in table Table XIII we see the same trends for SCR and SMR. Even worst maximum and average errors are shown in table Table XIV with respect to the ideal breast model. The reasons for this behaviour have been explained in the previous section. Since even without approximation the detection with real breast model and SKAR calibration has a very low accuracy, it is no surprise that in this case we obtain again inaccurate reconstruction. Reconstructed images of the breast obtained from the simulations are reported in Figure 31. As expected, with real breast model the images are even more distorted.

This amount of errors influences the outcome of the tumor diagnosis, leading to some unsuccessful detections. However, the positions of the maximum energy values detected shown in table Table XV are not so much worse than in the previous case. But again, remember that the maximum energy value position in just a rough guide to evaluate the performance. In any case, this analysis leads us to the conclusion that this approximation strategy is not suitable for our purposes.

### 7.3.3 Medium resources approximation

We have shown how applying the previously described approximation the precision loss in the detection is too large to be accepted. By deeply analysing the data representation at each stage using Modelsim software, we deduced that the highest amount of approximation is introduced by the bit shift performed before the square operation. This is because even if the results of the sum tree is represented on 32 bits, the actual values require much less bits. We use 32 bits to be conservative, but it is very unlikely that all of them are needed since it would happen only if all of the weights and samples values were maximum. Even in that case, the
Figure 31. Breast images for ideal breast model
TABLE XV
TUMOR DETECTION OUTCOMES, FIRST APPROXIMATION

<table>
<thead>
<tr>
<th>Actual Position</th>
<th>Breast Model</th>
<th>Calibration</th>
<th>Detected Position</th>
<th>Detection</th>
</tr>
</thead>
<tbody>
<tr>
<td>44 ; 88</td>
<td>ideal</td>
<td>IDEAL</td>
<td>46 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>ideal</td>
<td>IDEAL</td>
<td>88 ; 89</td>
<td>YES</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>ideal</td>
<td>IDEAL</td>
<td>132 ; 89</td>
<td>YES</td>
</tr>
<tr>
<td>44 ; 88</td>
<td>ideal</td>
<td>SKAR</td>
<td>45 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>ideal</td>
<td>SKAR</td>
<td>88 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>ideal</td>
<td>SKAR</td>
<td>131 ; 88</td>
<td>YES</td>
</tr>
<tr>
<td>44 ; 88</td>
<td>real</td>
<td>IDEAL</td>
<td>42 ; 95</td>
<td>YES</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>real</td>
<td>IDEAL</td>
<td>76 ; 90</td>
<td>NO</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>real</td>
<td>IDEAL</td>
<td>144 ; 96</td>
<td>NO</td>
</tr>
<tr>
<td>44 ; 88</td>
<td>real</td>
<td>SKAR</td>
<td>49 ; 77</td>
<td>NO</td>
</tr>
<tr>
<td>88 ; 88</td>
<td>real</td>
<td>SKAR</td>
<td>139 ; 1</td>
<td>NO</td>
</tr>
<tr>
<td>131 ; 88</td>
<td>real</td>
<td>SKAR</td>
<td>139 ; 1</td>
<td>NO</td>
</tr>
</tbody>
</table>

actual number of bits would depend on the number of antennas. We cannot reduce the number of bits used for the sum since we cannot know the actual number of bits the computation would require.

Another possibility is to keep all the 32 bits for the square computation, obtaining a 64 bits result. The 32 least significant bits of the result are then cut away. This solution requires of course a larger multiplier, and with that a higher complexity and amount of resources. The scheme of this approximation process is shown in Figure 32, while the Matlab/Octave code written to simulate this behaviour is in listing 7.2.
Listing 7.2. Octave code for approximated energy computation with bit cut after energy square

```octave
td=single(zeros(NRX,NCAMP+na));
g=single(zeros(NRX,NCAMP+na));
zdiv=double(zeros(NRX,NCAMP+na+L-1));
z=zeros(1,NCAMP+na+L-1);

%%Compute time delay
for ant=1:NRX
    td(ant,delay(ant,ii,jj)+1:delay(ant,ii,jj)+NCAMP)=xfin(ant,:);
end

%%compute window g
for instant=0:NCAMP-1
    for ant=1:NRX
        zdiv(ant,na+instant)=sum(fix(bitshift(int32(g(ant,na+instant-L+1:na+instant).*double(reshape(w_conv((ant-1)*L+1:ant*L),1,L))),-16)));
    end
end

%%%%% CUTTING 32 BITS FROM ENERGY VALUES
z=mod(double(sum(zdiv))+2^((bitQ*2-1),2^((bitQ*2)))-2^((bitQ*2-1));
p=(double(mod(sum(floor((z+nh+lh)*2^32)))+2^((bitQ*2-1),2^((bitQ*2)))-2^((bitQ*2-1)))*.2^32;
```

This new configuration was analysed performing again all the simulations. Again, we first consider the ideal breast model. Effects on the SCR and SMR values are shown in table Table XVI.

Surprisingly, the SMR and SCR results see an improvement with respect to a simulation without approximation. This behaviour can be understood considering that this new approxi-
Figure 32. Scheme of the medium resources approximation operations for the image reconstruction

Information strategy applies a highpass process on the results. This implies that the lowest energy values are wiped out. For the SMR this means a lower average value that increases the contrast factor. Similar effect applies to the SCR.

Even if the variations in SCR and SMR may lead to think of an improvement in the accuracy, we should consider that this variations are not due to better calibration (that is actually the same as in the previous case) but just to a wipe out of the lowest energy values. This means a loss of information, and the results can only be deteriorated. What we can say is that there is a chance that this deterioration does not compromises the tumor detection since it does not decrease the contrast indexes, but instead helps to highlight the most significant information from the computation.

The overall maximum error values, reported in table Table XVII, are a little lower than in the previous case with the exception of deep tumor position and SKAR calibration. As long as
TABLE XVI

SCR AND SMR COMPARISON BETWEEN IDEAL AND SECOND APPROXIMATED COMPUTATION

<table>
<thead>
<tr>
<th>Calibration</th>
<th>Tumor pos.</th>
<th>SCR</th>
<th>SMR</th>
<th>variation</th>
<th>SCR</th>
<th>SMR</th>
<th>variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>low</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>18.781</td>
<td>27.738</td>
<td>+47.69%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>med</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>18.557</td>
<td>24.842</td>
<td>+33.87%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>deep</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>17.199</td>
<td>23.059</td>
<td>+34.07%</td>
</tr>
<tr>
<td>SKAR</td>
<td>low</td>
<td>13.818</td>
<td>14.252</td>
<td>+3.14%</td>
<td>16.012</td>
<td>25.646</td>
<td>+60.17%</td>
</tr>
<tr>
<td>SKAR</td>
<td>med</td>
<td>8.842</td>
<td>10.030</td>
<td>+13.43%</td>
<td>16.602</td>
<td>23.169</td>
<td>+39.55%</td>
</tr>
<tr>
<td>SKAR</td>
<td>deep</td>
<td>1.719</td>
<td>2.072</td>
<td>+20.54%</td>
<td>12.506</td>
<td>23.200</td>
<td>+85.52%</td>
</tr>
</tbody>
</table>

This values are small, we can think that the points in which the maximum errors occur (whose value has been wiped out by the highpass effect) are points with very low detected energy, and that maybe are not very significant for the tumor detection. Maximum error values like 34% on the other hand imply that even points with medium energy have been wiped out, and this may induce errors like bad tumor shape detection.

We report the images that represent the reflected energy for this simulations in Figure 33. We can now see how the performance improve with this new approximation strategy. Accuracy is still lower than in the ideal case without approximations.

As always, moving to the real breast model we see worse performance as shown in Table XVIII. We see an SMR increase so large that it makes us suspicious on the validity of this value as index of algorithm accuracy in this case. The same thing is suggested by the values in table Table XIX. With error values as high as the 82%, it is clear that the information excluded
Figure 33. Breast images for ideal breast model and medium resources approximation
TABLE XVII

MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE SECOND APPROXIMATION STRATEGY

<table>
<thead>
<tr>
<th>CALIBRATION</th>
<th>TUMOR POS.</th>
<th>MAX ERR</th>
<th>AVG ERR</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>low</td>
<td>23.28%</td>
<td>0.60%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>med</td>
<td>4.94%</td>
<td>0.56%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>deep</td>
<td>6.70%</td>
<td>0.74%</td>
</tr>
<tr>
<td>SKAR</td>
<td>low</td>
<td>25.02%</td>
<td>1.17%</td>
</tr>
<tr>
<td>SKAR</td>
<td>med</td>
<td>8.67%</td>
<td>0.89%</td>
</tr>
<tr>
<td>SKAR</td>
<td>deep</td>
<td>34.46%</td>
<td>2.68%</td>
</tr>
</tbody>
</table>

by the approximation effect are not only some low energy value but also significant values for the tumor detection.

TABLE XVIII

SCR AND SMR COMPARISON BETWEEN IDEAL AND APPROXIMATED COMPUTATION, REAL BREAST MODEL

<table>
<thead>
<tr>
<th>Calibration</th>
<th>Tumor pos.</th>
<th>SCR</th>
<th>SMR</th>
<th>variation</th>
<th>variation</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>low</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>14.920</td>
</tr>
<tr>
<td>IDEAL</td>
<td>med</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>11.136</td>
</tr>
<tr>
<td>IDEAL</td>
<td>deep</td>
<td>Inf</td>
<td>Inf</td>
<td>NaN</td>
<td>13.970</td>
</tr>
</tbody>
</table>

Figure 34 shows the real breast model reconstructed images obtained with this approximation. Even if this approximation strategy is better than the previous one, we could still use some
TABLE XIX

MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE SECOND APPROXIMATION STRATEGY

<table>
<thead>
<tr>
<th>CALIBRATION</th>
<th>TUMOR POS</th>
<th>MAX ERROR</th>
<th>AVG ERROR</th>
</tr>
</thead>
<tbody>
<tr>
<td>IDEAL</td>
<td>low</td>
<td>82.37%</td>
<td>1.58%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>med</td>
<td>46.73%</td>
<td>3.38%</td>
</tr>
<tr>
<td>IDEAL</td>
<td>deep</td>
<td>18.49%</td>
<td>1.46%</td>
</tr>
</tbody>
</table>

accuracy improvement. As we see, it is still possible to identify the location of the tumor, but we have lost a lot of information about its shape and surroundings. Since so much information is lost, it is possible that in case of weaker reflected signals (for example due to some unusual permittivity values) some small tumor could remain undetected. Also, sometimes analysing the lower energy reflections shape can help us understand if we are dealing with actual tumors or with unwanted artifacts.

7.3.4 Full accuracy approximation

Reducing the approximations performed in the computation process, we can still add some more accuracy to our design. In the previous configuration we decided to perform a bit reduction on the result to obtain a 32bit from a 64bit one. This seems a logic choice if we consider that all the memories in the design have a 32bit parallelism. With this bit manipulation we obtain data that reduce the amount of memory required, together with the amount of data to be transmitted to the PC.

If we want to obtain more accuracy, we should however consider to avoid this loss of information.
Figure 34. Breast images for real breast model and medium resources approximation.
We have seen how critical is the tumor detection with real breast model, and thus we should keep any flavour of possible information within the computation in order to be able to later add some additional processing step to improve the performance. To keep the maximum level of accuracy, we decide then to maintain a 64\textit{bit} result. This decision does not affect the design as much as one may think. To store the 64\textit{bit} energy value we still use a 32\textit{bit}-wide memory. Every energy value is split in two and stored in two contiguous memory locations in little-endian fashion. To memorize this two half values no additional time is required since the FPGA memories are natively implemented with double port access. Of course additional memory will be needed to store the energy values, but this increase is still minor with respect to the amount of memory required to store the weights values. Similarly, more time will be needed to transmit
the computed energy values to the PC but this overhead is not very significant compared to
the time required to transmit the weights and delay values. Figure 35 illustrates the steps of
the data manipulation process.

With this approach we are able to keep almost all of the information within the output data. If
we observe table Table XXI we will see how in the ideal breast case the SCR and SMR values
are basically the same, with very little variations probably due to the bit reduction after the
FIR multiplications. Same behaviour we see in table Table XXII for the real breast model.

Indeed in table Table XXIII we can see how the maximum difference between this new
simulations outcome and those performed with Matlab/Octave native format are very small
, such that the average error was omitted because insignificant. The outcome of this kind of
design is so close to the simulation without approximations that we do not present here images
and tumor detected positions since the reader would not find any interesting difference from the ones previously shown.
## TABLE XXIII
MAXIMUM AND AVERAGE DIFFERENCE FOR ENERGY VALUES FOR THE THIRD APPROXIMATION STRATEGY

<table>
<thead>
<tr>
<th>Breast Model</th>
<th>Calibration</th>
<th>Tumor Pos.</th>
<th>Max Error</th>
</tr>
</thead>
<tbody>
<tr>
<td>ideal</td>
<td>IDEAL</td>
<td>low</td>
<td>0.25%</td>
</tr>
<tr>
<td>ideal</td>
<td>IDEAL</td>
<td>med</td>
<td>0.07%</td>
</tr>
<tr>
<td>ideal</td>
<td>IDEAL</td>
<td>deep</td>
<td>0.16%</td>
</tr>
<tr>
<td>ideal</td>
<td>SKAR</td>
<td>low</td>
<td>0.44%</td>
</tr>
<tr>
<td>ideal</td>
<td>SKAR</td>
<td>med</td>
<td>0.16%</td>
</tr>
<tr>
<td>ideal</td>
<td>SKAR</td>
<td>deep</td>
<td>0.48%</td>
</tr>
<tr>
<td>real</td>
<td>IDEAL</td>
<td>low</td>
<td>0.72%</td>
</tr>
<tr>
<td>real</td>
<td>IDEAL</td>
<td>med</td>
<td>0.72%</td>
</tr>
<tr>
<td>real</td>
<td>IDEAL</td>
<td>deep</td>
<td>0.58%</td>
</tr>
<tr>
<td>real</td>
<td>SKAR</td>
<td>low</td>
<td>1.12%</td>
</tr>
<tr>
<td>real</td>
<td>SKAR</td>
<td>med</td>
<td>1.12%</td>
</tr>
<tr>
<td>real</td>
<td>SKAR</td>
<td>deep</td>
<td>1.22%</td>
</tr>
</tbody>
</table>
CHAPTER 8

DESIGN FLEXIBILITY AND PERFORMANCE

Now that we have defined the design characteristics, we must evaluate its efficiency. Our main goal, as previously discussed, is to accelerate the image reconstruction process by implementing in hardware some of its steps. But is this hardware version really faster than the software one? In this chapter we analyse this issue together with the resources requirements of the design.

We should consider that the test design described so far fits a specific case in which nine antennas are used to sample the reflected signals. This choice derives from the fact that this design was easy enough to simulate on Octave with the available data. Anyway this will not necessarily be the actual design choice when the system will be completely implemented. A different number of antennas could be required to increase accuracy of the scan, and actually we already presented this possibility when discussing the C language software implementation.

Beside that, until now we have been thinking of the computation of 2D images of the breast. It is although probable that an image reconstruction unit able to handle 3D images could be used in order to obtain more detailed information on the tumor position. For this kind of design, even if the algorithm would be substantially the same, the number of antennas required to obtain consistent information could be significantly higher.

All this considerations lead to a design implementation that is as flexible as possible, in order to be easily adapted to various possible applications.
8.1 **Parametric implementation**

As said, the number of antennas actually used to scan the breast depends on a number of factors that should be taken in consideration when performing the complete design of the breast cancer detection system. While designing the image reconstruction unit, we should not remain locked to only one possible configuration: we want our design to be as flexible as possible so that it could be adapted to various applications.

Part of the solution was to transform the source code so that the resources declaration and implementation was parametric. That’s to say, given a fixed parameter defined in a package, the design is automatically derived based on the value of that parameter. In this way, in order to adapt the design to a different implementation the designer just needs to change the value of that parameter. The actual resizing is automatically performed by the synthesizer. In our case, we defined the two parameters $N_{\text{ANT}}$ and $N_{\text{FIR}}$ that specify respectively the number of antennas and filters we are taking in consideration for our design.

However, the design cannot simply scale up with the number of antennas. Each FIR requires a large number of multipliers and adders, so we should consider that there will be a maximum amount of FIRs and resources in general that we will be able to allocate given some area and power constraints. At the same time, we don’t want the design to be dependent on the specific application we consider for the performance evaluation. The development board used for testing is not the final platform that will be part of the complete breast cancer detection system. We need a solution that allows us to have any number of antennas without depending on the number of FIRs we can implement on the board.
8.1.1 **Iterative energy computation concept**

The solution is actually quite simple. As we know each filter is used to process the sample coming from one of the antennas with the associated weights. Until now, we have considered an implementation in which for each antenna a dedicated filter is defined because this solution is for sure the fastest and least complex available.

Analysing the FIR computation processes it becomes clear that the operations performed by each of the filters are independent from each other in the BEAF algorithm. This means that the computations on the data coming from different antennas does not need to be necessarily performed in parallel at the same time. Two opposite possibilities then arise: a parallel implementation, in which all the data is processed at the same time, and a serial implementation in which one filter is iteratively used to process data coming from each of the antennas one after the other\(^{(12)}\). Since the FIR outputs must be summed to compute the energy value, an accumulator will be inserted at the output of the filter in the serial implementation. Figure 36 shows the block scheme of this two possible designs. This two possible versions of the image reconstruction unit datapath were already analysed in \((12)\). In that case the parallel one was selected to minimize computation time. The serial one would solve our resources problem, but we want to avoid the excessive increase of the computation time.

Analysing our case it becomes clear that it is actually possible a hybrid solution between parallel and serial implementations. We could instantiate a number of filters \(N_{FIR}\) that matches our timing and resources requirements, and use them iteratively to process the data coming from \(N_{FIR}\) antennas at each time until all the data have been processed. A scheme of this solution
This solution is truly flexible, since allows to work with any amount of resources for any number of antennas. At each iteration, the results of the partial sum of the FIR outputs are fed back to a shift register. This values are then retrieved at the next iteration and summed to the new FIR outputs. When all the antennas sets have been processed, the final result is fed to the square block instead of the shift register. Of course this solution causes a little increase in complexity requiring additional memory and control logic, but this overhead is small enough to not compromise the benefits of this approach. The shift register used to store the partial results is actually quite small, since it needs to store only the values that are significant for the
energy computation, that are those in the interval defined by the values $n_h$ and $l_h$. This is why the loop back feeds the shift register from the output of block Window 2.

### 8.1.2 BEAF modifications

To realize the above described design, several layers of complexity need to be added to our source code. We must add some control signals to the BEAF block in order to correctly handle the data feed back:

- **loop rst**: this signal is used to reset the shift register that stores de partial results. This operation typically needs to be performed once for every energy value.

- **last FIR loop**: this signal is triggered by the control logic when the current filters processing iteration is the last we need to complete the computation of the energy value.
In this case, the output of the Window 2 block is fed to the Energy computation block that computes the energy value. This signal also allows to enable the Energy Memory at the right time to store this value.

- **loop enable**: this signal is actually controlled by the Window 2 block. We have discussed how we only need to store in the shift register some of the values produced by the sum tree. To correctly enable the shift register at the right time and for the right interval, we extended the functionalities of the Window 2 block which already had a similar job for controlling the Energy Computation block.

Of course the declaration of filters and Window 1 blocks must be parametrized, just as shown in Listing 8.1.

**Listing 8.1. FIR parametric declaration**

```vhdl
Instance_window_1: for i in 0 to NFIR-1 generate
    Inst_window_1: window_1 PORT MAP(
        datain => Window_1_in(i),
        dataout => filter_in(i),
        clk => clk,
        go_filter => go(0)
    );
end generate Instance_window_1;

Instance_fir_array: for i in 0 to NFIR-1 generate
    Inst_fir: fir PORT MAP(
        sample_in => filter_in(i),
        q=>q(i+1),
        clk => clk,
        rst => rst,
        sample_out => filter_out(i)
    );
end generate Instance_fir_array;
```
The FIR declaration is quite straightforward, but we cannot say the same for the sum tree. We know that the outputs of the FIR filters are added together thanks to a sum tree. Now that the number of FIR filters has become variable, we also need to parametrize the sum tree so that its connections and its number of levels automatically adapts to the number of filters we are using. The code in Listing 8.2 shows this connections.

**Listing 8.2. Sum tree parametric declaration**

```vhdl
type arrayvectoradder is array (0 to NFIR) of std_logic_vector(double_word_length-1 downto 0);
type arrayvectortreeadder is array (1 to integer(ceil(log2(real(NFIR+1))))+1) of arrayvectoradder;

signal adder_connections: arrayvectortreeadder;

[...]

Inst_tree_inputs: for i in 0 to NFIR-1 generate
    adder_connections(1)(i) <= filter_out(i);
end generate Inst_tree_inputs;

adder_connections(1)(NFIR)<= loop_register_out when loop_enable = '1' else (OTHERS => '0');

Inst_adder_tree: for j in 1 to integer(ceil(log2(real(NFIR+1))))-1 generate
    Inst_adder_level: for l in 0 to integer(floor(real(NFIR+1)/real(2*j))-real(1)) generate
        Inst_adder_leaf: adder

            GENERIC MAP (Nbit => double_word_length)
            PORT MAP(
                a => adder_connections(j)(2*l),
                b => adder_connections(j)(2*l+1),
                clk => clk,
                c => adder_connections(j+l)(1)
            );
    end generate Inst_adder_level;
end generate Inst_adder_tree;

Inst_adder_retard: if not((integer(ceil(real(NFIR+1)/real(2**(j-1)))) mod 2) = 0) generate
    Inst_tree_reg: register_par

        Generic map (bit_size=>double_word_length)
        Port map (
```
The array `adder_connections` is a collection of signals: 32 bits vectors are grouped in a number of layers equal to the number of layers in the sum tree. The real difficulty derives from the fact that the number of layers depends on the number of FIR filters plus one for the loop shift register output. Each of this layers contains a number of vectors equal to the number of filters. Every layer of vectors is used to connect one stage of the sum tree to the next one. The number of layers is given by the following formula:

\[ N_{\text{layers}} = \lceil \log_2(N_{\text{FIR}} + 1) \rceil \]  \hspace{1cm} (8.1)
Of course when moving on along the tree we will need less and less vectors to carry the data since each addition stage divides by two the remaining number of data values. The fact that every layer has the same number of arrays is just due to avoid excessive complexity in the code writing (we are not even sure that the code could be written in any other way). The unused signals are fixed to zero and should be removed by the synthesizer optimizations.

In this tree declaration we can see not only adders but also registers. The reason for this is that at each layer the partial values are added together in couples, but their number is not always a multiple of two. This means that sometimes, depending on the number of filters, a value exiting from one of the sum layers passes through the next layer without being added with any value. Since the other values are delayed by the adders (as you may remember, each adder has a couple of registers at the input), we need to add a register to delay its travelling to the next level. Figure 38 shows an example of this mechanism for a $N_{FIR} = 4$ design.

To determine the number of adders in each layer, we must consider both the geometric scaling and the data overhead due to the presence of uneven elements. Thanks to the floor operation, we can compute the number of adders with the following formula:

$$N_{add} = \left\lfloor \frac{N_{FIR} + 1}{2 \cdot j} \right\rfloor, \quad 1 \leq j \leq \lceil \log_2(N_{FIR} + 1) \rceil$$

(8.2)

where $j$ identifies the layer we are considering. When we have analysed the design layout in the previous chapters, we have seen how the start control signal for the BEAF block is connected to the Window 2 block after being delayed for an amount of clock cycles such that
the FIR and sum tree have the time to process the data. The number of clock cycles this signal needs to be delayed depends thus on the sum tree depth. The declaration of the shift register used for the delay needs to be parametric, as in Listing 8.3.

**Listing 8.3. Control signal shift register declaration**

```vhdl
signal go : std_logic_vector(1+7+integer(ceil(log2(real(NFIR+1)))+1 downto 0);

Inst_control_go : for i in 0 to 1+7+integer(ceil(log2(real(NFIR+1)))) generate --NUMBER OF CLOCKS REQUIRED TO COMPUTE SUM
    Inst_ffd1 : ffd1 PORT MAP(
        a => go(i),
        clk => clk,
        o => go(i+1)
    );
end generate Inst_control_go;
```

Figure 38. Sum tree connection mechanism in case of four filters
8.1.3 Memories

As explained, each of the Sample Memories contains the sampled signal values from two antennas. It is then clear that the number of memories needed depends on the number of antennas in our design. The memories parametric generation is reported in Listing 8.4.

```
Listing 8.4. Signal Memory parametric declaration
SmemInst: for i in 1 to integer(ceil(real(NANT)/real(2))) generate
  smemInst: SAMPLES_MEM,512x32 port map (  
    clka=>clk,  
    dina=>SMEM_Din(i),  
    addra=>SMEM_Addr(i),  
    ena=>SMEM_En(i),  
    wea=>SMEM_We(i),  
    douta=>SMEM_Dout(i) );
end generate;
```

We have discussed how the number of antennas may not match the number of FIR filters in the BEAF block. At each iteration of the FIR computation the appropriate Samples Memories will need to be accessed, so that at the end all the samples will have been processed. Some control logic is thus needed that keeps count of the FIR loop stage we are in and connects the correct memories to the BEAF block. For this purpose an additional counter was included in the design, reported in Listing 8.5. This counter uses as control input the end signals generated by Counter FIR and Counter ANT. Before each iteration, new weights data needs to be loaded in the shift register so that they are available to the FIRs. When this operation is performed the new counter understands that a new FIR iteration is starting, and increments its value.
The value of this counter is then used to address the correct Samples Memories output and feed them to the FIR filters, as in Listing 8.6. Note that even if at each time only some of the memories are connected to the filters, all of them are active at the same time. This probably leaves room for improvements to reduce power consumption.

Listing 8.5. Samples memory counter

```vhdl
count_smem_proc : process(clk, rst, count.FIR_end, count.ANT_end)
    variable flag : std_logic;
begin
    if rst = '1' then
        count.smem_index <= 0;
        flag := '0';
    elsif clk'event and clk = '1' then
        if loop_rst = '1' then
            count.smem_index <= 0;
            flag := '0';
        elsif (count.FIR_end = '1' or count.ANT_end = '1') and EN.CFIR = '1' then
            if (flag = '0') then
                flag := '1';
            elsif count.smem_index < integer(cei(real(NANT)/real(NFIR)))-1 then
                count.smem_index <= count.smem_index + 1;
            end if;
        end if;
    end if;
end process count_smem_proc;
```

Listing 8.6. Samples memory connections

```vhdl
type arrayvectorin is array (NFIR-1 downto 0) of std_logic_vector(word_length-1 downto 0);
type arrayvectorsmem is array (NANT-1 downto 0) of std_logic_vector(word_length-1 downto 0);
type arrayvectorsmemqueue is array (integer(cei(real(NANT)/real(NFIR)))-1 downto 0) of arrayvectorin;
signal smem_array : arrayvectorsmem;
signal smem_shift : arrayvectorsmemqueue;

[...]
```
smem_array.in: for i in 0 to NANT-1 generate
smem_array (i) <= SMEM_Dout (integer (floor (real (i)/real (2))) + 1)(31 - integer (i mod 2) * 16 downto 16 -
integer (i mod 2) * 16);
end generate;

smem_connection: for i in 0 to integer (ceil (real (NANT)/real (NFIR))) - 1 generate
    smem_distribution: for j in 0 to NFIR-1 generate
        too_many_fir: if i*NFIR+NFIR > NANT generate
            order_like_delay: if i*NFIR+j < NANT generate
                smem_shift (i)(i*NFIR+NFIR-NANT+j) <= smem_array (i*NFIR+j);
            end generate order_like_delay;
        smem_out_bound: if i*NFIR+j>NANT-1 generate
            smem_shift (i)(NFIR-1-j) <= (OTHERS => '0');
        end generate smem_out_bound;
    end generate too_many_fir;
    smem_in_bound: if i*NFIR+NFIR-1 < NANT generate
        smem_shift (i)(j) <= smem_array (i*NFIR+j);
    end generate smem_in_bound;
    end generate smem_distribution;
end generate smem_connection;
samples_in <= smem_shift (count_smem_index);

Since the amount of input required for the filters is now variable, even the shift register used to lead the weights values needs to be parametric. The amount of registers needed will be equal to \( N_{reg} = (L + 1) \cdot NFIR \).

8.2 FPGA vs serial software

Now that the architecture of our design has been completely defined, we can estimate its timing efficiency. We want to be able to compare the time required by this hardware design to perform the beamforming algorithm with the time required by a computer to perform the same task via software. To do so we express the performance variation with a speedup value that is computed as the ratio between the software execution time and the hardware execution time.
We built an emulation prototype of the Imaging Unit. This prototype relies on a general purpose microprocessor to perform the portions of processing that we decided to execute in software, while the hardware part of the BEAF algorithm was deployed on a Xilinx Virtex-4 XC4VLX160-FF1513 FPGA. In particular, we were able to connect via USB the Intel Core-i5 of a laptop with the FPGA located in an AVNET Virtex-4 LX development board.

The prototype has multiple purposes. It serves as an accelerator of the system-level simulation, because it permits a much faster execution of the part of the algorithm executed by the FPGA, compared to a logic-level simulation. The overall architecture, with the software and the hardware running concurrently is clearly a close approximation of the final system, even though the processor and the digital hardware will be different in the final target. The RTL VHDL code that gets synthesized on the FPGA is the same that we will eventually implement in an ASIC, therefore we can fully verify it before fabrication. Finally, we can anticipate what the exact performance will be in terms of clock cycles. (The only difference in performance will be then determined by the different clock frequency in FPGA and ASIC.) Both the software and the hardware code are fully instrumented to permit a cycle accurate evaluation of timing performance.

Of course, the performance will depend on the amount of hardware resources we decide to allocate on the FPGA. Thanks to the changes introduced in the previous paragraph we are now able to program the FPGA with the desired degree of parallelism by simply changing a couple of values in a package. It would then be interesting to see how the performance of the design change with different degree of parallelism.
8.2.1 Direct measurement

This timing operation is performed by the computer that interacts with the FPGA. The actual time required to perform the computation of the energy values would be too small to be directly measured by a general purpose computer. The communication overhead on the other hand is significantly larger, being responsible of most of the execution time. The host application was then rewritten to allow an indirect measurement: as a first step the energy computation process of the entire map is executed, obtaining the energy map. Then the whole process is repeated, but this time only the communication operations are performed while the execution commands are omitted. Of course this procedure produces a meaningless energy map, but that’s not our goal. By measuring the execution time of the two different processes and comparing them we should identify the communication overhead and thus obtain the actual computation time. In the appendix the main function of this modified host application can be found.

Unfortunately, the results of this measurements revealed to be inconsistent. Running repetitively the entire measurement process we obtained outcomes that were different one from the other, sometimes even significantly. Trying to repeat the process a larger number of times and measuring the result as an arithmetic medium did not help. This issue in quite understandable for various reasons. First of all, the native resolution of the ANSI C functions that allow to measure time intervals is very low: about 10\text{ms}. But that’s not the only problem: even having a higher resolution timer we should keep in mind that we are working with a general purpose
computer. While executing our application many other processes are running at the same time, meaning that they are scheduled together. Depending on the amount of other jobs the computer is handling at every time may void the validity of the timing operation: if for example the time measuring function is delayed of even just one millisecond because another process has been scheduled before, the outcome will be inconsistent and useless for our measure. This timing approach appears to be unsuitable for our purposes.

8.2.2 Extrapolated timing

We came to the conclusion that we cannot measure directly the time needed by the FPGA to execute the computation. Let’s take a step backward: we have the complete design of our system, and we know exactly how it works. It is possible to extrapolate the timing information and obtain a theoretical estimate of the number of clock cycles required by each of the computation steps. This solution is actually convenient for our purposes: the estimates obtained in this way will be completely independent from possible overheads due to communication and they will also be independent from the clock period used on the programmable support. In this way the same consideration we are about to make can be extended to other cases with different hardware support.

The FPGA energy computing process can be divided into a series of steps with different purposes. Since we are excluding the communication operations, our first step will consist in loading the BEAF weights and delay values in the shift register (so that they can be fed in parallel to the FIR blocks). The time required by this operation depends directly on the amount
of weights and delay values, which in our case is \( N_{\text{coeff}} = (L + 1) \cdot 9 = 504 \). The load time can be thus computed as:

\[
T_{\text{load}} = 504 \cdot T_{\text{CLK}}
\]  

(8.3)

The load operation however is not performed completely in one time. We have seen how the shift register size is such that it can allocate the required data for the available filters. As a consequence, the load operation must be split and partially executed every time the filters have processed one set of input data, that’s to say once for every iteration. Unfortunately, every time the load operation is started some overhead is required to correctly handle the control signals. This introduces a time delay that altogether can be expressed with the following formula:

\[
N_{\text{iter}} = \left\lceil \frac{N_{\text{ANT}}}{N_{\text{FIR}}} \right\rceil
\]  

(8.4)

\[
T_{\text{overhead}} = N_{\text{iter}} \cdot 2 \cdot T_{\text{CLK}}
\]  

(8.5)

where \( N_{\text{iter}} \) is the number of iterations in which the energy computation is split.

The second step is of course the energy computation. This step includes the time required to feed the FIR filters and compute the energy values. This includes the overheads due to Sample Memory access and the delays introduced by Control Start, Window 1 and Window 2 blocks, plus the time required by the sum tree to perform the additions. A portion of this delay is fixed since it derives from the hardware design of blocks that have not been parametrized. A smaller part depends on the number of filters available and more precisely on the depth of the
sum tree used to add together the results of the filters, and another part on the sum tree inside
the filters whose depth depends on $L$. Additional delay is introduced by the need to wait for
the sample interval defined by $n_h$ and $l_h$.

$$E_{exec\_delay} = n_a + \lceil \log_2(L) \rceil + n_h + l_h + 7 + \lceil \log_2(N_{FIR} + 1) \rceil \quad (8.6)$$

This delay is required for each computation iteration, and thus the total computation time is:

$$T_{comp} = E_{exec\_delay} \cdot N_{iter} \cdot T_{CLK} \quad (8.7)$$

Finally, besides load and computation times we also need some time to store the computed
values in the energy memory or in the shift register used to loop back the values. Together
with the energy storing there is also some additional operations required to handle the end of
the processing of one iteration as we have seen back in Figure 16. All this additional steps can
be added to form a energy save component of the overall execution time, which is computed as
follows:

$$T_{energy\_save} = 3 \cdot N_{iter} \cdot T_{CLK} \quad (8.8)$$

We now have computed all the components that build the execution time. The final formula
of the execution time per voxel can be summarized as:
\[
\frac{time}{voxel} = (504 + 2 \cdot N_{iter} + (n_a + \lceil \log_2(L) \rceil) + n_h + l_h + 7 \\
+ \log_2(N_{FIR} + 1)) \cdot N_{iter} + 3 \cdot N_{iter})T_{CLK}
\] (8.9)

This equation was verified by comparing its results with the outcomes of the Modelsim simulation of the system. If expressed in this terms, this formula can be used only for a 9 antennas system which was the test case we implemented for FPGA testing. However, since our code can be adapted to have any number of filters, it can also be adapted to run for systems with any number of antennas, and we can rewrite the timing formula in a more general form:

\[
\frac{time}{voxel} = ((L + 1) \cdot N_{ANT} + 2 \cdot N_{iter} + (n_a + \lceil \log_2(L) \rceil) + n_h + l_h + 7 \\
+ \log_2(N_{FIR} + 1)) \cdot N_{iter} + 3 \cdot N_{iter})T_{CLK}
\] (8.10)

\section*{8.2.3 Acceleration trend}

Even if we have so far discussed an FPGA implementation, we could be interested in implementing the system with other technologies such as ASIC. Thanks to the general formula we have described we can now obtain the timing performance for any type of possible technology we want to consider. We now compare the some possible hardware solutions with the parallel software performance over 16 threads that we have estimated in the previous chapters. In Figure 39 a comparison between three different technologies is shown: the only difference
between this solutions is the maximum system frequency at which the computation can be
executed, while from any other point of view the timing computation remains exactly the same
(being that the description code is the same).

The first kind of technology we consider

![Speedup comparison for different hardware technologies](image)

**Figure 39.** Speedup comparison for different hardware technologies

is of course the Xilinx Virtex 4 FPGA: the clock frequency we use in this case is not actually
the maximum possible but was chosen to meet easily some design constraints due the control
unit characteristics and the communication interface. It is then probably possible to optimize
this design to run at higher speed, but for the purpose of this paper we can just consider this
sub-optimal solution to have a general idea of the FPGA behaviour. The second solution is a
ASIC implemented with 45nm technology, which is a type of implementation we could actually
consider in a real life scenario. As we see, there is a speedup in performance of 8 times with respect to the FPGA implementation, due to the higher available maximum frequency as we will explain later. Finally, we also considered a possible hardware implementation running at the same speed of the microprocessor used for the software testing to have comparison between hardware and software implementations at the same given frequency, obtaining a speedup of 4.8 with respect to the 45nm ASIC.

As we expected, the real impact to the execution time when changing the number of filters mainly depends on the value of $\lceil N_{\text{ANT}}/N_{\text{FIR}} \rceil$. Increasing the number of filters gives us a significant boost in performance only when it allows us to perform the computation in a lower number of iterations. For example, in this case we would not get a significant advantage from using 49 filters: if we cannot use 50, than we could as well use 26 and still obtain basically the same performance we would see with 49 filters.

We can see how the performance appear to have a saturation trend: when using a large amount of parallelism, the increase in speedup becomes smaller and smaller. This behaviour is due to the memory access time. In the formula discussed before we included a component due to the time required to load the weights into the shift register: even if we are not considering communication overheads due to the interface with the external system, this memory load time reduces the maximum degree of acceleration we can obtain. When increasing the parallelism the portion of the execution time due to the computation is reduced, while the memory access time remains the same and constitutes thus a bottleneck.

Since the different hardware solutions we have considered produce the same speedup with the
exception of a linear increase due to the clock frequency, I will from now on refer to the 45nm ASIC implementation to explore the differences with the software solution.

### 8.3 ASIC vs parallel software

For the analysis of the ASIC implementation performance, we consider a design with 50 antennas that appears to be a realistic number of antennas for a real application. Since the results of the software simulation for this design case appeared to be disturbed by memory effect, even if we have excluded this effects by considering only the two smaller maps for extrapolating the performance trend we include in this section also an analysis for a system with 43 antennas.

All the considerations regarding the software version are the same discussed in Chapter 4. In this case we use as comparison the version with highest parallelism (16 threads).

#### 8.3.1 Single accelerator

For this part of our experiments we targeted a standard cell library in a 1.1 V 45 nm CMOS technology. We synthesized with Synopsys Design Compiler various instances of our architecture varying the number of filters and determined the relationship between speed-up (with respect to a software execution) and silicon area. The design, when synthesized with the maximum optimization effort available, can run at a clock period of 2 ns. Therefore, compared to the FPGA results reported in the previous section, all the performance times can be multiplied by a factor 8 (16 ns / 2 ns).

We imagine to use this hardware accelerator as a high performance peripheral directly connected to a computer as discussed before. For design optimization, we suppose that the accelerator can
have a direct access to memory thanks to some kind of DMA or that anyway it can read from the memory through a high speed link with bandwidth that is not limited by a low performance communication interface such as the USB connection, allowing us to exploit all the memory bandwidth with no significant latency introduced by third elements.

This assumption alone gives us a noticeable reduction of the execution time: if in the FPGA case we needed to first load the weights values into a local memory and then move them to the shift register, now there is no need for a local memory an this means that the memory access time is reduced. To see how the performance has changed, let us consider a breast map with a number of inbreast points equal to the worst case considered in (12), that is a map of $4.2M_{\text{voxel}}$.

The software execution time for a map of this size was extrapolated from the trend highlighted by the results in Chapter 4 for a small and medium map, while the result for the large map was ignored since we have seen how its value is influenced by a lack of memory that should not occur in an optimized breast scan system, and the result was $92916.25s$ for a 50 antennas design.

In the case of a 43 antennas design instead we used the medium and big map to extrapolate the trend, with a resulting time of and $74945s$ for the given map size. The results are shown in Table XXIV, and the trend is plotted in Figure 40 for 50 antennas and Figure 41 for 43 antennas.

As we see the initial values for a serial hardware show that a fully serial design has worst performance than a software one with high parallelism. However we are still able to reach a good speed-up if we introduce some degree of parallelism in the hardware.

The table also indicates the required area with the reference technology, which scales linearly
TABLE XXIV

ASIC SPEED-UP OVER 4.2M VOXELS, WITH 50 AND 43 ANTENNAS AND 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916.25 S AND 74945 S RESPECTIVELY.

<table>
<thead>
<tr>
<th>Ñ</th>
<th>Ñ</th>
<th>T_{exe}[s]</th>
<th>Acceleration</th>
<th>T_{exe}[s]</th>
<th>Acceleration</th>
<th>Area[µm²]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43</td>
<td>138625.20</td>
<td>0.67</td>
<td>119221.20</td>
<td>0.63</td>
<td>217438</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>81295.20</td>
<td>1.14</td>
<td>71072.40</td>
<td>1.05</td>
<td>405976</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>62815.20</td>
<td>1.48</td>
<td>54902.40</td>
<td>1.37</td>
<td>594515</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>53684.40</td>
<td>1.73</td>
<td>45754.80</td>
<td>1.64</td>
<td>783053</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>46729.20</td>
<td>1.99</td>
<td>41118.00</td>
<td>1.82</td>
<td>971591</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>44100.80</td>
<td>2.09</td>
<td>38799.60</td>
<td>2.05</td>
<td>1160129</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>42092.40</td>
<td>2.21</td>
<td>36481.20</td>
<td>2.05</td>
<td>1348667</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>39832.80</td>
<td>2.33</td>
<td>34213.20</td>
<td>2.19</td>
<td>1537205</td>
</tr>
<tr>
<td>9</td>
<td>5</td>
<td>37506.00</td>
<td>2.48</td>
<td>31886.40</td>
<td>2.35</td>
<td>1725743</td>
</tr>
<tr>
<td>10</td>
<td>5</td>
<td>35179.20</td>
<td>2.64</td>
<td>29559.60</td>
<td>2.54</td>
<td>1914281</td>
</tr>
<tr>
<td>11</td>
<td>4</td>
<td>35179.20</td>
<td>2.64</td>
<td>29559.60</td>
<td>2.54</td>
<td>2102819</td>
</tr>
<tr>
<td>12</td>
<td>4</td>
<td>35179.20</td>
<td>2.64</td>
<td>29559.60</td>
<td>2.54</td>
<td>2291357</td>
</tr>
<tr>
<td>13</td>
<td>4</td>
<td>32852.40</td>
<td>2.83</td>
<td>27258.00</td>
<td>2.75</td>
<td>3045510</td>
</tr>
<tr>
<td>14</td>
<td>4</td>
<td>32852.40</td>
<td>2.83</td>
<td>27258.00</td>
<td>2.75</td>
<td>324048</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>32852.40</td>
<td>2.83</td>
<td>27258.00</td>
<td>2.75</td>
<td>3422586</td>
</tr>
<tr>
<td>16</td>
<td>3</td>
<td>32886.00</td>
<td>3.04</td>
<td>27258.00</td>
<td>2.75</td>
<td>361124</td>
</tr>
<tr>
<td>17</td>
<td>3</td>
<td>30550.80</td>
<td>3.04</td>
<td>27258.00</td>
<td>2.75</td>
<td>3899662</td>
</tr>
<tr>
<td>18</td>
<td>3</td>
<td>30550.80</td>
<td>3.04</td>
<td>27258.00</td>
<td>2.75</td>
<td>4054302</td>
</tr>
<tr>
<td>19</td>
<td>3</td>
<td>30550.80</td>
<td>3.04</td>
<td>27258.00</td>
<td>2.75</td>
<td>42092.40</td>
</tr>
<tr>
<td>20</td>
<td>3</td>
<td>30550.80</td>
<td>3.04</td>
<td>27258.00</td>
<td>2.75</td>
<td>4365276</td>
</tr>
<tr>
<td>21</td>
<td>3</td>
<td>30550.80</td>
<td>3.04</td>
<td>27258.00</td>
<td>2.75</td>
<td>4521357</td>
</tr>
<tr>
<td>22</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>4521357</td>
</tr>
<tr>
<td>23</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>4677333</td>
</tr>
<tr>
<td>24</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>4833333</td>
</tr>
<tr>
<td>25</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>4989333</td>
</tr>
<tr>
<td>26</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>5145333</td>
</tr>
<tr>
<td>27</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>5301333</td>
</tr>
<tr>
<td>28</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>5457333</td>
</tr>
<tr>
<td>29</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>5613333</td>
</tr>
<tr>
<td>30</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>5769333</td>
</tr>
<tr>
<td>31</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>5925333</td>
</tr>
<tr>
<td>32</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>6081333</td>
</tr>
<tr>
<td>33</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>6237333</td>
</tr>
<tr>
<td>34</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>6393333</td>
</tr>
<tr>
<td>35</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>6549333</td>
</tr>
<tr>
<td>36</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>6705333</td>
</tr>
<tr>
<td>37</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>6861333</td>
</tr>
<tr>
<td>38</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>7017333</td>
</tr>
<tr>
<td>39</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>7173333</td>
</tr>
<tr>
<td>40</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>7329333</td>
</tr>
<tr>
<td>41</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>7485333</td>
</tr>
<tr>
<td>42</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>7641333</td>
</tr>
<tr>
<td>43</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>7797333</td>
</tr>
<tr>
<td>44</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>7953333</td>
</tr>
<tr>
<td>45</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>8109333</td>
</tr>
<tr>
<td>46</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>8265333</td>
</tr>
<tr>
<td>47</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>8421333</td>
</tr>
<tr>
<td>48</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>8577333</td>
</tr>
<tr>
<td>49</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>8733333</td>
</tr>
<tr>
<td>50</td>
<td>2</td>
<td>28215.60</td>
<td>3.29</td>
<td>24922.80</td>
<td>3.01</td>
<td>8889333</td>
</tr>
</tbody>
</table>
with the number of filters that appear to constitute the largest portion of the design. In Figure 42 the speedup is compared with the trend of the area normalized over the area required for a single filter design.

8.3.1.1 Masked memory access

There is actually another way to increase the speed-up by excluding the memory access time. The hardware design we have described so far is based on the concept that the total execution time is composed of a looping sequence of memory access and computation steps one after the other. Since the computation of each voxel is completely unrelated from the others,
there isn’t actually anything that forbids us from computing the energy from one voxel and at the same time load and store the weights values from the next voxel. If the timing of this two operations are chosen correctly we can completely mask the memory access time with the exception of some set up time at the beginning of the image reconstruction that in any case would be minor with respect to the whole process. To ensure this we must provide a memory bandwidth that allows the system to complete the memory access with a time that is not higher than the computation time. Moreover we can say that the overall execution time is equal to

Figure 41. Acceleration trend for ASIC 43 antennas design with single accelerator
As shown from the values in Table XXV and Table XXVI and represented in Figure 43 and Figure 44 ideally the saturation effect has disappeared if we suppose to have an infinite available bandwidth, even if the relation between filter parallelism and speed-up is still geometric. Of course in a real case performance would be limited by the actual maximum bandwidth of our memory. This data confirms the close relation between the required BW and the speed-up: the faster the execution is, the faster data will have to be read from memory to keep masking the
TABLE XXV

ASIC SPEED-UP OVER 4.2M VOXELS WITH MASKED MEMORY ACCESS, WITH 50 ANTENNAS AND 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916.25S

memory access time. The area was not included in this table because the values are just the same as in Table XXIV.

Please note that the given speedup values in this case are not valid to actually perform a comparison between software and hardware implementations, since modifying the memory bandwidth would affect also the performance of the software program while this results are obtained against a fixed reference just to provide a clearer idea of the performance trend instead of using the execution time.
TABLE XXVI

ASIC SPEED-UP OVER 4.2M VOXELS WITH MASKED MEMORY ACCESS, WITH 43 ANTENNAS AND 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 74945S

<table>
<thead>
<tr>
<th>N_FIR</th>
<th>N_ITER</th>
<th>T_exe [s]</th>
<th>Acceleration</th>
<th>Required BW [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>43</td>
<td>98994.00</td>
<td>0.76</td>
<td>408.66</td>
</tr>
<tr>
<td>2</td>
<td>22</td>
<td>50845.20</td>
<td>1.47</td>
<td>795.64</td>
</tr>
<tr>
<td>3</td>
<td>15</td>
<td>34675.20</td>
<td>2.16</td>
<td>1166.67</td>
</tr>
<tr>
<td>4</td>
<td>11</td>
<td>25527.60</td>
<td>2.94</td>
<td>1584.73</td>
</tr>
<tr>
<td>5</td>
<td>9</td>
<td>20890.80</td>
<td>3.59</td>
<td>1936.47</td>
</tr>
<tr>
<td>6</td>
<td>8</td>
<td>18572.40</td>
<td>4.04</td>
<td>2178.20</td>
</tr>
<tr>
<td>7</td>
<td>7</td>
<td>16254.00</td>
<td>4.61</td>
<td>2488.89</td>
</tr>
<tr>
<td>8</td>
<td>6</td>
<td>13986.00</td>
<td>5.36</td>
<td>2892.49</td>
</tr>
<tr>
<td>9 - 10</td>
<td>5</td>
<td>11659.20</td>
<td>6.43</td>
<td>3469.74</td>
</tr>
<tr>
<td>11 - 14</td>
<td>4</td>
<td>9332.40</td>
<td>8.03</td>
<td>4334.83</td>
</tr>
<tr>
<td>15</td>
<td>3</td>
<td>7005.60</td>
<td>10.70</td>
<td>5774.58</td>
</tr>
<tr>
<td>16 - 21</td>
<td>3</td>
<td>7030.80</td>
<td>10.66</td>
<td>5753.88</td>
</tr>
<tr>
<td>22 - 31</td>
<td>2</td>
<td>4695.60</td>
<td>15.96</td>
<td>8615.38</td>
</tr>
<tr>
<td>32 - 42</td>
<td>2</td>
<td>4712.40</td>
<td>15.90</td>
<td>8584.67</td>
</tr>
<tr>
<td>43</td>
<td>1</td>
<td>2368.80</td>
<td>31.64</td>
<td>17078.01</td>
</tr>
</tbody>
</table>

8.3.2 Multiple accelerators

Until now we have been comparing hardware and software with two different types of parallelism: while the software exploits the parallelism in the computation of different voxels, the hardware uses the parallelism between the filters in the computation of a single voxel.

Actually there is nothing that forbids us to implement multiple accelerators to let them work in parallel, especially considering that the required area for a single accelerator is little. To simplify the analysis of this possible implementation let us consider the parallelization of accelerators with only one filter each, and the results are in Table XXVII where the possible range of parallel accelerators is in the same range of the possible number of filters considered in the
previous case: this choice was made only to make easier a comparison with the previous data, but in this case there is actually no conceptual limit to the amount of parallelism we can add in this case (of course, the limit is the number of voxels that is in any case to actually limit the implementation). It is clear that even if the acceleration appears slightly higher there is no significant difference with respect to the multiple filters solution, and the reason as always is the memory access bound. Even if the computation time was reduced to zero, the maximum acceleration would be bounded to a maximum factor of 4. This trend is shown in Figure 45
Figure 44. Acceleration trend for ASIC 43 antennas design with single accelerator and masked memory access

and Figure 46.

Once again we see that the memory access time is what really limits the performance of the system.

8.3.2.1 Multiple accelerators with masked memory access

We can imagine to apply as before a solution in which the memory access time is masked by the execution, but this time with multiple accelerators. Of course adding more accelerators
TABLE XXVII

ASIC SPEED-UP OVER 4.2M VOXELS WITH MULTIPLE ACCELERATORS, FOR 50 AND 43 ANTENNAS SYSTEMS AND WITH 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916S AND 74945S RESPECTIVELY

<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>138625.20</td>
<td>0.67</td>
<td>119221.20</td>
<td>0.63</td>
</tr>
<tr>
<td>2</td>
<td>81072.60</td>
<td>1.15</td>
<td>69724.20</td>
<td>1.07</td>
</tr>
<tr>
<td>3</td>
<td>61888.40</td>
<td>1.50</td>
<td>53225.20</td>
<td>1.41</td>
</tr>
<tr>
<td>4</td>
<td>52296.30</td>
<td>1.78</td>
<td>44975.70</td>
<td>1.67</td>
</tr>
<tr>
<td>5</td>
<td>46541.04</td>
<td>2.00</td>
<td>40026.00</td>
<td>1.87</td>
</tr>
<tr>
<td>6</td>
<td>42704.20</td>
<td>2.18</td>
<td>36726.20</td>
<td>2.04</td>
</tr>
<tr>
<td>7</td>
<td>39963.60</td>
<td>2.33</td>
<td>34369.20</td>
<td>2.18</td>
</tr>
<tr>
<td>8</td>
<td>37908.15</td>
<td>2.45</td>
<td>32601.45</td>
<td>2.30</td>
</tr>
<tr>
<td>9</td>
<td>36309.47</td>
<td>2.56</td>
<td>31226.53</td>
<td>2.40</td>
</tr>
<tr>
<td>10</td>
<td>35030.52</td>
<td>2.65</td>
<td>30126.60</td>
<td>2.49</td>
</tr>
<tr>
<td>15</td>
<td>31193.68</td>
<td>2.98</td>
<td>26826.80</td>
<td>2.79</td>
</tr>
<tr>
<td>20</td>
<td>29275.26</td>
<td>3.17</td>
<td>25176.90</td>
<td>2.98</td>
</tr>
<tr>
<td>25</td>
<td>28124.21</td>
<td>3.30</td>
<td>24186.96</td>
<td>3.10</td>
</tr>
<tr>
<td>30</td>
<td>27356.84</td>
<td>3.40</td>
<td>23527.00</td>
<td>3.19</td>
</tr>
<tr>
<td>35</td>
<td>26808.72</td>
<td>3.47</td>
<td>23055.60</td>
<td>3.25</td>
</tr>
<tr>
<td>43</td>
<td>26196.87</td>
<td>3.55</td>
<td>22529.39</td>
<td>3.33</td>
</tr>
<tr>
<td>47</td>
<td>25969.05</td>
<td>3.58</td>
<td></td>
<td></td>
</tr>
<tr>
<td>50</td>
<td>25822.10</td>
<td>3.60</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
means that at each computation cycle we will need more data to be fed to them, and thus that the required bandwidth will be higher. From Table XXVIII and Table XXIX we see that in this case we get a small improvement in performance with respect to the multiple filter case. This is mainly due to the pruning of the sum tree that in that case was needed to add the results of the filters together. Another advantage of this solution is that the speed-up trend is much more regular while in the previous design there was an improvement only when there was a change in the number of iterations $N_{\text{iter}} = \lceil \frac{N_{\text{ANT}}}{N_{\text{FIR}}} \rceil$. Just as seen before, this small improvements in performance result in a small increase in the required memory bandwidth. The resulting
Figure 46. Acceleration trend for ASIC 43 antennas design with multiple accelerators

The trend is very similar to the previous one, as shown in Figure 47 Figure 48. Required area is basically the same as in the previous design since the largest part is due to the filter block while the other elements (windowing blocks, FSM etc.) have a very small impact on the occupied area.

As mentioned before, to actually obtain a fair comparison of software versus hardware we would need to estimate the variation of the software performance when changing the available memory bandwidth. To obtain an idea of the actual speedup that the hardware implementation can provide in this case, some tests were performed to find the memory bandwidth of the machine.
### TABLE XXVIII

ASIC SPEED-UP OVER 4.2M VOXELS WITH MULTIPLE ACCELERATORS AND MASKED MEMORY ACCESS, FOR A 50 ANTENNAS SYSTEM AND WITH 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 92916.25S.

<table>
<thead>
<tr>
<th>N_ACC</th>
<th>T_exe[s]</th>
<th>Acceleration</th>
<th>Required BW[MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>115105.20</td>
<td>0.81</td>
<td>408.67</td>
</tr>
<tr>
<td>2</td>
<td>57552.60</td>
<td>1.61</td>
<td>817.34</td>
</tr>
<tr>
<td>3</td>
<td>38368.40</td>
<td>2.42</td>
<td>1226.01</td>
</tr>
<tr>
<td>4</td>
<td>28776.30</td>
<td>3.23</td>
<td>1634.68</td>
</tr>
<tr>
<td>5</td>
<td>23921.04</td>
<td>4.04</td>
<td>2043.35</td>
</tr>
<tr>
<td>6</td>
<td>19184.20</td>
<td>4.84</td>
<td>2452.02</td>
</tr>
<tr>
<td>7</td>
<td>16443.60</td>
<td>5.65</td>
<td>2860.69</td>
</tr>
<tr>
<td>8</td>
<td>14388.15</td>
<td>6.46</td>
<td>3269.36</td>
</tr>
<tr>
<td>9</td>
<td>12789.47</td>
<td>7.27</td>
<td>3678.63</td>
</tr>
<tr>
<td>10</td>
<td>11510.52</td>
<td>8.07</td>
<td>4086.70</td>
</tr>
<tr>
<td>11</td>
<td>10464.11</td>
<td>8.88</td>
<td>4495.37</td>
</tr>
<tr>
<td>12</td>
<td>9592.04</td>
<td>9.69</td>
<td>4904.04</td>
</tr>
<tr>
<td>13</td>
<td>8854.25</td>
<td>10.49</td>
<td>5312.71</td>
</tr>
<tr>
<td>14</td>
<td>8221.80</td>
<td>11.30</td>
<td>5721.37</td>
</tr>
<tr>
<td>15</td>
<td>7673.68</td>
<td>12.11</td>
<td>6130.04</td>
</tr>
<tr>
<td>16</td>
<td>7194.08</td>
<td>12.92</td>
<td>6538.71</td>
</tr>
<tr>
<td>17</td>
<td>6770.89</td>
<td>13.72</td>
<td>6947.38</td>
</tr>
<tr>
<td>18</td>
<td>6394.73</td>
<td>14.53</td>
<td>7366.05</td>
</tr>
<tr>
<td>19</td>
<td>6058.17</td>
<td>15.34</td>
<td>7764.72</td>
</tr>
<tr>
<td>20</td>
<td>5755.26</td>
<td>16.14</td>
<td>8173.39</td>
</tr>
<tr>
<td>21</td>
<td>5481.20</td>
<td>16.95</td>
<td>8582.06</td>
</tr>
<tr>
<td>22</td>
<td>5232.05</td>
<td>17.76</td>
<td>8990.73</td>
</tr>
<tr>
<td>23</td>
<td>5004.57</td>
<td>18.57</td>
<td>9399.40</td>
</tr>
<tr>
<td>24</td>
<td>4796.05</td>
<td>19.37</td>
<td>9808.07</td>
</tr>
<tr>
<td>25</td>
<td>4604.21</td>
<td>20.18</td>
<td>10216.74</td>
</tr>
<tr>
<td>30</td>
<td>3836.84</td>
<td>24.22</td>
<td>12260.09</td>
</tr>
<tr>
<td>35</td>
<td>3288.72</td>
<td>28.25</td>
<td>14303.44</td>
</tr>
<tr>
<td>40</td>
<td>2877.63</td>
<td>32.29</td>
<td>16346.79</td>
</tr>
<tr>
<td>45</td>
<td>2557.89</td>
<td>36.33</td>
<td>18390.13</td>
</tr>
<tr>
<td>50</td>
<td>2302.10</td>
<td>40.36</td>
<td>20433.48</td>
</tr>
</tbody>
</table>
TABLE XXIX

ASIC SPEED-UP OVER 4.2M VOXELS WITH MULTIPLE ACCELERATORS AND MASKED MEMORY ACCESS, FOR A 43 ANTENNAS SYSTEM AND WITH 2NS CLOCK PERIOD, COMPARED WITH SOFTWARE EXECUTION TIME OF 74945S

<table>
<thead>
<tr>
<th>$N_{ACC}$</th>
<th>$T_{exec}$ [s]</th>
<th>Acceleration</th>
<th>Required BW [MB/s]</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>98994.00</td>
<td>0.76</td>
<td>408.66</td>
</tr>
<tr>
<td>2</td>
<td>49497.00</td>
<td>1.51</td>
<td>817.31</td>
</tr>
<tr>
<td>3</td>
<td>32998.00</td>
<td>2.27</td>
<td>1225.97</td>
</tr>
<tr>
<td>4</td>
<td>24748.50</td>
<td>3.03</td>
<td>1634.62</td>
</tr>
<tr>
<td>5</td>
<td>19798.80</td>
<td>3.79</td>
<td>2043.28</td>
</tr>
<tr>
<td>6</td>
<td>16499.00</td>
<td>4.54</td>
<td>2451.93</td>
</tr>
<tr>
<td>7</td>
<td>14142.00</td>
<td>5.30</td>
<td>2860.59</td>
</tr>
<tr>
<td>8</td>
<td>12374.25</td>
<td>6.06</td>
<td>3269.24</td>
</tr>
<tr>
<td>9</td>
<td>10999.33</td>
<td>6.81</td>
<td>3677.90</td>
</tr>
<tr>
<td>10</td>
<td>9899.40</td>
<td>7.57</td>
<td>4086.55</td>
</tr>
<tr>
<td>11</td>
<td>8999.45</td>
<td>8.33</td>
<td>4495.21</td>
</tr>
<tr>
<td>12</td>
<td>8249.50</td>
<td>9.08</td>
<td>4903.86</td>
</tr>
<tr>
<td>13</td>
<td>7614.92</td>
<td>9.84</td>
<td>5312.52</td>
</tr>
<tr>
<td>14</td>
<td>7071.00</td>
<td>10.60</td>
<td>5721.17</td>
</tr>
<tr>
<td>15</td>
<td>6599.60</td>
<td>11.36</td>
<td>6129.83</td>
</tr>
<tr>
<td>16</td>
<td>6187.13</td>
<td>12.11</td>
<td>6538.48</td>
</tr>
<tr>
<td>17</td>
<td>5823.18</td>
<td>12.87</td>
<td>6947.14</td>
</tr>
<tr>
<td>18</td>
<td>5499.67</td>
<td>13.63</td>
<td>7355.79</td>
</tr>
<tr>
<td>19</td>
<td>5210.21</td>
<td>14.38</td>
<td>7764.45</td>
</tr>
<tr>
<td>20</td>
<td>4949.70</td>
<td>15.14</td>
<td>8173.10</td>
</tr>
<tr>
<td>21</td>
<td>4714.00</td>
<td>15.90</td>
<td>8581.76</td>
</tr>
<tr>
<td>22</td>
<td>4499.73</td>
<td>16.66</td>
<td>8990.41</td>
</tr>
<tr>
<td>23</td>
<td>4304.09</td>
<td>17.41</td>
<td>9399.07</td>
</tr>
<tr>
<td>24</td>
<td>4124.75</td>
<td>18.17</td>
<td>9807.72</td>
</tr>
<tr>
<td>25</td>
<td>3959.76</td>
<td>18.93</td>
<td>10216.38</td>
</tr>
<tr>
<td>30</td>
<td>3299.80</td>
<td>22.71</td>
<td>12259.65</td>
</tr>
<tr>
<td>35</td>
<td>2828.40</td>
<td>26.50</td>
<td>14302.93</td>
</tr>
<tr>
<td>40</td>
<td>2474.85</td>
<td>30.28</td>
<td>16346.20</td>
</tr>
<tr>
<td>41</td>
<td>2414.49</td>
<td>31.04</td>
<td>16754.86</td>
</tr>
<tr>
<td>42</td>
<td>2357.00</td>
<td>31.80</td>
<td>17163.51</td>
</tr>
<tr>
<td>43</td>
<td>2302.19</td>
<td>32.55</td>
<td>17572.17</td>
</tr>
</tbody>
</table>
used for the software implementation performance test. Even if it is difficult to determine the main memory performance without the interference of the caches, a possible value for the appears to be about 1.9GB/s. Consider that this is a conservative estimate and that it is possible that the memory could provide higher bandwidth if used in the right conditions, for examples without the need of continuous cache paging due to the fact that we used very large data sets to account for cache effects (using the tool LLCbench(20) from University of Tennessee).
In this case, we can say that with equal available bandwidth the hardware implementation can provide a speedup whose value is around 4.
CHAPTER 9

CONCLUSIONS

Thanks to the parametric implementation, the design could be easily adapted to various configurations with different number of transmitters and receivers and for various programmable platform with different available resources. This gives to the described design a high degree of flexibility and scalability. Thanks to this improvements the design can also adapt to provide either 2D or 3D images, since to produce a 3D image we would just need to produce a series of 2D images each of which would be associated to different delay and weights sets of course. At the same time, the design can adapt to the presence of any number of antennas in the design, which is a parameter that could depend on the specific application requirements.

From our analysis we have seen how there is a maximum value to the acceleration we can obtain, even if this value depends on a number of factors like clock timing. Anyway, even if we cannot define the absolute maximum possible acceleration value, this value will exist for any design choices we could make and represent thus a significant limit to our performance.

After the detailed analysis we have performed on the results, we can say that the performance of hardware implementation of the Imaging Reconstruction Unit is strongly limited by the available memory bandwidth. To perform an even comparison, we should determine the bandwidth supplied by the system used for the software testing. Unfortunately this was not possible due to time limitations in the use of the machine. However, if we suppose to have a reasonable bandwidth value such as 2GB/s from the data shown in the previous chapter we see that the
hardware implementation would provide performance that is about 4 times faster. This is a significant improvement especially when considering real case applications of our system. In the worst case we have considered that corresponds to a 4.2 million voxels for a 50 antennas system, the software execution time is around 20 minutes. Even if it does not seem like a long time, it means that during a screening test the physician would have to wait that amount of time before proceeding with the analysis of the results, or that in case of performing the screening in a specialized lab there could not be more than three screenings every our, lowering *de facto* the efficiency of the process.

We should also consider that the considered worst case is just a reference value and also that it is valid for a 1mm resolution 3D breast scan of a 3D space of 160 voxels per dimension. If considering a more realistic 0.5mm for a 3D breast scan the total size of the map would scale up dramatically producing a 33 million voxels map, resulting in an execution time of about 3 hours for the software computation. Such a long time would of course be a strong limit to the capillar diffusion of this screening solution, while the performance boost obtained from the hardware implementation would take it down to a more reasonable value.

### 9.1 Possible improvements and future work

The design was modelled so that it is parametrized depending on the most important design choices like the number of antennas and the number of filters. There are however other design parameters whose values were chosen after some analysis and considerations that have a direct impact on the design layout. It could be interesting to modify the code so that it is flexible to possible changes on some design choices like the number of weights used for each FIR filter.
The described design had as main purpose the acceleration of the beamforming algorithm, and because of that the main issue we have been dealing with was the execution time. From a power consumption point of view this design is not optimized: several improvements could be performed, like enabling the right portions of the filters only when needed to limit the unnecessary phase changing of the arithmetic logics. We can say that in general it would be interesting to explore other tradeoffs between hardware and software such as power consumption, power density, cost etc.

Also our work group is currently exploring the possibility of a software implementation based on GPU computing, since this kind of devices have a very large number of cores and we have seen how large is the degree of parallelism of the described system.

A more detailed test of the variation of both software and hardware performance depending on the memory bandwidth would be relevant to obtain accurate comparison between this two implementations.
APPENDICES
Appendix A

VHDL SOURCES

In this Appendix all the VHDL sources are reported. Memory blocks code is not included being that they have been generated via ISE Block Memory Generator.

A.1 Top entity and communication interface

Listing A.1. Declaration pack for constants and signal types

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.all;
use IEEE.NUMERIC_STD.ALL;
use ieee.math_real.all;

package declaration_pack is

  -- Declare constants

  constant word_length       : integer := 16; -- il numero di bit dei dati
  constant double_word_length : integer := 32;
  constant total_word_length_out : integer := 2*double_word_length;
  constant na_length         : integer := 9;
  constant slow,high_length  : integer := 9;
  constant NANT             : integer := 9;
  constant NANT2MAX         : integer := ((2**5)-1)*2;
  constant NFIR             : integer := 9;
  -- constant NVOX_cycle     : integer := 2;
```
Appendix A (continued)

—constant NVOX_tst : integer := 14495;
constant nlow : integer := 252;
constant nhigh : integer := 257;

—Declare types

type arrayvector55 is array (0 to 54) of std_logic_vector (word_length-1 downto 0);
type arrayvector55x2 is array (0 to 54) of std_logic_vector (double_word_length-1 downto 0);
type arrayvector32 is array (0 to 31) of std_logic_vector (word_length-1 downto 0);
type arrayvector32x2 is array (0 to 31) of std_logic_vector (double_word_length-1 downto 0);
type arrayvector16 is array (0 to 15) of std_logic_vector (word_length-1 downto 0);
type arrayvector16x2 is array (0 to 15) of std_logic_vector (double_word_length-1 downto 0);
type arrayvector9 is array (0 to 8) of std_logic_vector (word_length-1 downto 0);
type arrayvector8 is array (0 to 7) of std_logic_vector (word_length-1 downto 0);
type arrayvector8x2 is array (0 to 7) of std_logic_vector (double_word_length-1 downto 0);
type arrayvector4 is array (0 to 3) of std_logic_vector (word_length-1 downto 0);
type arrayvector3 is array (0 to 2) of std_logic_vector (word_length-1 downto 0);
type arrayvector2 is array (0 to 1) of std_logic_vector (word_length-1 downto 0);
type arrayvector9x2 is array (0 to 8) of std_logic_vector (double_word_length-1 downto 0);
type arrayvector4x2 is array (0 to 3) of std_logic_vector (double_word_length-1 downto 0);
type arrayvector3x2 is array (0 to 2) of std_logic_vector (double_word_length-1 downto 0);
type arrayvector2x2 is array (0 to 1) of std_logic_vector (double_word_length-1 downto 0);

type arrayvectorin is array (NFIR-1 downto 0) of std_logic_vector (word_length-1 downto 0);
type arrayvecorsmem is array (NANT-1 downto 0) of std_logic_vector (word_length-1 downto 0);
type arrayvecormemqueue is array (integer (ceil(real(NANT)/real(NFIR))))-1 downto 0) of
arrayvectorin;
type arrayvecormemdata is array (1 to integer (ceil(real(NANT)/real(2)))) of std_logic_vector (double_word_length-1 downto 0);
type arrayvecormemaddress is array (1 to integer (ceil(real(NANT)/real(2)))) of
std_logic_vector (8 downto 0);
type arrayvecormemrwe is array (1 to integer (ceil(real(NANT)/real(2)))) of std_logic_vector (0
downto 0);
type arrayvectorinx2 is array (NFIR-1 downto 0) of std_logic_vector (double_word_length-1
downto 0);
type arrayvectorout is array (NFIR-1 downto 0) of std_logic_vector (1-1 downto 0);
type FIRINPUT is array (1 to NFIR) of arrayvector55;
type arrayvectoradder is array (0 to NFIR) of std_logic_vector (double_word_length-1 downto 0);
type arrayvectoradder is array (1 to integer (ceil(log2(real(NFIR+1))))+1) of
arrayvectoradder;
Listing A.2. Design top entity

--- Francesco Colonna
--- 8/2012
--- adapted from Phuong Nguyen
--- 22/8/2011
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity main_instance is
  port (  
    rst_n : in  std_logic;
    gclk : in  std_logic;
    fx2_clk : in  std_logic;
    fx2_ctl : in  std_logic_vector(2 downto 0);
    fx2_fd : inout std_logic_vector(7 downto 0);
    fx2_fifoadr : out std_logic_vector(1 downto 0);
    fx2_sloe : out std_logic;
    fx2_s1rd : out std_logic;
    fx2_s1wr : out std_logic;
    fx2_flagd : in  std_logic;
    fx2_pktend : out std_logic;
    leds : out std_logic_vector(7 downto 0);
    switch : in  std_logic_vector(7 downto 0)
  );
end main_instance;

architecture Behavioral of main_instance is
component mydcm_div

port ( 
  CLKin_IN : in std_logic;
  DADDR_IN : in std_logic_vector (6 downto 0);
  DCLK_IN : in std_logic;
  DEN_IN : in std_logic;
  DI_IN : in std_logic_vector (15 downto 0);
  DWE_IN : in std_logic;
  RST_IN : in std_logic;
  CLKDV_OUT : out std_logic;
  CLKFX_OUT : out std_logic;
  CLKFX180_OUT : out std_logic;
  CLKin_BUF_OUT : out std_logic;
  CLK0_OUT : out std_logic;
  CLK2X_OUT : out std_logic;
  CLK2X180_OUT : out std_logic;
  CLK90_OUT : out std_logic;
  CLK180_OUT : out std_logic;
  CLK270_OUT : out std_logic;
  DO_OUT : out std_logic_vector (15 downto 0);
  DRDY_OUT : out std_logic;
  LOCKED_OUT : out std_logic;
); 
end component;

--

component mydcm_cascade

port ( 
  GCLK : in std_logic;
  DADDR : in std_logic_vector (6 downto 0);
  DCLK : in std_logic;
  DEN : in std_logic_vector (1 downto 0);
  DI : in std_logic_vector (15 downto 0);
  DWE : in std_logic_vector (1 downto 0);
  RST : in std_logic;
  GCLK_BUF : out std_logic;
  CLKFX : out std_logic;
  DO : out std_logic_vector (15 downto 0);
  DRDY : out std_logic_vector (1 downto 0);
  LOCKED : out std_logic_vector (1 downto 0);
); 
end component;
-- component fx2interface is  -- interface for usb communication

port (
	rst  : in std_logic;

fx2_clk  : in std_logic;

fx2_ctl : in std_logic_vector(2 downto 0);

fx2_flagd : in std_logic;

dx2_sloe  : out std_logic;

fx2_sird : out std_logic;

fx2_slwr : out std_logic;

fx2_fifoadr : out std_logic_vector(1 downto 0);

fx2_pktend : out std_logic;

fx2_fd : inout std_logic_vector(7 downto 0);

--

fpga_clk : in std_logic;

bram16k_din : out std_logic_vector(31 downto 0);

bram16k_dout : in std_logic_vector(31 downto 0);

bram16k_en : out std_logic;

bram16k_we : out std_logic;

bram16k_addr : out std_logic_vector(16 downto 0);

--

bram2k_din0 : out std_logic_vector(7 downto 0);

bram2k_din1 : out std_logic_vector(7 downto 0);

bram2k_din2 : out std_logic_vector(7 downto 0);

bram2k_din3 : out std_logic_vector(7 downto 0);

bram2k_dout0 : in std_logic_vector(7 downto 0);

bram2k_dout1 : in std_logic_vector(7 downto 0);

bram2k_dout2 : in std_logic_vector(7 downto 0);

bram2k_dout3 : in std_logic_vector(7 downto 0);

bram2k_en : out std_logic;

bram2k_we : out std_logic;

bram2k_addr : out std_logic_vector(14 downto 0);

--

reset_cpu : out std_logic;

intrackF : out std_logic;

dataackM : out std_logic;

enmem_en : out std_logic;

STARTBEAF : out std_logic;

leds_debug : out std_logic_vector(31 downto 0);

NVOX_cycle : out std_logic_vector(8 downto 0);
Appendix A (continued)

```
data_processed : in std_logic
);
end component fx2interface;

component BEAF_interface is        -- Mist BEAF main instantiation block
port (  
clk : in std_logic;
rst : in std_logic;
switch : in std_logic_vector(7 downto 0);
rst_mips : out std_logic_vector(31 downto 0);
adx_mips : out std_logic_vector(31 downto 0);
do_mips : in std_logic_vector(31 downto 0);
rst_dcm : out std_logic;
den_dcm : out std_logic_vector(1 downto 0);
dwc_dcm : out std_logic_vector(1 downto 0);
di_dcm : out std_logic_vector(15 downto 0);
daddr_dcm : out std_logic_vector(6 downto 0);
do_dcm : in std_logic_vector(15 downto 0);
drdo_dcm : in std_logic_vector(1 downto 0);
locked_dcm : in std_logic_vector(1 downto 0);
zf_timer : in std_logic;
init_timer : out std_logic_vector(31 downto 0);
leds : out std_logic_vector(7 downto 0);
data_processed : out std_logic;

-- fx2interface
bram16k_din : in std_logic_vector(31 downto 0);
bram16k_dout : out std_logic_vector(31 downto 0);
bram16k_en : in std_logic;
bram16k_we : in std_logic;
bram16k_addr : in std_logic_vector(16 downto 0);

--
bram2k_din0 : in std_logic_vector(7 downto 0);
bram2k_din1 : in std_logic_vector(7 downto 0);
bram2k_din2 : in std_logic_vector(7 downto 0);
bram2k_din3 : in std_logic_vector(7 downto 0);
bram2k_dout0 : out std_logic_vector(7 downto 0);
bram2k_dout1 : out std_logic_vector(7 downto 0);
bram2k_dout2 : out std_logic_vector(7 downto 0);
bram2k_dout3 : out std_logic_vector(7 downto 0);  
```
Appendix A (continued)

```vhdl
bram2k_en : in std_logic;
bram2k_we : in std_logic;
bram2k_addr : in std_logic_vector(14 downto 0);

--
instrackF : in std_logic;
dataackM : in std_logic;
enmem_en : in std_logic;
START : in std_logic;
NVOX_cycle : in std_logic_vector(8 downto 0);
);

end component;

--
signal rst : std_logic;
signal clk, clk0, clk0_div : std_logic;
signal locked_dcm_div : std_logic;

-- mydcm_cascade
signal daddr_dcm : std_logic_vector(6 downto 0);
signal denv_dcm : std_logic_vector(1 downto 0);
signal din_dcm : std_logic_vector(15 downto 0);
signal dwe_dcm : std_logic_vector(1 downto 0);
signal rst_dcm : std_logic;
signal do_dcm : std_logic_vector(15 downto 0);
signal drdy_dcm : std_logic_vector(1 downto 0);
signal locked_dcm : std_logic_vector(1 downto 0);

-- control signals
signal rst_ctrl : std_logic;
signal reset_cpu : std_logic;
signal rst_mips : std_logic_vector(31 downto 0);
signal adx_mips : std_logic_vector(31 downto 0);
signal do_mips : std_logic_vector(31 downto 0);
signal zf_timer : std_logic;
signal init_timer : std_logic_vector(31 downto 0);
signal leds_ctrl : std_logic_vector(7 downto 0);

-- weights memory signals
signal bram16k_din, bram16k_dout : std_logic_vector(31 downto 0);
signal bram16k_en, bram16k_we : std_logic;
signal bram16k_addr : std_logic_vector(16 downto 0);

-- samples/energy signals
signal bram2k_din0, bram2k_din1, bram2k_din2, bram2k_din3 : std_logic_vector(7 downto 0);
```
Appendix A (continued)

```vhdl
signal bram2k_dout0, bram2k_dout1, bram2k_dout2, bram2k_dout3: std_logic_vector(7 downto 0);
signal bram2k_en, bram2k_we: std_logic;
signal bram2k_addr: std_logic_vector(14 downto 0);

signal instrackF, dataackM, enmem_en, data_processed: std_logic;
signal NVOX_cycle: std_logic_vector(8 downto 0);
signal START_BEAF: std_logic;
signal leds_beaf: std_logic_vector(7 downto 0);
signal leds_debug: std_logic_vector(31 downto 0);

signal timer_cntr: std_logic_vector(15 downto 0);
signal start_timer, load_timer: std_logic;

begin
    rst <= not rst_n;

    umydcm_div: mydcm_div

    port map (  
        CLKN_IN => clk0,  
        DADDR_IN => (others => '0'),  
        DCLK_IN => '0',  
        DEN_IN => '0',  
        DI_IN => (others => '0'),  
        DWE_IN => '0',  
        RST_IN => rst,  
        CLKDV_OUT => clk0_div,  
        CLKFX_OUT => open,  
        CLKFX180_OUT => open,  
        CLKN_IBUF_OUT => open,  
        CLK0_OUT => open,  
        CLK2X_OUT => open,  
        CLK2X180_OUT => open,  
        CLK90_OUT => open,  
        CLK180_OUT => open,  
        CLK270_OUT => open,  
        DQ_OUT => open,  
        DRDY_OUT => open,  
        LOCKED_OUT => locked_dcm_div  
    );
```
Appendix A (continued)

```
  umydcm_cascade: mydcm_cascade

  port map (    
      GCLK => gclk,    
      DADDR => daddr_dcm(6 downto 0),    
      DCLK => clk0_div,    
      DEN => den_dcm,    
      DI => di_dcm,    
      DWE => dwe_dcm,    
      RST => rst_dcm,    
      GCLK_BUF => clk0,    
      CLKFX => clk,    
      DO => do_dcm,    
      DRDY => drdy_dcm,    
      LOCKED => locked_dcm);

  rst_ctrl <= rst or not locked_dcm_div or reset_cpu;

  fx2int: fx2interface
  port map (    
      rst => rst,    
      fx2_clk => fx2_clk,    
      fx2_ctl => fx2_ctl,    
      fx2_flagd => fx2_flagd,    
      fx2_sloe => fx2_sloe,    
      fx2_slrd => fx2_slrd,    
      fx2_slwr => fx2_slwr,    
      fx2_fifoadr => fx2_fifoadr,    
      fx2_pktend => fx2_pktend,    
      fx2_fd => fx2_fd;

  --
      fpga_clk => clk0_div, -- #######
      bram16k_din => bram16k_din,    
      bram16k_dout => bram16k_dout,    
      bram16k_en => bram16k_en,    
      bram16k_we => bram16k_we,    
      bram16k_addr => bram16k_addr,    

  --
      bram2k_din0 => bram2k_din0,    
      bram2k_din1 => bram2k_din1,    
      bram2k_din2 => bram2k_din2;
```
Appendix A (continued)

```
bram2k.din3  =>  bram2k.din3,
bram2k.dout0  =>  bram2k.dout0,
bram2k.dout1  =>  bram2k.dout1,
bram2k.dout2  =>  bram2k.dout2,
bram2k.dout3  =>  bram2k.dout3,
bram2k.en   =>  bram2k.en,
bram2k.we   =>  bram2k.we,
bram2k.addr =>  bram2k.addr,

reset.cpu  =>  reset.cpu,
intrackF =>  instrackF,
dataackM =>  dataackM,
enmem_en  =>  enmem_en,
START_BEAf  =>  START_BEAf,
leds.debug =>  leds.debug,
NVOX_cycle  =>  NVOX_cycle,
data_processed =>  data_processed

): Inst_BEAf: BEAF_interface

port map(
  clk => clk0_div,
  rst => rst,
  switch => switch,
  rst_mips => rst_mips,
  adx_mips => adx_mips,
  do_mips => do_mips,
  rst_dcm => rst_dcm,
  den_dcm => den_dcm,
  dwe_dcm => dwe_dcm,
  di_dcm => di_dcm,
  daddr_dcm => daddr_dcm,
  do_dcm => do_dcm,
  drdy_dcm => drdy_dcm,
  locked_dcm => locked_dcm,
  zf_timer => zf_timer,
  init_timer => init_timer,
  leds => leds_beaF,
  data_processed => data_processed,
);```
Appendix A (continued)

--- fslinterface

\[ \begin{align*}
&\text{bram16k\_din} \Rightarrow \text{bram16k\_din}, \\
&\text{bram16k\_dout} \Rightarrow \text{bram16k\_dout}, \\
&\text{bram16k\_en} \Rightarrow \text{bram16k\_en}, \\
&\text{bram16k\_we} \Rightarrow \text{bram16k\_we}, \\
&\text{bram16k\_addr} \Rightarrow \text{bram16k\_addr}, \\
&\text{bram2k\_din0} \Rightarrow \text{bram2k\_din0}, \\
&\text{bram2k\_din1} \Rightarrow \text{bram2k\_din1}, \\
&\text{bram2k\_din2} \Rightarrow \text{bram2k\_din2}, \\
&\text{bram2k\_din3} \Rightarrow \text{bram2k\_din3}, \\
&\text{bram2k\_dout0} \Rightarrow \text{bram2k\_dout0}, \\
&\text{bram2k\_dout1} \Rightarrow \text{bram2k\_dout1}, \\
&\text{bram2k\_dout2} \Rightarrow \text{bram2k\_dout2}, \\
&\text{bram2k\_dout3} \Rightarrow \text{bram2k\_dout3}, \\
&\text{bram2k\_en} \Rightarrow \text{bram2k\_en}, \\
&\text{bram2k\_we} \Rightarrow \text{bram2k\_we}, \\
&\text{bram2k\_addr} \Rightarrow \text{bram2k\_addr}, \\
&\text{instrackF} \Rightarrow \text{instrackF}, \\
&\text{dataackM} \Rightarrow \text{dataackM}, \\
&\text{enmem\_en} \Rightarrow \text{enmem\_en}, \\
&\text{START} \Rightarrow \text{START\_BEAF}, \\
&\text{NVOX\_cycle} \Rightarrow \text{NVOX\_cycle}
\end{align*} \]

\[ \text{ptimer: process (clk0\_div, rst\_ctrl)} \]
\[ \text{variable counting : std\_logic := '0';} \]

begin --- process ptimer
\[ \text{if rst\_ctrl = '1' then} \quad \text{-- asynchronous reset (active high)} \]
\[ \text{timer\_cntn} <= ('\text{others} => '1');} \]
\[ \text{counting := '0';} \]
\[ \text{elsif clk0\_div'event and clk0\_div = '1' then} \quad \text{-- rising clock edge} \]
\[ \text{if ((start\_timer or counting) and not if\_timer) = '1' then} \]
\[ \text{timer\_cntn} <= \text{timer\_cntn} - 1; \]
\[ \text{counting := '1';} \]
\[ \text{end if;} \]
\[ \text{if load\_timer = '1' then} \quad \text{-- has priority} \]
\[ \text{timer\_cntn} <= \text{init\_timer}(15 \text{ downto } 0); \]
\[ \text{counting := '0';} \]
Appendix A (continued)

```vhdl
end if;
end if;
end process ptimer;

load_timer <= init_timer(16);
start_timer <= init_timer(17);
sf_timer <= '1' when timer_cnt = 0 else '0';

-- debug signals to control led lights
leds <= leds_debug(31 downto 24) when switch = "00000000" else
       leds_debug(23 downto 16) when switch = "00000001" else
       leds_debug(15 downto 8) when switch = "00000010" else
       leds_debug(7 downto 0);
end Behavioral;
```

Listing A.3. Xilinx automatic generated dcm divisor

```vhdl
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;

entity mydcm_div is
  port ( CLKIN_IN : in std_logic;
          DADDR_IN : in std_logic_vector (6 downto 0);
          DCLK_IN : in std_logic;
          DEN_IN : in std_logic;
          DI_IN : in std_logic_vector (15 downto 0);
          DWE_IN : in std_logic;
          RST_IN : in std_logic;
          CLKDIV_OUT : out std_logic;
          CLKFX_OUT : out std_logic;
          CLKFX180_OUT : out std_logic;
          CLKIN_IBUFG_OUT : out std_logic;
          CLK0_OUT : out std_logic;
          CLK2X_OUT : out std_logic;
          CLK2X180_OUT : out std_logic;
          CLK90_OUT : out std_logic;
```
Appendix A (continued)

```vhdl
architecture BEHAVIORAL of mydcm_div is
signal CLKDV_BUF : std_logic;
signal CLKFB_IN : std_logic;
signal CLKFX_BUF : std_logic;
signal CLKFX180_BUF : std_logic;
signal CLKIN_IBUFG : std_logic;
signal CLK0_BUF : std_logic;
signal CLK2X_BUF : std_logic;
signal CLK2X180_BUF : std_logic;
signal CLK90_BUF : std_logic;
signal CLK180_BUF : std_logic;
signal CLK270_BUF : std_logic;
signal GND_BIT : std_logic;
begin
GND_BIT <= '0';
CLKIN_IBUFG_OUT <= CLKIN_IBUFG;
CLK0 OUT <= CLKFB_IN;
CLKDV_BUF INST : BUFG
port map (I=>CLKDV_BUF,
O=>CLKDV_OUT);

CLKFX_BUF INST : BUFG
port map (I=>CLKFX_BUF,
O=>CLKFX_OUT);

CLKFX180_BUF INST : BUFG
port map (I=>CLKFX180_BUF,
O=>CLKFX180_OUT);

-- CLKIN_IBUFG INST : IBUF
-- port map (I=>CLKIN_IN,
-- O=>CLKIN_IBUFG);
CLKIN_IBUFG <= CLKIN_IN;
end mydcm_div;
```
CLK0_BUFG_INST : BUFG
  port map (I=>CLK0_BUF,
            O=>CLKFB_IN);

CLK2X_BUFG_INST : BUFG
  port map (I=>CLK2X_BUF,
            O=>CLK2X_OUT);

CLK2X180_BUFG_INST : BUFG
  port map (I=>CLK2X180_BUF,
            O=>CLK2X180_OUT);

CLK90_BUFG_INST : BUFG
  port map (I=>CLK90_BUF,
            O=>CLK90_OUT);

CLK180_BUFG_INST : BUFG
  port map (I=>CLK180_BUF,
            O=>CLK180_OUT);

CLK270_BUFG_INST : BUFG
  port map (I=>CLK270_BUF,
            O=>CLK270_OUT);

DCM_ADV_INST : DCM_ADV
  generic map( CLK_FEEDBACK => "1X",
               CLKDIV_DIVIDE => 2.0,
               CLKFX_DIVIDE => 5,
               CLKFX_MULTIPLY => 4,
               CLKIN_DIVIDE_BY_2 => FALSE,
               CLKIN_PERIOD => 8.000,
               CLKOUT_PHASE_SHIFT => "NONE",
               DCM_AUTOCALIBRATION => TRUE,
               DCM_PERFORMANCE_MODE => "MAX_SPEED",
               DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
               DFS_FREQUENCY_MODE => "LOW",
               DLL_FREQUENCY_MODE => "LOW",
               DUTY_CYCLE_CORRECTION => TRUE,
               FACTORY_JF => x"F0F0",}
Appendix A (continued)

```
PHASE_SHIFT => 0,
STARTUP_WAIT => FALSE)

port map (CLKFB => CLKFB_IN,
          CLKIN => CLKIN_IBUFG,
          DADDR(6 downto 0) => DADDR_IN(6 downto 0),
          DCLK => DCLK_IN,
          DEN => DEN_IN,
          DI(15 downto 0) => DI_IN(15 downto 0),
          DWE => DWE_IN,
          PSLCK => GND_BIT,
          PSEN => GND_BIT,
          PSINCDEC => GND_BIT,
          RST => RST_IN,
          CLKDV => CLKDV_BUF,
          CLKFX => CLKFX_BUF,
          CLKFX180 => CLKFX180_BUF,
          CLK0 => CLK0_BUF,
          CLK2X => CLK2X_BUF,
          CLK2X180 => CLK2X180_BUF,
          CLK30 => CLK30_BUF,
          CLK180 => CLK180_BUF,
          CLK270 => CLK270_BUF,
          DO(15 downto 0) => DO_OUT(15 downto 0),
          DRDY => DRDY_OUT,
          LOCKED => LOCKED_OUT,
          PSDONE => open);

end BEHAVIORAL;
```

**Listing A.4. Xilinx automatic generated dcm cascade**

```
library ieee;
use ieee.std_logic_1164.ALL;
use ieee.numeric_std.ALL;
library UNISIM;
use UNISIM.Vcomponents.ALL;

entity mydcm_cascade is
  port ( GCLK : in std_logic;
```
Appendix A (continued)

```vhdl
DADDR : in std_logic_vector (6 downto 0);
DCLK : in std_logic;
DEN : in std_logic_vector (1 downto 0);
DI : in std_logic_vector (15 downto 0);
DWE : in std_logic_vector (1 downto 0);
RST : in std_logic;
GCLK_BUF : out std_logic;
CLKFX : out std_logic;
DO : out std_logic_vector (15 downto 0);
DRDY : out std_logic_vector (1 downto 0);
LOCKED : out std_logic_vector (1 downto 0));
end mydcm_cascade;
architecture BEHAVIORAL of mydcm_cascade is
signal CLKFB_IN : std_logic_vector (1 downto 0);
signal CLKFX0 : std_logic;
signal CLKFX_BUF : std_logic;
signal CLKin_IBUFG : std_logic_vector (1 downto 0);
signal CLK0_BUF : std_logic_vector (1 downto 0);
signal GND_BIT : std_logic;
signal CLKin : std_logic_vector (1 downto 0);
signal RST_DCM : std_logic_vector (1 downto 0);
signal LOCKED_i : std_logic_vector (1 downto 0);
signal DO0, DO1 : std_logic_vector (15 downto 0);
signal rst_delay : std_logic_vector (3 downto 0);
begin
LOCKED <= LOCKED_i;
CLKin(0) <= GCLK;
GND_BIT <= '0';
--CLKin_IBUFG_OUT <= CLKin_IBUFG;
--CLK0 <= CLKFBI_IN(0);
GCLK_BUF <= CLKin_IBUFG(0);

CLKin(1) <= CLKFX0;
```
CLKIN_IBUFG_INST_0 : IBUFG
    port map (I=>CLKIN(0),
             O=>CLKIN_IBUFG(0));

CLKIN_IBUFG_INST_1 : IBUFG
    port map (I=>CLKIN(1),
             O=>CLKIN_IBUFG(1));

CLK0_BUFG_INST_0 : BUFG
    port map (I=>CLK0_BUF(0),
              O=>CLKFB_IN(0));

CLK0_BUFG_INST_1 : BUFG
    port map (I=>CLK0_BUF(1),
              O=>CLKFB_IN(1));

RST_DCM(0) <= RST;
RST_DCM(1) <= not LOCKED(0);

prst_dcm1 : process (CLKIN_IBUFG(0), RST)
begin -- process prst_dcm1
    if RST = '1' then -- asynchronous reset (active high)
        rst_delay <= (others => '0');
    elsif CLKIN_IBUFG(0)'event and CLKIN_IBUFG(0) = '1' then -- rising clock edge
        rst_delay(2 downto 0) <= rst_delay(3 downto 1);
        rst_delay(3) <= not LOCKED(0);
    end if;
end process prst_dcm1;
RST_DCM(1) <= rst_delay(0);

DCM_ADV_INST0 : DCM ADV
    generic map( CLKF_FEEDBACK => "1X",
                 CLKDIVITY => 2.0,
                 CLKFX_DIVIDE => 5,
                 CLKFX_MULTIPLY => 4,
                 CLKIN_DIVIDE_BY_2 => FALSE,
                 CLKPERIOD => 8.000,
                 CLKOUT_PHASE_SHIFT => "NONE",
                 DCM_AUTOCALIBRATION => TRUE,
                 DCM_PERFORMANCE_MODE => "MAX_SPEED"
Appendix A (continued)

DESKW\_ADJUST \=> "SYSTEM\_SYNCHRONOUS",
DFS\_FREQUENCY\_MODE \=> "LOW",
DLL\_FREQUENCY\_MODE \=> "LOW",
DUTY\_CYCLE\_CORRECTION \=> TRUE,
FACTORY\_JF \=> x"F0F0",
PHASE\_SHIFT \=> 0,
STARTUP\_WAIT \=> FALSE)

port map (CLKFB\_IN(0),
       CLKIN\_IBUFG(0),
       DADDR(6 downto 0)\= DADDR(6 downto 0),
       DCLK\= DCLK,
       DEN\= DEN(0),
       DI(15 downto 0)\= DI(15 downto 0),
       DWE\= DWE(0),
       PSCLK\= GND\_BIT,
       PSEN\= GND\_BIT,
       PSINCDEC\= GND\_BIT,
       RST\= RST\_DCM(0),
       CLKD\= open,
       CLKF\= CLKF\_O,
       CLKF\_180\= open,
       CLK0\= CLK0\_BUF(0),
       CLK2\= open,
       CLK2\_180\= open,
       CLK90\= open,
       CLK180\= open,
       CLK270\= open,
       DO(15 downto 0)\= DO(15 downto 0),
       DRDY\= DRDY(0),
       LOCKED\= LOCKED\_I(0),
       PSDONE\= open);

DCM\_ADV\_INST1 \: DCM\_ADV

generic map( CLK\_FEEDBACK \= "1X",
    CLKDV\_DIVIDE \=> 2.0,
    CLKFX\_DIVIDE \=> 10,
    CLKFX\_MULTIPLY \=> 9,
    CLKIN\_DIVIDE\_BY\_2 \=> FALSE,
    CLKIN\_PERIOD \=> 10.000,
    CLKOUT\_PHASE\_SHIFT \= "NONE"
DCM_AUTOCALIBRATION => TRUE,
DCM_PERFORMANCE_MODE => "MAX_SPEED",
DESKEW_ADJUST => "SYSTEM_SYNCHRONOUS",
DFB_FREQUENCY_MODE => "LOW",
DLL_FREQUENCY_MODE => "LOW",
DUTY_CYCLE_CORRECTION => TRUE,
FACTORY_JF => x"F0F0",
PHASE_SHIFT => 0,
STARTUP_WAIT => FALSE)

port map (CLKFB=>CLKFB_IN(1),
    CLKN=>CLKIN_IBUFG(1),
    DADDR(6 downto 0) => DADDR(6 downto 0),
    DCLK=>DCLK,
    DEN=>DEN(1),
    DI(15 downto 0) => DI(15 downto 0),
    DIW=>DIW(1),
    PSCLK=>GND_BIT,
    PSEN=>GND_BIT,
    PSINCDEC=>GND_BIT,
    RST=>RST_DCM(1),
    CLKDV=>open,
    CLKFX=>CLKFX_BUF,
    CLKFX180=>open,
    CLK0=>CLK0_BUF(1),
    CLKX=>open,
    CLKX180=>open,
    CLK90=>open,
    CLK180=>open,
    CLK270=>open,
    DO(15 downto 0) => DO(15 downto 0),
    DRDY=>DRDY(1),
    LOCKED=>LOCKED(1),
    PSDONE=>open);

DO <= DO[0] when den(0) = '1' else DO[1];

CLKFX_BUFG_INST : BUFG
    port map (I=>CLKFX_BUF,
               O=>CLKFX);
Listing A.5. FX2 chip interface top entity

--- Phuong Nguyen, Francesco Colonna
--- 20/08/2012

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity fx2interface is
  port(
    rst : in std_logic;
    fx2_clk : in std_logic;
    fx2_ctl : in std_logic_vector(2 downto 0);
    fx2_flagd : in std_logic;
    fx2_sloeu : out std_logic;
    fx2_slrdu : out std_logic;
    fx2_sldr : out std_logic;
    fx2_sldr : out std_logic;
    fx2_slwr : out std_logic;
    fx2_slwr : out std_logic;
    fx2_fifoadr : out std_logic_vector(1 downto 0);
    fx2_pktend : out std_logic;
    fx2_ifd : inout std_logic_vector(7 downto 0);
    --
    fpga_clk : in std_logic;
    bram16k_din : out std_logic_vector(31 downto 0);
    bram16k_dout : in std_logic_vector(31 downto 0);
    bram16k_en : out std_logic;
    bram16k_we : out std_logic;
    bram16k_addr : out std_logic_vector(16 downto 0);
    --
    bram2k_din0 : out std_logic_vector(7 downto 0);
    bram2k_din1 : out std_logic_vector(7 downto 0);
    bram2k_din2 : out std_logic_vector(7 downto 0);
    bram2k_din3 : out std_logic_vector(7 downto 0);
    bram2k_dout0 : in std_logic_vector(7 downto 0);
    bram2k_dout1 : in std_logic_vector(7 downto 0);
    bram2k_dout2 : in std_logic_vector(7 downto 0);
    bram2k_dout3 : in std_logic_vector(7 downto 0);
  );
Appendix A (continued)

```vhdl
architecture behavioral of fx2interface is
  component ctrl1 is
    port (  
      rst : in std_logic;
      clk : in std_logic;
      --
      flaga : in std_logic; -- full
      flagb : in std_logic; -- empty
      flagc : in std_logic;
      flagd : in std_logic;
      sloe : out std_logic;
      slrd : out std_logic;
      slwr : out std_logic;
      pktend : out std_logic;
      fifoadr : out std_logic_vector(1 downto 0);
      fd : in std_logic_vector(7 downto 0);
      --
      rxfifo_full : in std_logic;
      rxfifo_wr_en : out std_logic;
      --
      txfifo_empty : in std_logic;
      txfifo_rd_en : out std_logic;
      read_data : out std_logic;
      leds : out std_logic_vector(7 downto 0)
    );
  end component;
  
  component ctrl2 is
    port (  
      rst : in std_logic;
      clk : in std_logic;
      --
      bram2k_en : out std_logic;
      bram2k_we : out std_logic;
      bram2k_addr : out std_logic_vector(14 downto 0);
      --
      reset_cpu : out std_logic;
      instrackF : out std_logic;
      dataackM : out std_logic;
      enmem_en : out std_logic;
      START_BEAF : out std_logic;
      leds_debug : out std_logic_vector(31 downto 0);
      NVOX_cycle : out std_logic_vector(8 downto 0);
      data_processed : in std_logic
    );
  end component;
  
  component ct1 is
  end component;
end fx2interface;
```
Appendix A (continued)

```vhdl
end component ctrl1;

component ctrl2
port(
    clk : in std_logic;
    rst : in std_logic;

    --
    rx fifo dout : in std_logic_vector(31 downto 0);
    rx fifo rd en : out std_logic;
    rx fifo empty : in std_logic;

    --
    tx fifo din : out std_logic_vector(31 downto 0);
    tx fifo wr en : out std_logic;
    tx fifo full : in std_logic;

    --
    bram16k din : out std_logic_vector(31 downto 0);
    bram16k dout : out std_logic_vector(31 downto 0);
    bram16k en : out std_logic;
    bram16k we : out std_logic;
    bram16k addr : out std_logic_vector(16 downto 0);

    --
    bram2k din0 : out std_logic_vector(7 downto 0);
    bram2k din1 : out std_logic_vector(7 downto 0);
    bram2k din2 : out std_logic_vector(7 downto 0);
    bram2k din3 : out std_logic_vector(7 downto 0);
    bram2k dout0 : in std_logic_vector(7 downto 0);
    bram2k dout1 : in std_logic_vector(7 downto 0);
    bram2k dout2 : in std_logic_vector(7 downto 0);
    bram2k dout3 : in std_logic_vector(7 downto 0);
    bram2k en : out std_logic;
    bram2k we : out std_logic;
    bram2k addr : out std_logic_vector(14 downto 0);

    --
    reset cpu : out std_logic;
    instrackF : out std_logic;
    dataackM : out std_logic;
    enmem_en : out std_logic;
    START BEAF : out std_logic;
    leds debug : out std_logic_vector(31 downto 0);

    --
);```
NVOX_cycle : out std_logic_vector(8 downto 0);
data_processed : in std_logic
);
end component ctrl2;

component fifo_8in_32out_2clks is
  port ( 
    din : IN std_logic_VECTOR(7 downto 0);
    rd_clk : IN std_logic;
    rd_en : IN std_logic;
    rst : IN std_logic;
    wr_clk : IN std_logic;
    wr_en : IN std_logic;
    dout : OUT std_logic_VECTOR(31 downto 0);
    empty : OUT std_logic;
    full : OUT std_logic);
end component fifo_8in_32out_2clks;

component fifo_32in_8out_2clks is
  port ( 
    din : IN std_logic_VECTOR(31 downto 0);
    rd_clk : IN std_logic;
    rd_en : IN std_logic;
    rst : IN std_logic;
    wr_clk : IN std_logic;
    wr_en : IN std_logic;
    dout : OUT std_logic_VECTOR(7 downto 0);
    empty : OUT std_logic;
    full : OUT std_logic);
end component fifo_32in_8out_2clks;

signal rxfifo_din : std_logic_vector(7 downto 0);
signal rxfifo_dout : std_logic_vector(31 downto 0);
signal rxfifo_rd_en : std_logic;
signal rxfifo_wr_en : std_logic;
signal rxfifo_full : std_logic;
signal rxfifo_empty : std_logic;
--
signal txfifo_wr_en : std_logic;
signal txfifo_rd_en : std_logic;
begin

−− usb chip control interface

controller1: ctrl1

port map (  
  rst => rst ,  
  clk => fx2_clk ,  
  flaga => fx2_ctl(0) ,  
  flagb => fx2_ctl(1) ,  
  flagc => fx2_ctl(2) ,  
  flagd => fx2_flagd ,  
  sloe => fx2_sloe ,  
  slrd => fx2_slrd ,  
  slwr => fx2_slwr ,  
  pktend => fx2_pktend ,  
  fifoadr => fx2_fifoadr ,  
  fd => fx2_fd ,  
  rxifo_full => rxifo_full ,  
  rxifo_wr_en => rxifo_wr_en ,  
  txfifo_empty => txfifo_empty ,  
  txfifo_rd_en => txfifo_rd_en ,  
  read_data => read_usb_data ,  
  leds => leds
 );

leds_debug<=leds_int(31 downto 8)&leds;

rxifo: fifo_8in_32out_2clks

port map (  
  din => rxifo_din ,  
  rd_clk => fpga_clk ,  
  rd_en => rxifo_rd_en ,  
  rst => rst ,
Appendix A (continued)

```vhdl
wr_clk => fx2_clk,
wr_en => rxfifo_wr_en,
dout => rxfifo_dout,
empty => rxfifo_empty,
full => rxfifo_full
);

txfifo: fifo_32in_8out_2clks
port map (
  din => txfifo_din,
  rd_clk => fx2_clk,
  rd_en => txfifo_rd_en,
  rst => rst,
  wr_clk => fpga_clk,
  wr_en => txfifo_wr_en,
  dout => txfifo_dout,
  empty => txfifo_empty,
  full => txfifo_full
);

-- commands interpreter and communication FSM
controller2: ctrl2
port map (  
  clk => fpga_clk,
  rst => rst,
  rx fifo_dout => rx fifo_dout,
  rx fifo_rd_en => rx fifo_rd_en,
  rx fifo_empty => rx fifo_empty,
  txfifo_din => txfifo_din,
  txfifo_wr_en => txfifo_wr_en,
  txfifo_full => txfifo_full,
  
  --
  bram16k_din => bram16k_din,
  bram16k_dout => bram16k_dout,
  bram16k_en => bram16k_en,
  bram16k_we => bram16k_we,
  bram16k_addr => bram16k_addr,
  
  --
  bram2k_din0 => bram2k_din0,
  bram2k_din1 => bram2k_din1,
```
Appendix A (continued)

```
bram2k_din2 => bram2k_din2,
bram2k_din3 => bram2k_din3,
bram2k_dout0 => bram2k_dout0,
bram2k_dout1 => bram2k_dout1,
bram2k_dout2 => bram2k_dout2,
bram2k_dout3 => bram2k_dout3,
bram2k_en => bram2k_en,
bram2k_we => bram2k_we,
bram2k_addr => bram2k_addr,

---
reset_cpu => reset_cpu,
instrackF => instrackF,
dataackM => dataackM,
enmem_en => enmem_en,
START_BEAF => START_BEAF,
leds_debug => leds_int,

---
NVOX_cycle => NVOX_cycle,
data_processed => data_processed
```

Listing A.6. FX2 chip interface block
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL,
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ctrl1 is
    port (  
        rst : in std_logic; -- not rst_n
        clk : in std_logic; -- f#2.clk
        flaga : in std_logic;
        flagb : in std_logic; -- full
        flagc : in std_logic; -- empty
        flagd : in std_logic;
        sloe : out std_logic;
        slrd : out std_logic;
        slwr : out std_logic;
        pktend : out std_logic;
        fifoadr : out std_logic_vector(1 downto 0);
        fd : in std_logic_vector(7 downto 0);
        rxfifo_full : in std_logic;
        rxfifo_wr_en : out std_logic;
        txfifo_empty : in std_logic;
        txfifo_rd_en : out std_logic;
        read_data : out std_logic;
        leds : out std_logic_vector(7 downto 0)
    );
end ctrl1;

architecture Behavioral of ctrl1 is
Appendix A (continued)

type state_type is (IDLE, S0, S1, S1b, S2, S2b, S2c, S3, S3b, S3c, S4, S4b, S4c, S5, W1, W1b, W2, W3, W4, W4b, W4c, W5, HANDSHAKE, HANDSHAKE1, HANDSHAKE2, HANDSHAKE2b, HANDSHAKE2c, HANDSHAKE2d, RST_STATE);

signal state, nextstate: state_type;

-- instruction registers
signal ir_reg1, ir_reg2, ir_reg3, ir_reg4: std_logic_vector(7 downto 0);
signal ir_clr1, ir_clr2, ir_clr3, ir_clr4: std_logic;
signal ir_en1, ir_en2, ir_en3, ir_en4: std_logic;

-- program counter
signal pc_reg: std_logic_vector(23 downto 0);
signal pc_clr: std_logic;
signal pc_en: std_logic;

signal op: std_logic_vector(8 downto 1);
signal pc_eq, pc_eq4: std_logic;
signal read_data_sig: std_logic;

signal leds_sig: std_logic_vector(7 downto 0);

begin

read_data <= read_data_sig;
fifoadr <= "00" when read_data_sig = '1' else "10";

leds <= leds_sig;

-- instruction registers
irregproc: process(rst, clk)
begin

if rst = '1' then
  ir_reg1 <= (others => '0');
  ir_reg2 <= (others => '0');
  ir_reg3 <= (others => '0');
  ir_reg4 <= (others => '0');
elsif clk = '1' and clk'event then
  if ir_clr1 = '1' then
    ir_reg1 <= (others => '0');
  elsif ir_en1 = '1' then
    ir_reg1 <= fd;
  end if;
end if;
Appendix A (continued)

if \texttt{ir\_clr2} = '1' then
    \texttt{ir\_reg2} <= (\texttt{others} => '0');
elsif \texttt{ir\_en2} = '1' then
    \texttt{ir\_reg2} <= \texttt{fd};
end if;

--
if \texttt{ir\_clr3} = '1' then
    \texttt{ir\_reg3} <= (\texttt{others} => '0');
elsif \texttt{ir\_en3} = '1' then
    \texttt{ir\_reg3} <= \texttt{fd};
end if;

--
if \texttt{ir\_clr4} = '1' then
    \texttt{ir\_reg4} <= (\texttt{others} => '0');
elsif \texttt{ir\_en4} = '1' then
    \texttt{ir\_reg4} <= \texttt{fd};
end if;

end if;
end process \texttt{ir\_regproc};

process(\texttt{ir\_reg1})
begin
    if \texttt{ir\_reg1}(2 downto 0) = "001" then
        \textit{Reset command}
        \texttt{op(1)} <= '1';
        \texttt{op(2)} <= '0';
        \texttt{op(3)} <= '0';
        \texttt{op(4)} <= '0';
        \texttt{op(5)} <= '0';
        \texttt{op(6)} <= '0';
        \texttt{op(7)} <= '0';
        \texttt{op(8)} <= '0';
    elsif \texttt{ir\_reg1}(2 downto 0) = "010" then
        \textit{Write Weights mem command}
        \texttt{op(1)} <= '0';
        \texttt{op(2)} <= '1';
        \texttt{op(3)} <= '0';
        \texttt{op(4)} <= '0';
        \texttt{op(5)} <= '0';
        \texttt{op(6)} <= '0';
    end if;
end process;
op(7) <= '0';
op(8) <= '0';

elsif ir_reg1(2 downto 0) = "011" then
    -- read Weights mem command
    op(1) <= '0';
op(2) <= '0';
op(3) <= '1';
op(4) <= '0';
op(5) <= '0';
op(6) <= '0';
op(7) <= '0';
op(8) <= '0';

elsif ir_reg1(2 downto 0) = "100" then
    -- read energy mem command
    op(1) <= '0';
op(2) <= '0';
op(3) <= '0';
op(4) <= '1';
op(5) <= '0';
op(6) <= '0';
op(7) <= '0';
op(8) <= '0';

elsif ir_reg1(2 downto 0) = "101" then
    -- write signal mem command
    op(1) <= '0';
op(2) <= '0';
op(3) <= '0';
op(4) <= '0';
op(5) <= '1';
op(6) <= '0';
op(7) <= '0';
op(8) <= '0';

elsif ir_reg1(2 downto 0) = "110" then
    -- read signal mem command
    op(1) <= '0';
op(2) <= '0';
op(3) <= '0';
op(4) <= '0';
op(5) <= '0';
op(6) <= '1';
Appendix A (continued)

op(7) <= '0';
op(8) <= '0';

elsif ir_reg1(2 downto 0) = "111" then

-- start computation command

op(1) <= '0';
op(2) <= '0';
op(3) <= '0';
op(4) <= '0';
op(5) <= '0';
op(6) <= '0';
op(7) <= '1';
op(8) <= '0';

end if;
end process;

-- program counter

process(clk, rst)
begin

if rst = '1' then
    pc_reg <= (others => '0');
elsif clk'event and clk = '1' then
    if pc_clr = '1' then
        pc_reg <= (others => '0');
    elsif pc_en = '1' then
        pc_reg <= pc_reg + 1;
    end if;
end if;
end process;

process(pc_reg, ir_reg2, ir_reg3, ir_reg4)
begin

if pc_reg = ir_reg2(5 downto 0)&ir_reg3&ir_reg4&"00" then
    pc_eq_ir <= '1';
else
    pc_eq_ir <= '0';
end if;
end process;

process(pc_reg)
begin
Appendix A (continued)

if \texttt{pc\_reg} = "00000000000000000000000100" then
  \texttt{pc\_eq\_4} <= '1';
else
  \texttt{pc\_eq\_4} <= '0';
end if;
end process;

\textit{--- state machine}

\texttt{nextstateproc} : \texttt{process(clk, rst)}
begin
  if \texttt{rst} = '1' then
    \texttt{state} <= \texttt{IDLE};
  elsif \texttt{clk} = '1' and \texttt{clk\_event} then
    \texttt{state} <= \texttt{nextstate};
  end if;
end process \texttt{nextstateproc};

\texttt{combproc} : \texttt{process(state, rxfifo\_full, txfifo\_empty, op, pc\_eq\_ir, flagb, flagc)}
begin
  \texttt{nextstate} <= \texttt{state};
  \textit{--- set default values}
  \texttt{rxfifo\_wr\_en} <= '0';
  \texttt{txfifo\_rd\_en} <= '0';
  \texttt{sloe} <= '0';
  \texttt{slrd} <= '0';
  \texttt{slwr} <= '0';
  \texttt{pktend} <= '0';
  \texttt{ir\_en1} <= '0';
  \texttt{ir\_en2} <= '0';
  \texttt{ir\_en3} <= '0';
  \texttt{ir\_en4} <= '0';
  \texttt{ir\_clr1} <= '0';
  \texttt{ir\_clr2} <= '0';
  \texttt{ir\_clr3} <= '0';
  \texttt{ir\_clr4} <= '0';
  \texttt{pc\_en} <= '0';
  \texttt{pc\_clr} <= '0';
  \texttt{read\_data\_sig} <= '1';
  \texttt{case state is}
    when \texttt{IDLE} =>
Appendix A (continued)

\[\text{nextstate} \leftarrow \text{S0} ;\]
\[\text{leds\_sig} \leftarrow "00000001" ;\]

\textit{when S0 =>}
\[\text{if flagc} = '0' \text{ then}\]
\[\text{nextstate} \leftarrow \text{S1} ;\]
\[\text{else}\]
\[\text{nextstate} \leftarrow \text{S0} ;\]
\[\text{end if} ;\]
\[\text{leds\_sig} \leftarrow "00000010" ;\]

\textit{when S1 =>}
\[\text{-- Read the first word from ep2}\]
\[\text{sloe} \leftarrow '1' ;\]
\[\text{rxfifo\_wr\_en} \leftarrow '1' ;\]
\[\text{ir\_en1} \leftarrow '1' ;\]
\[\text{nextstate} \leftarrow \text{S1b} ;\]
\[\text{leds\_sig} \leftarrow "00000011" ;\]

\textit{when S1b =>}
\[\text{sloe} \leftarrow '0' ;\]
\[\text{rxfifo\_wr\_en} \leftarrow '0' ;\]
\[\text{ir\_en1} \leftarrow '0' ;\]
\[\text{sird} \leftarrow '1' ;\]
\[\text{nextstate} \leftarrow \text{S2} ;\]
\[\text{leds\_sig} \leftarrow "00000010" ;\]

\textit{when S2 =>}
\[\text{-- read the second word from ep2}\]
\[\text{sird} \leftarrow '0' ;\]
\[\text{if flagc} = '0' \text{ then}\]
\[\text{nextstate} \leftarrow \text{S2b} ;\]
\[\text{else}\]
\[\text{nextstate} \leftarrow \text{S2} ;\]
\[\text{end if} ;\]
\[\text{leds\_sig} \leftarrow "00000101" ;\]

\textit{when S2b =>}
\[\text{sloe} \leftarrow '1' ;\]
\[\text{rxfifo\_wr\_en} \leftarrow '1' ;\]
\[\text{ir\_en2} \leftarrow '1' ;\]
\[\text{nextstate} \leftarrow \text{S2c} ;\]
\[\text{leds\_sig} \leftarrow "00000110" ;\]

\textit{when S2c =>}
\[\text{sloe} \leftarrow '0' ;\]
Appendix A (continued)

```plaintext
rxifo_wr_en <= '0';
ir_en2 <= '0';
sldr <= '1';
nextstate <= S3;
leds_sig <= "00000111";

when S3 =>
  -- read the third word from ep2
  sldr <= '0';
  if flagc = '0' then
    nextstate <= S3b;
  else
    nextstate <= S3;
  end if;
  leds_sig <= "00001000";

when S3b =>
  sloe <= '1';
  rxifo_wr_en <= '1';
  ir_en3 <= '1';
  nextstate <= S3c;
  leds_sig <= "00001001";

when S3c =>
  sloe <= '0';
  rxifo_wr_en <= '0';
  ir_en3 <= '0';
  sldr <= '1';
  nextstate <= S4;
  leds_sig <= "00001010";

when S4 =>
  -- read the forth word from ep2
  sldr <= '0';
  if flagc = '0' then
    nextstate <= S4b;
  else
    nextstate <= S4;
  end if;
  leds_sig <= "00001011";

when S4b =>
  sloe <= '1';
  rxifo_wr_en <= '1';
  ir_en4 <= '1';
```

Appendix A (continued)

nextstate <= S4c;
leds_sig <= "00001100";

when S4c =>
sloe <= '0';
rxfifo_wr_en <= '0';
ir_en4 <= '0';
sld <= '1';
nextstate <= S5;
leds_sig <= "00001101";

when S5 =>
sld <= '0';
if op(1) = '1' then
  nextstate <= RST_STATE;
elsif op(2) = '1' then
  nextstate <= W1;
elsif op(3) = '1' then
  nextstate <= R1;
elsif op(4) = '1' then
  nextstate <= R1;
elsif op(5) = '1' then
  nextstate <= W1;
elsif op(6) = '1' then
  nextstate <= R1;
elsif op(7) = '1' then
  nextstate <= HANDSHAKE1;
else
  nextstate <= IDLE;
end if;
leds_sig <= "00001110";

when W1 =>
  -- read data from ep2 and write it to rx fifo
  if pc_eq_lr = '1' then
    nextstate <= W2;
  else
    nextstate <= W1b;
  end if;
  leds_sig <= "00001111";

when W1b =>
  if flagc = '0' and rxfifo_full = '0' then
    nextstate <= W1c;
else
    nextstate <= W1b;
end if;
leds_sig <= "00010000";

when W1c =>
    -- ep2 is not empty
    sloe <= '1';
    rxfifo_wr_en <= '1';
    pc_en <= '1';
    nextstate <= W1d;
    leds_sig <= "00010001";

when W1d =>
    sloe <= '0';
    rxfifo_wr_en <= '0';
    pc_en <= '0';
    slrd <= '1';
    nextstate <= W1;
    leds_sig <= "00010010";

when W2 =>
    -- all data has been read, clear ir and pc
    ir_clr1 <= '1';
    ir_clr2 <= '1';
    ir_clr3 <= '1';
    ir_clr4 <= '1';
    pc_clr <= '1';
    nextstate <= IDLE;
    leds_sig <= "00010011";

when R1 =>
    -- read data from txfifo and write to ep6
    read_data_sig <= '0';
    if pc_eq_ir = '1' then
        nextstate <= R2;
    else
        nextstate <= R1b;
    end if;
    leds_sig <= "00010100";

when R1b =>
    read_data_sig <= '0';
    if flagb = '0' and txfifo_empty = '0' then
        nextstate <= R1c;
else
    nextstate <= R1b;
end if;
leds_sig <= "00010101";
when R1c =>
    read_data_sig <= '0';
txifo_rd_en <= '1';
slwr <= '1';
pc_en <= '1';
nextstate <= R1;
leds_sig <= "00010110";
when R2 =>
    -- all data has been written, assert pktend, clear ir and pc
    read_data_sig <= '0';
    ir_clr1 <= '1';
    ir_clr2 <= '1';
    ir_clr3 <= '1';
    ir_clr4 <= '1';
    pc_clr <= '1';
    if flagb = '0' then
        nextstate <= R2b;
    else
        nextstate <= R2;
    end if;
    leds_sig <= "00010111";
when R2b =>
    read_data_sig <= '0';
    pktend <= '1';
    nextstate <= R2c;
    leds_sig <= "00011000";
when R2c =>
    read_data_sig <= '0';
    pktend <= '1';
    nextstate <= R2d;
    leds_sig <= "00011001";
when R2d =>
    read_data_sig <= '0';
    pktend <= '0';
    nextstate <= IDLE;
    leds_sig <= "00011010";
when HANDSHAKE =>
  -- send HANDSHAKE code
  read_data_sig <= '0';
  if pc_eq_4 = '1' then
    nextstate <= HANDSHAKE2;
  else
    nextstate <= HANDSHAKE1;
  end if;
  leds_sig <= "00010100";
when HANDSHAKE1 =>
  read_data_sig <= '0';
  if flag b = '0' and txfifo_empty = '0' then
    nextstate <= HANDSHAKE1b;
  else
    nextstate <= HANDSHAKE1;
  end if;
  leds_sig <= "00010101";
when HANDSHAKE1b =>
  read_data_sig <= '0';
  txfifo_rd_en <= '1';
  slwr <= '1';
  pc_en <= '1';
  nextstate <= HANDSHAKE;
  leds_sig <= "00010110";
when HANDSHAKE2 =>
  -- all data has been written, assert pktend, clear ir and pc
  read_data_sig <= '0';
  ir_clr1 <= '1';
  ir_clr2 <= '1';
  ir_clr3 <= '1';
  ir_clr4 <= '1';
  pc_clr <= '1';
  if flag b = '0' then
    nextstate <= HANDSHAKE2b;
  else
    nextstate <= HANDSHAKE2;
  end if;
  leds_sig <= "00010111";
when HANDSHAKE2b =>
  read_data_sig <= '0';
Appendix A (continued)

```vhdl
pktend <= '1';
nexstate <= HANDSHAKE2c;
leds_sig <= "00011000";
when HANDSHAKE2c =>
  read_data_sig <= '0';
pktend <= '1';
nexstate <= HANDSHAKE2d;
leds_sig <= "00011001";
when HANDSHAKE2d =>
  read_data_sig <= '0';
pktend <= '0';
nexstate <= IDLE;
leds_sig <= "00011010";
when RST_STATE =>
  nexstate <= IDLE;
leds_sig <= "00011011";
when others =>
  nexstate <= IDLE;
leds_sig <= "00011100";
end case;

end process combproc;

end Behavioral;
```

Listing A.7. Communication FSM

---

-- Company:
-- Engineer: PHUONG NGUYEN, Francesco Colonna
--
-- Create Date: 20:19:52 08/22/2011
-- Design Name:
-- Module Name: ctrl2 - Behavioral
-- Project Name:
-- Target Devices:
-- Tool versions:
-- Description:
-- Dependencies:
library IEEE;

use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity ctrl2 is
  port ( 
    clk : in std_logic;
    rst : in std_logic;
    
    rx fifo dout : in std_logic_vector(31 downto 0);
    rx fifo rd_en : out std_logic;
    rx fifo empty : in std_logic;
    
    tx fifo din : out std_logic_vector(31 downto 0);
    tx fifo wr_en : out std_logic;
    tx fifo full : in std_logic;
    
    bram16k din : out std_logic_vector(31 downto 0);
    bram16k dout : in std_logic_vector(31 downto 0);
    bram16k en : out std_logic;
    bram16k we : out std_logic;
    bram16k addr : out std_logic_vector(16 downto 0);
    
    bram2k din0 : out std_logic_vector(7 downto 0);
    bram2k din1 : out std_logic_vector(7 downto 0);
    bram2k din2 : out std_logic_vector(7 downto 0);
    bram2k din3 : out std_logic_vector(7 downto 0);
    bram2k dout0 : in std_logic_vector(7 downto 0);
    bram2k dout1 : in std_logic_vector(7 downto 0);
    bram2k dout2 : in std_logic_vector(7 downto 0);
    bram2k dout3 : in std_logic_vector(7 downto 0);
    bram2k en : out std_logic;
  );
Appendix A (continued)

```vhdl
architecture Behavioral of ctrl2 is
    component register_par is
        Generic (bit_size : integer :=16);
        Port ( din : in STD_LOGIC_VECTOR (bit_size-1 downto 0);
            clk : in std_logic;
            rst : in std_logic;
            enable: in std_logic;
            dout : out STD_LOGIC_VECTOR (bit_size-1 downto 0));
    end component;

    type state_type is (IDLE, S0, S1, S2, W1, W2, W1_SM, W2_SM, R1, R1b, R1e_SM, R2, R2_SM, R1b_SM,
        R2_SM, R1e_SM, R1b_EM, R2_EM, START_CALC1, START_CALC2, START_CALC3, START_CALC4,
        START_CALC5, RST_STATE, RST_STATE1, RST_STATE2, RST_STATE3);

    signal state, nextstate: state_type;
    --
    signal ir_reg: std_logic_vector(31 downto 0);
    signal ir_clr: std_logic;
    signal ir_en: std_logic;
    --
    signal pc_reg: std_logic_vector(23 downto 0);
    signal pc_clr: std_logic;
    signal pc_en: std_logic;
    --
    signal op: std_logic_vector(8 downto 1);
```
Appendix A (continued)

```vhdl
signal pc_eq_ir : std_logic;
signal read_bram_16k : std_logic;

signal handshake_en : std_logic;
signal state_leds : std_logic_vector(7 downto 0);

begin

bram2k_addr <= ir_reg(31 downto 27) & pc_reg(9 downto 0);
bram16k_addr <= pc_reg(16 downto 0);

NVOX_cycle <= ir_reg(8 downto 0);

-- instruction register load
ir_reg_proc : process(clk, rst)
begin
    if rst = '1' then
        ir_reg <= (others => '0');
    elsif clk'event and clk = '1' then
        if ir_clr = '1' then
            ir_reg <= (others => '0');
        elsif ir_en = '1' then
            ir_reg <= rxfifo_dout;
            leds_debug(31 downto 16) <= rxfifo_dout(31 downto 16);
        end if;
    end if;
end process ir_reg_proc;

leds_debug(7 downto 0) <= op;
leds_debug(15 downto 8) <= state_leds;

process (ir_reg, clk)
begin
    if ir_reg(26 downto 24) = "001" then -- Reset command
        op(1) <= '1';
        op(2) <= '0';
        op(3) <= '0';
        op(4) <= '0';
        op(5) <= '0';
        op(6) <= '0';
        op(7) <= '0';
        op(8) <= '0';
    end if;
end process ir_reg_proc;
```

Appendix A (continued)

```plaintext
elsif ir_reg(26 downto 24) = "010" then -- write Weights mem command
    op(1) <= '0';
    op(2) <= '1';
    op(3) <= '0';
    op(4) <= '0';
    op(5) <= '0';
    op(6) <= '0';
    op(7) <= '0';
    op(8) <= '0';
elsif ir_reg(26 downto 24) = "011" then -- read Weights mem command
    op(1) <= '0';
    op(2) <= '0';
    op(3) <= '1';
    op(4) <= '0';
    op(5) <= '0';
    op(6) <= '0';
    op(7) <= '0';
    op(8) <= '0';
elsif ir_reg(26 downto 24) = "100" then -- read energy mem command
    op(1) <= '0';
    op(2) <= '0';
    op(3) <= '0';
    op(4) <= '1';
    op(5) <= '0';
    op(6) <= '0';
    op(7) <= '0';
    op(8) <= '0';
elsif ir_reg(26 downto 24) = "101" then -- write signal mem command
    op(1) <= '0';
    op(2) <= '0';
    op(3) <= '0';
    op(4) <= '0';
    op(5) <= '1';
    op(6) <= '0';
    op(7) <= '0';
    op(8) <= '0';
elsif ir_reg(26 downto 24) = "110" then -- read signal mem command
    op(1) <= '0';
    op(2) <= '0';
    op(3) <= '0';
```

Appendix A (continued)

op(4) <= '0';
op(5) <= '0';
op(6) <= '1';
op(7) <= '0';
op(8) <= '0';
elif ir_reg(26 downto 24) = "111" then — start computation command
  op(1) <= '0';
op(2) <= '0';
op(3) <= '0';
op(4) <= '0';
op(5) <= '0';
op(6) <= '0';
op(7) <= '1';
op(8) <= '0';
else
  op <= (OTHERS=>'0');
end if;
end process;

— program counter
process_pc_reg: process(clk, rst)
begin
  if rst = '1' then
    pc_reg <= (others => '0');
  elsif clk'event and clk = '1' then
    if pc_clr = '1' then
      pc_reg <= (others => '0');
    elsif pc_en = '1' then
      pc_reg <= pc_reg + 1;
    end if;
  end if;
end process pc_reg;

process(ir_reg, pc_reg)
begin
  if pc_reg = ir_reg(23 downto 0) then
    pc_eq_ir <= '1';
  else
    pc_eq_ir <= '0';
  end if;
end process;

-- Connect data ports
bram16k_din <= rx fifo_dout;
bram2k_din0 <= rx fifo_dout(31 downto 24);
bram2k_din1 <= rx fifo_dout(23 downto 16);
bram2k_din2 <= rx fifo_dout(15 downto 8);
bram2k_din3 <= rx fifo_dout(7 downto 0);

tx fifo_din <= "00000000000000000000000000001111" when handshake_en = '1'
else bram16k_dout when read_bram_16k = '1' else bram2k_dout0 &
bram2k_dout1 & bram2k_dout2 & bram2k_dout3;

-- state machine
nextstateproc: process(rst, clk)
begin
  if rst = '1' then
    state <= IDLE;
  elsif clk'event and clk = '1' then
    state <= nextstate;
  end if;
end process nextstateproc;

combproc: process(state, rx fifo_empty, tx fifo full, op, pc eq ir, data processed)
begin
  nextstate <= state;
  case state is
    when IDLE =>
      -- first stage
      tx fifo wr en <= '0';
      rx fifo rd en <= '0';
      ir en <= '0';
      ir clr <= '0';
      pc en <= '0';
      pc clr <= '0';
      bram16k en <= '0';
      bram16k we <= '0';
      bram2k en <= '0';
      bram2k we <= '0';
      read bram 16k <= '1';
reset_cpu <= '0';
instrackF <= '1';
dataackM <= '1';
enmem_en <= '1';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= S0;

when S0 =>
  -- check for incoming commands
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '0';
  bram16k_en <= '0';
  bram16k_we <= '0';
  bram2k_en <= '0';
  bram2k_we <= '0';
  read_bram_16k <= '1';
  reset_cpu <= '0';
  instrackF <= '1';
dataackM <= '1';
enmem_en <= '1';
handshake_en <= '0';
START_BEAF <= '0';
if rxfifo_empty = '0' then
  nextstate <= S1;
else
  nextstate <= S0;
end if;

when S1 =>
  -- Read the first word from rxfifo
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '1';
  ir_en <= '1';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '0';
  bram16k_en <= '0';
Appendix A (continued)

bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= S2;

when S2 =>
  -- opcode analysis
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '0';
  bram16k_en <= '0';
  bram16k_we <= '0';
  bram2k_en <= '0';
  bram2k_we <= '0';
  read_bram_16k <= '1';
  reset_cpu <= '0';
  instrackF <= '0';
  dataackM <= '0';
  enmem_en <= '0';
  handshake_en <= '0';
  START_BEAF <= '0';
  if op(1) = '1' then
    nextstate <= RST_STATE;
  elsif op(2) = '1' then
    nextstate <= W1;
  elsif op(3) = '1' then
    nextstate <= R1;
  elsif op(4) = '1' then
    nextstate <= R1_EM;
  elsif op(5) = '1' then
    nextstate <= W1SM;
Appendix A (continued)

elsif op(6) = '1' then
    nextstate <= R1SM;
elsif op(7) = '1' then
    nextstate <= START_CALCI;
else
    nextstate <= IDLE;
end if;

when W1 =>
    -- Read data from rx fifo and write it to ram
    txfifo_wr_en <= '0';
    rxfifo_rd_en <= not rx fifo_empty;
    ir_en <= '0';
    ir_clr <= '0';
    pc_en <= not rx fifo_empty;
    pc_clr <= '0';
    bram16k_en <= not rx fifo_empty;
    bram16k_we <= not rx fifo_empty;
    bram2k_en <= '0';
    bram2k_we <= '0';
    read_bram_16k <= '1';
    reset_cpu <= '0';
    instrackF <= '0';
    dataackM <= '0';
    enmem_en <= '0';
    handshake_en <= '0';
    START_BEAF <= '0';
    if pc_eq_ir = '1' then
        nextstate <= W2;
    else
        nextstate <= W1;
    end if;
end if;

when W2 =>
    -- reset program counter
    txfifo_wr_en <= '0';
    rxfifo_rd_en <= '0';
    ir_en <= '0';
    ir_clr <= '0';
    pc_en <= '0';
    pc_clr <= '1';
    bram16k_en <= '0';
Appendix A (continued)

```
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '0';
intrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= RST_STATE;

when W1_SM =>
  -- Read data from rxfifo and write it to samples memory
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= not rxfifo_empty;
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= not rxfifo_empty;
  pc_clr <= '0';
  bram16k_en <= '0';
  bram16k_we <= '0';
  bram2k_en <= not rxfifo_empty;
  bram2k_we <= not rxfifo_empty;
  read_bram_16k <= '0';
  reset_cpu <= '0';
  instrackF <= '0';
  dataackM <= '0';
  enmem_en <= '0';
  handshake_en <= '0';
  START_BEAF <= '0';
  if pc_eq_ir = '1' then
    nextstate <= W2_SM;
  else
    nextstate <= W1_SM;
  end if;
when W2_SM =>
  -- reset program counter
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
```
Appendix A (continued)

```vhdl
ir_en <= '0';
ir_clr <= '0';
pc_en <= '0';
pc_clr <= '1';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '0';
reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= RST_STATE;
when R1 =>
    -- read data from Weights and write it to txfifo
    if txfifo_full = '0' then
        txfifo_wr_en <= '0';
        rxfifo_rd_en <= '0';
        ir_en <= '0';
        ir_clr <= '0';
        pc_en <= '1';
        pc_clr <= '0';
        bram16k_en <= '1';
        bram16k_we <= '0';
        bram2k_en <= '0';
        bram2k_we <= '0';
        read_bram_16k <= '1';
        reset_cpu <= '0';
        instrackF <= '0';
        dataackM <= '0';
        enmem_en <= '0';
        handshake_en <= '0';
        START_BEAF <= '0';
        nextstate <= R1b;
    else
        txfifo_wr_en <= '0';
        rxfifo_rd_en <= '0';
```

Appendix A (continued)

ir_en <= '0';
ir clr <= '0';
pc_en <= '0';
pc clr <= '0';
bram16k_en <= '1';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '0';
intrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= R1;

end if;
when R1b =>

-- Check for read operation completion

txfifo_wr_en <= '1';
rxfifo_rd_en <= '0';
ir_en <= '0';
ir clr <= '0';
pc_en <= '0';
pc clr <= '0';
bram16k_en <= '1';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '0';
intrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
if pc_eq_ir = '1' then

nextstate <= R2;
else

nextstate <= R1;
end if;
Appendix A (continued)

end if;

when R2 =>

-- reset program counter

txfifo_wr_en <= '0';
rxfifo_rd_en <= '0';
ir_en <= '0';
ir_clr <= '0';
pc_en <= '0';
pc_clr <= '1';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= IDLE;

when R1_EM =>

-- read data from ENRGY mem and write it to txfifo

if txfifo_full = '0' then

txfifo_wr_en <= '0';
rxfifo_rd_en <= '0';
ir_en <= '0';
ir_clr <= '0';
pc_en <= '1';
pc_clr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '0';
reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '1';
handshake_en <= '0';
Appendix A (continued)

START_BEAF <= '0';
nextstate <= R1b_EM;

else
    tx fifo wr en <= '0';
    rx fifo rd en <= '0';
    ir en <= '0';
    ir clr <= '0';
    pc en <= '0';
    pc clr <= '0';
    bram16k en <= '0';
    bram16k we <= '0';
    bram2k_en <= '0';
    bram2k_we <= '0';
    read bram_16k <= '0';
    reset_cpu <= '0';
    instrackF <= '0';
    dataackM <= '0';
    enmem_en <= '1';
    handshake_en <= '0';
    START_BEAF <= '0';
    nextstate <= R1_EM;
end if;

when R1b_EM =>

    -- Check for read operation completion
    tx fifo wr en <= '1';
    rx fifo rd en <= '0';
    ir en <= '0';
    ir clr <= '0';
    pc en <= '0';
    pc clr <= '0';
    bram16k en <= '0';
    bram16k we <= '0';
    bram2k_en <= '0';
    bram2k_we <= '0';
    read bram_16k <= '0';
    reset_cpu <= '0';
    instrackF <= '0';
    dataackM <= '0';
    enmem_en <= '1';
    handshake_en <= '0';
Appendix A (continued)

START_BEAF <= '0';
if pc_eq_ir = '1' then
    nextstate <= R2_EM;
else
    nextstate <= R1_EM;
end if;
when R2_EM =>
    -- reset program counter
    txfifo_wr_en <= '0';
    rxfifo_rd_en <= '0';
    ir_en <= '0';
    ir_clr <= '0';
    pc_en <= '0';
    pc_clr <= '1';
    bram16k_en <= '0';
    bram16k_we <= '0';
    bram2k_en <= '0';
    bram2k_we <= '0';
end when;
when R1_SM =>
    -- read data from samples memory and write it to txfifo
    if txfifo_full = '0' then
        txfifo_wr_en <= '0';
        rxfifo_rd_en <= '0';
        ir_en <= '0';
        ir_clr <= '0';
        pc_en <= '1';
        pc_clr <= '0';
        bram16k_en <= '0';
        bram16k_we <= '0';
        bram2k_en <= '1';
        bram2k_we <= '0';
        read_bram_16k <= '0';
    end if;
end when;

Appendix A (continued)

```vhdl
reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nexstate <= R1b_SM;
else
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '0';
  bram16k_en <= '0';
  bram16k_we <= '0';
  bram2k_en <= '1';
  bram2k_we <= '0';
  read_bram_16k <= '0';
  reset_cpu <= '0';
  instrackF <= '0';
  dataackM <= '0';
  enmem_en <= '0';
  handshake_en <= '0';
  START_BEAF <= '0';
  nexstate <= R1_SM;
end if;
when R1b_SM =>
  -- Check for read operation completion
  txfifo_wr_en <= '1';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '0';
  bram16k_en <= '0';
  bram16k_we <= '0';
  bram2k_en <= '1';
  bram2k_we <= '0';
  read_bram_16k <= '0';
```
Appendix A (continued)

reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
if pc.eq.ir = '1' then
    nextstate <= R2_SM;
else
    nextstate <= R1_SM;
end if;
when R2_SM =>
    -- reset program counter
    txfifo_wr_en <= '0';
    rxfifo_rd_en <= '0';
    ir_en <= '0';
    irclr <= '0';
    pc_en <= '0';
    pcclr <= '1';
    bram16k_en <= '0';
    bram16k_we <= '0';
    bram2k_en <= '0';
    bram2k_we <= '0';
    read_bram_16k <= '0';
    reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= IDLE;
when START_CALC1 =>
    -- initiate Computation process
    txfifo_wr_en <= '0';
    rxfifo_rd_en <= '0';
    ir_en <= '0';
    irclr <= '0';
    pc_en <= '0';
    pcclr <= '1';
    bram16k_en <= '0';
Appendix A (continued)

bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '0';
reset_cpu <= '0';
intrackF <= '1';
dataackM <= '1';
enmem_en <= '0';
handshake_en <= '0';
START_BEAFL <= '1';
nextstate <= START_CALC2;
when START_CALC2 =>
  -- wait for Computation completion
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '1';
  bram16k_en <= '0';
  bram16k_we <= '0';
  bram2k_en <= '0';
  bram2k_we <= '0';
  read_bram_16k <= '0';
  reset_cpu <= '0';
intrackF <= '1';
dataackM <= '1';
enmem_en <= '0';
handshake_en <= '0';
START_BEAFL <= '0';
if data_processed = '1' then
  nextstate <= START_CALC3;
else
  nextstate <= START_CALC2;
end if;
when START_CALC3 =>
  -- send handshake to txfifo
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
Appendix A (continued)

irclr <= '0';
pc_en <= '0';
pcclr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '0';
reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '1';
START_BEAF <= '0';
nextstate <= START_CALC4;

when START_CALC4 =>

-- signal completion via txfifo handshake

txfifo_wr_en <= '1';
rxfifo_rd_en <= '0';
iren <= '0';
irclr <= '0';
pc_en <= '0';
pcclr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '0';
reset_cpu <= '0';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '1';
START_BEAF <= '0';
nextstate <= START_CALC5;

when START_CALC5 =>

-- complete handshake

txfifo_wr_en <= '0';
rxfifo_rd_en <= '0';
iren <= '0';
Appendix A (continued)

ir,clr <= '0';
pc_en <= '0';
pc_clr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '0';
reset_cpu <= '0';
intrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '1';
START_BEAF <= '0';
nextstate <= RST_STATE;

when RST_STATE =>
  -- enter reset process
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '1';
intrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= RST_STATE1;

when RST_STATE1 =>
  -- reset BEAF block
  txfifo_wr_en <= '0';
  rxfifo_rd_en <= '0';
  ir_en <= '0';
Appendix A (continued)

```vhdl
ir.clr <= '0';
pc.en <= '0';
pc.clr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '1';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nexstate <= RST_STATE2;

when RST_STATE2 =>
  -- reset BEAF block
  txfifo_wr_en <= '0';
  rx fifo_rd_en <= '0';
  ir.en <= '0';
  ir.clr <= '0';
  pc.en <= '0';
  pc.clr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '1';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nexstate <= RST_STATE3;

when RST_STATE3 =>
  -- reset BEAF block
  txfifo_wr_en <= '0';
  rx fifo_rd_en <= '0';
  ir.en <= '0';
```

Appendix A (continued)

ir clr <= '0';
pc_en <= '0';
pc_clr <= '0';
bram16k_en <= '0';
bram16k_we <= '0';
bram2k_en <= '0';
bram2k_we <= '0';
read_bram_16k <= '1';
reset_cpu <= '1';
instrackF <= '0';
dataackM <= '0';
enmem_en <= '0';
handshake_en <= '0';
START_BEAF <= '0';
nextstate <= IDLE;

when others =>
  -- default
  txfifo_wr_en <= '0';
  rx fifo_rd_en <= '0';
  ir_en <= '0';
  ir_clr <= '0';
  pc_en <= '0';
  pc_clr <= '1';
  bram16k_en <= '0';
  bram16k_we <= '0';
  bram2k_en <= '0';
  bram2k_we <= '0';
  read_bram_16k <= '1';
  reset_cpu <= '0';
  instrackF <= '1';
  dataackM <= '1';
  enmem_en <= '1';
  handshake_en <= '0';
  START_BEAF <= '0';
  nextstate <= IDLE;
end case;

end process combproc;

-- debug leds setting
leds_proc : process(state)
begin
  case state is
    when IDLE =>
      state_leds := "00000000";
    when S0 =>
      state_leds := "00000001";
    when S1 =>
      state_leds := "00000010";
    when S2 =>
      state_leds := "00000011";
    when W1 =>
      state_leds := "00000100";
    when W2 =>
      state_leds := "00000101";
    when W1_SM =>
      state_leds := "00000110";
    when W2_SM =>
      state_leds := "00000111";
    when R1 =>
      state_leds := "00001000";
    when R1b =>
      state_leds := "00001001";
    when R1_SM =>
      state_leds := "00001010";
    when R2 =>
      state_leds := "00001011";
    when R1_SM =>
      state_leds := "00001100";
    when R1b_SM =>
      state_leds := "00001110";
    when R2_SM =>
      state_leds := "00001111";
    when R1_EM =>
      state_leds := "00010011";
    when R1b_EM =>
      state_leds := "00010000";
    when R2_EM =>
      state_leds := "00010001";
    when START_CALC1 =>
      state_leds := "00010010";
when START_CALC2 =>
    state_leds <= "00010011";
when START_CALC3 =>
    state_leds <= "00010100";
when START_CALC4 =>
    state_leds <= "00010101";
when START_CALC5 =>
    state_leds <= "00010110";
when RST_STATE =>
    state_leds <= "00010111";
when RST_STATE1 =>
    state_leds <= "00011000";
when RST_STATE2 =>
    state_leds <= "00011001";
when RST_STATE3 =>
    state_leds <= "00011010";
end case;
end process;
end Behavioral;

Listing A.8. BEAF interface

---

--- Company: PoliTO
--- Engineer: Paolo Rosingana, Francesco Colonna
---
--- Create Date: 20/08/2012
--- Design Name:
--- Module Name: BEAF_interface – Behavioral
--- Project Name:
--- Target Devices:
--- Tool versions:
--- Description:
---
--- Dependencies:
---
--- Revision:
--- Revision 1.0 final release
--- Additional Comments:
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use ieee.numeric_std.all;
use IEEE.STD_LOGIC_UNSIGNED.ALL;
use ieee.math_real.all;
use work.pkg_delay.all;
use work.declaration_pack.all;

entity BEAF_interface is
port(
  clk : in std_logic;
  rst : in std_logic;
  switch : in std_logic_vector(7 downto 0);
  rst_mips : out std_logic_vector(31 downto 0);
  adx_mips : out std_logic_vector(31 downto 0);
  do_mips : in std_logic_vector(31 downto 0);
  rst_dcm : out std_logic;
  den_dcm : out std_logic_vector(1 downto 0);
  dwe_dcm : out std_logic_vector(1 downto 0);
  dl_dcm : out std_logic_vector(15 downto 0);
  daddr_dcm : out std_logic_vector(6 downto 0);
  do_dcm : in std_logic_vector(15 downto 0);
  drdy_dcm : in std_logic_vector(1 downto 0);
  locked_dcm : in std_logic_vector(1 downto 0);
  zf_timer : in std_logic;
  init_timer : out std_logic_vector(31 downto 0);
  leds : out std_logic_vector(7 downto 0);
  data_processed : out std_logic;
  -- f2zinterface
  bram16k_din : in std_logic_vector(31 downto 0);
  bram16k dout : out std_logic_vector(31 downto 0);
  bram16k_en : in std_logic;
  bram16k_we : in std_logic;
  bram16k_addr : in std_logic_vector(16 downto 0);
  --
Appendix A (continued)

```vhdl
architecture Behavioral of BEAF_interface is

component Weights_MEM IS

port (  
  clka : IN std_logic;  
  dina : IN std_logic_VECTOR(15 downto 0);  
  addra : IN std_logic_VECTOR(16 downto 0);  
  ena : IN std_logic;  
  wea : IN std_logic_VECTOR(0 downto 0);  
  douta : OUT std_logic_VECTOR(15 downto 0));

END component;

component regN is generic (N: integer:=8);

port (clk, reset : in std_logic;  
  a : in std_logic_vector( N-1 downto 0);  
  b : out std_logic_vector(N-1 downto 0);  
  en, clr : in std_logic);

END component;
```
component BRAM_2K_CTRL_0
  port (
    addr : in std_logic_vector(10 downto 0);
    clk : in std_logic;
    din : in std_logic_vector(7 downto 0);
    dout : out std_logic_vector(7 downto 0);
    en : in std_logic;
    we : in std_logic);
end component;

COMPONENT SAMPLES_MEM_512x32 IS
  port (
    clka : IN std_logic;
    dina : IN std_logic_VECTOR(31 downto 0);
    addra : IN std_logic_VECTOR(8 downto 0);
    ena : IN std_logic;
    wea : IN std_logic_VECTOR(0 downto 0);
    douta : OUT std_logic_VECTOR(31 downto 0));
END COMPONENT;

COMPONENT ENERGY_MEM_1024x32 IS
  port (
    clka : IN std_logic;
    dina : IN std_logic_VECTOR(31 downto 0);
    addra : IN std_logic_VECTOR(9 downto 0);
    ena : IN std_logic;
    wea : IN std_logic_VECTOR(0 downto 0);
    douta : OUT std_logic_VECTOR(31 downto 0);
    clkb : IN std_logic;
    dinb : IN std_logic_VECTOR(31 downto 0);
    addrb : IN std_logic_VECTOR(9 downto 0);
    enb : IN std_logic;
    web : IN std_logic_VECTOR(0 downto 0);
    doutb : OUT std_logic_VECTOR(31 downto 0));
END COMPONENT;

component DelBEAM is
  port (
    clk , rst : in std_logic;
samples_in: in arrayvectorin;
q: in FIRINPUT;
delay_val: in arrayvectordelay;
start: in std_logic;
last_fir_loop: in std_logic;
loop_rst: in std_logic;
finish_mist: out std_logic;
n_low, n_high: in std_logic_vector(nlow_nhigh_length-1 downto 0);
energy_out: out STD_LOGIC_VECTOR(total_word_length_out-1 downto 0);
end component;

-- component FSM is

port ( rst, clk: in std_logic;
count_W_end: in std_logic;
count_Iter_end: in std_logic;
count_Ant_end: in std_logic;
count_FIR_end: in std_logic;
START: in std_logic;
finish_MIST: in std_logic;
EN_CW, RST_CW : out std_logic;
EN_CAWM, RST_CAWM: out std_logic;
EN_CNT, RST_CNT : out std_logic;
EN_CFIR, RST_CFIR : out std_logic;
EN_CITER, RST_CITER: out std_logic;
EN_WM, EN_RM, EN_WM, WE_RM: out std_logic;
EN_CAS, RST_CAS: out std_logic;
LOOP_RST: out std_logic;
Load_SHIFT, Clear_SHIFT: out std_logic;
EXE_START, RST_MIST: out std_logic
);

end component;

--
type  regs is array (0 to 31) of std_logic_vector(31 downto 0);
signal register_set, ctrl_regs_set : regs;
Appendix A (continued)

-- weights memory signals

signal WMemAddr: std_logic_vector(16 downto 0);
signal WMemDin, WMemDout: std_logic_vector(15 downto 0);
signal WMemEn, EN_WM, EN_WM_R, WR_WM: std_logic;
signal WMemWe: std_logic_vector(0 downto 0);

-- sample memory signals

signal SMEMAddr: array vectors_memaddress;
signal SMEMDin, SMEMDout: array vectors_memdata;
signal SMEMEn: std_logic_vector(1 to integer(ceil(real(NANT)/real(2))));
signal SMEMWe: array vectors_memwe;

-- energy memory signals

signal EMAddrA: std_logic_vector(9 downto 0);
signal EDMina, EDMouta: std_logic_vector(31 downto 0);
signal EMEna: std_logic;
signal EMWea: std_logic_vector(0 downto 0);
signal EMAddrB: std_logic_vector(9 downto 0);
signal EDMinb, EDMoutb: std_logic_vector(31 downto 0);
signal EMEnb: std_logic;
signal EMWeb: std_logic_vector(0 downto 0);

--
signal energy_write_val: std_logic_vector(total_word_length_out-1 downto 0);

-- weights memory address signals

signal count_AddrWM: std_logic_vector(16 downto 0);
signal EN_WM, RST_WM, count_AddrWM_end: std_logic;

-- Signals of counter for Loading Weights

signal count_W: std_logic_vector(8 downto 0);
signal EN_WM, RST_WM, count_W_end: std_logic;

-- Signals of counter for processed antennas

signal count_ANT: std_logic_vector(7 downto 0);
signal EN_ANT, RST_ANT, count_ANT_end: std_logic;

-- Signals of counter for loaded filters

signal count_FIR: std_logic_vector(7 downto 0);
signal count_smem_index: integer := 0;
signal EN_CFIR, RST_CFIR, count_FIR_end: std_logic;

-- Signals of counter for completed iterations

signal count_Iter: std_logic_vector(8 downto 0);
signal EN_CITER, RST_CITER, count_Iter_end: std_logic;

signal Load_SHIFT, Clear_SHIFT: std_logic; -- Signal for loading weights in the shift register.
Appendix A (continued)

type WEIGHTSDELAY is array (0 to NFIR*55+NFIR) of std_logic_vector(word_length−1 downto 0);
signal ShiftReg_Out: WEIGHTSDELAY; — weights and delay shift register signals

signal q: FIRINPUT; — Input to FIR filters

signal EXE_START, finish_MIST, RST_MIST, RST_MIST_aux: std_logic;
signal EN_SM, EN_SM_R: std_logic; — samples memories controls;
signal EN_EM, EN_EM_R, WE_EM, WE_EM_R: std_logic; — samples memories controls;
— counter for address samples memories
signal EN_CAS, RST_CAS, count_AddrSM_end: std_logic;
signal count_AddrSM: std_logic_vector(8 downto 0);
— SIGNALS for MIST
signal samples_in: array vector in;
signal smem_array: array vectors mem;
signal smem_shift: array vectors memqueue;
signal n_low, n_high: std_logic_vector(nlow_nhigh_length−1 downto 0);
signal delay_val: array vector delay;
signal loop_rst: std_logic;
signal Samples_read: std_logic_vector(31 downto 0);

signal clkn: std_logic;

begin

clkn<=not(clk);

— Instance memories

WMEM_inst: Weights_MEM port map (  
  clka=>clk ,  
  dina=>WMem_Din,  
  douta=>WMem_Dout,  
  addra=>WMem_Addr,  
  ena=>WMem_En,  
  wea=>WMem_We);

Smem_inst: for i in 1 to integer(cei{l}(real(NANT)/real(2))) generate  
  smem_int_inst: SAMPLES_MEM_512x32 port map (  

Appendix A (continued)

```vhdl
clk_a=>clk,
din_a=>SMEM_Din(i),
addr_a=>SMEM_Addr(i),
en_a=>SMEM_En(i),
we_a=>SMEM_We(i),
dout_a=>SMEM_Dout(i));
end generate;

ENERGYMEM_inst: ENERGY_MEM_1024x32 port map (;
clk=>clk,
din=>EM_Dina,
addr=>EM_Addra,
en=>EM_Ena,
we=>EM_Wea,
dout=>EM_Douta,
clk_b=>clk,
din_b=>EM_Dinb,
addr_b=>EM_Addrb,
en_b=>EM_Enb,
we_b=>EM_Web,
dout_b=>EM_Doutb);

-- Instance components

BEAFinst: DelBEAM
port map (;
clk=>clk,
rst=>RST_MIST_aux,
samples_in=>samples_in,
q=>q,
delay_val=>delay_val,
start=>EXE_START,
last_fir_loop=>count_ANT_end,
loop rst=>loop rst,
finish mist=>finish MIST,
n low=>n low,
```

n\_high\Rightarrow n\_high,
energy\_out\Rightarrow energy\_write\_val
);
Appendix A (continued)

-- Shift register for FIR inputs
Shift_reg_gen : for i in 0 to NFIR*55+NFIR-1 generate
    Inst_regs: regN
    generic map(N=>16)
    PORT MAP(
        a => ShiftReg.Out(i),
        b => ShiftReg.Out(i+1),
        clk => clkn,
        reset => rst,
        clr=> Clear_SHIFT,
        en=> Load_SHIFT);
end generate;

delayed_sig_proc: process (clk, rst)
begin
    if rst='1' then
        EN_WM_R<= '0';
        EN_SM_R<= '0';
        EN_EM_R<= '0';
    elsif (clk 'event and clk='0') then
        EN_WM_R<=EN_WM;
        EN_SM_R<=EN_SM;
        EN_EM_R<=EN_EM;
        WE_EM_R<=WE_EM;
    end if;
end process delayed_sig_proc;

-- connections
outer_qconn_gen : for j in 0 to NFIR-1 generate
    delay_val(j)<=ShiftReg.Out(NFIR*55+NFIR-J*(55+1))(pdelay=1 downto 0);
end generate outer_qconn_gen;

inner_qconn_gen : for i in 0 to 54 generate
    q(j+1)(54-i)<= ShiftReg.Out(NFIR*55+NFIR-(i+1)-j*(55+1));
end generate inner_qconn_gen;
end generate outer_qconn_gen;
Appendix A (continued)

−− Samples memory parametric connections

smem_connection: for i in 0 to integer(ceil(real(NANT)/real(NFIR)))-1 generate
    smem_distribution: for j in 0 to NFIR-1 generate
        too_many_fir: if i*NFIR+j>NANT generate
            order_like_delay: if i*NFIR+j<NANT generate
                smem_shift(i)(i*NFIR+j) <= smem_array(i*NFIR+j);
            end generate order_like_delay;
            smem_out_bound: if i*NFIR+j>NANT generate
                smem_shift(i)(NFIR-1-j) <= (OTHERS => '0');
            end generate smem_out_bound;
        end generate too_many_fir;
        smem_in_bound: if i*NFIR+j<NANT-1 generate
            smem_shift(i)(i*NFIR+j) <= smem_array(i*NFIR+j);
        end generate smem_in_bound;
    end generate smem_distribution;
end generate smem_connection;

samples_in <= smem_shift(count_smem_index);

smem_array_inst: for i in 0 to NANT-1 generate
    smem_array(i) <= SMEM_Dout(integer(floor(real(i)/real(2)))+1)(31-integer(i mod 2)*16 downto 16);
end generate;

ShiftReg_Out(0)<=WMem_Dout;

bram2k_dout0 <= EM_Douta(31 downto 24) when enmem_en = '1' else Samples_read(31 downto 24);
bram2k_dout1 <= EM_Douta(23 downto 16) when enmem_en = '1' else Samples_read(23 downto 16);
bram2k_dout2 <= EM_Douta(15 downto 8) when enmem_en = '1' else Samples_read(15 downto 8);
bram2k_dout3 <= EM_Douta(7 downto 0) when enmem_en = '1' else Samples_read(7 downto 0);

RST_MIST_aux<=RST_MIST or rst;
n_low<= std_logic_vector(to_unsigned(nlow-199+1,9)); -- "000110110"; --252-199+1
n_high<= std_logic_vector(to_unsigned(nhigh-199+1,9)); -- "000111011"; --257-199+1

-- COUNTERS

counter_iter_proc: process(clk, rst) -- signals the completion of NVOX cycle voxels
begin
if rst = '1' then
    count_Iter <= (others => '0');
    count_Iter_end <= '0';
elsif clk'event and clk = '0' then
    if RST_CITER = '1' then
        count_Iter <= (others => '0');
        count_Iter_end <= '0';
    elsif (count_Iter=NVOX_cycle) then --(NVOX_cycle = 195)
        count_Iter <= (others => '0');
        count_Iter_end <= '1';
    elsif EN_CITER = '1' then
        count_Iter <= count_Iter + 1;
        count_Iter_end <= '0';
    end if;
end if;
end process count_Iter_proc;

data_processed <= count_Iter_end;

count_W_proc : process(clk, rst)
begin
    if rst = '1' then
        count_W <= (others => '0');
        count_W_end <= '0';
    elsif clk'event and clk = '0' then
        if RST_CW = '1' then
            count_W <= (others => '0');
            count_W_end <= '0';
        elsif (count_W=std_logic_vector(to_unsigned(55+1-2.9)) and EN_CW = '1') then
            count_W <= (others => '0');
            count_W_end <= '1';
        elsif EN_CW = '1' then
            count_W <= count_W + 1;
            count_W_end <= '0';
        end if;
    end if;
end process count_W_proc;
count\_ANT\_proc: process(clk, rst) begin
  if rst = '1' then
    count\_ANT <= (others => '0');
    count\_ANT\_end <= '0';
  elsif clk'event and clk = '0' then
    if RST\_CANT = '1' then
      count\_ANT <= (others => '0');
      count\_ANT\_end <= '0';
    elsif count\_ANT = std\_logic\_vector(to\_unsigned(integer(NANT−1),8)) and EN\_CANT = '1' then
      --count\_ANT <= (others => '0');
      count\_ANT\_end <= '1';
    elsif EN\_CANT = '1' then
      count\_ANT <= count\_ANT + 1;
      count\_ANT\_end <= '0';
    end if;
  end if;
end process count\_ANT\_proc;

count\_FIR\_proc: process(clk, rst) begin
  if rst = '1' then
    count\_FIR <= (others => '0');
    count\_FIR\_end <= '0';
  elsif clk'event and clk = '0' then
    if RST\_CFIR = '1' then
      count\_FIR <= (others => '0');
      count\_FIR\_end <= '0';
    elsif (count\_FIR = std\_logic\_vector(to\_unsigned(NFIR−1,8)) and EN\_CFIR = '1') then
      count\_FIR <= (others => '0');
      count\_FIR\_end <= '1';
    elsif EN\_CFIR = '1' then
      count\_FIR <= count\_FIR + 1;
      count\_FIR\_end <= '0';
    end if;
  end if;
end if;
Appendix A (continued)

```vhdl
end process count_FIR_proc;

count_smem_proc: process(clk, rst, count_FIR_end, count_ANT_end)
variable flag: std_logic;
begin
  if rst = '1' then
    count_smem_index <= 0;
    flag := '0';
  elsif clk'event and clk = '1' then
    if loop_rst = '1' then
      count_smem_index <= 0;
      flag := '0';
    elsif (count_FIR_end = '1' or count_ANT_end = '1') and EN_CFR = '1' then
      if (flag = '0') then
        flag := '1';
      elsif count_smem_index < integer(cei(real(NANT)/real(NFIR)))-1 then
        count_smem_index <= count_smem_index + 1;
      end if;
    end if;
  end if;
end process count_smem_proc;

count_AddrWM_proc: process(clk, rst)
begin
  if rst = '1' then
    count_AddrWM <= (others => '0');
    count_AddrWM_end <= '0';
  elsif clk'event and clk = '0' then
    if RST_CAWM = '1' then
      count_AddrWM <= (others => '0');
      count_AddrWM_end <= '0';
    elsif count_AddrWM("1111111111111111") and EN_CAWM = '1' then
      count_AddrWM <= (others => '0');
      count_AddrWM_end <= '1';
    elsif EN_CAWM = '1' then
      count_AddrWM <= count_AddrWM + 1;
      count_AddrWM_end <= '0';
    end if;
  end if;
```
Appendix A (continued)

``` VHDL
end if;
end process count_AddrWM_proc;

count_AddrSM_proc: process(clk, rst)
begin
  if rst = '1' then
    count_AddrSM <= (others => '0');
    --count_AddrSM_end <= '0';
  elsif clk'event and clk = '0' then
    if RST_CAS = '1' then
      count_AddrSM <= (others => '0');
      --count_AddrSM_end <= '0';
    elsif (count_AddrSM = "011111111" and EN_CAS = '1') then
      count_AddrSM <= (others => '0');
      --count_AddrSM_end <= '1';
    elsif EN_CAS = '1' then
      count_AddrSM <= count_AddrSM + 1;
      --count_AddrSM_end <= '0';
    end if;
  end if;
end process count_AddrSM_proc;

---------------------------------------------------------------

--MEMORIES CONTROL MUXes

WMem_Addr <= bram16k_addr when instrackF = '0' else count_AddrWM;
WMem_Din <= bram16k_din(15 downto 0) when instrackF = '0' else (others => '0');
WMem_En <= bram16k_en when instrackF = '0' else EN_WMR;
WMem_We <= (others=>bram16k_we) when instrackF = '0' else (others=>'0');
bram16k_dout <="0000000000000000" & WMem_Dout;

smem_connect_inst: for i in 1 to integer(ceil(real(NANT)/real(2))) generate
  SMem_Addr(i) <= bram2k_addr (8 downto 0) when dataackM = '0' else count_AddrSM;
  SMem_Din(i) <= (bram2k_din0 & bram2k_din1 & bram2k_din2 & bram2k_din3) when dataackM = '0'
else (others=>'0');
  SMem_En(i) <= '1' when (bram2k_en = '1' and dataackM = '0' and bram2k_addr(14 downto 10) = i-1
  ) else
    '0' when dataackM = '0' else
    EN_SM_R;
```


SMem_We(i) <= (others=>"1") when bram2k_we = '1' and dataackM = '0' and bram2k_addr(14 downto 10) = i-1 else "0";
end generate;

Samples_read <= SMEM_Dout(to_integer(unsigned(bram2k_addr(14 downto 10)))+1) when bram2k_en = '1' else (others=>"1"); samples memory read via usb is not enabled

EM_Addra <= bram2k_addr (9 downto 0) when dataackM = '0' else count_iter&'0';
EM_Ema <= enmem_en when dataackM = '0' else EN_EM_R;
EM_Wea <= (others=>WE_EM_R) when dataackM = '1' else (others=>bram2k_we);
EM_Dina <= energy_write_val(64-1 downto 32);
EM_Addrb <= bram2k_addr (9 downto 0) when dataackM = '0' else count_iter&'1';
EM_Emb <= '0' when dataackM = '0' else EN_EM_R;
EM_Web <= (others=>WE_EM_R) when dataackM = '1' else (others=>bram2k_we);
EM_Dinb <= energy_write_val(32-1 downto 0);

leds<=(others=>"1") when bram16k_addr = "0000000000000000" else (others=>"0");

-- CONTROL REGISTERS SET
-- dcm ctrls in mapped to address 0x00000080
ctrl_regs_set(0) <= X"00000000" & drdy_dcm & locked_dcm;
-- dcm output mapped to address 0x00000084
ctrl_regs_set(1) <= X"000000" & do_dcm;
-- mips output mapped to address 0x00000088
ctrl_regs_set(2) <= do_mips;
-- switch value mapped to address 0x0000008C
ctrl_regs_set(3) <= X"000000" & switch;
-- timer mapper to address 0x00000090
ctrl_regs_set(4) <= X"000000" & "000" & zf_timer;

-- leds mapped to address 0x00000000
-- leds <= register_set(0)(7 downto 0);

-- dcm ctrls mapped to address 0x00000084
rst_dcm <= register_set(1)(0);
den_dcm <= register_set(1)(2 downto 1);
dwe_dcm <= register_set(1)(4 downto 3);
daddr_dcm <= register_set(1)(14 downto 8);
-- dcm data mapped to address 0x00000088
di_dcm <= register_set(2)(15 downto 0);
Appendix A (continued)

```vhdl
-- reset mips mapped to address 0x000000C
rst_mips <= register_set(3);
-- address of mips mapped to address 0x0000010
adr_mips <= register_set(4);
-- timer start and value mapped to address 0x00000014
init_timer <= register_set(5);
end Behavioral;
```

Listing A.9. Delay and Mist instantiation

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.declaration_pack.all;
use work.pkg_delay.all;

tenity DelBEAM is
port (clk, rst: in std_logic;
```

---

Company: Polito
---

Engineer: Paolo Rosingana, Francesco Colonna
---

Create Date: 20/08/2012
---

Design Name:
---

Module Name: DelBEAM — Behavioral
---

Project Name:
---

Target Devices:
---

Tool versions:
---

Description:
---

Dependencies:
---

Revision:
---

Revision 1.0 — final release
---

Additional Comments:
---

---

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.declaration_pack.all;
use work.pkg_delay.all;

tenity DelBEAM is
port (clk, rst: in std_logic;
```
Appendix A (continued)

```vhdl
samples_in: in array_vector_in;
q: in fir_input;
delay_val: in array_vector_delay;
start: in std_logic;
last_fir_loop: in std_logic;
loop_rst: in std_logic;
finish_mist: out std_logic;
n_low, n_high: in std_logic_vector(n FIR length-1 downto 0);
energy_out: out STD_LOGIC_VECTOR (total_word_length_out-1 downto 0)
); end DelBEAM;

architecture Behavioral of DelBEAM is

component Mist is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         In_signal: array_vector_in;
         q : in fir_input;
         Start : in STD_LOGIC;
         finish : out STD_LOGIC;
         last_fir_loop: in std_logic;
         loop_rst: in std_logic;
         n_low, n_high: in std_logic_vector(n FIR length-1 downto 0);
         Total_energy_out: out STD_LOGIC_VECTOR (total_word_length_out-1 downto 0));
end component;

component top_par_n_ant is
  port ( clock, reset : in std_logic;
         samples_in: in array_vector_in;
         samples_out: out array_vector_in;
         ritardi_in: in array_vector_delay;
         start: out std_logic_vector(N FIR-1 downto 0));
end component;

signal samples_del: array_vector_in;
signal energy: std_logic_vector(total_word_length_out-1 downto 0);
signal start_DP: std_logic_vector(N FIR-1 downto 0);
```
Appendix A (continued)

```vhdl
signal delayed_data_ready, start_and_ready : std_logic;

begin
  energy_out <= energy;

  Mist_inst: Mist port map (  
    clk => clk,  
    rst => rst,  
    ln_signal => samples_del,  
    q => q,  
    Start => start,  
    last_fir_loop => last_fir_loop,  
    loop_rst => loop_rst,  
    finish => finish_mist,  
    n_low => n_low, n_high => n_high,  
    Total_energy_out => energy  
  );

  delayed_data_ready <= '1' when start_DP = (NFIR-1 downto 0 => '1')  
    -- added for debug, no real need
  else '0';  
    -- added for debug
  , no real need

  Delay_inst: top_par_n_ant port map (  
    clock => clk,  
    reset => rst,  
    samples_in => samples_in,  
    samples_out => samples_del,  
    ritardi_in => delay_val,  
    start => start_DP  
  );

end Behavioral;
```

A.2 Delay block

Listing A.10. Delay declaration package
library IEEE;
use IEEE.std_logic_1164.all;
use ieee.numeric_std.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.declaration_pack.all;

Package pkg_delay is

--////////////////////////////////////////////////////////////////////---
--Variable--
--////////////////////////////////////////////////////////////////////---
constant ncount : integer := 4; -- log2(n_in_reg)
constant nb: integer := 16;
constant pdelay: integer := 8; -- log2(na)
constant na: integer := 256;
constant par: integer := 1;
constant n_in_reg: integer:= nbit/par;
constant nsamples : integer := 200;  -- numero campioni di ingresso per ogni voxel
constant module : integer := 8;
constant n_in_reg_bin : std_logic_vector := conv_std_logic_vector(n_in_reg-1, ncount);
constant cnt3 : integer := 9;  -- contatore per i campioni

type arrayvectordelay is array (NFIR-1 downto 0) of std_logic_vector(pdelay-1 downto 0);
--////////////////////////////////////////////////////////////////////---
--COMPONENT--
--////////////////////////////////////////////////////////////////////---
component cell_shift_register is
port (  
shift : in std_logic;
load: in std_logic;
clk, rst: in std_logic;
ld_sh : in std_logic;  -- shift attivo basso
out_cell: out std_logic);
end component;

component register_nbit is
generic( N: integer := nbit-1);
port (  
d : in std_logic_vector(N-1 downto 0);
Appendix A (continued)

```vhdl
clk, rst : in std_logic;
q : out std_logic_vector(N-1 downto 0);
end component;

component dec_shift is
port (  
   input, clk, rst : in std_logic;
   output : out std_logic;
   delay : in std_logic_vector(pdelay-1 downto 0)
);
end component;

component register_1bit is
port (  
   d : in std_logic;
   clk, rst : in std_logic;
   q : out std_logic;
);
end component;

component mux is
  port (    
   in_shift : in std_logic;
   in_load : in std_logic;
   load_shift : in std_logic;
   out_mux : out std_logic;
);
end component;

component Decoder_gen is
  Port (    
   input : in STD_LOGIC_VECTOR (pdelay-1 downto 0);
   output : out STD_LOGIC_VECTOR ((2**pdelay)-1 downto 0)
    );
end component;

component shift_register is
port (  
   shift : in std_logic;
   load : in std_logic;
   clk, rst : in std_logic;
);
Appendix A (continued)

```verilog
component orn IS
PORT (x : IN STD_LOGIC_vector(ncount-1 downto 0);
f : OUT STD_LOGIC);
END component orn;

component norn IS
PORT (x : IN STD_LOGIC_vector(ncount-1 downto 0);
f : OUT STD_LOGIC);
END component norn;

component counter is
generic (N : integer := module);
port (
en : in std_logic;
clk : in std_logic;
rst : in std_logic;
output_count : out std_logic_vector(N-1 downto 0)
)
end component;

component counter_gen is
generic (N : integer := module);
port (en : in std_logic;
clk : in std_logic;
rst : in std_logic;
output_count : out std_logic_vector(N-1 downto 0)
)
end component;

component start_block is
port (del : in std_logic_vector(pdelay-1 downto 0);
clk : in std_logic;
rst : in std_logic;
)
end component;
```
Appendix A (continued)

```vhdl
start : out std_logic);
end component;

−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
component delay_fsm
port (  
  clock, reset_fsm : in std_logic;
  count1 : in std_logic_vector(ncount-1 downto 0);
  count2 : in std_logic_vector(pdelay downto 0);
  count3 : in std_logic_vector(cnt3-1 downto 0);
  rst_count1, rst_count2, rst_count3 : out std_logic;
  busy, start, reset, load_in, on_count3 : out std_logic;
  delay_in : in std_logic_vector(pdelay-1 downto 0);
  delay_out : out std_logic_vector(pdelay-1 downto 0));
end component;
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
component delay_hybr
  Port
  (  
    input : in STD_LOGIC_VECTOR (nbit-1 downto 0);
    output : out STD_LOGIC_VECTOR (par-1 downto 0);
    delay : in std_logic_vector(pdelay-1 downto 0);
    load_sh : in std_logic;
    clk, rst : in std_logic
  );
end component;
−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−−
component delay_par_fsm is
  port (  
    clock, reset_fsm : in std_logic;
    count2 : in std_logic_vector(pdelay downto 0);
    count3 : in std_logic_vector(cnt3-1 downto 0);
  );
end component;
```
Appendix A (continued)

```vhdl
component delay_par is

Port
(
    input : in STD_LOGIC_VECTOR (nbit-1 downto 0);
    output : out STD_LOGIC_VECTOR (nbit-1 downto 0);
    delay : in std_logic_vector(pdelay-1 downto 0);
    clk, rst : in std_logic
);
end component;
```

```vhdl
component top_par

port ( 
    clock, reset : in std_logic;
    samples_in : in std_logic_vector(nbit-1 downto 0);
    samples_out : out std_logic_vector(nbit-1 downto 0);
    ritardi_in : in std_logic_vector(pdelay-1 downto 0);
    start : out std_logic);
end component;
```

```vhdl
component top_hybr

port ( 
    clock, reset : in std_logic;
    samples_in : in std_logic_vector(nbit-1 downto 0);
    samples_out : out std_logic_vector(par-1 downto 0);
    ritardi_in : in std_logic_vector(pdelay-1 downto 0);
    start, busy : out std_logic);
```

```vhdl
end component;
```
Appendix A (continued)

end component;
end package pkg_delay;

--Package body pkg_delay is
--
--end pkg_delay;

Listing A.11. N antennas Delay blocks instantiation

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.pkg_delay.all;
use work.declaration_pack.all;

entity top_par_n_ant is

port ( 
  clock, reset : in std_logic;
  samples_in: in arrayvectorin;
  samples_out: out arrayvectorin;
  ritardi_in: in arrayvectordelay;
  start: out std_logic_vector(NFIR-1 downto 0));
end top_par_n_ant;

architecture top_par_arch_n_ant of top_par_n_ant is

begin -- top_hybr_arch

inst_top_n_ant: for i in NFIR-1 downto 0 generate
begin
  inst.top_par_del: top_par
    port map( clock => clock, reset => reset, samples_in => samples_in(i), samples_out =>
     samples_out(i), ritardi_in => ritardi_in(i), start => start(i));
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.pkg_delay.all;

entity top_par is
  port(
    clock, reset : in std_logic;
    samples_in: in std_logic_vector(nbit-1 downto 0);
    samples_out: out std_logic_vector(nbit-1 downto 0);
    ritardi_in: in std_logic_vector(pdelay-1 downto 0);
    start: out std_logic);
end top_par;

architecture top_par_arch of top_par is
  signal int_delay: std_logic_vector (pdelay-1 downto 0);
  signal int_en3, int_reset, int_rst_count2, int_rst_count3 : std_logic;
  signal int_count2 : std_logic_vector(pdelay downto 0);
  signal int_count3 : std_logic_vector(cnt3-1 downto 0);

begin
  -- top_hybr_arch

del_hyb: delay_par port map (input => samples_in, delay => int_delay, clk => clock, rst => int_reset, output => samples_out);

del_fsm: delay_par_fsm port map (clock => clock, reset_fsm => reset, count2 => int_count2, count3 => int_count3,
  delay_in => ritardi_in, rst_count2 => int_rst_count2, rst_count3 => int_rst_count3,
  start => start, reset => int_reset, delay_out => int_delay, en_count3=> int_en3);

Listing A.13. Single Delay block top entity

library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.pkg_delay.all;

entity top_par is
  port (
    clock, reset : in std_logic;
    samples_in: in std_logic_vector(nbit-1 downto 0);
    samples_out: out std_logic_vector(nbit-1 downto 0);
    ritaldi_in: in std_logic_vector(pdelay-1 downto 0);
    start: out std_logic);
end top_par;

architecture top_par_arch of top_par is
  signal int_delay : std_logic_vector (pdelay-1 downto 0);
signal int_en3, int_reset, int_rst_count2, int_rst_count3 : std_logic;
signal int_count2 : std_logic_vector(pdelay downto 0);
signal int_count3 : std_logic_vector(cnt3-1 downto 0);
begin
end top_par_arch;
Appendix A (continued)

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.pkg_delay.all;

entity delay_par is
  Port
  (
    input : in STD_LOGIC_VECTOR (nbit−1 downto 0);
    output : out STD_LOGIC_VECTOR (nbit−1 downto 0);
    delay : in std_logic_vector(pdelay−1 downto 0);
    clk, rst : in std_logic
  );
end delay_par;

architecture delay_par_arch of delay_par is
  signal reg_nbit: std_logic_vector(nbit−1 downto 0);
```
Appendix A (continued)

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use ieee.numeric_std.all;
use work.pkg_delay.all;

entity delay_par_fsm is
  port (count2 : in std_logic_vector(pdelay downto 0);
        count3 : in std_logic_vector(cnt3−1 downto 0);
        en_count3 : out std_logic;
        rst_count2, rst_count3 : out std_logic;
        start, reset : out std_logic;
        delay_in : in std_logic_vector(pdelay−1 downto 0);
        delay_out : out std_logic_vector(pdelay−1 downto 0))
end delay_par_fsm;

architecture delay_par_fsm_arch of delay_par_fsm is
```

```vhdl
signal dec2shift : std_logic_vector(na−1 downto 0);

begin

  inst_delay : for i in 0 to nbit−1 generate
    instance_shift : shift_register port map(load=>reg_nbit(i), shift => reg_nbit(i), id_sh=>dec2shift,
                                             out_shift=>output(i), rst=>rst, clk=>clk);
  end generate;

  instance_dec : decoder_gen port map(input => delay, output=> dec2shift);
  instance_regnbit : register_nbit generic map(nbit)
  port map(d =>input, q=>reg_nbit, rst=>rst, clk=>clk);

end delay_par_arch;
```

Listing A.15. Delay fsm
Appendix A (continued)

```vhdl

type status is (state0, load, end_voxel);
signal presentstate, nextstate : status; -- stati per gestire lo slave
signal temp_count2 : std_logic := '0';
signal count2_start : std_logic_vector(pdelay downto 0);
begin

nextst : process (clock, reset_fsm)
begin -- process nstate
  if reset_fsm = '1' then -- asynchronous reset
    presentstate <= state0;
  elsif clock'event and clock='1' then -- rising clock edge
    presentstate <= nextstate;
  end if;
end process nextst;

fsm : process (presentstate, count3)
begin
  case presentstate is
    when state0 =>
      nextstate <= load;
    when load =>
      if count3 = conv_std_logic_vector(nsamples,8) then
        nextstate <= end_voxel;
      else
        nextstate <= load;
      end if;
    when end_voxel =>
      nextstate <= load;
    when others => null;
  end case;
end process fsm;
```
Appendix A (continued)

```vhdl
library ieee;

end delay_par_fsm_arch;

Listing A.16. Delay decoder

delay_out <= delay_in;
out fsm : process (presentstate, count2)
begin

case presentstate is

  when state0 =>
      start <= '0';
      reset <= '1';
      rst_count2 <= '1';
      rst_count3 <= '1';
      en_count3 <= '0';

  when load =>
      reset <= '0';
      rst_count2 <= '0';
      rst_count3 <= '0';
      count2_start <= temp_count2 & delay_in;
      if count2 = count2_start + 1 then
        start <= '1';
      end if;
      en_count3 <= '1';

  when end_voxel =>
      reset <= '1';
      start <= '0';
      rst_count2 <= '1';
      rst_count3 <= '1';
      en_count3 <= '0';

  when others => null;

end case;
end process out fsm;
```

end delay_par_fsm_arch;

library ieee;
Appendix A (continued)

use ieee.std_logic_1164.all;
use ieee.numeric_std.all;
use work.pkg_delay.all;

entity Decoder_gen is
  Port
  (   input : in STD_LOGIC_VECTOR (pdelay−1 downto 0);
      output : out STD_LOGIC_VECTOR ((2**pdelay)−1 downto 0)
  );
end Decoder_gen;

architecture decoder_arch of Decoder_gen is
begin

DECODE_PROC:
  process (input)
  VARIABLE out_var : std_logic_vector((2**pdelay)−1 DOWNTO 0);
  begin
    out_var := (OTHERS => '0');
    out_var(to_integer(unsigned(input))) := '1';
    output <= out_var;
  end process;
end decoder_arch;

---library ieee;
---use ieee.std_logic_1164.all;
---use ieee.std_logic_arith.all;
---
Appendix A (continued)

Listing A.17. Delay counter

```
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;
use work.pkg_delay.all;

entity counter_gen is
  generic ( 
    N : integer := module );
  -- entity decoder_gen is
  -- Generic( na: integer := 2 );
  -- Port
  -- ( 
  --   input : in STD_LOGIC_VECTOR (na-1 downto 0);
  --   output : out STD_LOGIC_VECTOR ((2**na)-1 downto 0)
  --   -- clk : in STD_LOGIC;
  --   -- rst : in STD_LOGIC
  --   );
  -- end decoder_gen;
  --
  -- architecture decoder_arch of decoder_gen is
  --
  -- begin
  --
  -- process (input)
  -- begin
  -- case (input) is
  --   dec: for i in 0 to (2**na)-1 generate
  --     begin
  --       when (i = conv.integer(input)) => output <= conv.std_logic_vector(i*2), output 'length);
  --     end generate dec;
  --   when others => output <= conv.std_logic_vector(0, output 'length);
  --   end case;
  -- end process;
  --
  -- end decoder_arch;
```
Listing A.18. Shift register

library IEEE;
use IEEE.std_logic_1164.all;
use work.pkg_delay.all;

entity shift_register is
  port (  
    shift : in std_logic;
  );
Appendix A (continued)

```vhdl
library IEEE;
use IEEE.std_logic_1164.all;
use work.pkg_delay.all;

entity cell_shift_register is

load : in std_logic;
clk, rst : in std_logic;
ld_sh : in std_logic_vector (na-1 downto 0); -- shift active-low
out_shift_reg : out std_logic);
end shift_register;

architecture shift_register_strct of shift_register is

signal cellout2shift : std_logic_vector (na-2 downto 0);

-- signal internal : std_logic;

begin

inst_shift_reg : for i in 0 to na-1 generate

begin

cell0 : if i=0 generate

c_0 : cell_shift_register port map (shift=>cellout2shift, load=>load, clk=>clk, rst=>rst, ld_sh=>ld_sh(na-i-1), out_cell=>cellout2shift (na-2-i));
end generate cell0;

cell_i : if i>0 and i<na-1 generate

c_i : cell_shift_register port map (shift=>cellout2shift (na-i-1), ld_sh=>ld_sh(na-i-1), clk=>clk, rst=>rst, load=>load, out_cell=>cellout2shift (na-2-i));
end generate cell_i;

cell_na : if i=na-1 generate

c_na : cell_shift_register port map (shift=>cellout2shift (na-1-1), load=>load, clk=>clk, rst=>rst,
ld_sh=>ld_sh(na-1-1), out_cell=>out_shift_reg);
end generate cell_na;
end generate inst_shift_reg;
end shift_register_strct;
```

Listing A.19. Shift register single cell
port (  
       shift: in std_logic;
       load: in std_logic;
       clk, rst: in std_logic;
       ld_sh: in std_logic; -- shift active-low
       out_cell: out std_logic);

end cell_shift_register;

architecture cell_shift_register_struct of cell_shift_register is
signal internal: std_logic;

begin -- shift_register_struct

regist: register_1bit
port map (d => internal, clk => clk, rst => rst, q => out_cell);

multiplexer: mux
port map (  
           in_shift => shift, in_load => load, load_shift =>ld_sh, out_mux => internal);

end cell_shift_register_struct;

library ieee;
use ieee.std_logic_1164.all;
use work.pkg_delay.all;

entity register_nbit is
generic( N: integer := nbit-1);
port (  
       d : in std_logic_vector(N-1 downto 0);
       clk, rst: in std_logic;
       q : out std_logic_vector(N-1 downto 0));
end register_nbit;
appendix a (continued)

architecture register_nbit_arch of register_nbit is

signal data : std_logic_vector(N downto 0);

begin

p1 : process (clk, rst)
begin

if rst = '1' then  -- asynchronous reset (active high)
data <= (others => '0');

elsif clk'event and clk = '1' then

data <= d;

end if;
end process p1;

q <= data;

end register_nbit_arch;

library ieee;
use ieee.std_logic_1164.all;

entity register_1bit is

port ( 
d : in std_logic;
clk, rst : in std_logic;
q : out std_logic);
end register_1bit;

architecture reg.beh_1bit of register_1bit is

signal data : std_logic;

begin  -- reg.beh

p1 : process (clk, rst)
begin  -- process p1

if rst = '1' then  -- asynchronous reset (active high)

end process p1;

end assume_1bit_arch;

Listing A.21. 1 bit register
Appendix A (continued)

```vhdl
data <= '0';
else
    if clk'event and clk = '1' then -- rising clock edge
data <= d;
    end if;
end process p1;

q <= data;
end reg_beh_1bit;
```

Listing A.22. 1 bit Multiplexer

```vhdl
library ieee;
use ieee.std_logic_1164.all;
use ieee.std_logic_arith.all;
use ieee.std_logic_unsigned.all;

entity mux is

    port(
in_shift : in std_logic;
in_load: in std_logic;
load_shift: in std_logic;

out_mux: out std_logic);
end mux;

architecture muxm of mux is
begin
mx: process (in_shift, in_load, load_shift)
begin

case load_shift is

when '1' =>
    out_mux <= in_load;
when '0' =>
    out_mux <= in_shift;
when others => null;
end case;
```

```
A.3 Mist Beamforming

Listing A.23. Mist top entity

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_SIGNED.ALL;
use work.declaration_pack.all;
use IEEE.NUMERIC_STD.ALL;
use ieee.math_real.all;

entity Mist is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         In_signal : array vector in;
         q : in FIRINPUT;
         Start : in STD_LOGIC;
         last_fir_loop : in std_logic;
         loop_rst : in std_logic;
         finish : out STD_LOGIC;
         write_fir1 : out std_logic;
         n_low, n_high : in std_logic_vector(nlow.nhigh_length-1 downto 0);
         Total_energy_out : out STD_LOGIC_VECTOR (total_word_length_out-1 downto 0));
  end Mist;

architecture Behavioral of Mist is

COMPONENT fir

PORT(
  sample_in : IN std_logic_vector(word_length-1 downto 0);
  q : in arrayvector55;
  clk : IN std_logic;
  rst : IN std_logic;
  sample_out : OUT std_logic_vector(double_word_length-1 downto 0)

```
Appendix A (continued)

```vhdl
COMPONENT Calcolo_Energia
PORT(
    clk, rst : IN std_logic;
    go : IN std_logic;
    d_in : IN std_logic_vector(double_word_length-1 downto 0);
    d_out : OUT std_logic_vector(total_word_length_out-1 downto 0);
);
END COMPONENT;

COMPONENT adder
Generic (Nbit : integer := 16);
PORT ( a : in STD_LOGIC_VECTOR (Nbit-1 downto 0);
       b : in STD_LOGIC_VECTOR (Nbit-1 downto 0);
       clk : in std_logic;
       c : out STD_LOGIC_VECTOR (Nbit-1 downto 0));
END COMPONENT;

COMPONENT window_1
PORT(
    datain : IN std_logic_vector(word_length-1 downto 0);
    dataout : OUT std_logic_vector(word_length-1 downto 0);
    clk : IN std_logic;
    go_filter : in std_logic
);
END COMPONENT;

COMPONENT control_start
PORT(
    clk : IN std_logic;
    rst : in STD_LOGIC;
    start : IN std_logic;
    na : IN std_logic_vector(na_length-1 downto 0);
    go_filter : OUT std_logic
);
END COMPONENT;

COMPONENT ffd1
```

PORT(
    a : IN std_logic;
    clk : IN std_logic;
    o : OUT std_logic
);
END COMPONENT;

COMPONENT window_2
PORT(
    datain : IN std_logic_vector(double_word_length-1 downto 0);
    clk : IN std_logic;
    rst : in STD_LOGIC;
    go : IN std_logic;
    go_loop : in std_logic;
    n_low : IN std_logic_vector(nlow_nhigh_length-1 downto 0);
    n_high : IN std_logic_vector(nlow_nhigh_length-1 downto 0);
    dataout : OUT std_logic_vector(double_word_length-1 downto 0);
    finish : out std_logic;
    calcola : OUT std_logic;
    loop_enable : out std_logic
);
END COMPONENT;

COMPONENT shift_register_generic
    generic (num_bit : integer := 16;
               num_stage : integer := 3);
port ( datain: in std_logic_vector(num_bit=1 downto 0);
       clk , rst , enable: in std_logic;
       dataout: out std_logic_vector(num_bit=1 downto 0))
end COMPONENT;

COMPONENT register_par
    Generic (bit_size : integer :=16);
    Port ( din : in STD_LOGIC_VECTOR (bit_size-1 downto 0);
           clk : in std_logic;
           rst : in std_logic;
           enable : in std_logic;
           dout : out STD_LOGIC_VECTOR (bit_size-1 downto 0))
end COMPONENT;
signal Window_1_in : array vector in;
signal filter_in : array vector in;
signal filter_out : array vector in x 2;
signal na : std_logic vector (na.length - 1 downto 0) := "011000111";
signal window_2_in, window_2_out : std_logic vector (double_word_length - 1 downto 0);
signal Calcolo_Energia_in : std_logic vector (double_word_length - 1 downto 0);

begin

Window_1_in <= In_signal;

Instance_window_1 : for i in 0 to NFIR - 1 generate
  Inst_window_1 : window_1 PORT MAP(
    a => go(i),
    clk => clk,
    o => go(i + 1)
  );
end generate Instance_window_1;

Instance_calcolo : for i in 0 to NFIR - 1 generate
  Inst_calcolo : Calcolo_Energia PORT MAP(
    clk => clk,
    rst => rst,
    start => start,
    na => na,
    go_filter => go(0)
  );
Appendix A (continued)

    datain => Window_1_in(i),
    dataout => filter_in(i),
    clk => clk,
    go_filter => go(0)
)
end generate Instance_window_1;

Instance_fir_array: for i in 0 to NFIR-1 generate
    Inst_fir: fir PORT MAP(
        sample_in => filter_in(i),
        q=>q(i+1),
        clk => clk,
        rst => rst,
        sample_out => filter_out(i)
    );
end generate Instance_fir_array;

-- link FIR outputs to tree adder input
Inst_tree_inputs: for i in 0 to NFIR-1 generate
    adder_connections(1)(i) <= filter_out(i);
end generate Inst_tree_inputs;
adder_connections(1)(NFIR)<= loop_register_out when loop_enable = '1' else (OTHERS => '0');

-- instantiate tree adder
Inst_adder_tree: for j in 1 to integer(ceil(log2(real(NFIR+1))))-1 generate
    Inst_adder_level: for l in 0 to integer(floor(real(NFIR+1)/real(2**j))-real(1)) generate
        Inst_adder_leaf: adder
            GENERIC MAP (Nbit => double_word_length)
            PORT MAP(
                a => adder_connections(j)(2*l),
                b => adder_connections(j)(2*l+1),
                clk => clk,
                c => adder_connections(j+1)(1)
            );
    end generate Inst_adder_level;
Inst_add_retard: if not(integer(ceil(real(NFIR+1)/real(2**(j-1)))) mod 2) = 0) generate
    Inst_tree_reg: register_par Generic map (bit_size=>double_word_length)
    Port map (
Appendix A (continued)

\[
din \rightarrow \text{adder}\_\text{connections}(j)(\text{integer}(\text{floor}(\text{real}(\text{NFIR+1})/\text{real}(2^j)))) \\
\text{clk} \rightarrow \text{clk} \\
\text{rst} \rightarrow \text{rst} \\
\text{enable} \rightarrow '1' \\
dout \rightarrow \text{adder}\_\text{connections}(j+1)(\text{integer}(\text{floor}(\text{real}(\text{NFIR+1})/\text{real}(2^j))))
\]

end generate Inst_add_retard;

Inst_add_void: for \(l\) in integer(\text{floor}(\text{real}(\text{NFIR+1})/\text{real}(2^j)))+1(1) to NFIR generate

\[
\text{adder}\_\text{connections}(j+1)(1) \leftarrow (\text{OTHERS} \Rightarrow '0')
\]

end generate Inst_add_void;

end generate Inst_adder_tree;

Inst_adder_tree_top: adder

\begin{align*}
\text{GENERIC MAP} & (\text{Nbit} \Rightarrow \text{double}\_\text{word}\_\text{length}) \\
\text{PORT MAP} & ( \\
& a \Rightarrow \text{adder}\_\text{connections}(\text{integer}(\text{ceil}(\text{log2}(\text{real}(\text{NFIR+1})))))0) \\
& b \Rightarrow \text{adder}\_\text{connections}(\text{integer}(\text{ceil}(\text{log2}(\text{real}(\text{NFIR+1})))))1) \\
& \text{clk} \Rightarrow \text{clk} \\
& c \Rightarrow \text{adder}\_\text{connections}(\text{integer}(\text{ceil}(\text{log2}(\text{real}(\text{NFIR+1}))))+1)0)
\end{align*}

end generate Inst_adder_tree_top;

Inst_adder_tree_top: for \(l\) in 1 to NFIR generate

\[
\text{adder}\_\text{connections}(\text{integer}(\text{ceil}(\text{log2}(\text{real}(\text{NFIR+1}))))+1)1(1) \leftarrow (\text{OTHERS} \Rightarrow '0')
\]

end generate Inst_adder_tree_top;

window_2\_in \leftarrow \text{adder}\_\text{connections}(\text{integer}(\text{ceil}(\text{log2}(\text{real}(\text{NFIR+1}))))+1)0);

Inst_window_2: window_2 \text{PORT MAP}(

\begin{align*}
datain & \Rightarrow \text{window}_2\_\text{in} \\
dataout & \Rightarrow \text{window}_2\_\text{out} \\
\text{clk} & \Rightarrow \text{clk} \\
\text{rst} & \Rightarrow \text{rst} \\
go & \Rightarrow \text{go}(1+7+\text{integer}(\text{ceil}(\text{log2}(\text{real}(\text{NFIR+1}))))+1) \\
go\_\text{loop} & \Rightarrow \text{write}\_\text{fir} \\
n\_\text{low} & \Rightarrow n\_\text{low} \\
n\_\text{high} & \Rightarrow n\_\text{high} \\
\text{finish} & \Rightarrow \text{finish} \\
\text{calc}\_\text{ola} & \Rightarrow \text{calc}\_\text{ola} \\
\text{loop}\_\text{enable} & \Rightarrow \text{loop}\_\text{enable}
\end{align*}

)
Listing A.24. Control Start block

```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.declaration_pack.all;

entity control_start is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         start : in STD_LOGIC;
         
```
Appendix A (continued)

na : in  STD_LOGIC_VECTOR (na_length−1 downto 0);
go_filter : out  STD_LOGIC);
end control_start;

architecture Behavioral of control_start is
signal count : unsigned(na_length−1 downto 0) := (others => '0');
signal do : std_logic := '0';
begin
  process (start,count,rst) is
  begin
    if (count = to_unsigned(401,na_length)) or (rst = '1') then
do <= '0';
elseif rst='0' and start = '1' then
do <= '1';
  end if;
end process;
  process(clk,na,rst) is
  begin
    if rst = '1' then
count <= (others => '0');
go_filter <= '0';
else
  if clk 'event and clk = '1' then
    if (do = '1') then
      count <= count + 1;
      if (count < unsigned(na)) then --put to zero all samples
        before na
      go_filter <= '0';
    else
      go_filter <= '1';
    end if;
    if (count = to_unsigned(401,na_length)) then -- last sample
      401 is "random"
count <= (others => '0');
    end if;
else
      go_filter <= '0';
  end if;
end if;
end process;
Appendix A (continued)

Listing A.25. Window 1

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.declaration_pack.all;

entity window_1 is
  Port ( datain : in STD_LOGIC_VECTOR (word_length-1 downto 0);
         dataout : out STD_LOGIC_VECTOR (word_length-1 downto 0);
         clk : in STD_LOGIC;
         go_filter : in STD_LOGIC );
end window_1;

architecture Behavioral of window_1 is
begin
  process(clk, datain, go_filter) is
  begin
    if clk'event and clk = '1' then
      if go_filter = '0' then
        dataout <= (others => '0');
      else
        dataout <= datain;
      end if;
    end if;
  end process;
end Behavioral;
```
Appendix A (continued)

Listing A.26. Control Start block

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_STD.ALL;
use work.declaration_pack.all;

entity control_start is
  Port ( clk : in STD_LOGIC;
         rst : in STD_LOGIC;
         start : in STD_LOGIC;
         na : in STD_LOGIC_VECTOR (na_length-1 downto 0);
         go_filter : out STD_LOGIC);
end control_start;

architecture Behavioral of control_start is
signal count : unsigned (na_length-1 downto 0) := (others => '0');
signal do : std_logic := '0';

begin
  process (start, count, rst) is
  begin
    if (count = to_unsigned(401, na_length)) or (rst = '1') then
      do <= '0';
    elsif rst='0' and start = '1' then
      do <= '1';
    end if;
  end process;

  process(clk, na, rst) is
  begin
    if rst = '1' then
      count <= (others => '0');
      go_filter <= '0';
    else
      if clk'event and clk = '1' then
        if (do = '1') then
          count <= count + 1;
          if (count < unsigned(na)) then
            -- put to zero all samples before na
```
Appendix A (continued)

```vhdl
appendix_a_code
```

Listing A.27. FIR filter

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.Numeric_STD.ALL;
use work.declaration_pack.all;

entity fir is
  Port ( sample_in : in STD_LOGIC_VECTOR (word_length-1 downto 0);
          q : in arrayvector55;
          clk : in STD_LOGIC;
          rst : in STD_LOGIC;
          sample_out : out STD_LOGIC_VECTOR (double_word_length-1 downto 0));
end fir;

architecture Behavioral of fir is

COMPONENT adder
  GENERIC (Nbit : integer := 32);
  PORT(
    a : IN std_logic_vector(Nbit-1 downto 0);
    b : IN std_logic_vector(Nbit-1 downto 0);
```
Appendix A (continued)

```vhdl
clk : IN std_logic;
c : OUT std_logic_vector(Nbit-1 downto 0)
);
END COMPONENT;

COMPONENT mult_half
PORT(
    a : IN std_logic_vector(word_length-1 downto 0);
b : IN std_logic_vector(word_length-1 downto 0);
c : OUT std_logic_vector(word_length-1 downto 0)
);
END COMPONENT;

COMPONENT register_par
Generic (bit_size : integer :=16);
Port (din : in STD_LOGIC_VECTOR (bit_size-1 downto 0);
    clk : IN std_logic;
    rst : in std_logic;
    enable : in std_logic;
    dout : OUT std_logic_vector(bit_size-1 downto 0)
);
END COMPONENT;

signal data_shift_reg : arrayvector55;
signal coefficient : arrayvector55;
signal mul_out_array : arrayvector55x2;
— signal mul_out_expanded : arrayvector55x2;
signal sample_out_in : STD_LOGIC_VECTOR (double_word_length-1 downto 0);
— first level of sum tree
signal data_sum_add32 : arrayvector32x2;
— first level of sum tree
signal data_sum_add16 : arrayvector16x2;
— second level of sum tree
signal data_sum_add8 : arrayvector8x2;
— third level of sum tree
signal data_sum_add4 : arrayvector4x2;
```
Appendix A (continued)

--- fourth level of sum tree

signal data_sum_add2 : array vector 2x2;

begin

coefficient <= q;

process (clk, sample_in) is
begin
if clk'event and clk = '1' then
  data_shift_reg(0) <= sample_in;
end if;
end process;

--- shift register

Inst_shiftreg: for i in 0 to 53 generate
  Inst_register_par: register_par GENERIC MAP (bit_size => word_length)
  PORT MAP(
    din => data_shift_reg(i),
    clk => clk,
    rst => rst,
    enable => '1',
    dout => data_shift_reg(i+1)
  );
end generate Inst_shiftreg;

--- multipliers

Inst_mult: for i in 0 to 54 generate
  Inst_mult_half: mult_half PORT MAP(
    a => data_shift_reg(i),
    b => coefficient(i),
    c => mul_out_array(i)(word_length-1 downto 0)
  );
  mul_out_array(i)(double_word_length-1 downto word_length) <= (double_word_length-1 downto word_length => mul_out_array(i)(word_length-1));
end generate Inst_mult;
Appendix A (continued)

```
--first level
Inst_first_level_add: for i in 0 to 26 generate
    Inst_adder16: adder GENERIC MAP (Nbit => double_word_length)
       PORT MAP(
           a => mul_out_array(2*i),
           b => mul_out_array(2*i+1),
           clk => clk,
           c => data_sum_add32(i)
       );
end generate Inst_first_level_add;

Inst_first_level_tail: register_par GENERIC MAP(bit_size => double_word_length)
       PORT MAP(
           din => mul_out_array(54),
           clk => clk,
           rst => rst,
           enable => '1',
           dout => data_sum_add32(27));

--second level
Inst_second_level_add: for i in 0 to 6 generate
    Inst_adder8: adder GENERIC MAP (Nbit => double_word_length)
       PORT MAP(
           a => data_sum_add16(2*i),
           b => data_sum_add16(2*i+1),
           clk => clk,
           c => data_sum_add8(i)
       );
```

Appendix A (continued)

end generate Inst_second_level_add;

-- third level
Inst_third_level_add: for i in 0 to 2 generate
  Inst_adder4: adder GENERIC MAP (Nbit => double_word_length)
    PORT MAP(
      a => data_sum_add8(2*i),
      b => data_sum_add8(2*i+1),
      clk => clk,
      c => data_sum_add4(i)
    );
end generate Inst_third_level_add;

Inst_third_level_tail: register_par GENERIC MAP (bit_size => double_word_length)
  PORT MAP(
    din => data_sum_add8(6),
    clk => clk,
    rst => rst,
    enable => '1',
    dout => data_sum_add4(3)
  );

Inst_fourth_level_add: for i in 0 to 1 generate
  Inst_adder5: adder GENERIC MAP (Nbit => double_word_length)
    PORT MAP(
      a => data_sum_add4(2*i),
      b => data_sum_add4(2*i+1),
      clk => clk,
      c => data_sum_add2(i)
    );
end generate Inst_fourth_level_add;

-- last level
Inst_last_adder: adder GENERIC MAP (Nbit => double_word_length)
  PORT MAP(
    a => data_sum_add2(0),
    b => data_sum_add2(1),
    clk => clk,
    c => sample_out_in
  );
Appendix A (continued)

Listing A.28. Window 2

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.declaration_pack.all;
use IEEE.NUMERIC_STD.ALL;

entity window_2 is
  Port ( datain : in STD_LOGIC_VECTOR (double_word_length−1 downto 0);
        dataout : out STD_LOGIC_VECTOR (double_word_length−1 downto 0);
        clk : in STD_LOGIC;
        rst : in STD_LOGIC;
        go : in STD_LOGIC;
        go_loop : in std_logic;
        n_low : in STD_LOGIC_VECTOR (n_low_nhigh_length−1 downto 0);
        n_high : in STD_LOGIC_VECTOR (n_low_nhigh_length−1 downto 0);
        finish : out STD_LOGIC;
        calcola : out std_logic;
        loop_enable : out std_logic);
end window_2;

architecture Behavioral of window_2 is
  --signal count : unsigned(8 downto 0) := (others => '0');
signal count : integer;
signal finish1 : std_logic := '0';
begin
```

Appendix A (continued)

--- control Calcola Energia block
--- and let only the meaningful values pass

process(rst, datain, clk, go, n_low, n_high) is
variable tmp: integer;
begin
  if rst = '1' then
    tmp := 0;
    dataout <= (others => '0');
    calcola <= '0';
    finish1 <= '0';
  else
    if clk'event and clk = '1' then
      if go = '1' then
        tmp := tmp + 1;
        if (count = unsigned(n_high)) then
          finish1 <= '1';
        end if;
        if ((tmp >= unsigned(n_low))and(tmp <= unsigned(n_high))) then
          dataout <= datain;
          calcola <= '1';
        else
          dataout <= (others => '0');
          calcola <= '0';
        end if;
      else
        dataout <= (others => '0');
        calcola <= '0';
        tmp := 0;
      end if;
    end if;
    count<=tmp;
  end if;
end process;

--- control parametric loop back block

process(clk, rst, go, loop, n_low, n_high) is
variable tmp: integer;
begin
  if rst = '1' then
    tmp := 0;
  end if;

Appendix A (continued)

loop_enable <= '0';

else
    if clk'event and clk = '1' then
        if go_loop = '1' then
            tmp := tmp + 1;
            if (tmp >= unsigned(n_low) and tmp <= unsigned(n_high)) then
                loop_enable <= '1';
            else
                loop_enable <= '0';
            end if;
        else
            loop_enable <= '0';
            tmp := 0;
        end if;
    end if;
end if;
end process;

finish <= finish1;
end Behavioral;

Listing A.29. Energy computation block

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.declaration_pack.all;
use IEEE.NUMERIC_STD.ALL;

entity Calcolo_Energia is
    Port ( clk, rst : in STD_LOGIC;
           go : in STD_LOGIC;
           d_in : in STD_LOGIC_VECTOR (double_word_length-1 downto 0);
           d_out : out STD_LOGIC_VECTOR (total_word_length,out-1 downto 0));
end Calcolo_Energia;

architecture Behavioral of Calcolo_Energia is

COMPONENT mult_full
Appendix A (continued)

```
GENERIC(
    INBIT : integer := double.word_length
);
PORT(
    a : IN std_logic_vector(INBIT-1 downto 0);
    b : IN std_logic_vector(INBIT-1 downto 0);
    clk : IN std_logic;
    c : OUT std_logic_vector((INBIT+2-1) downto 0)
);
END COMPONENT;

signal product : std_logic_vector((total_word_length_out-1) downto 0) := (others => '0');
signal product1 : std_logic_vector((total_word_length_out-1) downto 0) := (others => '0');
signal go1, go2 : std_logic := '0';
signal total_energy : std_logic_vector(total_word_length_out-1 downto 0) := (others => '0');
begin
    process (clk, product) is
    begin
        if clk 'event and clk = '1' then
            -- product1 <= product;
            go1 <= go ;
            go2 <= go1;
        end if;
    end process;
    product1<=product;
    process (clk) is
    begin
        if clk 'event and clk = '1' then
            if rst = '1' then
                -- prima if reset era go1 = '0'
                total_energy <= (others => '0');
            else
                total_energy <= std_logic_vector(signed(total_energy) + signed(product1));
            end if;
        end if;
    end process;
    d_out <= total_energy;
end process;
```
Appendix A (continued)

INBIT=>double_word_length
)
PORT MAP(
a => d_in,
b => d_in,
clk => clk,
c => product
);

del Behavioral;

Listing A.30. Parametric Shift register

library IEEE;
use IEEE.std_logic_1164.all;
use work.declaration_pack.all;

entity shift_register_generic is
  generic (num_bit: integer := 16;
            num_stage: integer := 3);
  port (datain: in std_logic_vector(num_bit-1 downto 0);
        clk, rst, enable: in std_logic;
        dataout: out std_logic_vector(num_bit-1 downto 0));
end shift_register_generic;

architecture shift_register_struct of shift_register_generic is
component register_par is
  Generic (bit_size : integer :=16);
  Port (  din : in STDLOGIC_VECTOR (bit_size-1 downto 0);
          clk : in std_logic;
          rst : in std_logic;
          enable: in std_logic;
          dout : out STDLOGIC_VECTOR (bit_size-1 downto 0));
end component;
Appendix A (continued)

```vhdl
subtype pipe is std_logic_vector (num_bit-1 downto 0);
type p_line is array (0 to num_stage) of pipe;
signal pipeline: p_line;

begin

inst_shift_reg: for i in 0 to num_stage-1 generate
  first_gen: if i=0 generate
    inst_regn_0: register_par generic map (num_bit)
      port map (din=>datain, clk=>clk, rst=>rst, enable=>enable, dout=>pipeline(0));
  end generate first_gen;
other_gen: if i>0 generate
  inst_regn: register_par generic map (num_bit)
    port map (din=>pipeline(i-1), clk=>clk, rst=>rst, enable=>enable, dout=>pipeline(i));
  end generate other_gen;
end generate inst_shift_reg;
dataout <= pipeline(num_stage-1);
end shift_register_struct;
```

Listing A.31. Parametric register

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use work.declaration_pack.all;

entity register_par is
  Generic (bit_size : integer :=16);
  Port ( din : in STD_LOGIC_VECTOR (bit_size-1 downto 0);
      clk : in std_logic;
      rst : in std_logic;
      enable: in std_logic;
      dout : out STD_LOGIC_VECTOR (bit_size-1 downto 0));
end register_par;

architecture Behavioral of register_par is
```
Appendix A (continued)

begin
  regproc: process (clk, din, rst)
  begin
    if rst = '1' then
      dout <= (others => '0');
    elsif clk'event and clk = '1' then
      if enable = '1' then
        dout <= din;
      end if;
    end if;
  end process;
end Behavioral;

Listing A.32. Multiply with bit shift

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.declaration_pack.all;

entity mult_half is
  Port (a : in STD_LOGIC_VECTOR (word_length-1 downto 0);
        b : in STD_LOGIC_VECTOR (word_length-1 downto 0);
        c : out STD_LOGIC_VECTOR (word_length-1 downto 0));
end mult_half;

architecture Behavioral of mult_half is
signal c1 : STD_LOGIC_VECTOR ((2*word_length-1) downto 0) :=(others => '0');

begin
  process (a, b)
  begin
    c1 <= std_logic_vector(signed(a)*signed(b));
  end process;
end Behavioral;
end process;
c <= c1((2*word_length−1) downto (word_length));
end Behavioral;

Listing A.33. Multiply with full accuracy

library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.declaration_pack.all;

entity mult_full is
    Generic (INBIT : integer := double_word_length);
    Port ( a : in STD_LOGIC_VECTOR (INBIT−1 downto 0);
    b : in STD_LOGIC_VECTOR (INBIT−1 downto 0);
    clk : in std_logic;
    c : out STD_LOGIC_VECTOR ((INBIT*2−1) downto 0));
end mult_full;

architecture Behavioral of mult_full is

signal a1 , b1 : STD_LOGIC_VECTOR (INBIT−1 downto 0) := (others => '0');
begin

process (clk , a,b) is

begin

    if clk 'event and clk = '1' then
        a1 <= a;
b1 <= b;
    end if;
end process;

process ( a1,b1)
begin

c <= std_logic_vector(signed(a1)*signed(b1));
end process;
end Behavioral;
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.NUMERIC_STD.ALL;
use work.declaration.pack.all;

entity adder is
  Generic (Nbit : integer := 16);
  Port ( a : in STD_LOGIC_VECTOR (Nbit−1 downto 0);
         b : in STD_LOGIC_VECTOR (Nbit−1 downto 0);
         clk : in std_logic;
         c : out STD_LOGIC_VECTOR (Nbit−1 downto 0)));
end adder;

architecture Behavioral of adder is
signal a1, b1 : STD_LOGIC_VECTOR (Nbit−1 downto 0);
begin
process (clk , a, b) is
begin
  if clk 'event and clk = '1' then
    a1 <= a;
    b1 <= b;
  end if;
end process;

process (a1, b1) is
begin
  C <= std_logic_vector(signed(a1) + signed(b1));
end process;
end Behavioral;
Appendix A (continued)

Listing A.35. D type flip flop

```vhdl
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;

entity ffd1 is
  Port ( a : in STD_LOGIC;
          clk : in STD_LOGIC;
          o : out STD_LOGIC);
end ffd1;

architecture Behavioral of ffd1 is

begin
  process (clk,a) is
    begin
      if clk'event and clk = '1' then
        o <= a;
      end if;
  end process;
end Behavioral;
```

A.4 FSM

Listing A.36. System FSM

```vhdl
-- Company: PoliTO
-- Engineer: Paolo Rosingana, Francesco Colonna
--
-- Create Date: 13:03:53 10/20/2011
-- Design Name: 
-- Module Name: FSM - Behavioral
-- Project Name: 
-- Target Devices: 
-- Tool versions: 
-- Description: 
```
library IEEE;
use IEEE.STD_LOGIC_1164.ALL;
use IEEE.STD_LOGIC_ARITH.ALL;
use IEEE.STD_LOGIC_UNSIGNED.ALL;

entity FSM is

    port (rst, clk: in std_logic;
          count_W, count_Iter, count_ANT, count_FIR: in std_logic;
          START, finish_MIST: in std_logic;
          EN_CW, RST_CW: out std_logic;
          EN_CAWM, RST_CAWM: out std_logic;
          EN_CANT, RST_CANT: out std_logic;
          EN_CFIR, RST_CFIR: out std_logic;
          EN_CITER, RST_CITER: out std_logic;
          EN_WM, EN_SM, EN_EM, WE_EM: out std_logic;
          EN_CAS, RST_CAS: out std_logic;
          LOOP_RST: out std_logic;
          Load_SHIFT, Clear_SHIFT: out std_logic;
          EXE2START, RST_MIST: out std_logic)

);

end FSM;

architecture Behavioral of FSM is
Appendix A (continued)

component regN is generic (N: integer:=8);
    port (clk, reset: in std_logic;
        a: in std_logic_vector(N downto 0);
        b: out std_logic_vector(N downto 0);
        en, clr: in std_logic);
end component;

type state_type is (Idle, LoadA, LoadB, LoadC, LoadD, ExeA, ExeB, SaveA, IncrIter, wait_for_count_iter_end);
signal PState, FState: state_type;

begin

State_proc: process (PState, START, count_W_end, finish_MIST)
begin
    case PState is
        when Idle =>
            if START='1' then FState<= LoadA;
            else FState<= Idle;
            end if;

            RST_CW<='1';
            EN_CW<='0';
            EN_CAWM<='0';
            RST_CAWM<='1';
            EN_WM<='0';
            Load_SHIFT<='0';
            Clear_SHIFT<='1';
            test<="0010001000010100";
            numstate<="000";
            EN_SM<='0';
            EXE_START<='0';
            EN_CAS<='0';
            RST_CAS<='1';
            RST_MIST<='1';
    end case;
end process State_proc;

end;
Appendix A (continued)

when LoadA =>
  -- Enable Weights memory
  if count_iter_end = '1' then FState <= Idle;
  else FState <= LoadB;
  end if;

  RST_CW <= '1';
  EN_CW <= '0';
  EN_CAWM <= '0';
  RST_CAWM <= '0';
  EN_WM <= '1';
  Load_SHIFT <= '0';
  Clear_SHIFT <= '1';
  -- test <= "0000000000100100";
  -- numstate <= "001";
  EN_SM <= '0';
  EXE_START <= '0';
  EN_CAS <= '0';
  RST_CAS <= '1';
  RST_MIST <= '1';

  EN_CNT <= '0';
  RST_CNT <= '0';
  EN_CFIR <= '0';
  RST_CFIR <= '1';
  LOOP_RST <= '1';

  EN_EM <= '0';
  WR_EM <= '0';
  EN_CITER <= '0';
  RST_CITER <= '1';
Appendix A (continued)

EN_CITER <= '0';
RST_CITER <= '0';

when LoadB =>

-- load weights in shift register for one FIR
  if count.W.end='1' then FState<=LoadC;
  else FState<=LoadB;
end if;

RST_CW <= '0';
EN_CW <= '1';
EN_CAWM <= '1';
RST_CAWM <= '0';
EN_WM <= '1';
Load_SHIFT <= '1';
Clear_SHIFT <= '0';
-- test <= "0101110000110111";
-- numstate <= "010";
EN_SM <= '0';
EXE_START <= '0';
EN_CAS <= '0';
RST_CAS <= '1';
RST_MIST <= '1';

EN_CANT <= count.W.end;
RST_CANT <= '0';
EN_CFIR <= count.W.end;
RST_CFIR <= '0';
LOOP_RST <= '0';

EN_EM <= '0';
WR_EM <= '0';
EN_CITER <= '0';
RST_CITER <= '0';

when LoadC =>

-- loop back load for each FIR
  if count.ANT.end='1' or count.FIR.end='1' then
    FState<= LoadD;
  else
    FState <= LoadB;
  end if;

end if;

RST_CW <= '1';
EN_CW <= '0';
EN_CAWM <= '1';
RST_CAWM <= '0';
EN_WM <= '1';
Load_SHIFT <= '1';
Clear_SHIFT <= '0';
— test <= "0000000000000000";
— numstate <= "011";
EN_SM <= '0';
EXE_START <= '0';
EN_CAS <= '0';
RST_CAS <= '1';
RST_MIST <= '1';

EN_CANT <= '1';
RST_CANT <= '0';
EN_CFIR <= '1';
RST_CFIR <= '0';
LOOP_RST <= '0';

EN_EM <= '0';
WR_EM <= '0';
EN_CITER <= '0';
RST_CITER <= '0';

when LoadD =>
— complete load step
FState <= ExeA;

RST_CW <= '1';
EN_CW <= '0';
EN_CAWM <= '0';
RST_CAWM <= '0';
EN_WM <= '0';
Load_SHIFT <= '0';
Clear_SHIFT <= '0';
— test <= "0000000000000000";
Appendix A (continued)

```plaintext
numstate <="100";
EN_SM <='0';
EXE_START <='0';
EN_CAS <='0';
RST_CAS <='1';
RST_MIST <='1';
EN_CANT <='0';
RST_CANT <='0';
EN_CFIR <='0';
RST_CFIR <='0';
LOOP_RST <='0';

EN_EM <='0';
WE_EM <='0';
EN_CITER <='0';
RST_CITER <='0';

when ExeA =>
  -- Enable sample memories
  Fstate <= ExeB;

  RST_CW <='1';
  EN_CW <='0';
  EN_CAWM <='0';
  RST_CAWM <='0';
  EN_WM <='0';
  Load_SHIFT <='0';
  Clear_SHIFT <='0';

  EN_SM <='1';
  EXE_START <='0';
  EN_CAS <='0';
  RST_CAS <='1';
  RST_MIST <='1';
  LOOP_RST <='0';

  EN_CANT <='0';
  RST_CANT <='0';
  EN_CFIR <='0';
```

when ExeA...
Appendix A (continued)

RST_CFIR <= '0';
EN_EM <= '0';
WE_EM <= '0';
EN_CITER <= '0';
RST_CITER <= '0';

when ExeB =>
  -- Enable Computation
  if finish.MIST = '1' then
    if count.ANT_end = '1' then
      Fstate <= SaveA;
    else
      Fstate <= LoadA;
    end if;
  else
    Fstate <= ExeB;
  end if;

RST_CW <= '1';
EN_CW <= '0';
EN_CAWM <= '0';
RST_CAWM <= '0';
EN_WM <= '0';
Load_SHIFT <= '0';
Clear_SHIFT <= '0';

EN_SM <= '1';
EXE_START <= '1';
EN_CAS <= '1';
RST_CAS <= '0';
RST_MIST <= '0';
LOOP_RST <= '0';

EN_CANT <= '0';
RST_CANT <= '0';
EN_CFIR <= '0';
RST_CFIR <= '0';

EN_EM <= '0';
WE_EM <= '0';
when \texttt{SaveA} \Rightarrow
--- \textit{Enable Energy memory}
\begin{align*}
\text{Fstate} &= \text{Incr. Iter} ; \\
\text{RST\_CW} &= '1' ; \\
\text{EN\_CW} &= '0' ; \\
\text{EN\_CAWM} &= '0' ; \\
\text{RST\_CAWM} &= '0' ; \\
\text{EN\_WM} &= '0' ; \\
\text{Load\_SHIFT} &= '0' ; \\
\text{Clear\_SHIFT} &= '0' ; \\
\text{EN\_SM} &= '0' ; \\
\text{EXE\_START} &= '1' ; \\
\text{EN\_CAS} &= '0' ; \\
\text{RST\_CAS} &= '1' ; \\
\text{RST\_MIST} &= '0' ;
\end{align*}

\begin{align*}
\text{EN\_CANT} &= '0' ; \\
\text{RST\_CANT} &= '1' ; \\
\text{EN\_CFIR} &= '0' ; \\
\text{RST\_CFIR} &= '1' ; \\
\text{LOOP\_RST} &= '1' ;
\end{align*}

\begin{align*}
\text{EN\_EM} &= '1' ; \\
\text{WR\_EM} &= '1' ; \\
\text{EN\_CITER} &= '0' ; \\
\text{RST\_CITER} &= '0' ;
\end{align*}

when \texttt{Incr. Iter} \Rightarrow
--- \textit{Increment Count Iter}
\begin{align*}
\text{Fstate} &= \text{wait for count iter end} ; \\
\text{RST\_CW} &= '1' ; \\
\text{EN\_CW} &= '0' ; \\
\text{EN\_CAWM} &= '0' ; \\
\text{RST\_CAWM} &= '0' ;
\end{align*}


Appendix A (continued)

EN,WM< '0';
Load,SHIFT< '0';
Clear,SHIFT< '0';

EN,SM< '0';
EXE,START< '1';
EN,CAS< '0';
RST,CAS< '1';
RST,MIST< '0';

EN,CANT< '0';
RST,CANT< '0';
EN,CFIR< '0';
RST,CFIR< '0';
LOOP,RST< '0';

EN,EM< '1';
WE,EM< '1';
EN,CITER< '1';
RST,CITER< '0';

\texttt{when \ wait\_for\_count\_iter\_end} \Rightarrow
\texttt{--wait 1 clock cycle for count\_iter\_end}
Fstate<LoadA;

RST,CW< '1';
EN,CW< '0';
EN,CAWM< '0';
RST,CAWM< '0';
EN,WM< '0';
Load,SHIFT< '0';
Clear,SHIFT< '0';

EN,SM< '0';
EXE,START< '0';
EN,CAS< '0';
RST,CAS< '1';
RST,MIST< '0';
RST,MIST< '0';
LOOP,RST< '0';
Appendix A (continued)

EN\textsubscript{CANT}\llt '0';
RST\textsubscript{CANT}\llt '0';
EN\textsubscript{CFIR}\llt '0';
RST\textsubscript{CFIR}\llt '0';

EN\textsubscript{EM}\llt '0';
WE\textsubscript{EM}\llt '0';
EN\textsubscript{CITER}\llt '0';
RST\textsubscript{CITER}\llt '0';

end case;
end process  \texttt{State}\_proc;

\texttt{nextstateproc} : \texttt{process(rst, clk)}
begin
  if rst = '1' then
    Pstate \leftarrow \texttt{Idle};
  elsif clk'event and clk = '1' then
    Pstate \leftarrow \texttt{FState};
  end if;
end process  \texttt{nextstateproc};

end  \texttt{Behavioral};
Appendix B

COMMUNICATION INTERFACE SOFTWARE CODE

This Appendix contains all the software code required to communicate with the FPGA.

B.1 Host application

Listing B.1. Host application code

```c
#include <stdio.h>
#include <stdlib.h>
#include <errno.h>
#include <string.h>
#include <time.h>
#include <libusb-1.0/libusb.h>

#define MAX_LEN 256
#define INTERFACE_NUMBER 0
#define BULK_OUT_ENDPOINT 0x02
#define BULK_IN_ENDPOINT 0x86
#define MAX_DATA_OUT_SIZE 512
#define MAX_DATA_IN_SIZE 512
#define TIMEOUT_MS 5000
#define NVOX_CYCLE 195
#define NWEIGHTS_PER_VOX 504

void reset_fpga(libusb_device_handle *mydevh);
void download_weights(libusb_device_handle *mydevh, int pktLen, FILE *weightsFile);
void read_weights(libusb_device_handle *mydevh, int pktLen);
void read_energy(libusb_device_handle *mydevh, int pktLen, FILE *receivedF);
void bin2asciihex(unsigned char input, char * asciibyte);
void download_samples(libusb_device_handle *mydevh, int file_num);
int start_FPGA(libusb_device_handle *mydevh, int pktLen);
void asciiextobin(char *input, char * out);
void asciiintoint(char *input, char * out, int length);
void read_samples(libusb_device_handle *mydevh, int pktLen);
```
Appendix B (continued)

```c
int64_t timespecDiff(struct timespec *timeA_p, struct timespec *timeB_p)
{
    return ( (timeA_p->tv_sec * 1000000000) + timeA_p->tv_nsec) -
            ( (timeB_p->tv_sec * 1000000000) + timeB_p->tv_nsec);
}

int main(void)
{
    char cmd_buf[MAX_LEN], file_name[30];
    static const int VENDOR_ID = 0x04b4;
    static const int PRODUCT_ID = 0x8613;
    struct libusb_device_handle *mydevh = NULL;
    int device_ready = 0;
    int result, pktLen, i;
    int voxel, Nvox, weightsFileLen;
    int err;
    // tempoInitio, tempoFine;
    struct timespec tempoInizio, tempoFine;
    double Tcycle;
    FILE* weightsFile, *samplesFile[5], *energyFile, *energy_test;
    err=0;
    // Initialize USB device
    result = libusb_init(NULL);
    if (result < 0)
    {
        fprintf(stderr, "Failed_to_initialize_libusb\n");
        exit(1);
    }
    mydevh = libusb_open_device_with_vid_pid(NULL, VENDOR_ID, PRODUCT_ID);
    if (mydevh == NULL)
    {
        fprintf(stderr, "Unable_to_find_the_device.\n");
        exit(1);
    }
    libusb_detach_kernel_driver(mydevh, INTERFACE_NUMBER);
    result = libusb_claim_interface(mydevh, INTERFACE_NUMBER);
    if (result >= 0)
    {
        device_ready = 1;
    }
    libusb_set_configuration(mydevh, 1);
    ```
else
{
    fprintf(stderr, "libusb_claim_interface_error\n");
    exit(1);
}

if (device_ready)
{
    printf("Press ENTER to start");
    getchar();
    weightsFile = fopen("fpga_feed_hex.txt", "r");
    if (!weightsFile)
    {
        fprintf(stderr, "Cannot open fpga_feed_hex.txt\n");
    }
    else
    {
        energyFile = fopen("energy_FPGA.txt", "w");
        if (!energyFile)
        {
            fprintf(stderr, "Cannot open energy_FPGA.txt \n", file_name);
            err=1;
        }
        if (err==0)
        {
            reset_fpga(mydevh);
            for (i=0; i<5; i++)
                download_samples(mydevh, i+1);
            fseek(weightsFile, 0, SEEK_END);
            weightsFileLen = ftell(weightsFile);
            fseek(weightsFile, 0, SEEK_SET);
            Nvox=(weightsFileLen /5)/NWEIGHTS_PER_VOX;
            printf("total number of voxels: %d\n", Nvox);
            voxel=0;
            while (voxel<Nvox)
            {
                printf("voxel %d\n", voxel+1);
                if (Nvox-voxel<NVOX-cycle)
                {
                    download_weights(mydevh, (Nvox-voxel)*NWEIGHTS_PER_VOX, weightsFile);
                    if (start_FPGA(mydevh, Nvox-voxel)==0)
Appendix B (continued)

```c
{
    read_energy(mydevh, (Nvox-voxel)*2, energyFile);
    reset_fpga(mydevh);
} else {
    printf("error computing energy\n");
    voxel=Nvox;
} else {
    download_weights(mydevh, NVOX_cycle*NWEIGHTS_PER_VOX, weightsFile);
    if (start_FPGA(mydevh, NVOX_cycle)==0) {
        read_energy(mydevh, NVOX_cycle*2, energyFile);
        reset_fpga(mydevh);
    } else {
        printf("error computing energy\n");
    }
    voxel+=NVOX_cycle;
}
}
fclose(energyFile);
for (i<5; i=0; i++) {
    fclose(samplesFile[i]);
}
fclose(weightsFile);
}
return 0;
}

// Function to reset mips
// Send command with opcode 00000001 to ep2out
```
void reset_fpga(libusb_device_handle *mydevh) {
    unsigned char data_out[4];
    int bytes_transferred = 0;
    int r = 0;
    data_out[0] = 0x01;
    data_out[1] = 0x00;
    data_out[2] = 0x00;
    data_out[3] = 0x00;
    // printf(“Sending reset command...\n”);
    r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 4, &bytes_transferred, TIMEOUT_MS);
    if (r<0)
    {
        fprintf(stderr, “Error sending reset command.\n”);
        exit(1);
    }
}

// Function to send weights and delays values to Weights Memory
void download_weights(libusb_device_handle *mydevh, int pktLen, FILE *weightsFile) {
    unsigned char data_out[MAX_DATA_OUT_SIZE];
    int bytes_transferred = 0;
    int total_bytes_sent = 0;
    int r = 0;
    unsigned long i = 0;
    unsigned long remainder = 0;
    int j;
    uint32_t fwFileRest = 0;
    unsigned char *buffer;
    FILE *copy_f;

    // Send command
    data_out[0] = 0x02; // write command
    // Determine the length of data to be sent 32 bit
    data_out[3] = (char)(pktLen%256);
    data_out[2] = (char)((pktLen-data_out[3])%256);
    data_out[1] = (char)((pktLen-data_out[2]*256-data_out[3])>>16);
    printf(“weight command :\n”, data_out[0], data_out[1], data_out[2], data_out[3]);
    r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 4, &bytes_transferred, TIMEOUT_MS);
if (r<0)
{
    fprintf(stderr, "Error sending command.\n");
    exit(1);
}
while (i<pktLen*4)
{
    fwFileRest=pktLen*4-i;
    i+=512;
    if (fwFileRest<512)
    {
        remainder = fwFileRest;
    }
    /* data packets are 512 bytes long by default,
    smaller if less data is available*/
    else
    {
        for (j=0; j<512; j=j+4)
        {
            fscanf(weightsFile,"%s",&data_out[j]);
            asciiextobin(&data_out[j+2],&data_out[j+3]);
            asciiextobin(&data_out[j],&data_out[j+2]);
            data_out[j]=0;
            data_out[j+1]=0;
        }
        r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 512, &bytes_transferred,
        TIMEOUT_MS);
        if (r<0)
        {
            fprintf(stderr, "Error sending data via bulk transfer.\n");
            exit(1);
        }
        total_bytes_sent=total_bytes_sent+bytes_transferred;
    }
    if ((remainder<512)&&(remainder>0))
    {
        for (j=0; j<remainder; j=j+4)
        {
            fscanf(weightsFile,"%s",&data_out[j]);
            asciiextobin(&data_out[j+2],&data_out[j+3]);
            asciiextobin(&data_out[j],&data_out[j+2]);
            data_out[j]=0;
            data_out[j+1]=0;
        }
        fwFileRest=pktLen*4-i;
        i+=512;
        if (fwFileRest<512)
        {
            remainder = fwFileRest;
        }
        /* data packets are 512 bytes long by default,
        smaller if less data is available*/
        else
        {
            for (j=0; j<512; j=j+4)
            {
                fscanf(weightsFile,"%s",&data_out[j]);
                asciiextobin(&data_out[j+2],&data_out[j+3]);
                asciiextobin(&data_out[j],&data_out[j+2]);
                data_out[j]=0;
                data_out[j+1]=0;
            }
            r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 512, &bytes_transferred,
            TIMEOUT_MS);
            if (r<0)
            {
                fprintf(stderr, "Error sending data via bulk transfer.\n");
                exit(1);
            }
            total_bytes_sent=total_bytes_sent+bytes_transferred;
        }
    }
Appendix B (continued)

```c
asciihex2bin(&data_out[j], &data_out[j+2]);
data_out[j]=0;
data_out[j+1]=0;
}
r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, remainder, &bytes_transferred,
TIMEOUT_MS);
if (r<0)
{
    fprintf(stderr, "Error sending data via bulk transfer.
    exit(1);
}
total_bytes_sent=total_bytes_sent+bytes_transferred;
}
}

// Function to send the samples values to the Samples Memory
void download_samples(libusb_device_handle *mydevh, int file_num)
{
    unsigned char data_out[MAX_DATA_OUT_SIZE];
    char line[33], tmp_c;
    int bytes_transferred = 0;
    int total_bytes_sent=0;
    int r = 0;
    unsigned long i = 0;
    unsigned long remainder = 0;
    int j;
    FILE *fwFile;
    uint32_t fwFileLen = 0;
    uint32_t fwFileRest = 0;
    uint32_t pktLen = 0;
    unsigned char *buffer;
    char file_name[30];

    sprintf(file_name,"samples%d.mif", file_num);
    fwFile = fopen(file_name, "r");
    if (!fwFile)
    {
        fprintf(stderr, "Cannot open samples%d.mif\n", file_num);
        exit(1);
    }
```
Appendix B (continued)

```c
{   tmp_c = fgetc(fwFile);
    if(tmp_c == 'n')
        pktLen++;
} while (tmp_c != EOF);
ftime(&ftime, SEEK_SET);
printf(" File packets sent: \n", pktLen);

// Send command
// five most significant bit show the energy memory to be addressed
data_out[0] = 0x05 + (file_num-1)*8;
// Determine the length of data to be sent 32 bit
data_out[1] = (char)(pktLen%256);
data_out[2] = (char)(((pktLen-data_out[1])%256)/256);
data_out[3] = (char)(((pktLen-data_out[2])*256-data_out[1])>>16);

r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 4, &bytes_transferred, TIMEOUT_MS);
if (r<0)
{
    fprintf(stderr, " Error sending command. \n");
    exit(1);
}
printf(" Bytes sent: \n", bytes_transferred);
while(i<pktLen*4)
{
    fwFileRest=pktLen*4-i;
i+=512;
    if(fwFileRest<512)
    {
        remainder = fwFileRest;
    }
    /* data packets are 512 bytes long by default, smaller if less data is available*/
    else
    {
        for (j=0; j<512; j=j+4)
        {
            fscanf(fwFile,"%a", line);
```
Appendix B (continued)

```c
ascii2int(&line[8], &data_out[j+1], 8);
ascii2int(line, &data_out[j], 8);
ascii2int(&line[16], &data_out[j+2], 8);
ascii2int(&line[24], &data_out[j+3], 8);
}

r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 512, &bytes_transferred, TIMEOUT_MS);
if (r < 0)
{
    fprintf(stderr, "Error sending data via bulk transfer.\n");
    exit(1);
}

total_bytes_sent = total_bytes_sent + bytes_transferred;
}

if ((remainder < 512) && (remainder > 0))
{
    for (j = 0; j < remainder; j += 4)
    {
        fscanf(fwFile, "%a", line);
        ascii2int(&line[8], &data_out[j+1], 8);
        ascii2int(line, &data_out[j], 8);
        ascii2int(&line[16], &data_out[j+2], 8);
        ascii2int(&line[24], &data_out[j+3], 8);
    }
    r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, remainder, &bytes_transferred, TIMEOUT_MS);
    if (r < 0)
    {
        fprintf(stderr, "Error sending data via bulk transfer 2.\n");
        exit(1);
    }
    total_bytes_sent = total_bytes_sent + bytes_transferred;
}

printf("Bytes sent: %d\n", total_bytes_sent);
}

// Function to read weights values from Weights memory
void read_weights(libusb_device_handle *mydevh, int pktLen)
{

unsigned char data_out[4], asciibyte[2];
unsigned char data_in[MAX_DATA_IN_SIZE];
int bytes_transferred = 0;
int bytes_received = 0;
int bytesLen, i, j, fwFileRest, remainder;
int r = 0;
FILE *receivedF, *debug_f;

data_out[0] = 0x03;
data_out[3] = (char)pktLen%256;
data_out[2] = (char)((pktLen-data_out[3])/256)%256;
data_out[1] = (char)((pktLen-data_out[2]*256-data_out[3])>>16);

r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 4, &bytes_transferred, TIMEOUT_MS);
if (r<0)
{
    fprintf(stderr, "Error sending read weights command.\n");
    exit(1);
}

r=0;
bytesLen=i*pktLen;
receivedF=fopen("received.txt","w");
if (!receivedF)
{
    fprintf(stderr,"Cannot create received.txt\n");
    exit(1);
}

i=0;
while(i<pktLen*4)
{
    fwFileRest(pktLen*4-i);
i+=MAX_DATA_IN_SIZE;
    if(fwFileRest<MAX_DATA_IN_SIZE)
    {
        remainder = fwFileRest;
    }
    else
    {

Appendix B (continued)

```
remainder = MAX_DATA_IN_SIZE;
}  data_in [remainder] = 0x0F;
for (j = 0; j < remainder; j++)

if (remainder > 0)
{
    r = libusb_bulk_transfer (mydevh, BULK_IN_ENDPOINT, data_in, remainder, &bytes_received, 
        TIMEOUT_MS);
    if (r < 0)
    {
        fprintf (stderr, " Error receiving data via bulk transfer 2.\n"");
    }
    for (j = 0; j < remainder; j++)
    {
        if (j % 4 == 0 &amp; j != 0)
        {
            fprintf (receivedF, "\n");
        }
        bin2asciihex (data_in [j], asciibyte);
        fprintf (receivedF, "%s " , asciibyte);
    }
    fprintf (receivedF, "\n");
}
fclose (receivedF);

// printf (" Finished.\n");

// Function to read samples values from Samples memory
// not supported yet from hardware side
void read_samples (libusb_device_handle *mydevh, int pktLen)
{
    unsigned char data_out [4], asciibyte [2];
    char data_in [MAX_DATA_IN_SIZE];
    int bytes_transferred = 0;
    int bytes_received = 0;
    int bytesLen, i, j;
    int r = 0;
```
int file_num, fwFileRest, remainder;
char file_name[30];
FILE *receivedF;

printf("Which memory do you want to read? (1-5)\n");
scanf("%d", &file_num);

sprintf(file_name, "received_samples,%d.mif", file_num);
receivedF = fopen(file_name, "w");

if (!receivedF)
{
    fprintf(stderr, "Cannot open received_samples,%d.mif\n", file_num);
    exit(1);
}

data_out[0] = 0x06 + (file_num - 1) * 8;
data_out[3] = (char)pktLen % 256;
data_out[2] = (char)(((pktLen - data_out[3]) / 256) % 256);

r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 4, &bytes_transferred, TIMEOUT_MS);

if (r < 0)
{
    fprintf(stderr, "Error sending read_samples command.\n");
    exit(1);
}

printf("Bytes sent: %d.\n", bytes_transferred);
r = 0;
i = 0;
while (i < pktLen * 4)
{
    fwFileRest = pktLen * 4 - i;
i += MAX_DATA_IN_SIZE;
    if (fwFileRest < MAX_DATA_IN_SIZE)
    {
        remainder = fwFileRest;
    }
    else
    {
        remainder = MAX_DATA_IN_SIZE;
    }
Appendix B (continued)

```c
if (remainder >0)
{
    r = libusb_bulk_transfer(mydevh, BULKIN_ENDPOINT, data_in, remainder, &bytes_received,
                             TIMEOUT_MS);
    if (r<0)
    {
        fprintf(stderr, "Error receiving data via bulk transfer2.\n");
    }
    for (j=0; j<remainder; j++)
    {
        if (j%4==0 && j!=0)
        {
            fprintf(receivedF, "\n");
        }
        bin2asciihex(data_in[j], ascii_byte);
        fprintf(receivedF, "%c", ascii_byte);
    }
    fprintf(receivedF, "\n");
    fclose(receivedF);

    printf("Finished.\n");
}

// Function to read energy values from Energy Memory
void read_energy(libusb_device_handle *mydevh, int pktLen, FILE *receivedF)
{
    unsigned char data_out[4], ascii_byte[2];
    char data_in[MAX_DATA_IN_SIZE];
    int bytes_transferred = 0;
    int bytes_received = 0;
    int bytesLen, i, j, fwFileRest, remainder;
    int r = 0;
    FILE *debug;
```
Appendix B (continued)

```c
data_out[0] = 0x04;
data_out[1] = (char)(pktLen%256);
data_out[2] = (char)(((pktLen-data_out[3])%256)/256);
data_out[1] = (char)(((pktLen-data_out[2]*256-data_out[3])>>16));

r = libusb_bulk_transfer(mydevh, BULK_OUT_ENDPOINT, data_out, 4, &bytes_transferred, TIMEOUT_MS);
if (r<0)
{
    fprintf(stderr, " Error sending reset command.\n");
    exit(1);
}
debug=fopen("debug_energy.txt","w");
r=0;
bytesLen=4*pktLen;
i=0;
fwFileRest=0;
while (i<pktLen*4)
{
    fwFileRest=pktLen*4-i;
i+=MAX_DATA_IN_SIZE;
    if (fwFileRest<MAX_DATA_IN_SIZE)
    {
        remainder = fwFileRest;
    }
    else
    {
        remainder=MAX_DATA_IN_SIZE;
    }
    if (remainder>0)
    {
        r = libusb_bulk_transfer(mydevh, BULK_IN_ENDPOINT, data_in, remainder, &bytes_received, TIMEOUT_MS);
        if (r<0)
        {
            fprintf(stderr, " Error receiving data via bulk transfer 2.\n");
        }
        for (j=0; j<remainder; j++)
        {
            if (j%4==0 && j!=0)
            {
```
Appendix B (continued)

```c
{
    fprintf(receivedF,"\n");
    fprintf(debug,"\n");
}
bin2asciihex(data_in[],asciibyte);
fprintf(receivedF,"%a",asciibyte);
fprintf(debug,"%a",asciibyte);
}
fprintf(receivedF,"\n");
}
fclose(debug);
}

// Function to launch the Energy values computation
int start_FPGA(libusb_device_handle *mydev, int pktLen)
{
    unsigned char data_out[4], asciibyte[2];
    char * data_in;
    int bytes_transferred = 0;
    int bytes_received = 0;
    int bytesLen, i;
    int r = 0;
    data_out[0] = 0x07;
    data_out[3] = (char)pktLen%256;
    data_out[2] = (char)(((pktLen-data_out[3])/256)%256);
    data_out[1] = (char)(((pktLen-data_out[2]*256-data_out[3])>>16);
    r = libusb_bulk_transfer(mydev, BULK_OUT_ENDPOINT, data_out, 4, &bytes_transferred, TIMEOUT_MS);
    if (r<0)
    {
        fprintf(stderr, "Error_sending_start_command \n");
        exit(1);
    }
    r=0;
    bytesLen=4*pktLen;
    data_in=(char *)malloc(4*sizeof(char));
    //Wait for maximum TIMEOUT_MS for the handshake to signal computation completion
    r = libusb_bulk_transfer(mydev, BULK_IN_ENDPOINT, data_in, 4, &bytes_received, TIMEOUT_MS);
```
Appendix B (continued)

```c
if (r<0)
{
    printf(stderr, "Error receiving data via bulk transfer2 (start_fpga).\n")
    exit(1);
}
if (data.in[3]!=0x0F)
{
    printf("ERROR: Handshake with FPGA did not complete, possible incorrect computation\n");
    return(1);
}
return(0);
}

void bin2asciihex(unsigned char input, char * ascii byte)
{
    unsigned char in0, in1;
    in0=input & 0x0f;
    in1=input & 0xf0;
    if (! (in1 ^ 0x00)) ascii byte[0]='0';
    else if (! (in1 ^ 0x10)) ascii byte[0]='1';
    else if (! (in1 ^ 0x20)) ascii byte[0]='2';
    else if (! (in1 ^ 0x30)) ascii byte[0]='3';
    else if (! (in1 ^ 0x40)) ascii byte[0]='4';
    else if (! (in1 ^ 0x50)) ascii byte[0]='5';
    else if (! (in1 ^ 0x60)) ascii byte[0]='6';
    else if (! (in1 ^ 0x70)) ascii byte[0]='7';
    else if (! (in1 ^ 0x80)) ascii byte[0]='8';
    else if (! (in1 ^ 0x90)) ascii byte[0]='9';
    else if (! (in1 ^ 0xa0)) ascii byte[0]='a';
    else if (! (in1 ^ 0xb0)) ascii byte[0]='b';
    else if (! (in1 ^ 0xc0)) ascii byte[0]='c';
    else if (! (in1 ^ 0xd0)) ascii byte[0]='d';
    else if (! (in1 ^ 0xe0)) ascii byte[0]='e';
    else if (! (in1 ^ 0xf0)) ascii byte[0]='f';
    else ascii byte[0]='H';
    if (! (in0 ^ 0x00)) ascii byte[1]='0';
    else if (! (in0 ^ 0x01)) ascii byte[1]='1';
}
```
else if (! (in0 ˆ0x02)) asciibyte[1]= '2';
else if (! (in0 ˆ0x03)) asciibyte[1]= '3';
else if (! (in0 ˆ0x04)) asciibyte[1]= '4';
else if (! (in0 ˆ0x05)) asciibyte[1]= '5';
else if (! (in0 ˆ0x06)) asciibyte[1]= '6';
else if (! (in0 ˆ0x07)) asciibyte[1]= '7';
else if (! (in0 ˆ0x08)) asciibyte[1]= '8';
else if (! (in0 ˆ0x09)) asciibyte[1]= '9';
else if (! (in0 ˆ0x0a)) asciibyte[1]= 'a';
else if (! (in0 ˆ0x0b)) asciibyte[1]= 'b';
else if (! (in0 ˆ0x0c)) asciibyte[1]= 'c';
else if (! (in0 ˆ0x0d)) asciibyte[1]= 'd';
else if (! (in0 ˆ0x0e)) asciibyte[1]= 'e';
else if (! (in0 ˆ0x0f)) asciibyte[1]= 'f';
else asciibyte[1]= 'L';

asciibyte[2]= '\0';

}

void asciihextobin(char *input, char * out)
{
    int i;
    char val;
    val=0;
    for(i=0;i<2;i++)
    {
        val=val*16;
        if (*(*(input+i))>=97)
        {
            val += ( *(input+i) - 97 ) + 10;
        }
        else if ( *(input+i) >= 65)
        {
            val += ( *(input+i) - 65 ) + 10;
        }
        else
        {
            val += *(input+i) - 48;
        }
    }
}
Appendix B (continued)

```c
void asciiintoint(char *input, char *out, int length)
{
    int i, j;
    unsigned char val;
    if (length > 8)
        length = 8;
    val = 0;
    for (i = 0; i < length; i++)
    {
        val = val * 2;
        if (input[i] == '1')
            val += 1;
    }
    *out = val;
}

B.2 Timer application

Listing B.2. Timer host application

int main(void)
{
    char cmd_buf[MAX_LEN], file_name[30];
    static const int VENDOR_ID = 0x04b4;
    static const int PRODUCT_ID = 0x8613;
    struct libusb_device_handle *mydevh = NULL;
    int device_ready = 0;
    int result, pktLen, i;
    int voxel, Nvox, weightsFileLen;
    int err, fir, repetitions;
    clock_t overhead[2], total[2];
    struct timespec tempoinizio, tempofine;
    double Tcycle;
    FILE* weightsFile, *samplesFile[5], *energyFile, *energy_test, *timer_file;
```
Appendix B (continued)

er = 0;
// Initialize USB device
result = libusb_init(NULL);
if (result < 0)
{
    fprintf(stderr, "Failed to initialize libusb\n");
    exit(1);
}
mydevh = libusb_open_device_with_vid_pid(NULL, VENDOR_ID, PRODUCT_ID);
if (mydevh == NULL)
{
    fprintf(stderr, "Unable to find the device.\n");
    exit(1);
}
libusb_detach_kernel_driver(mydevh, INTERFACE_NUMBER);
result = libusb_claim_interface(mydevh, INTERFACE_NUMBER);
if (result != 0)
    device_ready = 1;
else
{
    fprintf(stderr, "libusb_claim_interface error.\n");
    exit(1);
}
if (device_ready)
{
    printf("Specify number of FIR filters:\n");
    scanf("%d", &fir);
    weightsFile = fopen("fpga_feed_hex.txt", "r");
    if (!weightsFile)
    {
        fprintf(stderr, "Cannot open fpga_feed_hex.txt\n");
    }
    else
    {
        if (err != 0)
        {
            reset_fpga(mydevh);
            for (i = 0; i < 5; i++)
                download_samples(mydevh, i + 1);
            fseek(weightsFile, 0, SEEK_END);
        }
weightsFileLen = ftell(weightsFile);
fsseek(weightsFile, 0, SEEK_SET);
Nvox=(weightsFileLen/5)/NWEIGHTS_PER_VOX;
printf("total number of voxels: %d\n",Nvox);
total[0]=clock();
for(repetitions=0; repetitions<N.REPETITIONS; repetitions++)
{
    voxel=0;
    printf("repetition: %d\n", repetitions);
    // fsseek(weightsFile, 0, SEEK_SET);
    while(voxel<Nvox)
    {
        if(Nvox-voxel<NVOX_cycle)
        {
            download_weights(mydevh, (Nvox-voxel)*NWEIGHTS_PER_VOX, weightsFile);
            if(start FPGA(mydevh, Nvox-voxel)==0)
            {
                read_energy(mydevh, (Nvox-voxel)*2, energyFile);
                reset_fpga(mydevh);
            }
            else
            {
                printf("error computing energy\n");
            }
            voxel=Nvox;
        }
        else
        {
            printf("error computing energy\n");
        }
        voxel+=NVOX_cycle;
    }
Appendix B (continued)

```c
}
}
total[1] = clock();
printf("Complete computation finished \n");
overhead[0] = clock();
for (repetitions = 0; repetitions < NREPETITIONS; repetitions++)
{
    voxel = 0;
    // fseek(weightsFile, 0, SEEK_SET);
    while (voxel < Nvox)
    {
        // printf("voxel %d\n", voxel + 1);
        if (Nvox - voxel < NVOX_cycle)
        {
            download_weights(mydevh, (Nvox - voxel) * NWEIGHTS_PER_VOX, weightsFile);
            if (1)
            {
                read_energy(mydevh, (Nvox - voxel) * 2, energyFile);
                reset_fpga(mydevh);
            }
            else
            {
                printf("error computing energy\n");
            }
            voxel = Nvox;
        }
        else
        {
            download_weights(mydevh, NVOX_cycle * NWEIGHTS_PER_VOX, weightsFile);
            if (1)
            {
                read_energy(mydevh, NVOX_cycle * 2, energyFile);
                reset_fpga(mydevh);
            }
            else
            {
                printf("error computing energy\n");
            }
            voxel += NVOX_cycle;
        }
    }
}
```
Appendix B (continued)

```c
overhead[1] = clock();

fopen("timings\results", "a");

printf("Total time = %Lf - %Lf - - > %Lf seconds\n", (long double) (total[1] + 0.0), (long double) (total[0] + 0.0), (long double) (((total[1] - total[0] + 0.0) / N_REPETITIONS) / CLOCKS_PER_SEC));

printf("Overhead time = %Lf - %Lf - - > %Lf seconds\n", (long double) (overhead[1] + 0.0), (long double) (overhead[0] + 0.0), (long double) (((overhead[1] - overhead[0] + 0.0) / N_REPETITIONS) / CLOCKS_PER_SEC));

printf("Computation time = %Lf - %Lf - - > %Lf seconds\n", (long double) ((total[1] - total[0] + overhead[0] - overhead[1] + 0.0) / N_REPETITIONS) / CLOCKS_PER_SEC));

fprintf(timer_file, "%d\t%.Lf\t%.Lf\t%.Lf\n", fir, (long double) (((total[1] - total[0] + 0.0) / N_REPETITIONS) / CLOCKS_PER_SEC), (long double) (((overhead[1] - overhead[0] + 0.0) / N_REPETITIONS) / CLOCKS_PER_SEC), (long double) (((total[1] - total[0] + overhead[0] - overhead[1] + 0.0) / N_REPETITIONS) / CLOCKS_PER_SEC));

fclose(timer_file);

for (i = 0; i < 5; i++)
{
    fclose(samplesFile[i]);
}

fclose(energyFile);

return 0;
```
Appendix C

COMMUNICATION INTERFACE SOFTWARE CODE

This Appendix contains the C language implementation of the image reconstruction problem.

Listing C.1. Image reconstruction program

```c
#include <stdio.h>
#include <stdlib.h>
#include <math.h>
#include <pthread.h>
#include <stdint.h>
#include <string.h>
#include "./CLAPACK/INCLUDE/f2c.h"
#include "./CLAPACK/INCLUDE/clapack.h"
#include <complex.h>
#include "./CLAPACK/INCLUDE/blaswrap.h"
#include <fftw3.h>

#define INPUTSET "FDTD2D_DG Const_tmm_med"
#define SPACE "2D"
#define SIMULT "MONO"
#define pulseTX "DIFFGAUSS"
#define NCAMP_FDTD 4696
#define DX_FDTD 0.05
#define FC 50
#define RES 0.1
#define C0 2.99792458E10 //light speed cm/s
#define VF 0.95E10 //cm/s
#define DT_FDTD DX_FDTD/(2.0*C0)
```

#define MU0 4.0*M_PI*1.0E-7 // vacuum permeability V s/(Am)
#define EPS0 1.0/(1E-4*C0*C0*MU0) // vacuum permittivity F/m
#define RESPHANTOM 0.05 // phantom space resolution (cm)
#define DC (int) round(1/(DT_FDTD * FC *1E9)) // decimation constant
#define FC_REAL 1/(DC*DT_FDTD) // sampling frequency after decimation
#define TS DT_FDTD*DC // Time step beamforming
#define NCAMPfin 199 // (int) floor (NCAMP_FDTD/DC)

#define MAX_ANT 50
#define DIM 2
#define MAX_PHANTOMDIM 400 // verify !!!!!!!!
#define NHX 9

/// SKAR PARAMETERS
#define N0 30
#define NFIN 60
#define J 3
#define P_EIGH 20

/// MIST BEAF PARAMETERS
#define L 55
#define NH 252
#define LH 5
#define EPSI 4
#define PULSE_START 25
#define NA 199

/// MIST BEAF CONSTANTS
#define MU0 4.0*M_PI*1.0E-7
#define MINFUWB 0.5e9
#define MAXFUWB 12e9

// /////////////
#define NTHREAD 1

//namefiletum = 'Ez_2D_TM_tumor_rec';
//namefilenotum = 'Ez_2D_TM_no_tumor_rec';

int MIST_param[] = {55,252,5,4,25}; //MIST param: [L, nh, th, epsi, pulse_start]

int i_index, j_index;
double pulse[NCAMPfin];
double bt[NRX][NCAMPfin], b[NRX][NCAMPfin];
double c[NRX][NCAMPfin], ct[NRX][NCAMPfin];
double SCR, SMR, TCR, CBR;
int receivers_pos2D[MAX_ANT][2];
float ***phantom2D;

double ***weights_mat;
int ***delays_mat;
double *energy_mat;
int num_vox;

int tumpos[3], tumd;

typedef struct parameters {
    unsigned int threadnum;
    unsigned int num_thread;
} PARMS;

int load_breast_2D(void);
int antennapos2D(void);
int skarT_MONO(double xfin[NRX][NCAMPfin], double blocal[NRX][NCAMPfin]);
void * child(void * arg);
void joining_fn (pthread_t *pid, int n);
int BEAF(int num_thread);

int main()
{
    int keepon=1;
    char cmd;
    load_breast_2D();

    while (keepon==1)
    {
        printf("Choose action:\n");
        printf("1. Simulate\n");
        printf("2. Interact with Modelsim/FPGA\n");
        printf("3. Exit\n");
scan("%c", &cmd);
switch(cmd)
{
    case '1':
        simulation();
        break;
    case '2':
        printf("Not yet implemented\n");
        break;
    case '3':
        keepon=0;
        break;
    default:
        printf("Invalid command, please retry\n");
}
}
return(0);
}

int simulation(void)
{
    char calib;
    int  i, j;
    double diff[NRX][NCAMPfin], maxtum, maxb, maxc, maxct;
    double c_t octave[NRX][NCAMPfin], c_c octave[NRX][NCAMPfin];
    FILE *skarFile;
    do
    {
        printf("Choose a calibration type:\n");
        printf("1-Ideal remotion\n");
        printf("2-Mist Skin Artifact Remotion\n");
        scanf("%c", &calib);
    } while (calib!='1' && calib!='2');

    maxtum=0;
    maxb=0;
    for(i=0; i<NRX; i++)
        for(j=0; j<NCAMPfin; j++)

Appendix C (continued)

```c
{ 
    diff[i][j]=bt[i][j]-b[i][j];
    if(fabs(diff[i][j])>maxtum)
    {
        maxtum=fabs(diff[i][j]);
    }
    if(fabs(b[i][j])>maxb)
    {
        maxb=fabs(b[i][j]);
    }
}
```

```c
switch(calib)
{
    case '1':
        CBR=20*log10(maxtum/maxb);
        TCR=-1;
        //skarT_MONO_prova(c);
        break;
    case '2':
        printf("Aplying SKAR to non-tumour breast ...
");
        skarT_MONO(c,b);
        /*
         * skarFile=fopen("skar_c0.bin","rb");*/
        /*
         * for(i=0;i<NRX;i++)*/
        /*
         * {*/
        /*
         *     fread(c_octave[i],sizeof(double),NCAMPfin,skarFile);*/
        /*
         * for(j=0;j<NCAMPfin;j++)*/
        /*
         * {*/
        /*
         *     printf("%.3E : ",(c_octave[i][j]-c[i][j])/c_octave[i][j];*/
        /*
         * }*/
        /*
         * printf("\n");*/
        /*
         */
        fclose(skarFile);
        printf("Aplying SKAR to tumour breast ...
");
        skarT_MONO(ct,bt);
        skarFile=fopen("skar_ct0.bin","rb");
        for(i=0;i<NRX;i++)
        {
            fread(ct_octave[i],sizeof(double),NCAMPfin,skarFile);
        }
        */
    }*/
```
Appendix C (continued)

```c
for (j = 0; j < NCAMPfin; j++)
{
    printf("%.3E: octave=%.3E local=%.3E\n", (ct_octave[i][j] - ct[i][j]) / ct_octave[i][j], ct_octave[i][j], ct[i][j]);
}
printf("\n");
fclose(skarFile);

maxc = 0;
maxct = 0;
for (i = 0; i < NRX; i++)
{
    for (j = 0; j < NCAMPfin; j++)
    {
        if (fabs(c[i][j]) > maxc)
        {
            maxc = fabs(c[i][j]);
        }
        if (fabs(ct[i][j]) > maxct)
        {
            maxct = fabs(ct[i][j]);
        }
    }
}
CBR = 20 * log10(maxct / maxb);
TCR = 20 * log10(maxum / maxc);
break;
    default:
    break;
}
BEAPset();
return;
}

int BEAP(int num_thread) {
    unsigned int i, vox, ant, w, j;
    int r;
```
double x;
char d;
clock_t cstart, cfinish;
char weights_string[5], line[35];
struct timespec ts_start;
struct timespec ts_end;
pthread_t *pid;
PARMS *inputs;
FILE *energyFile;

                                      // /////////////////////////////
printf("Starting BEAFF computation with %d threads over %d voxels and %d antennas...\n",
       num_thread, num_vox, NRX);

pid=(pthread_t *) malloc((num_thread)*sizeof(*pid));

inputs=(PARMS*) malloc((num_thread)*sizeof(*inputs));

                                        // /////////////////////////////
// Create N thread

clock_gettime(CLOCK_MONOTONIC, &ts_start);
// cstart=clock();
for (i=0; i < num_thread; i++) {
    // Initialize parameters
    inputs[i].threadnum = i;
    inputs[i].num_thread = num_thread;
    r = pthread_create(&pid[i], NULL, child, (void *) &inputs[i]);
    if (r) {
        printf("Error creating Thread #%d!", i);
        joining_fn (pid, i);
        return -1;
    }
    // printf("thread %d running\n", i);
}
// wait for all threads to finish
Appendix C (continued)

```c
joining fn (pid, num_thread);
// cfinish=clock();
clock_gettime(CLOCK_MONOTONIC, &ts_end);
energyFile=fopen("energy_par_swcomplete.bin", "wb");
fwrite(energy_mat, 8, num_vox, energyFile);
fclose(energyFile);
printf("%e\n", energy_mat[0]);
// printf("Computation time: %f\n", (float) (cfinish-cstart)/CLOCKS_PER_SEC);
printf("BEAF time: %f + %f = %f msec\n", (float) (ts_end.tv_nsec-ts_start.tv_nsec)/1000000000,
(ts_end.tv_nsec-ts_start.tv_nsec)*1000/1000/1000000);
free(pid);
free(inputs);
return 0;
}

void * child(void * arg) {
  int i, j, ant, w, voxel;
  double tmp, a;
  double result;
  PARMS * inputs = (PARMS *) arg;
  for (voxel=inputs->threadnum; voxel<num_vox; voxel=voxel+inputs->num_thread)
  {
    result=0;
    for (j=0; j=LH; j++)
    {
      tmp=0;
      for (ant=0; ant<NRX; ant++)
      {
        for (w=0; w<L; w++)
        {
          if (NH+j+L+w-delays_mat[voxel][ant]>NA-delays_mat[voxel][ant])
          {
            a=ct[ant][NH-L+w-delays_mat[voxel][ant]+j]*weights_mat[voxel][ant][w];
            tmp+=ct[ant][NH-L+w-delays_mat[voxel][ant]+j]*weights_mat[voxel][ant][w];
          }
        }
      }
    }
  }
}
```
Appendix C (continued)

```c
result+=tmp*tmp;
energy_mat[voxel]=result;

pthread_exit(NULL);
}

void joining_fn (pthread_t *pid, int n) {
    register short i;

    for (i = 0; i < n; i++) pthread_join (pid[i], NULL);
}

int BEAFset(double ***weighttot) {
    int ant, i, j, k, i_wl, i_wu, Lmid, K_tot, vox;
    int na, delta_epsilon, epsilon_inf;
    float SMR, SCR, inbreast, sumrect;
    float ***Lpath, MAXpath, cond_debye, tdebye;
    double wstep, wL, wU, lvett[L], tau, delay, tau0;
    fftw_plan p;
    double complex *sumV, ***V, tauw, S, Sa, Sb, epsilon_d, alfaabs;
    struct timespec ts_start;
    struct timespec ts_end;
    clock_t cstart, cfinish;

    printf("Starting BEAF weights computation ...
    
    //COMPUTE ROUND TRIP DISTANCE IN CM FOR EACH ANTENNA AND EACH VOXEL

    // Resolution
    wstep=1/(L*TS)*2*M_PI;
    wL=round(MNFUWB*2*M_PI/wstep)*wstep; // lower angular frequency evaluated
```
wu = round(MAX_FU_W * 2 * M_PI/wstep); // upper angular frequency evaluated

i_wl = round(wl/wstep); // lower angular frequency index
i_wu = round(wu/wstep); // upper angular frequency index

K_tot = i_wu - i_wl + 1; // number of frequencies evaluated

Lmid = ceil(L/2);
tau0 = TS * (L-1)/2;

// Debye equation and absorption evaluation
epsilon_inf = 7;
delta_epsilon = 3;
tdebye = 7.00e-12;
cond_debye = 0.15;

// printf("Debug 1\n");
pulsefreq = (fftw_complex*) fftw_malloc(L * sizeof(fftw_complex));
p = fftw_plan_dft_r2c_1d(L, &pulse[PULSESTART-1], pulsefreq, FFTW_ESTIMATE);
fftw_execute(p);
fftw_destroy_plan(p);

WeightfreqL = (fftw_complex*) fftw_malloc(NRX * L * sizeof(fftw_complex));
Weighttime = (fftw_complex*) fftw_malloc(NRX * L * sizeof(fftw_complex));

sumV = (double complex*) malloc(sizeof(double)*K_tot);
V = (double complex**) malloc(sizeof(double)*NRX);

for (ant = 0; ant < NRX; ant++)
{
    V[ant] = (double complex*) malloc(sizeof(double)*K_tot);
}

// printf("Debug 2\n");
Lpath = (float***) malloc(sizeof(float*Lpath)*NRX);
for (ant = 0; ant < NRX; ant++)
{
    Lpath[ant] = (float***) malloc(sizeof(float*Lpath)*i_index);
    for (i = 0; i < i_index; i++)
    {
        Lpath[ant][i] = (float**) malloc(sizeof(float*Lpath)*j_index);
    }
}
clock_gettime(CLOCK_MONOTONIC, &ts_start);
MAXpath = 0;
for (ant = 0; ant < NRX; ant++)
{
Appendix C (continued)

```c
for (i = 0; i < i_index; i++)
{
    for (j = 0; j < j_index; j++)
    {
        if ((Lpath[ant][i][j] = 2*RES*sqrt((i - receivers_pos2D[ant][0]) * (i - receivers_pos2D[ant][0]) + (j - receivers_pos2D[ant][1]) * (j - receivers_pos2D[ant][1])) > MAXpath)
            MAXpath = Lpath[ant][i][j];
    }
}
MAXpath = MAXpath / VF;
na = ceil(MAXpath / TS);
SMR = 0;
SCR = 0;
inbreast = 0;
sumrect = 0;
// printf("i_index=%d j_index=%d\n", i_index, j_index);
// printf("Paths length computed\n");
c_start = clock();
clock_gettime(CLOCK_MONOTONIC, &ts_start);

num_vox = 0;
for (i = 0; i < i_index; i++)
{
    for (j = 0; j < j_index; j++)
    {
        /*
         * printf("i=%d j=%d i_index=%d j_index=%d\n", i, j, i_index, j_index); */
        if (phantom2D[i][j] >= 0)
            {
                num_vox++;
            }
    }
}
weights_mat = (double ** *) malloc(sizeof(* weights_mat) * num_vox);
delays_mat = (int ** *) malloc(sizeof(* delays_mat) * num_vox);
energy_mat = (double **) malloc(sizeof(* energy_mat) * num_vox);
for (i = 0; i < num_vox; i++)
{
    
```
Appendix C (continued)

delays_mat[i] = (int*) malloc(sizeof(*delays_mat)*NRX);
weights_mat[i] = (double**) malloc(sizeof(**weights_mat)*NRX);
for (j = 0; j < NRX; j++)
{
    weights_mat[i][j] = (double*) malloc(sizeof(**weights_mat)*L);
}
}

vox = 0;
for (i = 0; i < index; i++)
{
    for (j = 0; j < index; j++)
    {
        if (phantom2D[i][j] > 0)
        {
            for (ant = 0; ant < NRX; ant++)
            {

            }
            delays_mat[vox][ant] = na-round(tau/TS);
            for (k = 0; k < K_tot; k++)
            {

            }
            Sa = cexp(-I*(wl+wstep*k)*Lpath[ant][i][j]/2)/VF;
            epsilon_d = epsilon_inf + (delta_epsilon / (1+(I*(wl+wstep*k)*tdebye)) + (cond_debye / (1*(wl+wstep*k)*EPS0));
            alfaabs = cabs(cimag(epsilon_d)) *(wl+wstep*k) *(VF/(C0*C0)) ; // absorption coefficient
            Sb = cexp(-alfaabs*Lpath[ant][i][j]) / 2;
            S = Sa*Sb*(1/sqrt(Lpath[ant][i][j]) / 2);
            tauw = cexp(I*(wl+k*wstep)*round(tau/TS)*TS);
            V[ant][k] = S*tauw*pulsefreq[i, wl+k];
        }
    }
    vox ++;
}

for (k = 0; k < K_tot; k++)
{
    sumV[k] = 0;
    for (ant = 0; ant < NRX; ant++)
    {
        sumV[k] += V[ant][k];
    }
}
Appendix C (continued)

```c
for (ant = 0; ant < NRX; ant++)
{
    for (k = 0; k < L; k++)
    {
        WeightfreqL[ant*L + k] = 0;
    }
    for (k = 1; k < K_tot; k++)
    {
        WeightfreqL[ant*L + k+iwl] = (V[ant][k] * cexp(I*(wl+k*wstep)*tau0)) / (cabs(V[ant][k]) * (sumV[k]+EPSI));
    }
    for (k = 1; k < iwu-1; k++)
    {
        WeightfreqL[ant*L + L-k] = conj(WeightfreqL[ant*L+k]);
    }
}

p = fftw_plan_dft_2d(NRX, L, WeightfreqL, Weighttime, FFTW_BACKWARD, FFTW_ESTIMATE);
fftw_execute(p);
fftw_destroy_plan(p);

for (ant = 0; ant < NRX; ant++)
{
    for (k = 0; k < L; k++)
    {
        weights_mat[vox][ant][k] = creal(Weighttime[k*NRX+ant]);
    }
    vox++;
}
}

clock_gettime(CLOCK_MONOTONIC, &ts_end);
finish = clock();
```
Appendix C (continued)

```c
printf("BEAF_weights_time : \%f - \%f = \%f m sec\n", (float)(ts_end.tv_sec-ts_start.tv_sec)*1000,((float)(ts_end.tv_nsec-ts_start.tv_nsec))/1000000000,((float)(ts_end.tv_nsec-ts_start.tv_nsec))/1000000);
printf("\%f - \%f = \%f nsec\n", (float)(ts_end.tv_sec),(float)(ts_start.tv_sec),(float)(ts_end.tv_nsec-ts_start.tv_nsec));
printf("\%f - \%f = \%f \%\\n", (float)(ts_end.tv_nsec),(float)(ts_start.tv_nsec),(float)(ts_end.tv_nsec-ts_start.tv_nsec));
printf("BEAF_weights_time : \%f\n", (float)(cfinish-cstart)/CLOCKS_PER_SEC);
free(WeightfreqL);
free(Weighttime);
free(pulsefreq);
BEAF(NTHREAD);
free(sumV);

for(ant=0;ant<NRX;ant++)
{
    free(V[ant]);
    free(V);
    // printf("Debug 2\n");
}

for(ant=0;ant<NRX;ant++)
{
    for(i=0;i<i_index;i++)
    {
        free(Lpath[ant][i]);
    }    
    free(Lpath[ant]);
    free(Lpath);

printf("%.3E\n",energy_mat[0]);

for(i=0;i<num_vox;i++)
{
    for(j=0;j<NRX;j++)
    {
        free(weights_mat[i][j]);
    }
}
```
Appendix C (continued)

```c
free(delays_mat[i]);
free(weights_mat[i]);
}
free(weights_mat);
free(delays_mat);
free(energy_mat);
return 0;
}

/* int skarT_MONO proved(double afin[NRX][NCAMP][m])* /
/* */
/* int W, m, i, j, ant, n; */
/* float qr[NRX][(2*J+1)*(NRX-1)]; */
/* double real sommaR[3*3], sommap[3]; */
/* double real eig[3], z[3*3]; */
/* double real work[1960], abstol; */
/* double Rp,inv[(2*J+1)*(NRX-1)][(2*J+1)*(NRX-1)]; */
/* integer iwork[5*3], ifail[3], info, matorder, il, ia, v, lda, ldz; */
/* char jobz, range, uplo, tmp; */
/* */
/* for(i=0; i<3; i++) */
/* { */
/*   for(j=i; j<3; j++) */
/* { */
/*     sommaR[i*3+j]=i; */
/*     sommaR[i+j*3]=i; */
/*   } */
/* } */
/* for(i=0; i<3; i++) */
/* { */
/*   for(j=0; j<3; j++) */
/* { */
/*   printf("%lf ",sommaR[i*3+j]); */
/* } */
/*   printf("n"); */
/* } */
/* */
/* jobz='V'; */
/* range = 'A'; */
/* uplo = 'U'; */
```

Appendix C (continued)

```c
/*
  matorder = 3; */
/*
  lda = 3; */
/*
  ldz = 3; */
/*
  il = 1; */
/*
  iu = 3; */
/*
  tmp = 'S'; */
/*
  abstol = 0; // dlamch('E'); */
/*
  v = 1960; */

  dsyevx(&jobz, 'R', 'U', &matorder, &lda, &il, &iu, &abstol, &a, &eigv, &z, &ldz, &work, &v, &iwork, &ifail, &info);
  printf(" eigenvectors:\n");
  for (i=0; i<3; i++){
    for (j=0; j<3; j++)
    { printf("%2.2E \n", z[j+i*3]);
    }
    printf("\n");
  }
  printf(" eigenvalues:\n");
  for (i=0; i<3; i++)
  { printf("%E \n", eigv[i]);
  }
  printf(" \n");
  printf(" ifail:\n");
  for (i=0; i<3; i++)
  { printf("%d \n", ifail[i]);
  }
  printf(" \n");
  printf(" info: %d\n", (int)info);
  printf(" abstol: %d\n", (int)abstol);
  printf(" lwork: %ld\n", (int)lwork);
  printf(" num eigenvalues: %d\n", (int)a);
}*/
```
Appendix C (continued)

```c
int skarT_MONO(double xfin[NRX][NCAMPfin], double b_local[][NCAMPfin])
{
    int W, m, i, j, k, ant, n, j_adj, i_adj, camp, start, a;
    float qr[NRX][(2*J+1)*(NRX-1)];
    double sommar[(2*J+1)*(NRX-1)], sommap[(2*J+1)*(NRX-1)];
    double eigv[(2*J+1)*(NRX-1)], z[(2*J+1)*(NRX-1)*2];
    double real work[1960], abstol;
    double Rp_inv[(2*J+1)*(NRX-1)][(2*J+1)*(NRX-1)], octaveR[(2*J+1)*(NRX-1)*2];
    int iwork[5*(2*J+1)*(NRX-1)], ifail[(2*J+1)*(NRX-1)], info, matorder, il, iu, lda, ldz, neig;
    char jobz, range, uplo, tmp, line[30];
    FILE *skarFile;
    struct timespec ts_start;
    struct timespec ts_end;

    printf("Starting SKAR calibration...
" );
    W = 2*J+1;
    m = NFIN-N0+1;
    clock_gettime(CLOCK_MONOTONIC, &ts_start);
    for (ant = 0; ant < NRX; ant++)
    {
        for (i = 0; i < W*(NRX-1); i++)
        {
            for (j = 0; j < W*(NRX-1); j++)
            {
                sommar[i*(W)*(NRX-1)+j] = 0;
            }
            sommap[i] = 0;
        }
        for (n = N0; n < (N0+m-1); n++)
        {
            for (i = 0; i < W*(NRX-1); i++)
            {
                if (i < ant*W)
                {
                    i_adj = 0;
                }
                else
                {
```
Appendix C (continued)

```c
i_adj = 1;
}
for (j = 0; j < W * (NRX - 1); j++)
{
    if (i < ant * W)
    {
        j_adj = 0;
    }
    else
    {
        j_adj = 1;
    }
    sommaR[i * W * (NRX - 1) + j] += local[i_adj + i / W][n - J + j / W] * b_local[j_adj + j / W][n - J + j / W] / m;
}
sommap[i] = sommap[i] / m;
```
Appendix C (continued)

abstol = 0; //2*dlaunum(&tmp);

v = 1960;

//memcpy(octaveR, sommaR, (2*J+1)*(NRX-1)*(2*J+1)*(NRX-1)*sizeof(double));

dsyevx_(&jobz, &range, &uplo, &matorder, sommaR, &lda, NULL, NULL, &il, &iu, &abstol, &neig, eigv, 
   z, &ldz, work, &v, iwork, ifail, &info);

for (i=0; i<Ws*(NRX-1); i++)
{
    for (j=0; j<Ws*(NRX-1); j++)
    {
        Rp_inv[i][j]=0;
    }
}

for (n=(W)*(NRX-1)-1; n>=W*(NRX-1)-P_EIGH; n--)
{
    for (i=0; i<Ws*(NRX-1); i++)
    {
        for (j=0; j<Ws*(NRX-1); j++)
        {
            Rp_inv[i][j]+=(z[n*Ws*(NRX-1)+i]*z[n*Ws*(NRX-1)+j])/eigv[n];
        }
    }
}

for (i=0; i<Ws*(NRX-1); i++)
{
    qr[ant][i]=0;
    for (j=0; j<(W)*(NRX-1); j++)
    {
        qr[ant][i]+=Rp_inv[i][j]*sommap[j];
    }

    /*
    printf("%.3E : ", qr[ant][i]);
    */
    
    /*
    printf("\n"); */
    
}
clock_gettime(CLOCK_MONOTONIC, &ts_end);
printf("SKAR_weights_time: \%f [\%f\%f\%fnsec\n", (float)(ts_end.tv_sec - ts_start.tv_sec)*1000,(float)(ts_end.tv_nsec - ts_start.tv_nsec))/1000,(float)(ts_end.tv_sec - ts_start.tv_sec)*1000+((float)(ts_end.tv_nsec - ts_start.tv_nsec))/1000000);

for (i = 0; i < We(NRX-1); i++)
{
    for (j = 0; j < We(NRX-1); j++)
    {
        printf("%.2E ", Rp_inv[i][j]);
    }
    printf("\n");
}

printf("Weights\nfor SKAR computed\n");
printf(" First level SKAR\n");
clock_gettime(CLOCK_MONOTONIC, &ts_start);
for (ant = 0; ant < NRX; ant++)
{
    // printf("STARTING ANT RIDE %d\n", ant);
    for (j = 0; j < NCAMPfin; j++)
    {
        if (j > J)
        {
            start = −J;
        }
        else
        {
            start = −j;
        }
        // printf("STARTING K RIDE %d\n", k);
        xt[ant][j] = b_local[ant][j];
        // printf("Here? %d\n", j);
        for (k = 0; k < NRX-1; k++)
        {
            if (k < ant)
            {
                i_adj = 0;
            }
            else
Appendix C (continued)

{ 
    i_adj=1;
}
for(camp=start; camp<i & j+camp<NCAMPfin; camp++)
{
    // x1[ant][j]=qr[ant][j+camp]*b_local[i+adj][j+camp];
    x1[ant][j]=b_local[i+adj][j+camp]*qr[ant][(k*W)+camp-start];
}
// printf("STARTING i RIDE %d\n", j);
/*
if (j>=W)*/
/*
    /*
        start=j-W+1;*/
    /*
        // printf("start:%d\n", start);*/
    */
/*
close/*
/*
    start=0;*/
/*
*/
/*
    if (((j>=J) & (j<J+NCAMPfin)))*/
/*
    /*
        // printf("j-J=%d", j-J);*/
    /*
        x1[ant][j]=b_local[ant][j-J];*/
    /*
        printf("tOK\n");*/
    */
/*
close/*
/*
    x1[ant][j]=0;*/
/*
*/
for (camp=start; camp<=j & camp<NCAMPfin; camp++)
/*
    */
/*
    printf("camp=\n->", camp);*/
/*
    x1[ant][j]=b_local[i+adj+k][camp]*qr[ant][i+(k*W)+camp-start];
*/
/*
    printf("DONE\n");*/
/*
*/
printf("COMPLETED i RIDE %d\n", j); */
Appendix C (continued)

```c
/*
 * printf("ant=%d COMPLETED K RIDE %d\n",ant,k);*/
*/
/*
 * printf("COMPLETED ANT RIDE %d\n",ant);*/
*/
// printf("Completed first level SKAR\n");
/*
 * for(ant=0; ant<NRX; ant++)*/
/*
 */
/*
 */
/*
 * x1[ant][j]=x1[ant][j+J];*/
/*
 */
/*
 */
// printf("Second level SKAR...\n");
for(ant=0; ant<NRX; ant++)
{
    for( j=0; j<NCAMPfin; j++)
    {
        if(j>=J)
        {
            start=-J;
        }
        else
        {
            start=-j;
        }
        // printf("STARTING K RIDE %d\n",k);
        x2[ant][j]=x1[ant][j];
        // printf("Here\n%",j);
        for(k=0;k<NRX-1;k++)
        {
            if(k<ant)
            {
                i_adj=0;
            }
            else
            {
                i_adj=1;
            }
            for(camp=start; camp<=J &
            if(camp<NCAMPfin; camp++)
```
Appendix C (continued)

```c
{
    x2[ant][j] += qr[ant][(k*W)+start]*x1[i_adj][j+camp];
}
}

for (k=0; k<NRX-1; k++) {
    if (k<ant) {
        i_adj = 0;
    } else {
        i_adj = 1;
    }
    for (j=0; j<NCAMPfin-W-1; j++) {
        if (j>W) {
            start = j-W+1;
        } else {
            start = 0;
        }
        if (j>=J && j<J+NCAMPfin) {
            x2[ant][j] = x1[ant][j-J];
        } else {
            x2[ant][j] = 0;
        }
        for (camp=start; camp<=j; camp++) {
            x2[ant][j] += x1[i_adj+k][camp]*qr[ant][(camp-start)*W+k];
        }
    }
}
}
clock_gettime(CLOCK_MONOTONIC, &ts_end);
for (ant=0; ant<NRX; ant++) {
    for (j=0; j<NCAMPfin; j++) {
        xfin[ant][j] = x2[ant][j];
    }
}
```
Appendix C (continued)

```c

printf("SKAR time: \%f + \%f = \%fmsec\n", (float)(ts_end.tv_sec-ts_start.tv_sec)*1000,((float)(ts_end.tv_nsec-ts_start.tv_nsec))/1000,(float)(ts_end.tv_sec-ts_start.tv_sec)*1000+(float)((ts_end.tv_nsec-ts_start.tv_nsec)/1000000));

// WEIGHTS QUANTIZATION WAS NOT IMPLEMENTED: WE DON'T NEED TO EVALUATE THE HARDWARE IMPLEMENTATION

printf("SKAR completed! \n");

for (i = 0; i < NRX; i++){
    free(qr[i]);
}
free(qr);
free(sommaR);
free(sommap);
free(eige);
free(z);

for (i = 0; i < (NRX-1)*W; i++){
    free(Rp_inv[i]);
}
free(Rp_inv);

for (i = 0; i < NRX; i++){
    free(x1[i]);
    free(x2[i]);
}
free(x1);
free(x2);
free(iwork);
free(ifail);
return 0;
```

```c
int load_breast_2D(void)
```
Appendix C (continued)

```c
int ii, jj, kk, i, j, k, index, Kres, k, slice, a;
char filename[60], line[30], id[10];
float tmp;
double sample;
double pulsetime[NCAMP_FDTD];
printf("Open Breast Info File . . . \n\n");
strcpy(filename, INPUTSET);
infoFile = fopen(strcat(filename, "/breastInfo.txt"), "r");
if (infoFile == NULL)
{
    printf("Impossible to open breastInfo file \n");
    return -1;
}
else
{
    scanf(infoFile, "breast ID=%s\n", id);
    scanf(infoFile, "s1=%d\n", &ii);
    scanf(infoFile, "s2=%d\n", &jj);
    scanf(infoFile, "s3=%d\n", &kk);
    sprintf(filename, ". / Phantom/%s /mtype.txt", id);
    breastFile = fopen(filename, "r");
    if (breastFile == NULL)
    {
        printf("Error opening breast phantom file \n");
        return -1;
    }
    Kres = RES / RES_PHANTOM;
    i_index = ceil((float)ii / Kres);
    j_index = ceil((float)jj / Kres);
    k_index = ceil((float)kk / Kres);
    phantom2D = (float**) malloc(sizeof(float*) * i_index);
    for (i = 0; i < i_index; i++)
    {
        phantom2D[i] = (float*) malloc(sizeof(float) * j_index);
    }
    // //////////////////////////////////////////////////OPTIMIZE
    printf("Reading breast shape . . . \n");
```
Appendix C (continued)

```c
printf("breast_size:
    ii=%d jj=%d kk=%d
    ii_index=%d jj_index=%d kk_index=%d
", id, ii, jj, kk, ii_index, jj_index, kk_index);
k_slice=floor(kk/2);
for(k=0;k<k_slice-1;k++)
{
    for(i=0;i<ii;i++)
    {
        //printf("i=%d\n",i);
        for(j=0;j<jj;j++)
        {
            scanf(breastFile,"%f",&tmp);
        }
    }
    a=0;
    for(j=0;j<jj;j++)
    {
        //printf("j=%d\n",j);
        for(i=0;i<ii;i++)
        {
            scanf(breastFile,"%f",&tmp);
            if((j%Kres==0)&&(i%Kres==0))
            {
                phantom2D[(int)(i/Kres)][(int)(j/Kres)]=tmp;
            }
        }
        //printf("\n");
    }
    //printf("\n");
}
//fclose(breastFile);
printf("Reading_antennas_positions...
");
if(antennapos2D()==-1)
    return -1;
else
{
    for(i=0;i<NRX;i++)
    {
        for(j=0;j<DIM;j++)
        {
```
\textbf{Appendix C (continued)}

\begin{verbatim}
receivers_pos2D[i][j] = ceil(receivers_pos2D[i][j] / Kres);

strcpy(filename, INPUTSET);
printf("Reading	umor_position...
\n");
tumposFile=fopen strcat(filename, "/tumpos.txt", "r");

\textbf{fscanf(tumposFile, 
\texttt{"%s", line);} // discard text line
\textbf{for} (i=0; i<3; i++)
{
\textbf{fscanf(tumposFile, 
\texttt{"%d", &tumpos[i];}} // read tumor coordinate
  tumpos[i] = ceil(tumpos[i] / Kres);
}
\textbf{fscanf(tumposFile, 
\texttt{"%s", line);} // discard text line
\textbf{fscanf(tumposFile, 
\texttt{"%d", &tumd);} // read tumor diameter
  tumd=(int) round(tumd/(RES*0.1));
fclose(tumposFile);

strcpy(filename, INPUTSET);
pulsetimeFile=fopen strcat(filename, "/pulse_DIFFGAUSS.txt", "r");
\textbf{for} (i=0; i<NCAMP_FDTD; i++)
{
\textbf{fscanf(pulsetimeFile, 
\texttt{"%lf", &pulsetime[i];}}
}
fclose(pulsetimeFile);

printf("Reading	breast_signal_response...
\n");
\textbf{for} (i=0; i<NRX; i++)
{
\textbf{strcpy(filename, INPUTSET);}
\textbf{sprintf(line, 
\texttt{"/MONO/Ez_2D_TM_tumor_rec%d.txt", i+1);}
filetum=fopen strcat(filename, \
\texttt{line}, "r");
\textbf{if} (filetum==NULL)
{
  printf("Error_opening_tumor_file_%s\n", filename);
  \textbf{return} -1;
}
\textbf{strcpy(filename, INPUTSET);}
\textbf{sprintf(line, 
\texttt{"/MONO/Ez_2D_TM_no_tumor_rec%d.txt", i+1);}
filenotum=fopen strcat(filename, \
\texttt{line}, "r");
\end{verbatim}
Appendix C (continued)

```c
if (filenotum==NULL)
{
    printf("Error opening tumor file \"%s\n", filename);
    return -1;
}
for (j=0; j<NCAMPfin; j++)
{
    bt[i][j]=0;
    b[i][j]=0;
    for (k=0; k<DC; k++)
    {
        fscanf(filetum, "%.lf", &sample);
        bt[i][j]+=sample;
        fscanf(filenotum, "%.lf", &sample);
        b[i][j]+=sample;
    }
    bt[i][j]=bt[i][j]/DC;
    b[i][j]=b[i][j]/DC;
}
for (j=0; j<NCAMPfin; j++)
{
    bt[i][j]=bt[i][0];
    // printf("%.2E", bt[i][j]);
    b[i][j]=b[i][0];
}
printf("\n");
fclose(filetum);
fclose(filenotum);
for (k=0; k<NCAMPfin; k++)
{
    pulse[k]=0;
    for (j=0; j<DC; j++)
    {
        pulse[k]+=pulsetime[k+DC*j];
    }
    pulse[k]=pulse[k]/DC;
}
```
Appendix C (continued)

```c

int antennapos2D (void)
{
    char line[30];
    int tx, rx, i, j;
    int i_pos, j_pos, ant;
    int nant;
    FILE *antennaFile;
    strcpy(line, INPUTSET);
    antennaFile = fopen(strcat(line, "/antennas2D.txt"), "r");  // Open the file
    if(antennaFile=NULL)
    {
        printf("Impossible_to_open_antennas_file\n");
        return -1;
    }
    else
    {
        while(getc(antennaFile)!='\n');  // Read in the transmitter header
        fscanf(antennaFile,"%d\n",&tx);  // Get line number of transmitter
        printf("tx:%d\n",tx);
        for(i=0;i<tx;i++)
        {
            while(getc(antennaFile)!='\n');  // Read transmitter coordinates
        }
        while(getc(antennaFile)!='\n');  // Read in the receiver header
        fscanf(antennaFile,"%d\n",&rx);  // Get line number of receiver
        printf("rx:%d\n",rx);
        for(i=0;i<rx;i++)
        {
            fscanf(antennaFile,"%d%d%d",&ant,&i_pos,&j_pos);  // Read transmitter coordinates
        }
    }
}
```

```
Appendix C (continued)

```c
receivers_pos2D[ant - 1][0] = i_pos;
receivers_pos2D[ant - 1][1] = j_pos;
}
fclose(antennaFile);
nant = rx;
}
if (nant != NRX)
{
    printf("Error while reading antennas file: receivers number is higher than expected: %d\n", nant);
    return -1;
}
return 0;
```
Appendix D

USER MANUAL

Here all the steps required to correctly perform the image reconstruction are listed. The system used to communicate with the FPGA is a general purpose machine running Ubuntu 11.10 operating system.

1. As a first step, the FPGA needs to be programmed using Xilinx Impact tool;

2. Once the USB interface has been connected to the computer, we need to program the Cypress FX2 microchip.

To do so on Windows, a specific Cypress software tool can be used.

On Ubuntu, we need to find out the virtual bus on which the FPGA usb interface has been connected by the system. To do so, we need to run inside a shell the command lsusb. The output will be of the type:

Bus BBB Device DDD: ID XXXX:XXXX Cypress FX2...

Where BBB and DDD identify the virtual bus and device. With this two numbers, we will be able to program the FX2 chip. A specific command for this purpose is included in the Linux kernel, and is fxload. The form in which the command must be run is the following:

sudo fxload -D /dev/bus/usb/BBB/DDD -I [path-to-hostapp] -t fx2 -vv
3. The host application needs then to be compiled if it is not yet. To compile it the libusb 1.0 library needs to be included to the system. The compiling instruction is of the following form:

```
gcc -o [path-for-executable] [path-of-source-code] -lusb-1.0
```

4. Next step is to launch the host application. It needs to be executed with root privileges to correctly access the usb interface. The files containing the values to be stored into the system memories need to be placed in the same directory of the host application. The files containing the samples values need to be ascii text file with values expressed in binary form, with names going from `samples_1.mif` to `samples_5.mif`. The file containing the weights and delay values need to be an ascii text file with the values expressed in hexadecimal form, with name `fpga_feed_hex.txt`

5. Energy results will be automatically written into a file named `energy_FPGA.txt` with hexadecimal form.


VITA

<table>
<thead>
<tr>
<th>Name:</th>
<th>Francesco Colonna</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Master of Science in Electrical Engineering and Computer Science, University of Illinois at Chicago, 2013</td>
</tr>
<tr>
<td>Honors/Awards:</td>
<td>2011: winner of Piero Mossino scholarship for academic excellence in the field of engineering sponsored by FIAT group.</td>
</tr>
</tbody>
</table>